Univerzitet u Nišu
Elektronski fakultet

Maksimović M. Dejan

Simulacija digitalnih sistema hibridnim simulatorom Alecsis2.1

Magistarska teza

Niš, jun 1995.

Sadržaj

1 Uvod

Sa porastom složenosti elektronskih kola, mogućnost greške pri njihovom projektovanju raste, a otkrivanje greške postaje otežano. Greška se može otkriti proizvodnjom i testiranjem prototipa, ali takav postupak je skup i zahteva mnogo vremena. Simulacija elektronskih kola daje odgovor na ovaj problem. Simulacijom kola na nivou električne šeme (električna analiza, analogna simulacija) ili na nivou logičke šeme (logička, digitalna simulacija) može se verifikovati ispravan rad kola pre proizvodnje.

Električna analiza bazira se na formiranju i rešavanju sistema jednačina kojim se opisuje ponašanje kola na nivou električne šeme. Promenljive u sistemu su naponi, struje i druge analogne veličine u kolu. Rešavanje sistema sa velikim brojem promenljivih zahteva mnogo memorije i računarskog vremena. Zato se električnom analizom mogu simulirati samo kola relativno male složenosti.

Logička simulacija primenjuje se za analizu rada digitalnih sistema opisanih na nivou logičke šeme. Pri logičkoj simulaciji kroz kolo propagiraju logička stanja umesto analognih veličina, a primenom principa narednog dogadjaja i principa selekcije puteva postiže se maksimalno ubrzanje procesa simulacije. Zahvaljujući tome moguće je simulirati kola mnogo veće složenosti nego električnom analizom. Na žalost, logička simulacija primenjiva je samo kod digitalnih sistema. Treba imati na umu da su električna analiza i logička simulacija usko povezane. Električna analiza nalazi svoje mesto u karakterizaciji osnovnih gradivnih elemenata digitalnih kola. Recimo, NI kolo se simulira na nivou električne šeme da bi se ekstrahovala kašnjenja sa kojima kasnije radi logički simulator.

U novije vreme ASIC (Application Specific Integrated Circuit) sistemi kombinuju analogne i digitalne module u jednu nedeljivu celinu, te se pokazalo da simulatori sa mogućnošću kombinovanja električne analize i logičke simulacije predstavljaju nezamenjivo orudje pri njihovom projektovanju. Ideja hibridne simulacije nije nova, ali se hibridni simulatori značajnijih upotrebnih karakteristika pojavljuju tek zadnjih desetak godina. Hibridna simulacija je otvorila novo polje u simulacionim tehnikama i istraživači su zadnjih godina prionuli na razvoj novih hibridnih simulatora. Osmišljavanje novih principa simulacije i modeliranja daje ovim simulatorima niz novih mogućnosti, tako da se oni često mogu primeniti i van oblasti simulacije elektronskih kola.

Paralelno sa razvojem simulatora razvijaju se i jezici koji služe kao njihov interfejs sa korisnikom. Uloga jezika je da omogući modeliranje svih relevantnih pojava u simuliranom kolu, opisivanje topološke strukture kola i zadavanje komandi za simulaciju. Od pojave VHDL-a (Very high speed integrated circuits Hardware Description Language) došlo je do zaokreta u razvoju

jezika za opis elektronskih kola u smeru univerzalne primenjivosti. Ranije je ulazni jezik simulatora bio prilagodjen specifičnoj nameni simulatora. VHDL je okupio u jednu celinu sve najboje osobine različitih jezika za opis digitalnih elektronskih kola, obuhvativši na taj način sve specifične oblasti primene digitalnih simulatora. VHDL podrazumeva funkcionalno modeliranje tamo gde se ranije podrazumevalo ugradjivanje odgovarajućih mehanizama u samo programsko jezgro simulatora. Iz oblasti modeliranja digitalnih sistema, ova tendencija se zadnjih godina prenosi i u oblast analognih sistema, gde se pojavljuju mnogi novi algoritmi za modeliranje. Moderni simulatori realizuju se kao univerzalne simulacione mašine - sadrže samo minimalni skup simulacionih mehanizama, a sav posao oko prilagodjavanja konkretnim simulacionim potrebama prelazi u nadležnost ulaznog jezika. U takvoj situaciji, da bi simulator imao upotrebnu vrednost, neophodno je razviti simulacione biblioteke. Simulacione biblioteke posebno su značajne kod logičkih simulatora koji koriste VHDL ili neki sličan ulazni jezik.

VHDL je u simulaciju uneo još jednu važnu karakteristiku - objektno-orjentisan pristup modeliranju i simulaciji. Zahtev da se sve digitalne komponente modeliraju korišćenjem paralelnih procesa u skladu je sa objektno-orjentisanim pogledom na svet. Komponente sistema su nezavisni objekti koji neprekidno obavljaju svoje funkcije, a sinhronizovanjem njihove medjusobne razmene informacija ostvaruje se funkcija sistema. Kao najznačajniji objektno-orjentisan jezik danas, C++ vrši značajan uticaj na razvoj modernih jezika za opis hardvera.

Hibridni simulator pod imenom ALECSIS (Analog and Logic Electronic Circuit Simulation System) razvijen je na Elektronskom fakultetu u Nišu kao rezultat višegodišnjeg razvojnog procesa čiji je cilj bio stvaranje orudja za simulaciju sistema koji u sebi usko integrišu kako analogne, tako i digitalne komponente. Prva verzija pod radnim nazivom MIX zaživela je davne 1992. godine i sadržavala je analogni simulator sa osnovnim programskim jezgrom za hibridnu simulaciju, dok su mehanizmi za logičku simulaciju bili tek u razvoju. Godinu dana kasnije značajno usavršena verzija simulatora dobila je ime Alecsis2.0. Ovaj simulator bio je opskrbljen mehanizmima za funkcionalno modeliranje digitalnih kola dovoljnim za ozbiljnu logičku simulaciju. Simulator sa oznakom 2.0 sadržavao je osnovne mehanizme za objektno-orjentisani prisup modeliranju hardvera, a najnovija verzija 2.1 sadrži najnovija poboljšanja ovih mehanizama koja su rezultat testiranja mogućnosti simulatora i pokušaja proširenja polja njegove primene.

Ulazni jezik simulatora Alecsis2.1 nazvan je AleC++ i predstavlja nadgradnju programskog jezika C++ jezičkim konstrukcijama za opisivanje elektronskih kola i zadavanje parametara simulacije. To je objektno-orjentisan jezik za opis hibridnih kola. Istim jezičkim konstrukcijama opisuju se i analogni i digitalni sistemi, pri čemu je vodjeno računa da se postigne maksimalna moguća konzistentnost. Mehanizmi funkcionalnog modeliranja obezbedjuju modeliranje analognih i digitalnih komponenti do tog nivoa da ugradnja modela bilo koje komponente u sam simulator nije potrebna. Sve komponente, analogne i digitalne, mogu se modelirati spolja. Radi povećanja brzine rada simulatora, u programsko jezgro su ipak ugradjene osnovne električne komponente (otpornik, tranzistor, ...). Digitalne komponente nisu ugradjene i da bi se simulator koristio za logičku simulaciju, neophodno je razviti biblioteke osnovnih komponenti.

Tema ovog rada je razvoj digitalnih biblioteka za simulator Alecsis2.1 pomoću kojih se on osposobljava za efikasnu logičku simulaciju. Po pravilu, sa porastom kardinalnog broja logičkog simulatora (broj stanja u sistemu stanja), raste kvalitet modeliranja i količina informacije koju je moguće simulacijom dobiti. Pri razvoju biblioteka može se krenuti od različitih sistema stanja. Svaki sistem stanja daje odredjene mogućnosti i nameće odredjena ograničenja. Zato je razvijeno više biblioteka. Biblioteke zasnovane na jednostavnijim sistemima stanja koriste se u početnim fazama projektovanja, a u završnim fazama potrebne su biblioteke koje obezbedjuju detaljnu proveru funkcionisanja projektovanog kola.

Drugo poglavlje može se podeliti na dva dela. U prvom delu biće ukratko izloženi opšti koncepti i problemi koji se rešavaju u oblasti hibridne simulacije i logičke simulacije. U drugom delu biće predstavljen simulator Alecsis2.1. Pri tome ćemo se zadržati na mogućnostima njegove primene za logičku simulaciju. Biće ukratko opisane neke osobine jezika AleC++ i dati jednostavni primeri koji bi trebalo da omoguće praćenje izlaganja u narednim poglavljima.

Treće poglavlje sadrži detaljan opis postupka razvoja biblioteke za logičku simulaciju koja podržava sistem sa dva logička stanja. Osim modeliranja osnovnih logičkih elemenata, biće opisano i modeliranje PLA struktura.

Četvrto i peto poglavlje bave se razvojem biblioteka za sistem sa tri i četiri stanja. Kod sistema sa tri stanja razmotrene su specifičnosti koje proizilaze iz uvodjenja neodredjenog stanja. Proširivanjem sistema stanja stanjem visoke impedanse dobija se sistem sa četiri stanja. Ovaj sistem omogućava simulaciju rada logičkih elemenata poznatih pod imenom trostatički drajveri. Sistem sa četiri stanja izabran je i za ilustraciju modeliranja memorijskih struktura tipa ROM i RAM.

U šestom i sedmom poglavlju biće ilustrovane mogućnosti simulatora Alecsis2.1 da emulira rad logičkog simulatora HILO koji ima ugradjen sistem sa petnaest logičkih stanja i hibridni simulator PSpice koji ima sistem sa pet stanja. Razvijene su digitalne biblioteke koje "pretvaraju" simulator Alecsis2.1 po potrebi u simulator HILO ili PSpice. Modelirane su specifičnosti ovih simulatora i pokazano je da simulator Alecsis2.1 može uspešno da se nosi sa svim mehanizmima modeliranja koje ova dva simulatora nude korisniku.

U osmom poglavlju biće opisana specijalizovana digitalna biblioteka za simulaciju hazarda u kombinacionim kolima bez povratne sprege. Ova biblioteka zasniva se na sistemu sa devet logičkih stanja. Biće ilustrovana primena biblioteke za analizu hazarda u kolima projektovanim korišćenjem razvoja Reed-Muller-a [Davi87, Hurs85].

Digitalna kola sa dva logička stanja dominiraju današnjom elektronikom. Za specifične namene koriste se i kola sa više od dva stabilna stanja. U devetom poglavlju opisana je digitalna biblioteka za simulaciju višeznačnih digitalnih kola, a kao najčešće korišćen, izabran je sistem višeznačne logike sa četiri logička stanja. Izloženi opšti principi mogu se primeniti u razvoju biblioteka sa drugim brojem stanja. Modelirani su osnovni logički elementi nižeg nivoa apstrakcije i višeznačna PLA struktura.

Ilustracije u ovom radu nacrtane su korišćenjem programa CorelDRAW, verzija 2.0. Grafički postprocesor Art2.1 [Gloz94a] korišćen je za generisanje svih talasnih oblika (rezultata simulacije) priloženih u radu.

Na kraju treba istaći da se svaka od razvijenih biblioteka koristi nezavisno i pretvara simulator Alecsis2.1 u logički simulator odredjenih karakteristika. Može se reći da je u ovom radu opisano sedam različitih logičkih simulatora projektovanih primenom ulaznog jezika AleC++ i simulacionih mehanizama inkorporiranih u simulatoru Alecsis2.1.

2 Logička simulacija upotrebom hibridnog simulatora Alecsis2.1

U ovoj glavi pozabavićemo se osnovnim mehanizmima za logičku simulaciju koje nudi simulator Alecsis2.1. Pri tome nam nije namera da ih detaljno objašnjavamo, s obzirom da postoji literatura u kojoj je to učinjeno [Gloz94a, Gloz94b], već da čitaoca uvedemo u ove mehanizme kako bi lakše pratio naredna poglavlja u kojima se oni koriste za razvoj biblioteka za logičku simulaciju.

2.1 Hibridna simulacija

Hibridni simulator obavlja simulaciju elektronskih kola u kojima su istovremeno prisutni i analogni i digitalni signali. Stoga on obuhvata dva različita simulaciona mehanizma: električnu analizu i logičku simulaciju.

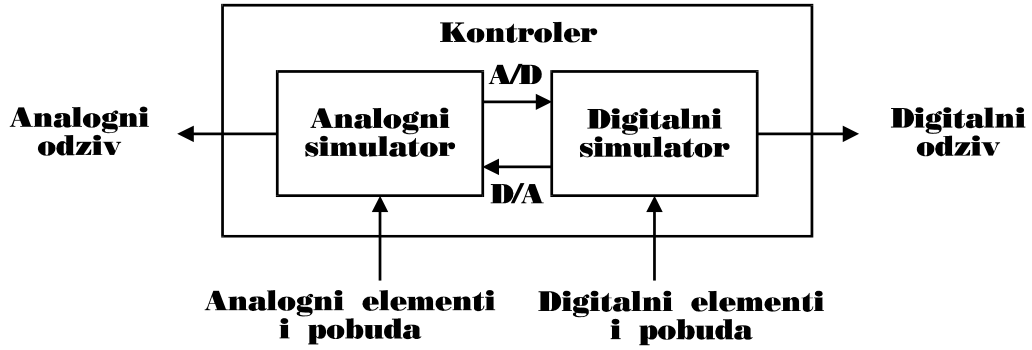
Iako su teorijski osnovi električne analize postavljeni mnogo ranije, može se smatrati da era električne analize elektronskih kola počinje pojavom simulatora SPICE [Nage75]. Do danas je razvijeno niz programa za električnu analizu koji se baziraju na istim osnovnim algoritmima na kojima je bio zasnovan SPICE. Mnogi od njih prilagodjeni su radu na PC računarima, poput MICROCAP-a, PSpice-a, IsSpice-a [Rowe91], što ih čini vrlo popularnim medju projektantima elektronskih kola.

Električna analiza zahteva formulisanje sistema linearnih jednačina kojima se opisuje ponašanje kola i njegovo rešavanje kako bi se odredile vrednosti napona, struja ili drugih analognih veličina koje su promenljive u sistemu jednačina. Ponašanje nelinearnih električnih kola opisuje se diferencijalnim nelinearnim jednačinama. Rešavanje sistema ovih jednačina zasnovano je na diskretizaciji vremenske ose (aproksimacija izvoda) i iterativnim postupcima (linearizacija nelinearnih jednačina) poput Newton-Raphson-ovog [Lito91a, Mrča92]. Električno kolo moguće je analizirati u jednosmernom režimu (DC analiza), u vremenskom domenu (analiza prelaznih režima, TR analiza) i u frekventnom domenu (AC analiza). DC analiza podrazumeva formiranje sistema jednačina kojima se kolo opisuje u jednosmernom režimu i njegovo rešavanje. Kod analize u vremenskom domenu kolo sadrži vremenski zavisne pobudne generatore, tako da je potrebno sistem jednačina formirati i rešavati iznova u svakom vremenskom trenutku. U najjednostavnijem slučaju, ako je kolo linearno, a korak diskretizacije vremenske ose konstantan, broj formiranja i rešavanja sistema jednačina proporcionalan je vremenu završetka analize. Ako je kolo nelinearno, u svakom vremenskom trenutku uvodi se iterativni postupak koji zahteva višestruko ponavljanje formiranja i rešavanja sistema jednačina, tako da je trajanje simulacije još mnogo puta duže. AC

analiza podrazumeva promenu frekvencije pobudnih generatora u kolu i analizu rada kola na svakoj zahtevanoj frekvenciji. Sistem kompleksnih jednačina formira se za svaku zadatu vrednost frekvencije. Treba napomenuti da se sistem kompleksnih jednačina obično rešava formiranjem sistema jednačina sa realnim koeficijentima dvostruko veće dimenzije, čime rastu memorijski zahtevi. Osim ovih osnovnih vrsta simulacije, neki električni simulatori imaju mogućnost da obave i analizu šuma, statističku (Monte Carlo) analizu, analizu najgoreg slučaja, parametarsku analizu, analizu osetljivosti i druge specifične tipove analiza.

Iz prethodno rečenog jasno je da problemi sa električnom analizom nastaju kod složenih elektronskih kola, kada je trajanje simulacije veoma dugo, a zahtevi simulatora za memorijom veoma veliki. Praksa je pokazala da već elektronska kola sa preko 100 promenljivih u sistemu jednačina predstavljaju skoro nepremostivu prepreku za elektronski simulator. Tada je, ako se radi o digitalnim elektronskim kolima, moguće pribeći logičkoj simulaciji [Lito91b]. Na ovaj način trajanje simulacije se drastično skraćuje, s tim što se redukuje količina dobijene informacije - umesto analognih vrednosti napona dobijaju se logička stanja.

Postoje, medjutim, situacije kada je neophodno da se manipuliše sa oba tipa signala. Primera radi, može da se poseduje biblioteka dobro karakterisanih logičkih ćelija koje, prirodno, mogu da se simuliraju digitalnim simulatorom, a istovremeno da se kreira nova logička ćelija koja je karakterisana samo na električnom nivou. Drugi, očigledniji primer je hibridno kolo koje je sastavljeno i od analognih elemenata (otpornika, tranzistora, kondenzatora i slično) i od logičkih elemenata (I, ILI, flip-flop i slično). Ovakvo kolo, jednostavno, mora da se simulira hibridnim simulatorom.



Slika 2.1 *Arhitektura hibridnog simulatora*

Arhitektura hibridnog simulatora prikazana je na slici 2.1. Može se smatrati da se hibridni simulator sastoji od tri komponente: analognog simulatora, digitalnog simulatora i kontrolera.

Analogni simulator prihvata opis analognog dela simuliranog sistema, opise A/D i D/A sprege, analogne pobudne signale i konvertovane signale na mestu D/A sprege. Na osnovu ovih podataka analogni simulator obavlja električnu analizu i kreira analogni odziv.

Digitalni simulator prihvata opis logičkog dela simuliranog sistema, opis A/D i D/A sprege, digitalne pobudne signale i konvertovane signale na mestu A/D sprege. Na osnovu ovih podataka digitalni simulator obavlja logičku simulaciju i kreira digitalni odziv.

Zadatak kontrolera je da na osnovu podataka koje korisnik saopšti pomoću ulaznog jezika i na osnovu fiksnih biblioteka modela kreira sve potrebne strukture podataka za analogni i digitalni simulator [Maks92]. Tokom hibridne simulacije, kontroler sinhronizuje rad oba simulatora, omogućava razmenu podataka u hibridnim čvorovima (mesta A/D i D/A konverzije) i upravlja strukturama podataka.

Moguća su četiri pristupa razvoju hibridnog simulatora. Prvi pristup je sprezanje programa za električnu analizu i programa za logičku simulaciju trećim programom [Corm88, Mrča94]. Drugi pristup je korišćenje analognog simulatora kao osnove i ugradnja logičke simulacije u njega. Ovaj pristup se sreće recimo kod simulatora PSpice [-91]. Treći pristup je korišćenje logičkog simulatora kao osnove [Agra80, Thel88]. Konačno, poslednji pristup je realizacija jedinstvenog programa za

hibridnu simulaciju - takozvanog integrisanog hibridnog simulatora [Acun90, Chad92]. Logično, ovaj pristup je najefikasniji, jer omogućava optimalnu realizaciju struktura podataka i simulacionih mehanizama. Ovakvim simulatorom mogu se analizirati i čisto analogna kola, kao da je u pitanju električni simulator i čisto digitalna kola, kao da je u pitanju logički simulator.

Simulatoru treba saopštiti neophodne informacije o strukturi simuliranog kola i komande za simulaciju. Projektanti simulatora razvijali su ulazne jezike za svoje simulatore unapredjujući već postojeće jezike svojim originalnim doprinosom, tako da postoji veliki broj različitih jezika za opis kako analognih, tako i digitalnih sistema. Jezici za opis analognih elektronskih kola znatno su unapredjeni tek zadnjih godina sa pojavom novih jezičkih mehanizama koji omogućavaju korisniku intervenciju direktno u strukturi matrice sistema, automatsko diferenciranje i linearizaciju nelinearnih komponenti [Kazm91, Feld92, Kazm93]. Medju jezicima za opis digitalnih sistema došlo je do standardizacije - zahvaljujući zalaganju velikih svetskih proizvodjača digitalnih integrisanih kola jezik VHDL [-90] usvojen je kao standardni jezik za razmenu informacija. Na razvoju jezika za opis hibridnih sistema intenzivno se radi [Zwol91] i treba očekivati skoru standardizaciju u ovoj oblasti.

2.2 Logička simulacija

Sa porastom složenosti integrisanih kola, iznalaženi su algoritmi koji su omogućavali njihovu simulaciju. Kod analognih elektronskih kola velike složenosti električna analiza moguća je zahvaljujući metodima dekompozicije sistema jednačina i makromodeliranju [Petk92]. Logička simulacija predstavlja odgovor na povećanje složenosti **digitalnih** integrisanih kola. Još davnih šezdesetih godina razradjeni su osnovni algoritmi logičke simulacije i nastali prvi logički simulatori [Szyg72, Flak74, Chap74, Giam79, Stev83, Bush83]. Danas je u upotrebi mnoštvo logičkih simulatora medju kojima su najznačajniji HILO [Harr84, Harr85] i TEGAS [Szyg72].

Umesto analognim vrednostima napona (struja, naelektrisanja, ...) u elektronskom kolu, logički simulator rukuje logičkim stanjima. Dakle, da bi se uopšte započela logička simulacija, potrebno je izvršiti konverziju analognih veličina u logička stanja. Osnovni sistem stanja u današnjim digitalnim kolima je sistem sa dva stanja, to su logička nula i logička jedinica. Prvi logički simulatori bazirali su se na ovom sistemu stanja. Problemi oko inicijalizacije (odredjivanja početnih stanja u svim tačkama simuliranog logičkog kola pre početka simulacije) uslovili su uvodjenje dodatnog, trećeg logičkog stanja u logičke simulatore. To je neodredjeno stanje, obično označavano sa 'U' (unknown), 'X' ili '*' [Nash80]. Sistem sa tri stanja, medjutim, nedovoljno je deskriptivan pri simulaciji trostatičkih kola. Zato je simulatorima sa tri stanja dodato i stanje visoke impedanse (uobičajena oznaka 'Z'), kao četvrto [Wilc79]. Sistem sa ova četiri stanja ('0', '1', 'X' i 'Z') pokazao se dovoljnim za simulaciju svih značajnijih pojava u digitalnoj logici i na njega se oslanja najveći broj logičkih simulatora koji su danas u upotrebi.

Treba, medjutim, primetiti da stanje visoke impedanse ne bi smelo da se smatra stanjem u pravom smislu te reči. Kad izlaz trostatičkog kola ode u stanje visoke impedanse, to ne znači da je na izlazu uspostavljen neki odredjeni naponski nivo, već samo označava da je izlaz drajvera postao neaktivan. Pri tome, ako je kapacitivnost izlazne veze velika, na njoj se može još izvesno vreme držati prethodno stanje. Ako ova veza povezuje izlaze više trostatičkih kola, onda neko od njih koje nije u stanju visoke impedanse može da odredi stanje na vezi. U stanju logičke jedinice i u stanju logičke nule drajver ima daleko manju izlaznu otpornost od otpornosti drajvera u stanju visoke impedanse. Može se reći da su stanja logičke jedinice i logičke nule jača od stanja visoke impedanse. Nameće se zaključak da nije važan samo naponski nivo na izlazu drajvera, već i njegova izlazna impedansa. Jači je drajver sa manjom izlaznom impedansom. Dakle, stanje na izlazu odlikuje se naponskim nivoom i jačinom. Konzistentno se pridržavajući ovakvog pristupa, moguće je razviti mnogo složenije sisteme stanja. Primena sistema sa tri stanja ('0', '1' i 'X') i dve jačine (mala i velika izlazna impedansa) opisana je u [Holt81]. I simulator HILO zasniva se na jednom takvom sistemu sa petnaest stanja [Flak83]. U [Coel89, Coel90] opisana je standardna

VHDL biblioteka za logičku simulaciju zasnovana na dva logička stanja ('0' i '1') i pet jačina: 'F', 'R', 'W', 'Z' i 'D'. Jačina 'F' (forced) predstavlja direktnu vezu na izvor za napajanje, jačina 'R' (strong resistive) predstavlja vezu sa izvorom za napajanje preko otpornosti male vrednosti, a jačina 'W' (weak resistive) preko otpornosti velike vrednosti. Jačinu 'Z' ima tačka u kolu izolovana beskonačnom otpornošću od izvora za napajanje u kojoj se zadržava prethodno stanje beskonačno dugo (prisutna je parazitna kapacitivnost bez mogućnosti oticanja, odnosno doticanja naelektrisanja). Jačina 'D' (disconnect) ne predstavlja nikakvo stanje, što odgovara izolovanom čvoru bez kapacitivnosti. Logički sistem formiran od ovih vrednosti stanja i jačina ima 46 vrednosti. Rukovanje tabelama istinitosti čak i najjednostavnijih logičkih elemenata u ovakvom sistemu stanja postaje naporno. Pri izboru sistema stanja treba se rukovoditi minimalističkim pristupom: treba izabrati sistem sa minimalnim brojem stanja koji daje zadovoljavajuću tačnost simulacije [Menc90].

Sledeća bitna karakteristika logičkog simulatora je način modeliranja kašnjenja logičkih elemenata. Osim propagacionog kašnjenja, u model kašnjenja spadaju i druge vremenske karakteristike poput vremena postavljanja i držanja flip-flopova [Evan78], ograničenja u trajanju impulsa u pojedinim tačkama kola i slične. Medju najvažnije modele kašnjenja spada nulto kašnjenje, jedinično kašnjenje, dodeljivo kašnjenje, rise/fall kašnjenje i precizno kašnjenje [Abra90, Lito91b]. Kašnjenje elementa moguće je u principu primeniti na ulazu (pre izračunavanja logičke funkcije) ili na izlazu (posle odredjivanja izlaznog stanja). Kašnjenje se približno linearno povećava sa kapacitivnim opterećenjem izlaza komponente i ovu osobinu modeliraju bolji logički simulatori. Postoje takodje i složeniji modeli kašnjenja [Faza88], ali se retko sreću kod postojećih simulatora. Da bi se omogućio proizvoljan izbor modela kašnjenja, potrebno je obezbediti mehanizam u ulaznom jeziku simulatora kojim bi korisnik definisao model kašnjenja, a ova osobina simulatora potencira se tek od pojave jezika VHDL.

Jedna od takodje nezaobilaznih pojava u digitalnim kolima koju logički simulator mora da modelira je žičana logika. Pod ovim izrazom se podrazumeva spajanje izlaza više logičkih elemenata u istu tačku. Ako su u pitanju standardni gejtovi koji na izlazu mogu imati stanje logičke nule i stanje logičke jedinice, žičana logika omogućava ostvarivanje I i ILI logičke funkcije bez upotrebe odgovarajućeg gejta. Ako se vezuju izlazi trostatičkih kola (nekad se nazivaju drajverima magistrale) koja imaju osobinu da svoj izlaz mogu da učine neaktivnim, odnosno da odu u stanje visoke impedanse, zajednička izlazna tačka naziva se magistrala ili bus. Ovaj način povezivanja koristi se da bi se istom vezom prenosili signali više drajvera u vremenskom multipleksu. Ako je više od jednog drajvera magistrale aktivno, rezultujuće stanje magistrale definiše se zavisno od jačina drajvera, odnosno od vrednosti njihovih izlaznih impedansi. Takodje, ako izlazi svih drajvera postanu neaktivni, stanje magistrale zavisi od ekvivalentne parazitne kapacitivnosti i odvodnosti prema izvorima za napajanje [Kamb83, Sher81]. Za odredjivanje rezultujućeg stanja magistrale, a takodje za slučaj žičane logike, simulator mora da raspolaže rezolucionim funkcijama i mehanizmom pomoću koga korisnik specificira rezolucionu funkciju koja treba da se primeni za odredjeni čvor u kolu. Sistemi stanja koji sadrže i informaciju o jačini drajvera pojednostavljuju rešavanje problema rezolucije [Levi81, Flak83].

Logička simulacija bazira se na manipulaciji veoma složenim strukturama podataka koje sadrže informacije o putevima prostiranja signala u simuliranom kolu i listom budućih dogadjaja [Maks92]. Da bi se obezbedila zadovoljavajuća brzina rada simulatora, razradjeni su mnogi algoritmi za efikasnu manipulaciju strukturama podataka [Ulri80, Kroh81, Shen90, Nich93]. Zahvaljujući povećanju brzine rada savremenih računara i uvodjenju paralelne obrade podataka, ovi algoritmi nemaju više tako veliki značaj tako da se pri projektovanju simulatora može posvetiti više pažnje njegovoj univerzalnosti i mehanizmima za modeliranje.

Da bi se digitalno kolo simuliralo, potrebno ga je opisati nekim jezikom razumljivim kako čoveku, tako i simulatoru. Od šezdesetih godina nastalo je više desetina jezika za opis digitalnog hardvera (Hardware Description Language - HDL). Većina ovih jezika bazira se na usvajanju jednog sistema logičkih stanja, razradi načina modeliranja osnovnih logičkih komponenti (ILI kolo, NI kolo, invertor, flip-flopovi, ...) i načina modeliranja osnovnih pojava u digitalnim kolima (žičana

logika, hazard, kašnjenje, jačine stanja, ...), te ugradnji ovih rešenja u strukturu jezika. VHDL predstavlja do danas najznačajniji pokušaj da se razvije univerzalniji alat za modeliranje i simulaciju [Lips89]. Nastao je kao jezik za opis digitalnog hardvera, pre samog pokušaja da se stvori simulator. Bio je to, isprva, ambiciozni spisak zahteva, ali vreme je pokazalo da je moguće razviti simulatore traženih karakteristika. Od kako je prihvaćen kao standard od strane uticajnih svetskih organizacija i proizvodjača integrisanih kola, VHDL je postao merilo za razvoj logičkih simulatora. Simulatori koji nemaju mogućnost simulacije pojava koje VHDL može da opiše nisu više "dovoljno dobri". U okviru VHDL-a definisane su ideje koje ćemo izneti u daljem tekstu, a moderni simulatori digitalnih kola zasnivaju rad na njima.

VHDL simulator nema predefinisan (ugradjen) sistem stanja. Korisnik simulatora definiše koji sistem stanja će se koristiti pri simulaciji.

VHDL simulator može, ali ne mora da ima ugradjene logičke primitive, odnosno osnovne digitalne komponente kao što su invertor, NI kolo i druge, čijom kombinacijom korisnik opisuje strukturu modeliranog kola. Simulator, medjutim, mora imati mehanizam kojim korisnik spolja modelira logičke primitive. Takodje, neophodno je omogućiti parametrizovanje razvijenih modela. Od ovakvih modela kreiraju se biblioteke koje moraju biti jednostavne za upotrebu i ne smeju usporavati rad simulatora, ni imati prevelike memorijske zahteve.

VHDL simulator mora omogućiti korisniku definisanje nekih neophodnih funkcija za manipulaciju dogadjajima u toku simulacije. Takve su, na primer, rezolucione funkcije, funkcije za kašnjenje, funkcije za dokumentovanje procesa simulacije i komunikaciju sa korisnikom.

Kod VHDL-a simulacija se zasniva na sinhronizovanom radu konkurentnih procesa. Tome odgovara objektno-orjentisan pristup modeliranju. Kao prirodniji i logičniji način gledanja na svet, ovaj pristup je zadnjih godina zadobio simpatije istraživača, tako da se javilo mnoštvo ideja i implementacija [Sand88, Moon90, Kazm91]. Kao što je to slučaj kod objektno-orjentisanih programskih jezika, i objektno-orjentisani simulatori imaju mogućnost iskorišćenja "minulog rada". Sa svakom novom bibliotekom modela koja se razvije, upotrebna vrednost simulatora raste.

Mora se napomenuti da uz VHDL simulatore uvek idu unapred razvijene biblioteke za logičku simulaciju. Simulaciona biblioteka zasniva se na jednom izabranom sistemu logičkih stanja i sadrži modele osnovnih logičkih elemenata, potrebne funkcije za modeliranje i slično. Posebni timovi stručnjaka za VHDL neprestano se bave razvojem novih biblioteka kako bi njima pokrili najnovije potrebe proizvodjača integrisanih kola. Da bi se sprečila nekompatibilnost ovih biblioteka, poseglo se za standardizovanjem nekih pravila na ovom polju [Coel88, Tani94].

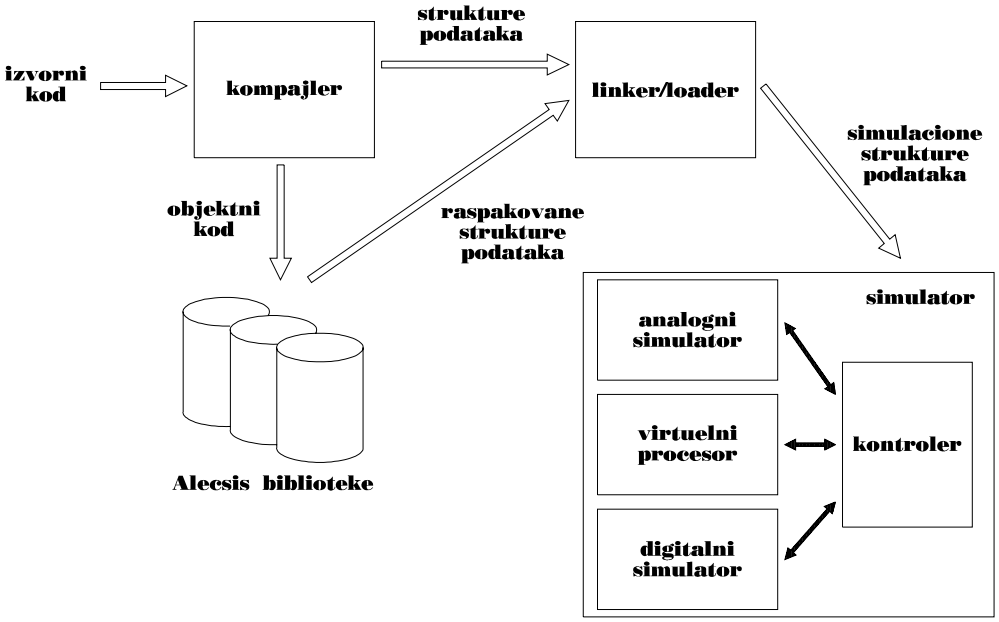
Neki autori [Kang94] izveštavaju o razvoju CASE (Computer-Aided Software Engineering) alata za automatsko projektovanje jednostavnijih logičkih simulatora (sa tri stanja, sa relativno jednostavnim modelima kašnjenja i detekcijom hazarda). Budućnost će pokazati da li je moguće pomoću CASE alata razvijati složenije logičke simulatore.

2.3 Hibridni simulator Alecsis2.1

Alecsis2.1 je integrisani hibridni simulator [Gloz94b]. Realizovan je kao minimalni skup simulacionih mehanizama koji čine osnovnu konstrukciju simulatora, sa moćnim ulaznim jezikom koji omogućava dogradnju simulacionog okruženja za različite primene i do proizvoljne složenosti. Podržava opisivanje simuliranog sistema na veoma različitim nivoima apstrakcije, kao i mešanje različitih nivoa apstrakcije u istom opisu, a složenost hijerarhijske strukture simuliranog kola ograničena je samo memorijskim mogućnostima hardverske platforme na kojoj je simulator instaliran.

Pri razvoju ulaznog jezika pošlo se od ideje da bi trebalo iskoristiti neki široko poznat jezik, kako bi se olakšalo učenje korisnicima, a na njega dodati što manji broj specifičnih konstrukcija za opisivanje strukture i funkcionisanja elektronskih kola. AleC++ - ulazni jezik simulatora Alecsis2.1, nadgradnja je programskog jezika C++. To je objektno-orjentisan jezik za opis hibridnih sistema. Kako je programskim jezikom C++ moguće opisati i rešiti mnoštvo raznorodnih

programerskih zadataka, tako i AleC++ nije ograničen samo na opisivanje elektronskih kola [Maks94b]. Pri razvoju simulatora vodilo se računa da strukture podataka na kojima se zasniva simulacija budu što univerzalnije. Simulator Alecsis2.1 primenjen je za simulaciju fuzzy sistema [Gloz93b], neuronskih mreža [Gloz93a], računarskih mreža [Maks95], kao i za simulaciju čisto digitalnih kola, dakle kao logički simulator [Maks93a, Maks94a, Toši95].



Slika 2.2 *Arhitektura simulatora Alecsis2.1*

Alecsis2.1 se sastoji od prevodioca, linkera i simulatora, kao što to pokazuje slika 2.2. Prevodilac generiše bibliotečke (objektne) fajlove iz kojih linker čita i povezuje objekte da bi se kompletirala struktura kola pre poziva simulatora. Simulator objedinjuje u jedinstvenu celinu algoritam analogne simulacije i algoritam digitalne simulacije. Sama simulacija obavlja se svodjenjem programskog koda do nivoa mašinskih instrukcija za virtuelni procesor (interpreter) programski realizovan po ugledu na Motorolin MC68020 mikroprocesor.

Težnja za univerzalnošću dovela je do razvoja mehanizama za spoljašnje modeliranje osnovnih digitalnih i analognih komponenti. Tako u simulator nije ugradjen ni jedna digitalna komponenta, dok je od analognih komponenti samo novi model idealnog prekidača [Mrča93] morao biti ugradjen, jer zbog svoje specifične prirode zadire u mehanizam kontrole vremenskog koraka. Ugradjene su, medjutim, i osnovne elektronske komponente (mada se one mogu modelirati i spolja) radi postizanja veće brzine rada simulatora.

Da bi se omogućilo efikasno modeliranje u analognom domenu, u ulazni jezik je ugradjen mehanizam za automatsko formulisanje doprinosa nelinearnih komponenti sistemu jednačina koji koristi metodu sečice umesto Newton-Raphson-ove metode linearizacije. Naime, za razliku od Newton-Raphson-ove metode, metod sečice ne zahteva poznavanje izvoda zadate nelinearne funkcije. Naravno, red konvergencije metoda sečice je nešto manji (1.62) od reda konvergencije Newton-Raphson-ovog metoda (2), ali se dobija na univerzalnoj primenjivosti. Osim ovoga, AleC++ dozvoljava i direktnu intervenciju u strukturi matrice sistema naredbom `eqn`. U zadavanju jednačina kojima se opisuju elektronske komponente mogu se koristiti operatori diferenciranja `ddt` i integracije `idt`. Razradjeni su mehanizmi za automatsku vremensku diskretizaciju na osnovu jednokoračnog Euler-ovog pravila i dvokoračnog Gear2 pravila [Mrča92].

Primenjivost simulatora je umnogome uvećana time što je obezbedjena delimična kompatibilnost sa SPICE sintaksom. Naime, moguće je direktno u opis kola za simulaciju simulatorom Alecsis2.1 priključiti delove SPICE biblioteka za poluprovodničke komponente: NMOS (PMOS), NPN (PNP), NJF (PJF) i D. Koncept modelskih kartica proširen je i na digitalne komponente korišćenjem `class` konstrukcije iz programskog jezika C++.

Za digitalnu simulaciju Alecsis2.1 (poput VHDL simulatora) zahteva razvoj digitalnih biblioteka. Razvoj biblioteke predstavlja zapravo kreiranje radnog okruženja, što obuhvata izbor sistema stanja, predefinisanje logičkih operatora za rad u tom sistemu, definisanje funkcija za kašnjenje i funkcija za rezoluciju, definisanje atributa signala, modeliranje osnovnih logičkih primitiva, počev od najjednostavnijih (invertor, NI kolo, ...), pa do proizvoljno složenih (ALU, PLA, RAM, ...). Moguće je razviti biblioteke pomoću kojih se simulatorom Alecsis2.1 "imitiraju" najpoznatiji logički i hibridni simulatori.

Od nestandardnih mogućnosti za modeliranje treba pomenuti parametrizovanje strukture kola naredbom `clone`. Kombinovanjem kloniranja i `for` petlje, Alecsis2.1 može da modelira ponavljanje odredjene komponente u strukturi simuliranog kola zadati broj puta, što je izuzetno korisna osobina kod modeliranja lančanih struktura kakvi su recimo registri, lestvičaste mreže i slično. Pri simulaciji takvog modela moguće je zadati dužinu strukture, čime se olakšava korisniku eksperinmentisanje sa strukturnim promenama.

Mogućnosti simulatora Alecsis2.1 istražene su samo delimično, budući da je to široko otvoren sistem koji se dogradjuje spoljašnjim intervencijama. Neke od osnovnih konstukcija jezika AleC++ i osobina simulatora Alecsis2.1 koje će biti korišćene u poglavljima o bibliotekama za digitalnu simulaciju opisaćemo u daljem tekstu.

2.3.1 Signali

Veza je pojam koji u jeziku AleC++ označava medijum za medjusobno sprezanje strukturnih elemenata simuliranog sistema. Veze u hibridnom sistemu mogu imati analogni aspekt (promenljive u sistemu jednačina kojim se opisuje analogni deo simuliranog kola), digitalni aspekt (veze u digitalnom delu kola) ili hibridni aspekt (veze izmedju analognog i digitalnog dela hibridnog kola). Veze sa hibridnim aspektom se automatski razlažu na dve veze: jednu sa analognim, a drugu sa digitalnim aspektom, pri čemu se ubacuje A/D ili D/A konvertor. AleC++ razlikuje pet vrsta veza: `node`, `current`, `charge`, `flow` i `signal`. Prve četiri vrste veza imaju analogni aspekt i koriste se pri analognoj i hibridnoj simulaciji. Veza tipa `signal` ima digitalni aspekt i biće predmet našeg interesovanja, s obzirom da ćemo se baviti razvojem biblioteka za digitalnu simulaciju simulatorom Alecsis2.1.

Simulator Alecsis2.1 modelira digitalne sisteme pomoću paralelnih procesa. Signal predstavlja komunikacioni kanal preko koga procesi razmenjuju informacije i na taj način sinhronizuju rad. U signal se može zapisivati informacija, sa njega se može čitati informacija ili i zapisivati i čitati. Zato signali imaju usmerenje `in`, `out` ili `inout`. Signali mogu biti proizvoljnog tipa: obično se koriste signali enumerisanog tipa, ali je moguće koristiti i bilo koji drugi tip (`int`, `double`, ...), kao i strukture i klase. Pre korišćenja signala potrebno je deklarisati njegov tip, a kod signala u interfejsu modula potrebno je navesti i usmerenje (difoltno usmerenje signala je `in`). Pri deklaraciji signala alocira se memorija za čuvanje njegove vrednosti i memorija za smeštaj vrednosti atributa signala. Vrednost signala u tekućem trenutku simulacionog vremena dobija se pozivom na njegovo ime, dok se atributima pristupa operatorom `@`. U jednoj digitalnoj biblioteci svi signali su obično istog tipa - tipa kojim je definisan korišćeni sistem logičkih stanja. Signale je moguće koristiti u logičkim izrazima, ako se osnovni logički operatori predefinišu (preopterete) za rad u korišćenom sistemu stanja.

Osim signala koje je korisnik deklarisao kao veze u simuliranom kolu, postoje i implicitni signali koje generiše sam simulator u odredjenim fazama rada. Implicitni signali su:

- `structural` (aktivira se u procesu konstruisanja hijerarhijskog stabla opisa kola, može se iskoristiti da se interveniše u strukturi kola)
- `post_structural` (aktivira se nakon što je tekući hijererhijski nivo formiran, može se iskoristiti za modifikaciju vrednosti atributa signala)
- `initial` (aktivira se na početku simulacije, u trenutku t=0 simulacionog vremena, a zatim suspenduje do kraja simulacije)

- `per_moment` (aktivira se u svakom novom trenutku simulacije pre formiranja matrice sistema, koristi se u analognoj simulaciji za intervenciju u matrici sistema)
- `post_moment` (aktivira se u svakom novom trenutku simulacije posle rešavanja sistema jednačina; ako je kolo čisto digitalno, nema rešavanja sistema jednačina, pa se ovaj i prethodni signal aktiviraju istovremeno)
- `per_iteration` (kod simulacije kola u kojima ima nelinearnih analognih elemenata, ovaj signal se aktivira u svakoj iteraciji, pre formiranja sistema jednačina)
- `final` (aktivira se po završetku simulacije, može se iskoristiti za obavljanje završnih aktivnosti simulacije: svodjenje statistike, štampanje odgovarajućih informacija, zatvaranje otvorenih datoteka i slično)

Signali koji predstavljaju veze u digitalnom kolu navode se u listi osetljivosti procesa ili u `wait` naredbi unutar procesa. Kad se proces sinhronizuje implicitnim signalom, koristi se posebna sintaksa. Razmotrimo nekoliko karakterističnih primera.

```
proc_1: process (a, b, c) {   // using sensitivity list
  ...
}
proc_2: process {
  ...
  wait a while a!='0';        // wait statement detecting falling edge on a
  ...
}
proc_3: process initial {     // process synchronized by implicit signal
  ...
}
```

Proces označen labelom `proc_1` ima listu osetljivosti u kojoj su signali `a`, `b` i `c`. Proces `proc_2` sadrži `wait` naredbu koja izaziva suspendovanje procesa sve do pojave logičke nule na signalu `a`. Procesi koji sadrže `wait` naredbe ne smeju imati drugi način sinhronizacije (listu osetljivosti ili implicitni signal). Kad se proces sinhnonizuje implicitnim signalom, taj signal se navodi bez zagrada, kao kod procesa `proc_3`.

2.3.2 Sistemi stanja

Pod sistemom stanja simulatora podrazumeva se skup simbola kojima se opisuju dozvoljene vrednosti signala. Ukupan broj ovih stanja naziva se kardinalnim brojem simulatora. Da bi se projektovala digitalna biblioteka prvo je potrebno izabrati sistem stanja. Pri izboru sistema stanja projektant biblioteke se rukovodi zahtevima budućih korisnika biblioteke. Odredjenom sistemu logičkih stanja odgovara odredjena tačnost simulacije. Veća tačnost zahteva složeniji sistem stanja. Mnogi simulatori imaju ugradjen sistem stanja. Alecsis2.1 takav sistem nema i on se mora definisati pre simulacije.

Sistem stanja se obično definiše korišćenjem enumerisanog tipa podataka. Da bi se omogućilo da se jednostavno predstavljaju i digitalne reči (vektori stanja), pogodno je (ali to nije pravilo) da sistem stanja bude definisan sa karakterima kao simbolima. Najšire korišćen sistem stanja kod postojećih logičkih simulatora je sistem sa četiri logička stanja koja se obično označavaju skupom simbola: `'X'`, `'0'`, `'1'` i `'Z'`. Ovaj sistem stanja u jeziku AleC++ definiše se sledećim kodom.

```
typedef enum { 'X', 'x'='X', '0', '1', 'Z', 'z'='Z', '_'=void } four_t;
```

Signali tipa `four_t` mogu uzeti vrednosti `'X'`, `'0'`, `'1'` i `'Z'`. Kao i kod drugih programskih jezika, pri definiciji enumerisanog tipa podataka pojedinim simbolima se dodeljuju celobrojni indeksi počev od vrednosti nula. Tako stanju `'X'` odgovara indeks 0, stanju `'0'` odgovara indeks 1, stanju `'1'` indeks 2 i stanju `'Z'` indeks 3. Konstrukcija `'x'='X'` kaže

prevodiocu da je stanje `'x'` ekvivalentno stanju `'X'`, odnosno i njemu odgovara indeks 0. Slično je i sa stanjem `'z'`. Ova stanja su definisana da bi se korisniku omogućilo ravnopravno korišćenje malih i velikih slova. Konstrukcijom `'_'=void` definisan je separator `'_'`. To nije logičko stanje i njemu ne odgovara indeks. Separator je namenjen poboljšanju preglednosti pri definisanju vektora stanja. Na primer, vrednost šesnaestobitnog vektora `"0001000011010101"` mnogo se preglednije zapisuje u obliku `"0001_0000_1101_0101"`. Na sledećim primerima objasnićemo postupak deklarisanja signala različitih tipova.

```
signal four_t a, b='0', vec1[8]="0000_1111", vec2[8]="0010_00";
four_t c, d, e;
signal int i1, i2=95;
signal double d1=0.2, d2;
```

Kao što se vidi, pri deklaraciji se može, ali ne mora, zadati inicijalna vrednost signala (vrednost signala u nultom trenutku simulacije). Ako nije zadata, simulator bira vrednost 0 (pri svakoj alokaciji memorije AleC++ anulira sadržaj memorije), što je u slučaju tipa four_t odgovara stanju `'X'`, jer ono ima indeks 0. Dakle, signal `a` ima inicijalnu vrednost `'X'`. Signalu `b` dodeljena je početna vrednost `'0'`. Kod vektorskog signala `vec1` definisana su početna stanja svih elemenata korišćenjem string notacije, dok su vektoru `vec2` definisana stanja za šest signala, a signali `vec2[6]` i `vec2[7]` imaju vrednost `'X'`. Ključna reč `signal` može se izostaviti ako su signali enumerisanog tipa, jer simulator podrazumeva da je u pitanju deklaracija signala. Medjutim, to se ne sme učiniti kod signala drugačijeg tipa. Signalu `i1` odgovara inicijalna vrednost `0`, a signalu `d2` inicijalna vrednost `0.0`.

2.3.3 Preopterećenje operatora

Da bi se olakšalo funkcionalno modeliranje pogodno je predefinisati osnovne logičke operatore za rad u izabranom sistemu stanja. Osnovni logički operatori u jeziku AleC++ su: `~` (binarna negacija), `&` (binarno I), `|` (binarno ILI), `^` (binarno ekskluzivno ILI), `~&` (NI), `~|` (NILI), `~^` (ekvivalencija), `&=` (binarno I sa dodeljivanjem vrednosti rezultata levom operandu), `|=` (ILI sa dodelom vrednosti) i `^=` (ekskluzivno ILI sa dodelom vrednosti). Svi se ovi operatori mogu predefinisati uvodjenjem odgovarajućih funkcija koje su obično vrlo jednostavne i svode se na čitanje tabela istinitosti logičkih opetatora. U funkcije je moguće preneti operande po vrednosti ili po referenci (za poslednja tri operatora). Na primer, moguće je predefinisati operator `&=` za rad u sistemu stanja `four_t` na sledeći način.

```
const four_t and_tab[4][4] = {
// X    0    1    Z
  'X', '0', 'X', 'X',   // X
  '0', '0', '0', '0',   // 0
  'X', '0', '1', 'X',   // 1
  'X', '0', 'X', 'X'    // Z
}
four_t operator &= (four_t & left, four_t right)
                              {  return (left = and_tab[left][right]);  }
```

Prvo smo definisali tabelu `and_tab` koja predstavlja tablicu istinitosti za logičku I operaciju u sistemu sa četiri stanja `four_t`. Funkcije za preopterećenje operatora označavaju se ključnom rečju `operator`. Naša funkcija je vrlo jednostavna. Ona samo čita rezultat logičke operacije `&` nad levim i desnim operatorom i upisuje ga na adresu levog operanda. Za tu svrhu levi operand je prenet u funkciju po adresi (operator `&` u ovom kontekstu označava prenos adrese). Pozivom na ime signala (recimo `left` ili `right`) dobija se vrednost signala - stanje u sistemu `four_t`. Ovom stanju odgovara celobrojni indeks, pa je moguće direktno imenom signala indeksirati tabelu `and_tab` i pročitati njen sadržaj.

2.3.4 Atributi signala

Bez obzira na tip, signali imaju implicitne atribute, a mogu imati i korisničke atribute. Atributi sadrže informacije o signalu u odredjenom trenutku simulacionog vremena. Predefinisani, implicitni atributi su sledeći:

- `active` (ima vrednost 1 ako je signal aktivan u tekućem trenutku, inače ima vrednost 0)
- `event` (ima vrednost 1 ako se na signalu dešava dogadjaj u tekućem trenutku, inače ima vrednost 0)
- `quiet` (ima vrednost 1 ako se signal ne obradjuje u tekućem trenutku, inače ima vrednost 0 - !active)
- `stable` (ima vrednost 1 ako signal ne menja vrednost u tekućem trenutku, inače ima vrednost 0 - !event)
- `fanin` (broj procesa osetljivih na signal)
- `fanout` (broj drajvera koje ima signal)
- `hybrid` (ima vrednost 1 ako signal ima i analogni aspekt, inače ima vrednost 0)

Pristup implicitnim atributima signala ostvaruje se operatorom `->`. Recimo, izraz `a->event` predstavlja vrednost atributa `event` signala `a` u tekućem trenutku simulacionog vremena. Ako je ova vrednost `1`, na signalu `a` se u tekućem trenutku dešava dogadjaj. Ova informacija se može iskoristiti pri modeliranju komponenti digitalnih kola. Na primer, ako se kašnjenje od `s` i `r` ulaza RS flip-flopa do `q` izlaza razlikuje, potrebno je detektovati na kome ulazu se dogadja dogadjaj koji je izazvao promenu stanja izlaza `q` (atribut `event` tog ulaza ima vrednost 1), kako bi se dogadjaj poslao na izlaz posle odgovarajućeg kašnjenja.

Korisnički atributi definišu se po potrebi i dodaju listi implicitnih atributa. Mogu biti proizvoljnog tipa, a uz signal se pridružuju operatorom `@`. Deklaracija

```
signal four_t @ double x1, x2;
```

kreira signale `x1` i `x2` koji osim sedam implicitnih atributa imaju i osmi korisnički, tipa `double`. Ovom atributu može se pristupiti takodje operatorom `@`. Izraz `@x1` predstavlja pokazivač na memorijsku lokaciju u kojoj se čuva vrednost osmog atributa signala `x1`, a `*@x1` je vrednost ovog atributa. Korisnički atributi mogu se efikasno iskoristiti za memorisanje informacije o osobinama veza u digitalnim kolima (recimo o parazitnoj kapacitivnosti veze), kao što ćemo videti u poglavljima o bibliotekama.

2.3.5 Modul - osnovna strukturna jedinica

Modul (`module`) je osnovni element hijerarhije u opisu hardvera za simulaciju simulatorom Alecsis2.1. Bez obzira na to da li sadrži čisto analogne elemente, čisto digitalne elemente ili i jedne i druge, modul se definiše na isti način. U C-u izvršenje programa počinje i završava se u funkciji `main`. Kod simulatora Alecsis2.1 hijerarhija opisa simuliranog kola počinje i završava se u modulu sa oznakom `root`. Tako `root` modul predstavlja celo simulirano kolo. U njemu se zadaju komande za kontrolu simulacije: vremenski korak i drugi parametri analogne simulacije, vreme trajanja simulacije, izgled izlazne liste i pobuda kola. `Root` modul sadrži proizvoljan broj modula koji čine prvi nivo hijerarhije opisa kola, a svaki modul sadrži takodje proizvoljan broj modula sledećeg nivoa hijerarhije. Broj nivoa hijerarhije u opisu kola nije ograničen.

Modul se sastoji od interfejsa i tela. U interfejsu modula su navedeni ulazno/izlazni terminali modula, njihov tip i usmerenje. Telo modula sastoji se od deklarativnog dela, strukturnog dela i funkcionalnog dela. Deklarativni deo sadrži deklaracije signala i modula koji će biti korišćeni. Strukturni deo omogućava strukturno modeliranje: opis topologije modula navodjenjem sastavnih elemenata i veza izmedju njih. Sastavni elementi su drugi moduli i/ili ugradjene analogne

komponente. Signali deklarisani u deklarativnom delu predstavljaju veze. Funkcionalni deo (takozvani akcioni blok naredbi) omogućava funkcionalno modeliranje i sadrži jedan ili više paralelnih procesa. Procesi se izvršavaju kvaziparalelno i mogu biti aktivni ili suspendovani. Rad procesa sinhronizuje se pomoću signala.

Operator `<-` služi za odloženo dodeljivanje signala. Proces u kome se javi opetator `<-` kreira drajver za signal koji je sa leve strane ovog operatora. Promena stanja signala naziva se dogadjaj. Drajver signala predstavlja lokalnu listu budućih dogadjaja za taj signal. Dogadjaj se opisuje trenutkom kada treba da se dogodi i stanjem koje treba da se uspostavi na signalu. Novi par vreme-stanje može da se pošalje u drajver signala sa inercijalnim ili sa transportnim kašnjenjem. U slučaju inercijalnog kašnjenja, svi parovi na listi čije je vreme manje od vremena novog para biće obrisani. Izuzetak čine oni parovi čija je vrednost stanja ista kao i kod novog para. U slučaju transportnog kašnjenja, novi par vrednost-vreme biće smešten na listu tako što će se svi parovi čije je vreme veće od njegovog obrisati. Inercijalno kašnjenje je karakteristika digitalnih komponenti - pojava na ulazu mora da ima odredjeno minimalno trajanje da bi izazvala promene na izlazu. Transportno kašnjenje je karakteristično za vodove - svaka pojava sa ulaza prenosi se na izlaz bez obzira na trajanje. Transportno kašnjenje zahteva se navodjenjem ključne reči `transport` iza operatora `<-`, inače se podrazumeva inercijalno kašnjenje. Način obrade ova dva tipa kašnjenja u simulatoru isti je kao kod VHDL-a [Lips89, -90, Gloz94a]. Sledeći kod predstavlja model voda i ilustruje upotrebu transportnog kašnjenja.

```
module delay_line (four_t in a; four_t out y) {
  action (double tranport_delay) {
    process (a) { y <- transport a after transport_delay; }
  }
}
```

Pomenimo ovde i operator `now` koji se često koristi u procesima, a vraća trenutnu vrednost simulacionog vremena.

```
if (now>=30ns) exit(1);        // stop simulation after 30ns
```

a) b)

Slika 2.3 *a) Dvoulazno I kolo b) Jednookidni impulsni generator*

Proces osetljiv na neki signal aktivira se pri svakom dogadjaju na tom signalu. Procesi se mogu sinhronisati na dva načina: korišćenjem liste osetljivosti ili pomoću naredbe `wait`. Kao primer sinhronizacije primenom liste osetljivosti neka posluži model dvoulaznog I kola (slika 2.3a) sa propagacionim kašnjenjem 1ns.

```
module and2 (signal four_t in a, b; signal four_t out y) {
  action {
    process (a, b) { y <- a & b after 1ns; }
  }
}
```

Modul `and2` ima ulazne terminale `a` i `b` i izlazni terminal `y`. Signali koji se mogu spregnuti za ove terminale moraju biti tipa `four_t`. Modul nema deklarativni i strukturni deo. U akcionom bloku definisan je samo jedan proces. Ovaj proces ima u listi osetljivosti signale `a` i `b`. Pri dogadjaju na nekom od ovih signala proces u drajver signala `y` šalje rezultat `&` operacije nad ulazima sa inercijalnim kašnjenjem 1ns.

Umesto liste osetljivosti može se koristiti naredba `wait`. Sledeći kod predstavlja kolo sa slike 2.3b koje generiše impuls trajanja 50ns na izlazu `y` pri pojavi logičke jedinice na `trigger` ulazu, ali samo jednom (one-shot kolo). Kašnjenje kola do početka generisanja impulsa je 1ns. Impuls može biti pozitivan ili negativan u zavisnosti od stanja izlaza u trenutku okidanja.

```
module one_shot (signal four_t in trigger; signal four_t out y) {
  action {
    process {
      wait trigger while trigger!='1'; // wait for rising edge on trigger
      if (y=='0') y <- '1' after 1ns, '0' after 51ns;
      else        y <- '0' after 1ns, '1' after 51ns;
      wait; // suspend process to the end of simulation
    }
  }
}
```

Prva `wait` naredba suspenduje proces sve do pojave logičke jedinice na ulazu `trigger`. Kad se takav dogadjaj desi, proces generiše impuls na izlazu `y`, a zatim se suspenduje do kraja simulacije s obzirom da u drugoj naredbi `wait` nije naveden signal na čiju se promenu čeka.

2.3.6 Rezolucione funkcije

Kad signal ima više od jednog drajvera, rezolucijom njihovih vrednosti odredjuje se rezultujuće stanje signala. Kod simulatora Alecsis2.1 definisanje rezolucione funkcije prepušteno je korisniku. Svaka biblioteka za digitalnu simulaciju treba da sadrži rezolucionu funkciju za vezu žičano I i za vezu žičano ILI, rezolucionu funkciju za magistralu (kratkospojeni izlazi trostatičkih kola) bez i sa pullup ili pulldown otpornikom.

Funkcija za rezoluciju mora da zadovolji odredjene uslove u pogledu rezultata i parametara. Ako je signal tipa `four_t`, njegova rezoluciona funkcija mora vraćati isti tip. Za signale koji nisu tipa strukture ili klase rezoluciona funkcija ima dva parametra. Prvi parametar je lista vrednosti drajvera signala, a drugi je pokazivač na celobrojnu promenljivu preko koje rezoluciona funkcija signalizira konflikt. Kod signala tipa strukture ili klase, rezoluciona funkcija ima tri parametra. Prvi je lista vrednosti drajvera, drugi je broj drajvera signala, a treći pokazivač na celobrojnu promenljivu preko koje funkcija javlja o pojavi konflikta. Navešćemo dva primera deklaracije rezolucione funkcije.

```
four_t resol_function (four_t *drivers, int *report);
complex_t resol_complex (complex_t *drivers, int n_drivers, int *report);
```

Rezoluciona funkcija `resol_function` služi za rezoluciju signala tipa `four_t`, a rezoluciona funkcija `resol_complex` služi za rezoluciju signala tipa `complex_t`. Tip `complex_t` je struktura ili klasa, pa funkcija ima i parametar `n_drivers` koji predstavlja ukupan broj drajvera signala.

U okviru rezolucione funkcije može se promeniti vrednost na koju ukazuje pokazivač `report`. Ako se ova vrednost postavi na 1, simulator će štampati na ekranu poruku da je došlo do konflikta, vremenski trenutak simulacije i ime signala. Ako se ova vrednost postavi na 2, simulator će štampati poruku da je došlo do potencijalnog konflikta (ovo je potrebno za slučaj neodredjenog stanja na izlazu drajvera). Ako nema konflikta, nije potrebno intervenisati na vrednosti na koju ukazuje pokazivač `report`, s obzirom da ona pri pozivu funkcije ima vrednost 0, što je znak simulatoru da je sve u redu.

Kako je redosled drajvera u vektoru `drivers` slučajan (zavisi od redosleda procesa pri podizanju hijererhijskog stabla opisa kola), rezultat funkcije rezolucije ne sme da zavisi od redosleda navodjenja drajvera.

2.3.7 Akcioni parametri

Da bi definisani modul imao opšti karakter i bio prilagodljiv potrebama korisnika, neophodno je parametrizovati njegove osobine. Na primer, definisano dvoulazno I kolo bi trebalo da ima dodeljivu vrednost kašnjenja umesti konstantne vrednosti 1ns. U suprotnom bi korisnik morao iznova da definiše novi modul za svako dvoulazno I kolo čije je kašnjenje različito od 1ns.

Kod simulatora Alecsis2.1 postoje dva mehanizma za prenos parametara modela do procesa koji modeliraju funkciju modula. Prvi mehanizam su akcioni parametri. Akcioni blok može imati listu parametara kao u sledećem primeru.

```
module and2 (signal four_t in a, b; signal four_t out y) {
  action (double tplh, double tphl) {
    process (a, b) {
      y <- a&b after (a&b=='0' ? tphl : a&b=='1' ? tplh : MAX(tplh, tphl));
    }
  }
}
```

Sada je modulu `and2` moguće pri pozivu zadati kašnjenja prednje i zadnje ivice signala na izlazu. U slučaju da je izračunata vrednost izlaza `'0'`, kašnjenje komponente je `tphl`, ako je `'1'`, kašnjenje je `tplh`, a ako je `'X'`, uzima se maksimalno od dva moguća kašnjenja (najgori slučaj). Sledeći primer pokazuje dva moguća načina navodjenja akcionih parametara pri definisanju konkretnog dvoulaznog I kola u strukturnom delu tela modula.

```
module and2 and2gate1, and2gate2;   // modules declarations
signal four_t input1, input2, output1, input3, input4, output2;   // signals
and2gate1 (input1, input2, output1) action (10ns, 9ns);     // instantiation
and2gate2 (input3, input4, output2) { tphl=9ns; tplh=10ns; }
```

Kod navodjenja akcionih parametara u listi iza ključne reči `action`, kao kod gejta `and2gate1`, redosled parametara mora biti isti kao u definiciji modula. Dakle, kod gejta `and2gate1` zadata je vrednost `10ns` za parametar `tplh`, a vrednost `9ns` za parametar `tphl`. Akcioni parametri mogu se definisati i direktnom dodelom vrednosti, kao kod gejta `and2gate2`. Pri tome se navode imena parametara, tako da je redosled naredbi dodeljivanja nebitan. Gejtovi `and2gate2` i `and2gate1` imaju iste parametre kašnjenja.

2.3.8 Modelske kartice

Parametrizovanje modela gejta korišćenjem akcionih parametara postaje nepraktično kad model ima veliki broj parametara. Zato je simulator Alecsis2.1 opremljen još jednim mehanizmom parametrizacije modula - to su modelske kartice. Ovaj mehanizam je razvijen po ugledu na SPICE modelske kartice i predstavlja njihovo uopštenje za modeliranje hibridnih komponenti. Modelske kartice definišu se korišćenjem mehanizma modelskih klasa (konstukcija `class`). Modelska klasa obezbedjuje grupisanje i zaštitu podataka modela i predstavlja osnovu objektno-orjentisanog pristupa modeliranju elemenata sistema. Definicija jedne modelske klase ima sledeći izgled.

```
class gates {
  private:
      double incap, outcap;
      double tplh, tphl;
  public:
      gates();
      >gates();
      ~gates();
      double delay(four_t, four_t, double, int);
      friend module buf, inv, and, or, xor, nand, nor, nxor;
```

```
};
```

Klasa `gates` sadrži podatke i metode. Podaci su `incap`, `outcap`, `tplh` i `tphl` i to su parametri modela gejtova koji su navedeni u `friend` deklaraciji kao "prijatelji" klase `gates`. Prijatelji klase imaju direktan pristup privatnim podacima klase. Svi ostali objekti klase mogu pristupiti podacima klase samo preko metoda klase. Standardni metodi klase su konstruktor, procesor i destruktor klase. Konstruktor klase je istoimena funkcija koja po potrebi alocira prostor za podatke klase i zadaje im inicijalne vrednosti (funkcija `gates` u gornjem primeru). Procesor klase je funkcija čije se ime formira dodavanjem prefiksa `>` imenu klase (`>gates` u gornjem primeru) i to je novina koju AleC++ donosi u odnosu na C++. Procesor služi za obradu podataka klase posle formiranja modela: proveru logičnosti zadatih vrednosti parametara modela, izračunavanje vrednosti simulacionih konstanti i slične aktivnosti. Destruktor klase je funkcija čije se ime formira dodavanjem znaka `~` imenu klase (funkcija `~gates` u gornjem primeru). Destruktor služi za oslobadjanje alociranog memorijskog prostora za podatke klase pri izlasku iz oblasti vidljivosti u kojoj je objekat klase definisan. Osim ovih metoda, klasa sadrži i druge metode za pristup i obradu podataka. Recimo, kod klase `gates` funkcija `delay` namenjena je izračunavanju kašnjenja. Prijateljski moduli `buf`, `inv`, `and` i ostali deklarišu se kao objekti klase `gates`.

```
module gates :: inv (four_t in a; four_t out y) {
   ...
}
```

Ovako deklarisanom modulu `inv` odgovara modelska kartica tipa `gates` u kojoj su navedene vrednosti parametara modela. Modelska kartica se definiše korišćenjem konstrukcije `model`.

```
model gates :: inverter_model_1 {
  incap = 1pF;  outcap = 1.5pF;  tplh = 12ns;  tphl = 10ns;
}
```

Definisana modelska kartica sa imenom `inverter_model_1` predstavlja statički objekat koji se pridružuje uz modul tipa `inv` pri njegovom korišćenju na sledeći način.

```
module gates :: inv g1; // module declaration
signal four_t x1, x2; // signals declaration
g1 (x1, x2) model = inverter_model_1; // instantiation
```

Važno je napomenuti da jedna ista modelska kartica može da se pridruži uz više različitih modula koji imaju iste vrednosti modelskih parametara. Recimo, ako u simuliranom kolu imamo 15-tak invertora sa istim kašnjenjima, onda im se svima može pridružiti ista modelska kartica.

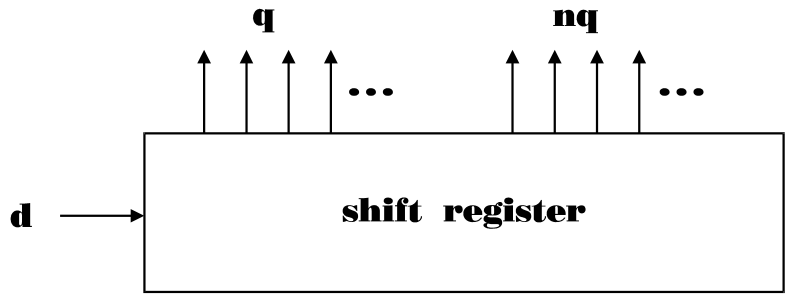
2.3.9 Parametrizacija strukture kola - kloniranje

Naredba `clone` služi za direktnu intervenciju na strukturi kola iz procesa sinhronizovanih implicitnim signalom `structural`. Ovim se izbegava korišćenje strukturnog regiona u telu modula. Umesto da se komponenta definiše u okviru opisa topologije kola, ona se u akcionom bloku ugradjuje u strukturu kola.

```
module inverter g1;
signal four_t s1, s2;
action {
  process structural {
    clone g1 (s1, s2) { tplh=10ns; tphl=8ns; }
  }
}
```

Ali postoje i sofisticiranije primene naredbe `clone`. S obzirom da su u okviru akcionog bloka poznate vrednosti akcionih parametara, moguće je naredbu `clone` iskoristiti za parametrizovanje strukture kola. Razmotrimo sledeći primer.

```
module shift_register (four_t in d; four_t out q[], nq[]) {
  module dtypeff ff;
  action (double tplh, double tphl) {
    process structural {
      int i;
      for (i=0; i<lengthof q; i++) {
        if (i==0) clone ff[i] (d, q[i], nq[i]) action (tplh, tphl);
        else      clone ff[i] (q[i-1], q[i], nq[i]) action (tplh, tphl);
      }
    }
  }
}
```



Slika 2.4 *Asinhroni šift registar proizvoljne dužine*

Modul `shift_register` predstavlja asihnroni šift registar sačinjen od proizvoljnog broja D flip-flopova (slika 2.4). Registar ima serijski ulaz `d` i paralelne izlaze `q` i `nq`. Dužina registra (broj flip-flopova) jednaka je dužini vektorskih signala `q` i `nq`. Registar prima akcione parametre - kašnjenja flip-flopa `tplh` i `tphl`. U procesu sinhronizovanom signalom `structural` kombinovana je naredba `clone` sa `for` petljom da bi se flip-flopovi povezali u niz. Parametri kašnjenja su prosledjeni flip-flopovima pri njihovom povezivanju. Operator `lengthof` vraća dužinu vektora. Može se koristiti sa ili bez zagrada.

```
lengthof vec;     // this is equivalent to:
lengthof (vec);
```

Modul `dtypeff` definisan je na sledeći način.

```
module dtypeff (four_t in d; four_t out q, nq) {
  action (double tplh, double tphl) {
    process (d) {
      if (d=='x' || d=='z') {
        warning ("d unknown or high impedanse, outputs set to 'X'");
        q  <- 'x' after MAX(tplh, tphl);
        nq <- 'x' after MAX(tplh, tphl);
      }
      else {
        q  <-  d after (d=='0' ? tphl : tplh);
        nq <- ~d after (d=='1' ? tphl : tplh);
      }
    }
  }
}
```

Dužina šift registra definiše se pri njegovom pozivu u opisu kola za simulaciju. `Root` modul za testiranje modula `shift_register` za različite dužine može da ima sledeći izgled.

```
# define LEN 8
```

```
root module shift_register_test {
  module shift_register sh;
  signal four_t d='0', q[LEN], nq[LEN];

  sh (d, q, nq) action (10ns, 20ns);

  out { signal four_t d, q, nq; }
  timing { tstop = 500ns; }
  action {
    process initial { d <- '1' after 50ns, '0' after 200ns; }
  }
}
```

Dužina vektorskih signala `q` i `nq` (a time i registra `sh`) definisana je makroom `LEN`. Promenom njegove vrednosti mogu se simulirati registri različite dužine.

Treba, medjutim, imati na umu da mogućnosti naredbe `clone` nisu neograničene. Njome je moguće parametrizovati samo jednodimenzionalne strukture. Matrične ili višedimenzionalne strukture moraju imati fiksirane sve dimenzije osim jedne. Pored toga, nije moguće proslediti ime modelske kartice u proces sinhronizovan signalom `structural`. Zato moduli koji se kloniraju ne mogu primati modelske kartice, već treba koristiti akcione parametre kao u gornjem primeru.

2.3.10 Inicijalizacija

Pod inicijalizacijom logičkog kola podrazumeva se odredjivanje početnih stanja (stanja u trenutku $t=0^-$) svih signala u kolu. Početna stanja odredjuju se na osnovu zadatih stanja na globalnim ulazima kola, a kod sekvencijalnih elemenata početna memorisana stanja zadaje korisnik ako ih nije moguće odrediti (recimo kod sinhronih flip-flopova treba da se sačeka aktivna ivica taktnog signala). Problem inicijalizacije rešava se kod različitih logičkih simulatora na različite načine. Većina simulatora obezbedjuje mehanizam automatske inicijalizacije kola propagacijom stanja sa globalnih ulaza [Ulri74, Chap74, Bush83]. Pokazano je da simulator koji razlikuje neodredjeno stanje i njegov komplement ima bolju mogućnost automatske inicijalizacije [Stev83]. Automatska inicijalizacija se često kombinuje sa interaktivnom inicijalizacijom - intervencijom korisnika inicijalizuju se signali za koje nije moguća automatska inicijalizacija [Bush83]. U slučaju nemogućnosti inicijalizacije, neinicijalizovani signali se ostavljaju u neodredjenom stanju sve do pojave prvog dogadjaja. Ovakvo neodredjeno stanje treba razlikovati od neodredjenog stanja koje nastaje usled konfliktnih situacija u logičkom kolu. Tako se u literaturi sreću sistemi stanja koji razlikuju više vrsta neodredjenih stanja: neinicijalizovano (nepoznato), konfliktno (neodredjeno), nebitno ("don't care") i druga [Chap74, Okaz83]. Moguć je i pristup slučajne inicijalizacije signala za koje nije moguće izvršiti inicijalizaciju (recimo stanja flip-flopova), što odgovara realnoj situaciji u logičkom kolu pri uključenju napajanja [Giam79]. Takodje se kod nekih logičkih simulatora koristi inicijalna sekvenca pobudnih signala (na primer taktni signali omogućavaju inicijalizaciju flip-flopova) u odredjenom trajanju [Stev83] ili se koristi zapisano stabilno stanje kola iz neke ranije simulacije [Giam79]. Zanimljiv pristup inicijalizaciji korišćenjem intervalne algebre izložen je u [Ruan91].

Kod simulatora Alecsis2.1 problem inicijalizacije logičkog kola prepušten je korisniku. U težnji da se simulator učini univerzalnim, nije ugradjen ni jedan od specifičnih mehanizama inicijalizacije koji se sreću kod logičkih simulatora.

Kao što znamo, logička simulacija se u simulatoru Alecsis2.1 bazira na komunikaciji kvazi-paralelnih procesa. Procesi mogu biti aktivni ili suspendovani. Da bi simulacija započela, potrebno je pokrenuti (aktivirati) sve procese. Postupak aktivacije procesa predstavlja zapravo inicijalizaciju. Procesi sinhronizovani implicitnim signalom `structural` izvršavaju se samo jednom za vreme konstruisanja hijerarhijskog stabla opisa kola, a zatim se suspenduju do kraja simulacije. Procesi sinhronizovani implicitnim signalom `post_structural` obradjuju se na isti način. O inicijalizaciji ovih procesa vodi računa sam simulator. Procesi sinhronizovani implicitnim signalom `initial`

startuju se pre početka simulacije, u trenutku simulacionog vremena 0.0s, ali ih je teško iskoristiti za inicijalizaciju simuliranog kola, jer redosled njihove aktivacije nije moguće unapred predvideti. To je stoga što simulaciona mašina simulatora Alecsis2.1 emulira paralelizam rada procesa (zapravo bi trebalo sve procese aktivirati istovremeno, ali to nije moguće na jednoprocesorskoj hardverskoj platformi). Svi ostali procesi aktiviraju se automatski u trenutku 0.0s. Ako u procesu nema naredbe `wait`, to jest, ako je proces sinhronizovan signalima sa liste osetljivosti, celokupni programski kod sadržan u procesu biće jednom izvršen. Korisniku je ostavljena mogućnost da u okviru procesa predvidi postupak inicijalizacije. U okviru procesa obično se nalazi jedno ili više dodeljivanja vrednosti drajveru nekog izlaznog signala operatorom `<-`. Ovo dodeljivanje se obično obavlja sa kašnjenjem, tako da se usled aktivacije procesa pojavljuju dogadjaji u trenucima posle početka simulacionog vremena. Ovo se može sprečiti ako se eliminiše kašnjenje kod slanja dogadjaja na izlaz komponete pri inicijalizaciji. Recimo, sledećim kodom definisano je dvoulazno NILI kolo kod koga je posebno razmatrana inicijalizacija.

```
module nor_init (four_t out y; four_t in a, b) {
  action (double tp) {
    behaviour: process (a, b) {
      int initialized=0;

      if (!initialized) {
        y <- ~(a | b);        // zero delay during initialization
        initialized=1;
      }
      else  y <- ~(a | b) after tp;
    }
  }
}
```

Promenljiva `initialized` je pri deklaraciji postavljena na vrednost `0` što znači da proces `behaviour` nije prošao kroz fazu inicijalizacije. Pri prvom ulasku u proces koristi se nulto kašnjenje, a kasnije se koristi normalan model kašnjenja. Svi moduli koji se koriste u simulaciji morali bi biti realizovani na sličan način kao modul `nor_init`. Ovakvo rešenje inicijalizacije može pomoći u nekim situacijama, ali nije opšteg karaktera zbog nepredvidivog redosleda aktivacije procesa pri simulaciji. Naime, obično je potrebno više od jednog prolaza sa nultim kašnjenjem kroz proces da bi se uspostavilo validno početno stanje u simuliranom kolu. U tom slučaju se uslov `(!initialized)` može zameniti uslovom `(now==0.0)`. Tako će komponenta raditi sa nultim kašnjenjem sve dok se ne potroše svi dogadjaji nastali kao rezultat inicijalizacije (odnosno dok se kolo ne stabilizuje). Treba samo obezbediti stabilnost kola u slučaju da kolo sadrži povratne sprege, jer može doći do oscilovanja. Osim toga, prvi dogadjaj pobude kola ne bi smeo da se zada u trenutku 0.0s, jer bi komponente na njega reagovale sa nultim kašnjenjem smatrajući ga delom inicijalizacione aktivnosti.

Naravno, ovaj način inicijalizacije nema smisla ako pri simulaciji nisu zadate konkretne vrednosti za stanja ulaznih terminala `a` i `b`. Zato AleC++ podržava inicijalizaciju signala pri deklarisanju. Kao što je već pomenuto, svaki signal se mora deklarisati pre upotrebe. Pri tome je moguće, ali nije obavezno zadati inicijalnu vrednost signala. U sledećem primeru se početna vrednost signala `a` postavlja na `'0'`, a početne vrednosti signala `vec[0]`, `vec[1]` i `vec[2]` postavljaju se na `'1'`, `'X'` i `'0'`, respektivno.

```
four_t a='0', vec[3]="1X0";
```

Da bi inicijalizacija uopšte bila moguća, inicijalizacija vrednosti signala je obavezna za globalne ulaze simuliranog kola. Kod relativno jednostavnih kola moguće je odrediti i zadati stanja internih signala. Medjutim, kod složenijih kola nije tako jednostavno odrediti početna stanja u svim čvorovima kola i tu je simulator zadužen da propagacijom stanja sa ulaza postavi stanja u internim čvorovima.

Kod sekvencijalnih logičkih elemenata (flip-flopovi, memorije) moguće je primeniti sličan mehanizam inicijalizacije. Ovde ćemo izložiti metod za definisanje inicijalno memorisanog stanja flip-flopa nezavisno od stanja ulaza i taktnog signala.

```
// Rising edge triggered d flip-flop:
module cffd_init (four_t in d, clk; four_t out q, nq) {
  action (double tp, int initialize=0, four_t init_state='X') {
    ff_beh: process (clk) {
      if (initialize && now==0.0s) {
        q  <-  init_state;
        nq <- ~init_state;
      }
      else if (now > 0.0s) {
        if (clk == 'x') {
          q  <- 'x' after tp;
          nq <- 'x' after tp;
        }
        else if (clk->event && clk == '1') {    // rising edge
          q  <-  d after tp;
          nq <- ~d after tp;
        }
      }
    }
  }
}
```

Modul `cffd_init` predstavlja sinhroni D flip-flop. Na ulazni terminal `d` dovode se podaci, a na ulaz `clk` dovodi se taktni signal. Flip-flop je realizovan tako da ima mogućnost postavljanja inicijalnog stanja na izlazima `q` i `nq` pri aktivaciji procesa `ff_beh` sa nultim kašnjenjem u izabrano stanje. Flip-flop ima kašnjenje `tp` u normalnom radnom režimu zadato kao akcioni parametar. Akcioni parametar `initialize` treba postaviti na vrednost različitu od nule ako se želi izlazima nametnuti početno stanje. Početno stanje izlaza `q` zadaje se kao vrednost parametra `init_state` pri pozivu komponente. Početno stanje izlaza `nq` je naravno komplement stanja zadatog pametrom `init_state`. Treba uočiti da je parametre `initialize` i `init_state` moguće izostaviti i komponentu koristiti bez inicijalizacije sa nultim kašnjenjem. To daje mogućnost korisniku da fiksira stanja samo nekih flip-flopova u simuliranom kolu, a da ostale ostavi u proizvoljnom stanju (stanje `'X'` sve do pojave prednje ivice `clk` signala). Svi pozivi flip-flopa `cffd_init` u narednom primeru su validni.

```
module cffd_init ff1, ff2, ff3;
signal four_t d1, d2, d3, clk_sig, q1, q2, q3, nq1, nq2, nq3;
ff1 (d1, clk_sig, q1, nq1) { tp=10ns; initialize=1; init_state='1'; }
ff2 (d2, clk_sig, q2, nq2) { tp=12ns }
ff3 (d3, clk_sig, q3, nq3) action (10ns, 1);
```

2.3.11 Modeliranje hazarda i vremenskih ograničenja

U toku rada digitalnog kola mogu se javiti pojave koje izazivaju neželjena stanja signala. Od logičkog simulatora se očekuje da takve pojave detektuje i prijavi korisniku kako bi ovaj preduzeo odgovarajuće mere za njihovo otklanjanje. Jedna od takvih pojava je hazard. Postoji statički i dinamički hazard. Ako kolo koje treba da obavlja odredjenu logičku funkciju posmatramo spolja, kao blok sa ulaznim i izlaznim terminalima, hazard predstavlja neželjenu kratkotrajnu oscilaciju na izlazu kola do koje dolazi pri promeni stanja ulaza. Obično se hazard javlja pri simultanoj promeni stanja više ulaznih terminala.

Neka kolo obavlja logičku funkciju `f`. Pod ulaznim vektorom podrazumevaćemo uredjeni skup stanja na ulazima kola. Neka su v_1 i v_2 dva različita ulazna vektora. Ako je $f(v_1)=f(v_2)$, postoji mogućnost da se na izlazu javi statički hazard pri promeni ulaznog vektora sa v_1 na v_2 ili obrnuto. Neka je, recimo, `f(v1)=f(v2)='1'`. Statički 1-hazard javio se ako se na izlazu dobije

oscilacija tipa '1'-'0'-'1'-...-'0'-'1'. Pri tome, minimalna izlazna sekvenca stanja je '1'-'0'-'1', a treba primetiti da sve hazardne sekvence sadrže neparan broj stanja (3, 5, 7, ...). Za $f(v_1)=f(v_2)$='0', statički 0-hazard javio se ako je na izlazu zabeležena sekvenca '0'-'1'-'0'-...-'1'-'0'. Ako je $f(v_1)\neq f(v_2)$, postoji mogućnost da se na izlazu javi dinamički hazard pri promeni ulaznog vektora sa v_1 na v_2 i obrnuto. Na primer, ako je $f(v_1)$='0', a $f(v_2)$='1' i na izlazu se javi sekvenca '0'-'1'-'0'-...-'0'-'1', u pitanju je dinamički hazard pri prelazu sa logičke nule na logičku jedinicu. Minimalna sekvenca koja predstavlja dinamički hazard je u ovom slučaju '0'-'1'-'0'-'1', a sve validne sekvence sadrže paran broj stanja (4, 6, 8, ...). Govoreno terminologijom logičke simulacije, pri statičkom hazardu mora doći do parnog broja dogadjaja (dogadjaj je promena stanja, sekvenca '0'-'1'-'0' sadrži dva dogadjaja), a pri dinamičkom hazardu do neparnog broja dogadjaja na izlazu. Potrebno je naglasiti da hazard predstavlja pojavu podložnu verovatnoći - u istoj seriji logičkih kola neka mogu ispoljavati pojavu hazarda, a druga ne, zavisno od tolerancija tehnoloških parametara pri izradi kola.

Kod simulatora Alecsis2.1 jednostavno je razviti model logičkog kola kod koga se detektuju hazardne situacije. Šta treba učiniti kada se detektuje hazard, pitanje je na koje nije moguće odgovoriti unapred - pri razvoju biblioteke za digitalnu simulaciju. Naime, odluku o akciji donosi korisnik u toku simulacije. Moguće je razviti model kod koga se izbor obavlja definisanjem vrednosti odredjenih modelskih parametara, ali takav model postaje komplikovan, nečitljiv za korisnika i nepotrebno usporava simulaciju.

U tretmanu hazarda moguća su dva pristupa. Prvi pristup je neaktivan: treba samo detektovati hazardnu situaciju i dati upozorenje korisniku. U tom slučaju dovoljno je u akcioni blok modeliranje komponente dodati proces koji prati stanja na ulaznim priključcima i detektuje pojavu koja izaziva hazard na izlazu. Na primer, kod sledećeg modela dvoulaznog I kola hazard na izlazu pojavljuje se ako se na ulazima jave suprotne ivice (na jednom prelaz sa nule na jedinicu, a na drugom sa jedinice na nulu) u vremenskom intervalu kraćem od zadatog.

```
module and2 (four_t in a, b; four_t out y) {
  action (double t_haz ...) {
    ...
    process (a, b) {
      double last_a=0, last_b=0;
      if (a->event) last_a = now;
      if (b->event) last_b = now;
      if (a->event && b->event && a!=b && abs(last_a-last_b)<t_haz)
        warning ("static 0-hazard detected");
    }
    ...
  }
}
```

Drugi pristup je aktivan i zasniva se na pretpostavci da je potrebno na izlazni terminal proslediti posledice pojave hazarda. Za ovu namenu moguće je definisati posebne sisteme stanja koji sadrže i hazardna stanja [Lito91b, Fant74]. Recimo, moguće je definisati sistem sa stanjima '0', '1', 'R', 'F' i '*', gde stanje 'R' predstavlja prelaz sa logičke nule na logičku jedinicu, stanje 'F' predstavlja prelaz sa logičke jedinice na logičku nulu, dok stanje '*' obuhvata neodredjeno stanje i hazarde.

```
typedef enum { '0', '1', 'R', 'F', '*' } five_t;
```

U ovakvom sistemu stanja mogu se razviti modeli osnovnih logičkih elemenata koji reaguju na hazardne situacije postavljanjem izlaza u stanje '*'. Kao primer neka posluži model dvoulaznog I kola.

```
const five_t and_tab[4][4] = {
//'0'  '1'  'R'  'F'  '*'
  '0', '0', '0', '0', '0',    // '0'
  '0', '1', 'R', 'F', '*',    // '1'
```

```
  '0', 'R', 'R', '*', '*',    // 'R'
  '0', 'F', '*', 'F', '*',    // 'F'
  '0', '*', '*', '*', '*'     // '*'
}
module and2 (five_t in a, b; five_t out y) {
  action (double tp) {
    process (a, b) { y <- and_tab[a][b] after tp; }
  }
}
```

Propagacijom stanja `'*'` kroz kolo moguće je analizirati uticaj hazarda na ispravan rad kola. Kod složenijih modela kašnjenja moguće je hazardnu situaciju tumačiti kao pojavu neodredjenog stanja na izlaznom terminalu u trajanju koje je moguće odrediti pažljivom analizom parametara kašnjenja modeliranog logičkog elementa. Treba primetiti da kod ovog drugog (aktivnog) pristupa modeliranju hazarda nije moguće jednostavno dodati još jedan proces modelu gejta, već je potrebno intervenisati u istom procesu u kome se modelira logička funkcija komponente.

Pod vremenskim ograničanjima koja je potrebno uzeti u obzir pri modeliranju logičkih kola podrazumevamo vremena postavljanja, vremena držanja (kod taktovanih sekvencijalnih logičkih elemenata, recimo kod flip-flopova), minimalna trajanja stanja (širine impulsa u odredjenim signalima) i druga. I ovde je mogućpasivni i aktivni pristup pri modeliranju. Kod pasivnog pristupa, vremenska ograničenja se proveravaju u posebnim dodatnim procesima. Kod aktivnog pristupa provera ograničenja ograničenja obavlja se u procesu u kome je definisana i logička funkcija komponente, čime ovaj proces postaje složeniji i teško razumljiv korisniku biblioteke.

Sledeći primer ilustruje pasivnu detekciju prekršaja vremena postavljanja signala na `d` ulazu D flip-flopa u odnosu na opadajuću ivicu na `clk` ulazu.

```
module dtypeff (four_t in d, clk; four_t out q, nq) {
  action (double tp, double dsetup) {
    process (d, clk) {
      double last_d = 0.0;
      wait d;                       // wait event on d
      last_d = now;
      wait clk while clk!='0';      // wait falling edge on clk
      if (now-last_d < dsetup) warning ("d setup violation detected");
    }
  }
}
```

Proveru trajanja impulsa na odredjenim signalima moguće je izvršiti vezivanjem posebnih komponenti za te signale, takozvanih vremenskih čekera, kao kod simulatora PSpice [-91]. Modul `width_checker_1`, čiji opis sledi, proverava da li je trajanje stanja logičke jedinice na signalu `y` duže od `min_width_1` i ako nije, javlja poruku korisniku o pojavi impulsa visoke frekvencije.

```
module width_checker_1 (four_t in y) {
  action (double min_width_1) {
    process {
      double last_time = 0.0;
      wait y while y!='1'; last_time = now;
      wait y while y=='1';
      if (now-last_time < min_width_1) warning ("impulse too short");
    }
  }
}
```

2.3.12 Automatska konverzija u hibridnim čvorovima

Pri simulaciji hibridnih kola, simulator razdvaja hibridne čvorove na dva čvora: analogni na analognoj strani i digitalni na digitalnoj strani. Pri tome se automatski ubacuju A/D i D/A konvertori. Konvertor je modul kao i svaki drugi, osim što u interfejsu mora imati jednu vezu tipa

`node` (analogni čvor) i jednu vezu tipa `signal` (digitalni čvor). Sve ostalo definiše korisnik. U okviru modula treba definisati strukturu kola za spregu i način konverzije signala. Pod kolom za spregu podrazumeva se analogni model ulaza, odnosno izlaza logičkog kola koji se ubacuje u analoni deo kola kako bi se očuvala tačnost analogne simulacije. Način konverzije signala podrazumeva pretvaranje analogne veličine (sa analogne strane hibridne veze) u digitalnu vrednost (koja se vodi na ulaze digitalnih kola) i pretvaranje digitalnog signala (sa izlaza digitalnog dela kola) u analognu vrednost (kojom se pobudjuje analogni deo kola). Opširnije informacije o konverziji mogu se naći u [Mrča92, Gloz94b].

Kod simulatora Alecsis2.1 konverzija je povezana sa digitalnim delom kola. Tako se konvertori definišu za svaki modul (gejt) u biblioteci za digitalnu simulaciju. Imena konverzionih modula navode se u deklaraciji `conversion`, a sami moduli se pri simulaciji pozivaju iz objektnih datoteka navedenih u komandi `library`.

```
conversion { a2d="a2d_converter_name"; d2a="d2a_converter_name"; }
library conv_lib;
```

2.3.13 Komunikacija sa korisnikom

Pri razvoju digitalnih (i analognih) biblioteka može da se predvidi niz uslužnih funkcija koje treba da omoguće jednostavniju komunikaciju izmedju simulatora i korisnika (ispis poruka na ekranu, foirmiranje raznih datoteka za dokomentovanje simulacije, vodjenje statistike o simulaciji i slično). Ovde ćemo pomenuti samo jednu funkciju ugradjenu u sam programski kod simulatora Alecsis2.1 koja će biti često korišćena u poglavljima o razvoju biblioteka za digitalnu simulaciju. To je funkcija `warning` koja je deklarisana na sledeći način.

```
void warning(const *char, int=0);
```

Prvi formalni parametar funkcije `warning` je tekst koji treba štampati na ekranu, a drugi parametar je ceo broj koji definiše akciju simulatora posle ispisivanja upozorenja na ekranu. Ako je drugi parametar postavljen na vrednost različitu od nule (recimo 1), simulator prekida simulaciju (smatra se da je došlo do greške fatalne za nastavak simulacije), a ako je postavljen na vrednost nula, simulacija se nastavlja. Drugi parametar ima difoltnu vrednost nula, tako da se može izostaviti. Primeri koji slede ilustruju upotrebu funkcije `warning`.

```
if (lengthof vec < 2) warning ("vector too short", 1);
warning ("static 0-hazard detected, but ignored", 0);
warning ("static 0-hazard detected, but ignored");
```

Funkcija `warning`, osim same poruke, štampa na ekranu i informaciju o modulu u okviru kojeg je došlo do njenog poziva, vrednost simulacionog vremena u trenutku poziva, kao i redni broj poziva.

2.4 Razvoj biblioteka i njihovo korišćenje

U današnjim uslovima oštre konkurencije na tržištu integrisanih kola, projektovanje VLSI integrisanih kola sve redje se izvodi full custom metodom. Većina proizvodjača koristi neki od metoda projektovanja koji obezbedjuje skraćeno vreme projektovanja (metod standardnih ćelija, metod gejtovskih polja i drugi). Svi se ovi metodi zasnivaju na odredjenom stepenu pretprojektovanja koje podrazumeva postojanje biblioteke ćelija. Projektovanje biblioteke ćelija počinje od layout-a, nastavlja se električnom karakterizacijom, a završava logičkom karakterizacijom. Da bi se projekat verifikovao logičkim ili hibridnim simulatorom, potrebno je ćelije modelirati korišćenjem ulaznog jezika raspoloživog simulatora. Od uvodjenja VHDL

standarda za opis digitalnih kola, biblioteke se uglavnom razvijaju na VHDL-u [Nous93]. VHDL simulatori nemaju unapred ugradjene logičke elemente, pa dobijaju upotrebnu vrednost tek sa razvojem biblioteka logičkih elemenata. Simulator Alecsis2.1 im je u tom pogledu vrlo sličan.

Razvoj biblioteke za digitalnu simulaciju simulatorom Alecsis2.1 predstavlja kompleksan zadatak koji podrazumeva sledeće aktivnosti:

- izbor sistema stanja,
- izbor skupa osnovnih logičkih elemenata (gejtova) koje treba modelirati,
- izbor modela kašnjenja,
- izbor načina parametrizovanja osnovnih gejtova tako da budu univerzalno upotrebljivi i
- kreiranje samog programskog koda na jeziku AleC++.

Posao programera u okviru poslednje aktivnosti obuhvata definisanje sistema stanja, predefinisanje (preopterećenje) logičkih operatora za rad u novom sistemu stanja, definisanje funkcija za kašnjenje koje odgovaraju izabranom modelu kašnjenja, definisanje rezolucionih funkcija (za magistralu bez i sa pullup i pulldown otpornikom, za žičanu logiku i slično), definisanje drugih uslužnih funkcija koje olakšavaju korišćenje biblioteke, modeliranje osnovnih logičkih elemenata, kao i definisanje nekih osnovnih modela za potrebe testiranja biblioteke. Testiranje biblioteke obavlja se najpre pojedinačno za svaki razvijeni logički element, a zatim i za složenija digitalna kola formirana od osnovnih elemenata. Takodje je potrebno testirati razvijene funkcije za kašnjenje i rezoluciju u svim mogućim situacijama. Po testiranju biblioteke potrebno je napisati detaljno uputstvo za njenu upotrebu koje će krajnjem korisniku omogućiti jednostavno rukovanje. Medjutim, ma kako dobro bilo napisano uputstvo za rukovanje bibliotekom, zbog specifičnih zahteva pri simulaciji pojedinih digitalnih kola neophodno je intervenisati na programskom kodu.

Da bi se programski kod biblioteke učinio čitljivim i razumljivim čak i za manje upućenog korisnika, usvojene su odredjene konvencije. Ove konvencije umnogome podsećaju na nepisana pravila programiranja u programskom jeziku C++, tako da ih je lako razumeti i prihvatiti njihovu svrsishodnost.

******* Svaka digitalna biblioteka sadrži jednu ili više ***deklaracionih datoteka*** sa ekstanzijom **.h** u kojima su smeštene definicije makroa, deklaracije korišćenih struktura podataka, globalnih podataka, tabela, modelskih klasa, funkcija, modula. U jednoj od ovih datoteka definisan je i sistem logičkih stanja koji je osnova biblioteke. Deklaracijske datoteke priključuju se preprocesorskom naredbom `#include` drugim datotekama, kao i datoteci koja sadrži opis kola za simulaciju (`root` modul). Ovo je potrebno stoga što kod jezika AleC++ postoji pravilo da svaki objekat mora biti ili deklarisan ili definisan pre upotrebe. S obzirom da se objekti obično linkuju pozivom iz bibliotečkih datoteka, neophodno ih je deklarisati kako bi prevodilac mogao proveriti ispravnost njihovog korišćenja u konkretnom kontekstu. Deklaracijske datoteka obično nose naziv **ssXXX.h** (state system XXX header file), gde **XXX** predstavlja broj stanja u korišćenom sistemu stanja ili mnemonik koji asocira na sistem stanja.

******* Svaka digitalna biblioteka sadrži jednu ili više ***funkcijskih datoteka***. Funkcijske datoteke sadrže definicije globalnih podataka, definicije funkcija za preopterećenje logičkih operatora, funkcija za kašnjenje, rezolucionih funkcija, funkcija u vezi sa korisničkim atributima signala i druge slične objekte. Ove datoteke nose ekstenziju **.hi,** a obično se nazivaju **funXXX.hi** ili **opXXX.hi** (functions/opetators in state system XXX). Funkcijske datoteke se prevode u objektni kod (datoteke sa ekstenzijom **.ao**) i priključuju pri linkovanju korišćenjem naredbe `library`.

*** Svaka digitalna biblioteka sadrži jednu ili više ***strukturnih (modulskih) datoteka***. Modulske datoteke sadrže definicije modula koji predstavljaju osnovne logičke elemente - gejtove. Ovi moduli se koriste kao strukturni blokovi za opis simuliranih digitalnih kola. Modulske datoteke

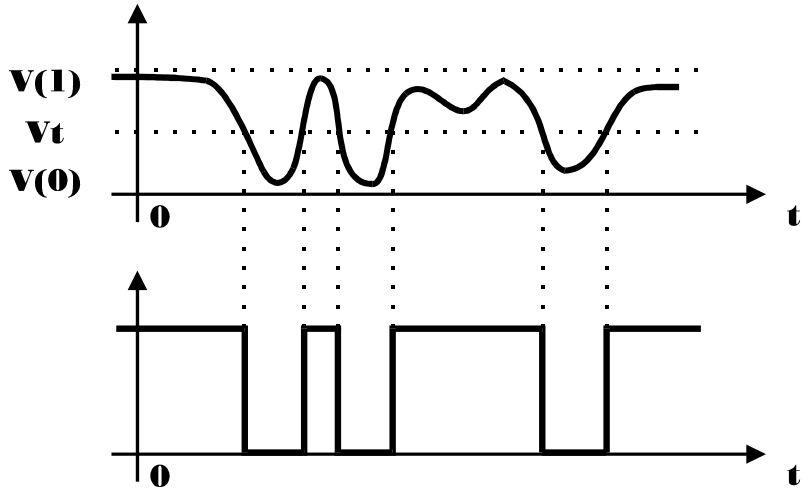
se obično nazivaju **gatesXXX.hi** ili **strXXX.hi** i prevode se u objektni kod. Priključuju se pri linkovanju korišćenjem naredbe library.

*** U većini slučajeva digitalna biblioteka sadrži jednu ili više ***modelskih datoteka*** u kojima su definisane modelske kartice gejtova iz strukturnih datoteka. Pri razvoju biblioteke sa iole složenijim modelom kašnjenja koristi se mehanizam modelskih klasa. Svakoj modelskoj klasi odgovara difoltni model sa imenom **classname**`_default_model` gde je **classname** ime modelske klase. Recimo, modelskoj klasi `m15` odgovara difoltni model `m15_default_model`. Difoltni model se definiše u modelskoj datoteci i omogućava neiskusnom korisniku lakše snalaženje sa bibliotekom. Naime, pri prvom kontaktu sa bibliotekom korisnik ne mora da poznaje detaljno parametre modela (detaljno upoznavanje može zahtevati mnogo sati proučavanja literature i simulacije), već može da koristi difoltni model pri probnim simulacijama. Modelska datoteka u kojoj su smešteni difoltni modeli obično nosi ime **modelXXX.hi** ili **modXXX.hi**. Pored toga, biblioteka može da sadrži mnogo različitih modelskih datoteka koje korisnik kreira da bi modelirao konkretne komponente pri nekoj specifičnoj simulaciji. Modelske datoteke se prevode u objektni kod i priključuju pri linkovanju komandom `library`.

3 Biblioteka za sistem sa dva stanja

Najjednostavniji sistem logičkih stanja sadrži samo dva stanja - logičku jedinicu i logičku nulu. Konverzija analogne veličine u sistem sa dva stanja izvodi se poredjenjem sa pragom promene stanja V_t, kao što je prikazano je na slici 3.1.

Simulator sa dva stanja može se iskoristiti za grubu verifikaciju logičke funkcije kola. Kako je izabrani sistem stanja jednostavan, obično se modeli logičkih elemenata ne opterećuju složenijim modelima kašnjenja. Ovakav simulator se koristi uglavnom u preliminarnim fazama projektovanja, dok se precizne vremenske karakteristike projektovanog kola proveravaju simulatorom sa složenijim sistemom stanja i modelom kašnjenja.



Slika 3.1 *Konverzija analogne veličine u sistem sa dva stanja*

Biblioteka za simulaciju u sistemu sa dva stanja zasnovana je na usvojenom stilu za razvoj logičkih biblioteka za simulator Alecsis2.1. Organizovana je u četiri datoteke:

ss2.h - deklaraciona datoteka
op2.hi - funkcijska datoteka
gates2.hi - modulska datoteka
model2.hi - modelska datoteka

3.1 Definisanje sistema stanja i preopterećenje operatora

Sistem sa dva logička stanja kreira se na sledeći način.

```
typedef enum { '0', '1', '_'=void } two_t;
```

Definisan je novi enumerisani tip podataka sa imenom `two_t` koji sadrži karakter konstante `'0'` i `'1'` (logička stanja) i separator `'_'`. Separator služi za poboljšanje preglednosti pri ispisivanju vektora stanja. Treba uočiti da svi signali i promenljive tipa `two_t` uzimaju za početnu vrednost stanje `'0'`, jer je ono prvo navedeno u definiciji tipa `two_t`.

Kod sistema stanja definisanih korišćenjem enumerisanih tipova podataka pogodno je definisati tabelu sa imenima stanja radi olakšavanja ispisa logičkih stanja na ekranu funkcijom `printf` i drugim sličnim funkcijama. Za sistem sa dva stanja definisana je tabela po imenu `names2`. Na ovom mestu ćemo pomenuti još jednu konvenciju koje ćemo se držati u svim razvijenim simulacionim bibliotekama: tabele sa imenima stanja označavaćemo sa `namesXXX`, gde `XXX` predstavlja broj stanja u sistemu ili neki mnemonik koji asocira na sistem stanja.

```
extern const char *names2[];
```

Gornja deklaracija nalazi se u deklaracijskoj datoteci **ss.h**, dok se definicija tabele nalazi u funkcijskoj datoteci **op2.hi**.

```
const char *names2[] = { "0", "1" };
```

Sada je moguće napisati:

```
two_t x[8] = "0110_0001";
for (int i=0; i<8; i++) printf ("x[%d] = %s\n", i, names2[x[i]]);
```

Da bi se pojednostavilo modeliranje logičkih funkcija, poželjno je predefinisati (preopteretiti) logičke operatore za rad u novom sistemu stanja. Funkcije za preopterećenje operatora deklarisane su u deklaracijskoj datoteci na sledeći način.

```
extern two_t operator  ~ (two_t op);              // complement
extern two_t operator  & (two_t op1, two_t op2); // and
extern two_t operator  | (two_t op1, two_t op2); // or
extern two_t operator ~& (two_t op1, two_t op2); // nand
extern two_t operator ~| (two_t op1, two_t op2); // nor
extern two_t operator  ^ (two_t op1, two_t op2); // xor
extern two_t operator ~^ (two_t op1, two_t op2); // xnor
```

Pogodno je definisati tabele istinitosti logičkih funkcija, tako da funkcije za preopterećenje operatora budu vrlo jednostavne - svode se na iščitavanje odgovarajuće vrednosti iz tabele istinitosti. Deklaracije potrebnih tabela takodje se nalaze u deklaracionoj datoteci.

```
extern two_t const  not_tab    [2];
extern two_t const  and_tab [2][2];
extern two_t const nand_tab [2][2];
extern two_t const   or_tab [2][2];
extern two_t const  nor_tab [2][2];
extern two_t const  xor_tab [2][2];
extern two_t const nxor_tab [2][2];
```

Same tabele istinitosti i funkcije za preopterećenje operatora definisane su u funkcijskoj datoteci. Navešćemo primer unarne logičke operacije komplementiranja i binarne NI logičke operacije za operande tipa `two_t`.

```
two_t const not_tab[2] = { '1', '0' };
two_t operator  ~ (two_t op) { return not_tab[op]; }
```

```
two_t const nand_tab[2][2] = { '1', '1',
                               '1', '0'  };
two_t operator ~& (two_t op1, two_t op2) { return nand_tab[op1][op2]; }
```

Kako se enumerisani tipovi u programskim jezicima predstavljaju pomoću odgovarajućih celobrojnih indeksa, to se izraz `nand_tab['1']['0']` tumači kao "element matrice `nand_tab` iz prve vrste, a nulte kolone", jer konstanti `'1'` odgovara celobrojni indeks 1, a konstanti `'0'` indeks 0 (prema redosledu pojavljivanja u definiciji tipa `two_t`). Očigledno je da AleC++ u ovakvom slučaju automatski konvertuje enumerisani tip `two_t` u tip `int`. U nekim slučajevima implicitna konverzija nije moguća i tada treba koristiti *cast* operator na sledeći način.

```
nand_tab[(int)op1][(int)op2]; }
```

3.2 Modeli osnovnih logičkih elemenata - klasični pristup

Sledeći korak u razvoju biblioteke je modeliranje osnovnih logičkih elemenata (gejtova, logičkih primitiva). Oni se deklarišu u deklaracionom fajlu **ss2.h** na sledeći način.

```
module sinv  (two_t in a;    two_t out y) action (double tr, double tf);
module sbuf  (two_t in a;    two_t out y) action (double tr, double tf);
module sand  (two_t in a, b; two_t out y) action (double tr, double tf);
module sandx (two_t in a[];  two_t out y) action (double tr, double tf);
module snand (two_t in a, b; two_t out y) action (double tr, double tf);
module snandx(two_t in a[];  two_t out y) action (double tr, double tf);
module sor   (two_t in a, b; two_t out y) action (double tr, double tf);
module sorx  (two_t in a[];  two_t out y) action (double tr, double tf);
module snor  (two_t in a, b; two_t out y) action (double tr, double tf);
module snorx (two_t in a[];  two_t out y) action (double tr, double tf);
module sxor  (two_t in a, b; two_t out y) action (double tr, double tf);
module snxor (two_t in a, b; two_t out y) action (double tr, double tf);
module smux21(two_t in x0, x1, c; two_t out y)
                                          action (double tr, double tf);
module smux41(two_t in x0, x1, x2, x3, cmsb, clsb; two_t out y)
                                          action (double tr, double tf);
module sfa  (two_t in a, b, ci; two_t out s, co)
                                          action (double tr, double tf);
module sffd  (two_t in d, reset, c; two_t out q, nq)
                                          action (double tr, double tf);
module sffjk (two_t in j, k, reset, c; two_t out q, nq)
                                          action (double tr, double tf);
```

Navedene deklaracije govore dovoljno same za sebe. Recimo, modul `sinv` ostvaruje funkciju invertora. Prefiks `s` znači da je u pitanju jednostavni (simple) model, kasnije ćemo govoriti i o složenijim modelima invertora u istoj biblioteci. U ulazno/izlaznoj listi modul `sinv` ima dva terminala: ulazni `a` i izlazni `y`. Na ulazni i izlazni terminal mogu se spregnuti samo signali tipa `two_t`. Kod nekih deklarisanih modula sreće se postfiks `x`, koji označava da su ulazni terminali modula zadati u vektorskom obliku. Moduli `smux21` i `smux41` su multiplekseri 2 u 1 i 4 u 1, respektivno. Modul `sfa` je potpuni sabirač. Deklarisana su i dva sinhrona flip-flopa: modul `sffd` je D flip-flop, a modul `sffjk` je JK flip-flop.

Deklaracije služe kompajleru za proveru ispravnosti U/I interfejsa gejta pri njegovom korišćenju u opisu simuliranog kola. Usvojeno je pravilo u jeziku AleC++ da objekat mora biti ili deklarisan ili definisan pre upotrebe. Stoga se ove deklaracije priključuju na početku simulacione datoteke korišćenjem naredbe `include`.

```
# include "ss2.h"
```

Kao što se u deklaracijama vidi, opredelili smo se za dodeljivo rise/fall kašnjenje. Kod tog modela kašnjenja razlikuje se kašnjenje uzlazne i opadajuće ivice signala na izlazu gejta. Akcioni

parametri `tr` i `tf` predstavljaju ova kašnjenja i njihova vrednost se mora zadati za svaki modul pri njegovom uključivanju u opis kola za simulaciju.

Definicije modula smeštene su u modulsku datoteku **gates2.hi**. Kao primer ćemo navesti modul koji predstavlja NI kolo sa dva ulaza.

```
module snand (two_t a,b; two_t out y) {
  action (double tr, double tf) {
    process (a, b) { y <- a~&b after ((a~&b)=='0' ? tf : tr); }
  }
}
```

Modul `snand` ima ulaze `a` i `b` i izlaz `y`. Parametri modela su mu kašnjenje prednje ivice `tr` i kašnjenje zadnje ivice `tf`. Unutar akcionog bloka nalazi se samo jedan proces osetljiv na promene na ulazima `a` i `b`, a koji kreira drajver za signal `y` (sadrži operator `<-`). U drajver signala `y` se pri svakoj promeni stanja nekog od ulaza šalje novo stanje izlaza (odredjeno primenom ranije predefinisanog logičkog operatora `~&`) posle odgovarajućeg kašnjenja. Za odredjivanje vrednosti kašnjenja upotrebljen je uslovni izraz poznat iz jezika C i C++. Izraz `(a~&b)=='0' ? tf : tr` vraća vrednost `tf` ako je izračunato stanje izlaza `'0'`, a vrednost `tr` ako je izračunato stanje izlaza `'1'`.

Ovakav opis NI kola podseća na VHDL kod koga je takodje moguć prenos generičkih parametara (čemu odgovara `action` u AleC++). Ako, medjutim, želimo da iskoristimo prednosti objektno orjentisanog modeliranja koje nudi simulator Alecsis2.1, onda ćemo gejtove definisati kao entitete koji primaju modelsku karticu.

3.3 Modeli osnovnih logičkih elemenata - objektno orjentisani pristup

Prvo treba definisati klasu koja će sadržati parametre modela logičkih elemenata. Kako je Alecsis2.1 hibridni simulator, bitno je da se definišu A/D i D/A konvertori koje simulator automatski ubacuje na mestima sprezanja analognih i digitalnih delova opisivanog kola. Konvertori se definišu u digitalnim bibliotekama ako su ove namenjene i za hibridnu simulaciju. Kod simulatora Alecsis2.1 usvojili smo konvenciju da se konvertori sa svojim parametrima smeštaju u jednu baznu klasu iz koje se zatim izvode klase modela pojedinih logičkih komponenti. U deklaracionoj datoteci **ss2.h** deklarisana je modelska klasa `io2` na sledeći način:

```
typedef struct { double time; double level; } Point;
enum Status { Rising, Falling, Steady };
class io2 {
  protected:
            // parameters for simple CMOS D/A converter
      double rise_time;       // rising ramp duration
      double fall_time;       // falling ramp duration
      double zero_level;      // voltage level of logic 0
      double one_level;       // voltage level of logic 1
      double Rout;            // logic gate output resistance, const
      double Cout;            // logic gate output capacitance, const
      double LevelTol;        // voltage tolerance
            // parameters for simple CMOS A/D converter
      double Cin;             // digital input strey capacitance
      double treshold;        // '0' - '1' treshold voltage
  public:
      io2();
      >io2 ();
      void set_breakpoints (double, two_t, two_t, Point*, Point*, Status*);
      double evaluate_vramp (double, Point*, Point*);
      friend module cmos_a2d, cmos_d2a;
};
```

Kao "prijatelji" (deklaracija `friend`) klase `io2` navedeni su moduli `cmos_a2d` i `cmos_d2a`. Moduli `cmos_a2d` i `cmos_d2a` su AD i DA konvertor za CMOS logička kola i deklarisani su takodje u datoteci **ss2.h** kao objekti klase `io2`, tako da im se pridružuje modelska kartica klase `io2`.

```
module io2::cmos_a2d (node analog; signal two_t out digital);
module io2::cmos_d2a (signal two_t in digital; node analog);
```

O konvertorima će biti više reči kasnije.

Iz klase `io2` izvedena je klasa `gm2` koja sadrži parametre modela osnovnih logičkih elemenata. Ova klasa deklariše se na sledeći način.

```
class gm2 : public io2 {            // gate models in 2-state system
      double tplh, tphl;
  public:
      gm2 ();
      double delay2 (two_t, int);
      friend module buf, inv, and, or, andx, orx, nand, nor, mux21,
                    mux41, fa, add, add_alternative, shift, ffd, ffjk;
};
```

Konstruktori klasa `io2` i `gm2` definišu difoltne vrednosti privatnih podataka, a ove se vrednosti kasnije mogu menjati u definiciji modelskih kartica ili pomoću funkcija koje imaju odgovarajuće pravo pristupa (metodi klase ili prijateljske funkcije).

```
io2::io2 () {     // defaults:
  rise_time  = fall_time = 2ns;
  zero_level = 0v;
  one_level  = 5v;
  Rout       = 1kohm;
  Cout       = 10fF;
  LevelTol   = 1nv;

  Cin        = 100fF;
  threshold  = (zero_level + one_level)/2.0;
}
gm2::gm2 () { tplh = tphl = 10ns; }
```

Logički primitivi sada su objekti klase `gm2`, čime im automatski pripada modelska kartica u kojoj se definišu vrednosti podataka koje klasa "krije" kao privatne članove.

```
module gm2::buf   (two_t in a;   two_t out y);
module gm2::inv   (two_t in a;   two_t out y);
module gm2::and   (two_t in a,b; two_t out y);
module gm2::or    (two_t in a,b; two_t out y);
module gm2::andx  (two_t in a[]; two_t out y);
module gm2::orx   (two_t in a[]; two_t out y);
module gm2::xor   (two_t in a,b; two_t out y);
module gm2::nand  (two_t in a,b; two_t out y);
module gm2::nor   (two_t in a,b; two_t out y);
module gm2::nxor  (two_t in a,b; two_t out y);
module gm2::nxor  (two_t in a,b; two_t out y);
module gm2::mux21 (two_t in x0, x1, c; two_t out y);
module gm2::mux41 (two_t in x0, x1, x2, x3, cmsb, clsb; two_t out y);
module gm2::fa    (two_t in a, b, ci; two_t out s, co);
module gm2::ffd   (two_t in d, reset, c; two_t out q, nq);
module gm2::ffrs  (two_t in r, s, reset, gate; two_t out q, nq);
module gm2::ffjk  (two_t in j, k, reset, c; two_t out q, nq);
module gm2::clkgen (signal two_t inout y)
                  action (double width1, double width0, double startdelay=0.0);
```

Ovim modulima je potreban slobodan pristup privatnim elementima klase (kašnjenja `tplh` i `tphl`), pa su deklarisani kao "prijatelji" klase (ključna reč `friend`).

Sada se logički elementi definišu na sličan način kao kod klasičnog pristupa, ali imaju pridružene modelske kartice (slično SPICE modelskim karticama). Modelske kartice definišu vrednosti privatnih parametara klasa `io2` i `gm2`, tako da se ove ne prenose kao akcioni parametri. Kao što se iz deklaracije klase `gm2` vidi, funkcija za kašnjenje `delay2` definisana je kao metod klase `gm2`, tako da ima slobodan pristup parametrima kašnjenja `tplh` i `tphl`. Ova funkcija smeštena je u funkcijsku datoteku **op2.hi** i ima sledeći izgled.

```
double gm2::delay2 (two_t ns, int hyb) {
  if (hyb)
    return (ns=='0' ? tphl-fall_time/2.0 : tplh-rise_time/2.0);
  else
    return (ns=='0' ? tphl : tplh);
}
```

Ako izlazni čvor gejta ima hibridni aspekt, simulator automatski ubacuje D/A konvertor. Konvertor počinje da "gradi" rampu u trenutku pojave digitalnog signala, a trajanje rampe je definisano kao `fall_time` ili `rise_time` u modelskoj kartici (parametri klase `io2`). Da bi se dogadjaj promene stanja odigrao u pravom trenutku (promenom se smatra prelazak 50% napona izmedju logičke nule i jedinice), potrebno je kašnjenje gejta umanjiti za polovinu trajanja rampe. Funkcija pravi razliku izmedju kašnjenja prednje i kašnjenja zadnje ivice, a koristi promenljive `tplh` i `tphl` iz modelske kartice odgovarajuće komponente iz koje je funkcija pozvana. Formalni parametar `ns` predstavlja novo stanje na izlazu gejta.

Modeliranje gejtova ilustrovaćemo na primeru višeulaznog ILI kola. Sledeći kod nalazi se u datoteci **gates2.hi**.

```
module gm2::orx (two_t in a[];  two_t out y) {
  conversion { a2d = "cmos_a2d"; d2a = "cmos_d2a"; }
  action {
    process (a) {
      two_t result;
      int i;
      result = a[0];
      for (i=1; result!='1' && i<lengthof a; i++) result = result | a[i];
      y <- result after this.delay2(result, y->hybrid);
    }
  }
}
```

Komandom `conversion` definisani su A/D i D/A konvertori koje Alecsis2.1 automatski ubacuje u slučaju da je neki ulaz ili izlaz modula `orx` vezan za hibridni čvor. Modul `orx` ima poizvoljan broj ulaza, onoliki kolika je dimenzija vektora `a`. Proces definisan unutar akcionog bloka osetljiv je na promenu bilo kog signala iz vektora `a`. Unutar procesa korišćen je operator `lengthof` da bi se dobila dužina vektora `a` (ova vrednost se kao "skriveni" parametar prenosi u modul). Kašnjenje gejta odredjuje se pozivom funkcije `delay2`, a ključna reč `this` odredjuje da funkcija `delay2` koristi modelsku karticu pridruženu **ovom** gejtu da bi iz nje čitala vrednosti parametara kašnjenja. Vrednost atributa `hybrid` izlaznog signala spušta se u funkciju za kašnjenje. Implicitni atribut `hybrid` automatski se generiše uz svaki signal pri podizanju hijerarhijskog stabla opisa kola i `y->hybrid` ima vrednost 1 ako je signal `y` hibridni, a 0 ako ima čisto digitalni aspekt.

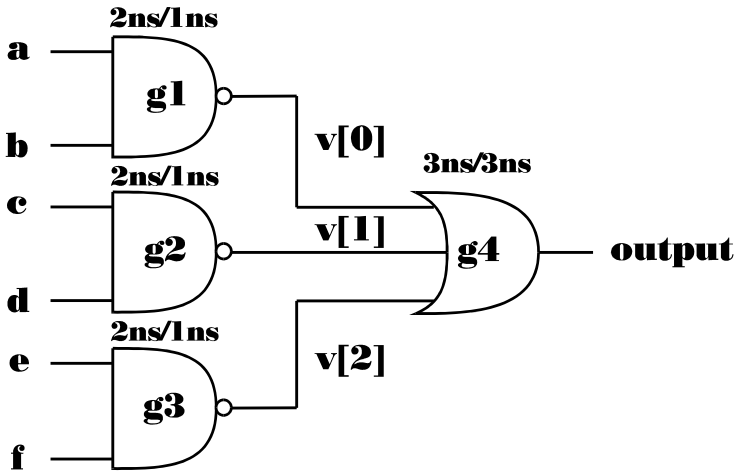
Modeli gejtova sadrže parametre konkretnih digitalnih komponenata iz simuliranog kola i smeštamo ih u posebne modelske datoteke. Modelskih datoteka može biti veliki broj, s obzirom da je za svaku simulaciju pogodno modele smestiti u posebnu modelsku datoteku. Jedan od modela višeulaznog ILI kola smešten je u osnovnu modelsku datoteku u našoj biblioteci **model2.hi** i ima sledeći izgled.

```
model gm2::orx_model_3ns {
  fall_time = 0.1ns;
  io2::rise_time = 0.2ns;     // this is O.K., too
  tplh = 3ns;
```

```
  gm2::tphl = 3ns;
};
```

Za baznu modelsku klasu `io2`, a takodje i za izvedenu klasu `gm2`, prema usvojenoj konvenciji definisani su difoltni modeli sa imenima `io2_default_model` i `gm2_default_model`. Ove dve modelske kartice nalaze se u datoteci **model2.hi**.

```
model io2::io2_default_model { }
model gm2::gm2_default_model { }
```



Slika 3.2 *Jednostavno logičko kolo*

Ilustrujmo korišćenje biblioteke sa dva stanja na jednom jednostavnom primeru. Opišimo kolo sa slike 3.2 za simulaciju programom Alecsis2.1.

```
# include "ss2.h"
library model2, gates2, op2;
root module example_test () {
  module snand g1, g2, g3;
  module gm2::orx g4;
  signal two_t a='0', b='1', c, d='0', e, f='0', v[3]="111", output;

  g1 (a, b, v[0]) action (2ns, 1ns);
  g2 (c, d, v[1]) action (2ns, 1ns);
  g3 (e, f, v[2]) action (2ns, 1ns);
  g4 (v, output) model = orx_model_3ns;

  timing { tstop = 50ns; }
  out { signal two_t a, b, c, d, e, f, v, output; }
  action {
    process initial {  // stimulus
      a <- '1' after 10ns, '0' after 20ns;
      d <- '1' after 15ns;
    }
  }
} // root
```

U navedenom opisu prva linija je naredba preprocesoru da priključi deklaracionu datoteku **ss2.h** kako bi bio poznat sistem stanja i deklarisane potrebne funkcije i moduli. Komanda `library` kaže linkeru da će biti korišćeni objekti iz objektnih datoteka **model2.ao**, **gates2.ao** i **op2.ao** (koje su dobijene kompajliranjem odgovarajućih datoteka sa ekstenzijom **.hi**). Modul označen sa `root` ima ulogu "korena" hijerarhijskog stabla opisa simuliranog sistema, odnosno, on obuhvata celo simulirano kolo, te nema interfejs sa okolinom. U njemu se prvo deklarišu korišćeni moduli i signali, zatim se definiše struktura kola, zada vreme završetka simulacije i imena signala koje treba štampati u izlaznoj datoteci, a na kraju se u akcionom bloku pomoću procesa sinhronizovanog

implicitnim signalom `initial` zada pobuda. Pri deklaraciji nekih signala zadate su inicijalne vrednosti (stanja u trenutku $t=0^-$). Preostali signali uzimaju vrednost `'0'`, jer je to prvo stanje (stanje sa celobrojnim indeksom 0) u definiciji tipa `two_t`. U deklaraciji modula `g4` može se izostaviti ime klase, tako da je deklaracija

```
module orx g4;
```

ekvivalentna onoj korišćenoj u našem `root` modulu. Ime klase ne dodaje se imenu modula pri kompajliranju, tako da imena `gm2::orx` i `orx` predstavljaju isti objekat. U slučaju pokušaja da se definišu dva modula istog imena tako da jedan bude objekat neke klase, a drugi ne, ili kao objekti dve različite klase, kompajler daje poruku o grešci.

3.4 Žičana logika

U sistemu sa dva stanja može se modelirati žičana logika. Za tu svrhu potrebno je definisati odgovarajuće rezolucione funkcije. Rezolucione funkcije za ostvarivanje veze "žičano I" i "žičano ILI" smeštene su u fajlu **op2.hi** i imaju sledeći izgled.

```
two_t wired_and (const two_t *drivers, int *rprt) {
  two_t result='1';           // non-dominant state
  for (int i=0; i<lengthof(drivers); i++) {
    if (drivers[i]=='0') { result='0'; break; }
  }
  *rprt=0;
  return result;
}

two_t wired_or (const two_t *drivers, int *rprt) {
  two_t result='0';           // non-dominant state
  for (int i=0; i<lengthof(drivers); i++) {
    if (drivers[i]=='1') { result='1'; break; }
  }
  return result;
}
```

Funkcija `wired_and` definisana je tako da vraća stanje logičke nule ako je makar jedan drajver signala u stanju logičke nule, a funkcija `wired_or` na sličan način proglašava dominantnim stanje logičke jedinice. Kako kod žičane logike ne može doći do konflikta, promenljiva `rprt` (pointer na celobrojnu promenljivu koja se setuje na vrednost 1 da bi simulator javio poruku o konfliktu) ne mora se koristiti. Rezoluciona funkcija se pridružuje signalu korišćenjem operatora `:`, kao u sledećem primeru.

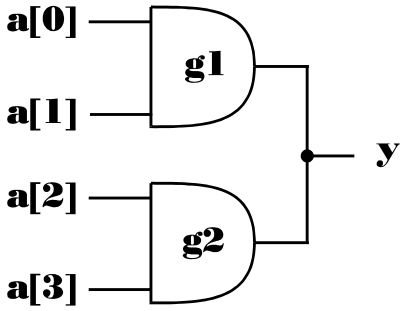
```
signal two_t : wired_or wired_signal1, wired_signal2;
signal two_t:wired_and a, b, c='1';
two_t:wired_and variable1;          // incorrect!!
```

Rezoluciona funkcija može se pridružiti samo signalima, a ne i promenljivim, tako da je treća deklaracija u gornjem primeru nedozvoljena.

Kao ilustraciju upotrebe funkcija rezolucije simuliraćemo kolo sa slike 3.3. To je kolo kod koga je iskorišćena žičana veza izlaza dva I kola za ostvarivanje složenije logičke funkcije nego što se može ostvariti sa dva I kola. Upotrebom rezolucione funkcije `wired_and` dobija se dominacija logičke nule na izlazu. Koristićemo nulto kašnjenje gejtova `g1` i `g2`.

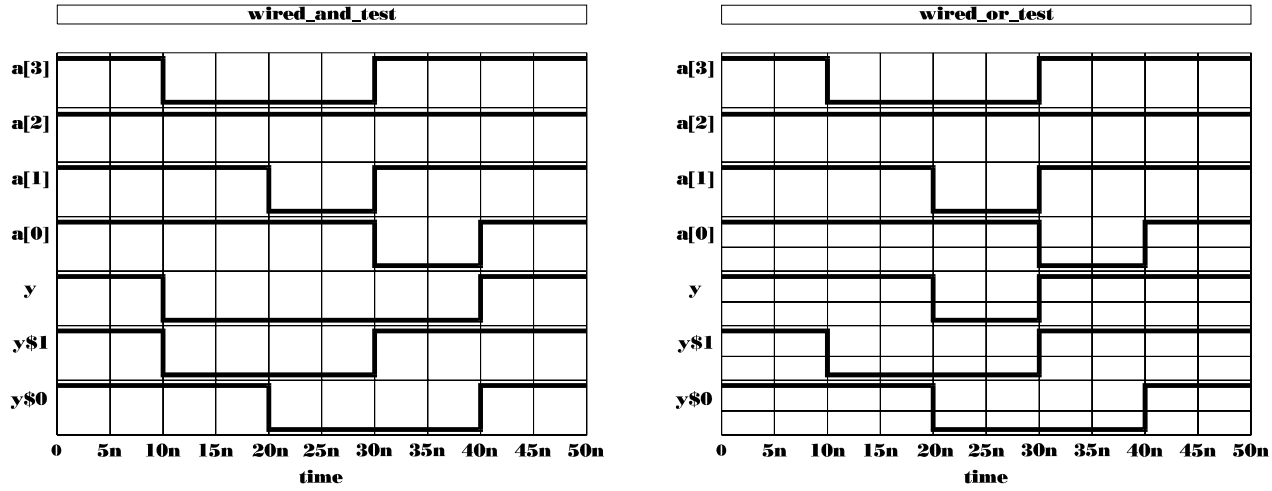
```
# include "ss2.h"
library gates2, op2;
root wired_and_test () {
  module sand g1, g2;                          // modules declaration
  signal two_t a[4]="1111";                    // input signals
```

```
  signal two_t:wired_and y='1';                  // output signal, wired logic

  g1 (a[0], a[1], y) { tr=0ns; tf=0ns; }         // zero delay AND gate
  g2 (a[2], a[3], y) { tr=0ns; tf=0ns; }         // zero delay AND gate

  timing { tstop = 50ns; }                       // simulation lasts 50ns
  out { signal two_t inout y, a; }               // output list
  action {
    process initial {                            // circuit stimulus
      a <-  "1110" after 10ns, "1010" after 20ns,
            "0111" after 30ns, "1111" after 40ns;
    }
  }
}
```



Slika 3.3 *Žičana logika*

Rezultat simulacije kola sa slike 3.3 dat je na slici 3.4a, a rezultat simulacije istog kola kada je umesto funkcije `wired_and` upotrebljena rezoluciona funkcija `wired_or` za izlazni signal `y` dat je na slici 3.4b. Promenljive označene sa `y$0` i `y$1` na slici 3.4 dobijene su zahvaljujući specificiranju usmerenja `inout` (može i usmerenje `out`) za vezu `y` u naredbi `out` i predstavljaju vrednosti drajvera signala `y` (stanja na izlazima dva I kola) na osnovu kojih rezoluciona funkcija odredjuje rezultujuće stanje ovog signala.



a) *b)*

Slika 3.4 *Rezultat simulacije žičane logike a) žičano I b) žičano ILI*

Može se uočiti da u prvom slučaju (slika 3.4a) na izlazu dominira stanje logičke nule kad se pojavi na makar jednom izlazu I kola, izlaz `y` je u stanju `'0'` od 10ns do 40ns, a u drugom slučaju (slika 3.4b) dominira stanje logičke jedinice tako da se stanje `'0'` javlja samo kada su oba drajvera u stanju logičke nule - od 20ns do 30ns.

3.5 Detekcija hazarda

Hazardnom se naziva situacija kada kao rezultat simultane promene vrednosti signala na više ulaza logičkog kola njegov izlaz proizvodi neželjeni kratkotrajni impuls ili visokofrekventnu oscilaciju. Ako je generisan kratkotrajni impuls (pik ili dip), pojava se naziva statički hazard, a ako je u pitanju oscilacija pri promeni stanja izlaza sa logičke nule na jedinicu ili obrnuto, pojava se naziva dinamički hazard. Ovakve pojave najčešće nemaju negativnih posledica po funkcionisanje kola, jer su vrlo kratkotrajne, pa ne raspolažu dovoljnom energijom da pobude kola na čijem se ulazu pojave. Algoritam kontrole vremena u digitalnoj simulaciji koji je ugradjen u Alecsis2.1 automatski potiskuje ovakve impulse. Dodavanjem još jednog procesa u model odgovarajućeg gejta, može se postići njihova detekcija i javljanje poruka korisniku. Navešćemo dva jednostavna oblika procesa za detekciju hazarda kod dvoulaznih gejtova.

```
hazard_detect: process (a, b) {
  if (a->event && b->event && a!=b) warning ("static hazard detected", 0);
}
```

Ovaj proces, medjutim, može da detektuje hazard samo u izuzetnim situacijama (recimo na globalnim ulazima kola). Naime, potrebno je da se poklope trenuci pojave dva dogadjaja na ulazima, što je malo verovatna pojava u simulatoru kod koga je simulaciono vreme realan broj (`double`), kao kod simulatora Alecsis2.1. Zapravo, hazard treba prijaviti ako se odgovarajuće promene na ulazima jave u dovoljno bliskim vremenskim trenucima, toliko bliskim da rezultat prvog dogadjaja ne stigne da se ispolji na izlazu kola, a drugi dogadjaj, koji ovaj rezultat poništava, javi se na drugom ulazu. Obično se za minimalno dozvoljeno vreme izmedju dve uzastopne promene na ulazima bira veća od dve vrednosti propagacionog kašnjenja (`tplh` i `tphl`).

```
hazard_detect: process (a, b) {
  double lasta=0.0, lastb=0.0;
  if (a->event) lasta = now;
  if (b->event) lastb = now;
  if (a!=b && fabs(lasta-lastb) <= MAX(tplh, tphl))
    warning ("static hazard detected", 0);
}
```

3.6 A/D i D/A konverzija

Pri simulaciji hibridnih kola povezivanje analognih i digitalnih delova kola zasniva se na konverziji signala i pravilnom sprezanju. Kao što je rečeno u odeljku 2.3.12, pod konverzijom signala podrazumeva se pretvaranje analognih signala u digitalne (obično poredjenjem sa vrednostima naponskih pragova izmedju stanja) i pretvaranje digitalnih signala u analogne (obično linearnom aproksimacijom promene vrednosti signala). Pravilno sprezanje podrazumeva umetanje modela ulaza odnosno izlaza digitalnog dela kola u analogni deo kola da bi se očuvala verodostojnost rezultata analogne simulacije [Mrča92].

Kao i drugi hibridni simulatori, Alecsis2.1 raspolaže mehanizmom za automatsko razdvajanje analognog od digitalnog dela kola ubacivanjem A/D i D/A konvertora. Da bi se održala otvorenost i fleksibilnost simulatora, korisniku je omogućeno da sam definiše ove konvertore u obliku klasičnih modula koji imaju jedan analogni i jedan digitalni čvor. Tako se algoritam konverzije, kao i model ulaza i izlaza mogu proizvoljno izabrati.

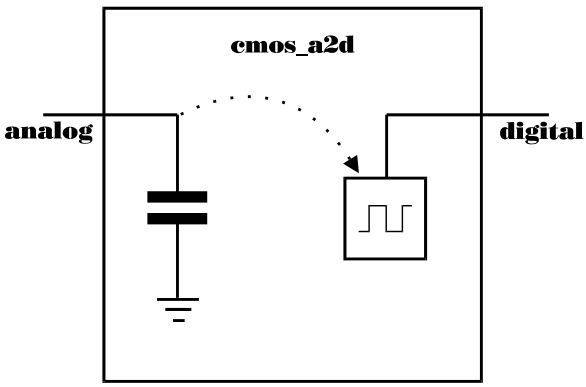
Već smo videli kako je definisana klasa `io2`, bazna klasa za sve gejtove biblioteke sa dva stanja. Ona sadrži parametre A/D i D/A konvertora za sve module kojima pripada modelska kartica njenih izvedenih klasa (poput klase `gm2`). Modeliran je jedan modul za A/D konverziju (`cmos_a2d`) i jedan modul za D/A konverziju (`cmos_d2a`). Njihove deklaracije smeštene su u datoteku **ss2.h**, a definicije u datoteku **gates2.hi**. Funkcije korišćene pri modeliranju D/A konverzije smeštene su u datoteku **op2.hi**. U konstruktoru klase `io2` postavljene su difoltne vrednosti parametara modela

konvertora. Pri definisanju modelske kartice moguće je promeniti ove vrednosti, a zbog mogućnosti greške definisan je i procesor klase `io2` koji proverava logičnost zadatih vrednosti. Procesor ima sledeći oblik.

```
io2::>io2() {
  if (rise_time<=0.0) warning ("incorrect D/A ramp rise time", 1);
  if (fall_time<=0.0) warning ("incorrect D/A ramp fall time", 1);
  if (Rout<=0.0) warning ("incorrect D/A output resistance", 1);
  if (zero_level==one_level) warning ("incorrect zero/one level", 1);
}
```

3.6.1 A/D konverzija

Struktura konvertora je najjednostavnija kado su u pitanju CMOS logička kola, jer se ulaz CMOS kola može modelirati samo parazitnom kapacitivnošću gejta koju ćemo smatrati konstantnom. Sprega analognog dela kola i ulaza digitalnog CMOS kola ostvaruje se tako što se u analogno kolo vezuje kapacitivnost gejta, a digitalno kolo se direktno pobudjuje digitalnim signalom nastalim konverzijom analognog napona, kao na slici 3.5. A/D konvertor ima jedan čvor tipa `node` i signal usmerenja `out` tipa `two_t`. Vrednost napona u analognom delu kola poredi se sa naponom praga (parametar `threshold`). Ako je napon manji od napona praga, digitalnom čvoru dodeljuje se stanje `'0'`, u suprotnom se dodeljuje stanje `'1'`.



Slika 3.5 *A/D konvertor za CMOS logička kola*

```
module io2::cmos_a2d (node analog; signal two_t out digital) {
  capacitor Capin;
  Capin (analog, 0);    // gate input capacitance
  action {
    assign: process post_structural { Capin->value = Cin; }
    conv: process post_moment {
      two_t new_event, last_event='0';    // logic low assumed initially

      if (analog >= threshold) new_event = '1';
      else new_event = '0';
      if (new_event != last_event) {
          last_event = new_event;
          digital <- new_event;           // schedule event
      }
    }
  }
}
```

U deklaracionom delu tela modula `cmos_a2d` deklarisan je kondenzator (ugradjena analogna komponenta sa oznakom `capacitor`) `Capin`, a u strukturnom delu povezan u kolo izmedju ulaznog čvora `analog` i mase. Čvor sa oznakom `0` predstavlja masu, kao i kod simulatora SPICE. Vrednost kapacitivnosti je zadata u okviru akcionog bloka u procesu sinhronizovanom implicitnim signalom `post_structural` (proces sa labelom `assign`). Ukoliko se ne zada vrednost kapacitivnosti,

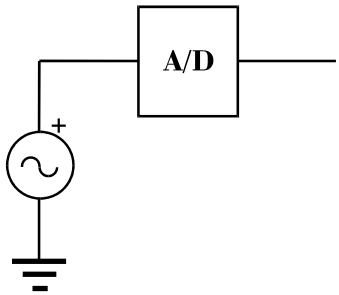
simulator podrazumeva vrednost `0.0F`. Na sličan način kondenzator `Capin` mogao je biti vezan u kolo u procesu sinhronizovanom signalom `structural` i korišćenjem naredbe `clone`. U procesu sa labelom `conv` ostvaruje se konverzija signala. Proces `conv` sinhronizovan je implicitnim signalom `post_moment`, dakle, aktivira se posle izračunavanja vrednosti promenljivih u analognom delu kola (medju ovim promenljivim nama je važna vrednost napona u čvoru `analog`). Promenljiva `last_event` čuva prethodnu vrednost konvertovanog napona u čvoru `analog` i ima početnu vrednost `'0'` (ova vrednost je izabrana, jer je to difoltna inicijalna vrednost za signale tipa `two_t`). Kad napon u čvoru `analog` prelazi nivo zadat parametrom `threshold`, stanje se menja, a novo stanje se šalje u drajver signala `digital` u istom trenutku (bez kašnjenja). Dakle, u istom vremenskom trenutku, u digitalnom delu kola pojavljuje se novi dogadjaj i digitalni simulator se aktivira.

Za svaki gejt koji je objekat modelske klase `gm2` može da se izabere konvertor koji će biti korišćen pri automatskoj konverziji tipa čvora (naredba `conversion`). Modelskom karticom gejta mogu se zadati konkretne vrednosti parametara modela konvertora: ulazna kapacitivnost `Cin` i napon praga izmedju logičke nule i jedinice `threshold`. Recimo, u sledećem modelu višeulaznog I kola (modul `andx`) definisana su vremena kašnjenja prednje i zadnje ivice na izlazu, ali i vrednost ulazne kapacitivnosti koja je parametar modela A/D konvertora.

```
model gm2::andx_model {
  Cin = 150fF;    // io2::Cin = 150fF is OK too
  tplh = 2.4ns;
  tphl = 2.9ns;
}
```

Konvertor `cmos_a2d` moguće je koristiti i kao poseban modul koji se ugradjuje u simulirano kolo nezavisno od mehanizma automatske konverzije i komande `conversion`. Pri tome mu odgovara modelska kartica tipa `io2`. Modelska kartica jednog konkretnog modula `cmos_a2d` ima sledeći izgled.

```
model io2::a2d_model_1 {
  Cin = 2pF;
  threshold = 2.75V;
}
```



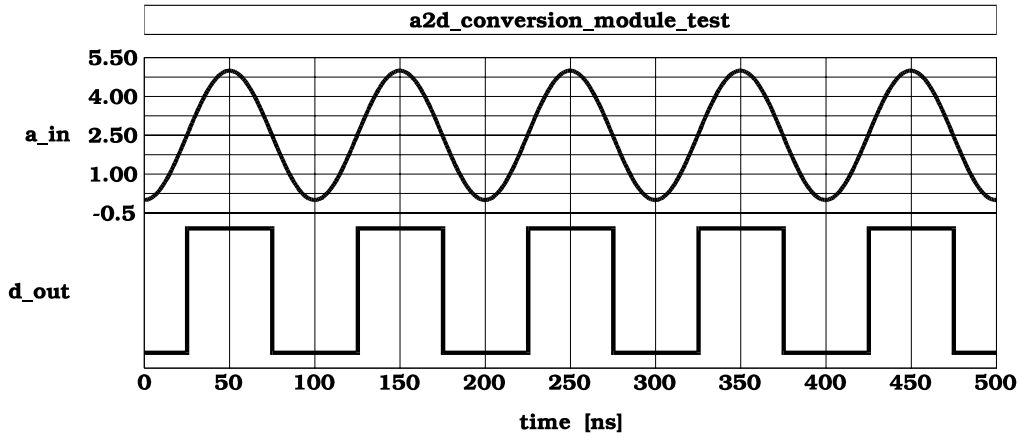
Slika 3.6 *Kolo za testiranje A/D konvertora*

Funkcionalnost modela A/D konvertora `cmos_a2d` proverićemo dovodjenjem sinusoide na njegov ulaz, kako to pokazuje slika 3.6. Opis kola sa slike 3.6 za simulaciju programom Alecsis2.1 je sledeći.

```
# include "ss2.h"
model default_a2d_model {}
library gates2, op2;
root a2d_conversion_module_test () {
  module io2::cmos_a2d adconv;
  vsin sg;                    // sinusoidal voltage generator
  signal two_t d_out;         // digital output
  node a_in;                  // analog input
```

```
  sg (a_in, 0) { amp=2.5v; freq=10MHz; phase=1.5*3.1415rad;
                                                dc_offset=2.5v; }
  adconv (a_in, d_out) model=default_a2d_model;

  timing { tstop = 500ns; a_step=0.01ns; }
  out { signal two_t d_out; node a_in; }
}
```

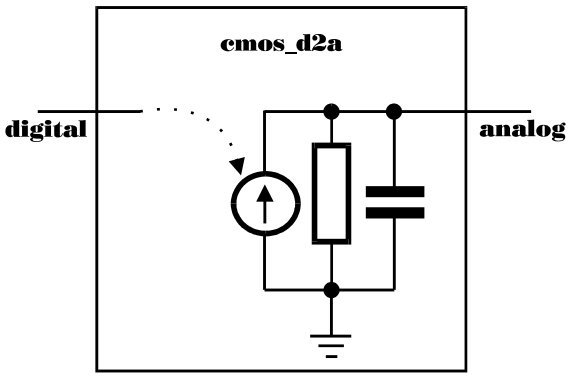
A/D konvertor je direktno ugradjen u kolo sa modelskom karticom `default_a2d_model` (ona sadrži parametre modela postavljene u konstruktoru klase `io2`). Sinusni naponski generator, ugradjena analogna komponenta sa oznakom `vsin`, ima jednosmerni ofset 2.5V, amplitudu 2.5V (tako da se napon kreće od 0V do 5V), frekvenciju 10MHz i početni fazni stav $3\pi/2$. Osim trajanja simulacije (parametar `tstop`), zadata je i vrednost vremenskog koraka za analognu simulaciju (parametar `a_step`). Ovo je hibridno kolo kome nije potrebno posebno definisati pobudu, jer pobudu predstavlja sinusni generator. Rezultat simulacije prikazan je na slici 3.7.



Slika 3.7 *Konverzija sinusidnog talasnog oblika u sistem sa dva stanja A/D konvertorom* `cmos_a2d`

3.6.2 D/A konverzija

D/A konverzija je nešto složeniji problem, jer zahteva izbor aproksimacije talasnog oblika promene vrednosti izlazne analogne veličine pri promeni vrednosti ulaznog digitalnog signala. Ako signal na izlazu digitalne komponente promeni stanje od logičke nule (recimo V(0)=0) na logičku jedinicu (recimo V(1)=5V), nije moguće ovaj dogadjaj uvesti u analogni deo kola kao trenutnu promenu napona sa 0V na 5V. Ovakav postupak onemogućio bi konvergenciju rešenja sistema jednačina kojim se opisuje analogni deo kola. Neophodno je ovu promenu napona na neki način "usporiti". Pri tome se postavlja pitanje kojim talasnim oblikom definisati tu promenu. Obično se bira jednostavna linearna promena (rampa), ali je takodje moguće iskoristiti eksponencijalnu ili neku drugu funkciju koja više odgovara realnom stanju u kolu.



Slika 3.8 *D/A konvertor za CMOS logička kola*

Jednostavni D/A konvertor za CMOS logička kola ima izlaznu otpornost, izlaznu kapacitivnost i kontrolisani strujni generator sa talasnim oblikom u obliku rampe upravljan digitalnim signalom sa ulaza, kao na slici 3.8. Korišćen je strujni umesto naponskog generatora na izlazu da bi se uštedeo dodatni čvor koji bi se inače pojavio izmedju naponskog generatora i izlazne otpornosti i tako uvećao složenost analognog dela kola. Konvertor se obično realizuje sa najmanje dva procesa, od kojih jedan registruje promene na digitalnom ulazu i definiše potrebna vremena i naponske nivoe, a drugi upravlja vrednostima analognog modela izlaza (u ovom slučaju samo vrednošću strujnog generatora). Vreme trajanja rampe ulazi u ukupno vreme kašnjenja gejta na čiji se izlaz konvertor dodaje, tako da je neophodno da se od kašnjenja gejta koji pobudjuje konvertor oduzme polovina trajanja rampe. To je jednostavno izvesti u okviru funkcije za kašnjenje gejta, jer se proverom atributa `hybrid` izlaznog čvora detektuje da li će se za njega koristiti konvertor (hibridni čvor) ili neće (digitalni čvor).

Kod kojim se opisuje D/A konvertor `cmos_d2a` u datoteci **gates2.hi** ima sledeći izgled.

```
module io2::cmos_d2a (signal two_t in digital; node analog) {
  capacitor Capout;
  resistor  Resout;
  cgen      Iout;                          // controlled current source

  Capout (analog, 0);
  Resout (analog, 0);
  Iout   (0, analog);

  action {
    static Status istat=Steady;
    static two_t old_state='0';           // guessed initial input state
    static two_t new_state='0';
    static Point start={0,0}, stop={0,0};

    assign: process post_structural {     // assign component values
      *Capout = Cout;
      *Resout = Rout;
    }

    monitor: process (digital) {          // monitor digital node
      // when an event occurs
      old_state = new_state;
      new_state = digital;
      set_breakpoints (now, old_state, new_state, &start, &stop, &istat);
    }

    control: process per_moment {         // control analog devices
      double level;
      level = Iout->value * this.Rout;
      if (istat != Steady) {
        if (fabs(level - stop.level) <= LevelTol) istat = Steady;
        else {                            // still in transition
          level = evaluate_vramp (now, &start, &stop);
          Iout->value = level / this.Rout;
        }
      }
    }
  }
}
```

U deklaracijskom delu modula `cmos_d2a` deklarisane su tri analogne komponente modela D/A konvertora sa slike 3.8: otpornik `Resout`, kondenzator `Capout` i kontrolisani strujni generator `Iout`. Rezervisane reči `resistor`, `capacitor` i `cgen` označavaju ove ugradjene analogne komponente kod simulatora Alecsis2.1. U strukturnom delu modula ove komponente su vezane u kolo izmedju čvora `analog` i mase (čvor `0`). Redosled navodjenja čvorova kod otpornosti i

kapacitivnosti nije bitan. Po definiciji smer struje kod komponente `cgen` je od prvog navedenog čvora prema drugom navedenom čvoru. Vrednosti izlazne otpornosti i kapacitivnosti su parametri modela konvertora, pa su postavljene u procesu `assign`. Vrednost kontrolisanog strujnog generatora zavisi od stanja na ulaznom signalu `digital` i zadaje se u svakom trenutku pre rešavanja sistema jednačina za analogni deo kola. Zato je proces `control`, u kome se ova vrednost postavlja, sinhronizovan implicitnim signalom `per_moment`. Proces sa labelom `monitor` detektuje svaku promenu vrednosti ulaznog signala `digital` i postavlja vremenske i naponske koordinate početne (promenljiva `start`) i krajnje tačke (promenljiva `stop`) rampe koju je potrebno izgraditi u izlaznom čvoru `analog` (ovo je zadatak funkcije `set_breakpoints`). Na osnovu ovih vrednosti i trenutne vrednosti simulacionog vremena u procesu `control` izračunava se vrednost strujnog generatora `Iout` (funkcija `evaluate_vramp`). Promenljiva `istat` pamti trenutni status izlazne struje, da li ona raste (`Rising`), opada (`Falling`) ili ne menja vrednost (`Steady`). Tipovi podataka `Status` i `Point` definisani su u datoteci **ss2.h** zajedno sa klasom `io2`. Kad se rampa približi ciljnom naponskom nivou na `LevelTol` (parametar modela konvertora), usvaja se ciljna vrednost i struja uzima status `Steady` do sledeće promene na ulazu `digital`. Korišćena funkcija `fabs` odredjuje apsolutnu vrednost realnog broja.

Funkcije `set_breakpoints` i `evaluate_vramp` definisane su u datoteci **op2.hi** i imaju sledeći oblik.

```
void io2::set_breakpoints (double time, two_t old_s, two_t new_s,
                           Point *start, Point *stop, Status *istat) {
                    // Determine output voltage status:
  if (new_s == '1') *istat = Rising;
  else              *istat = Falling;
                    // Set breakpoints:
  start->level = ((old_s=='0') ? zero_level : one_level);
  stop->level  = ((new_s=='0') ? zero_level : one_level);
  start->time  = time;
  stop->time   = time + ((*istat==Rising) ? rise_time : fall_time);
}

double io2::evaluate_vramp (double time, Point *start, Point *stop) {
  double slope = (stop->level - start->level)/(stop->time - start->time);
  double new_level = start->level + slope * (time - start->time);
  if (slope < 0.0) new_level = MAX(new_level, stop->level);
  else new_level = MIN(new_level, stop->level);
  return new_level;
}
```

Makroi MIN i MAX, kao i funkcija `fabs` postoje u standardnoj deklaracijskoj datoteci **alec.h**, pa je potrebno priključiti ovu datoteku da bi bili vidljivi kompajleru.

```
# include <alec.h>
```

3.7 Pobudni generatori

Pri logičkoj simulaciji sekvencijalnih kola vrlo često su nam potrebni taktni generatori periodičnog oblika, a promenljive frekvencije i odnosa trajanja nula/jedinica. Iz tog razloga se u logičkoj biblioteci definišu ovakvi moduli. Generatori taktnih impulsa imaju samo izlaze, a parametri talasnog oblika zadaju se ili modelskom karticom ili preko akcionih parametara. Takt generatori mogu biti objekti nove klase izvedene iz klase `io2`.

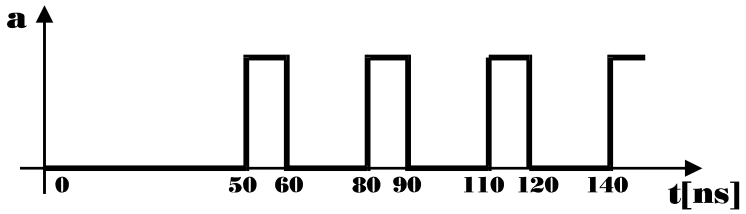
```
class gen : public io2 {
      double width1, width0, startdelay;
  public:
      gen();
      >gen();
};
```

Ovakav način realizacije, medjutim, nepotrebno komplikuje modele taktnih generatora. Drugi, jednostavniji način, koji ćemo ovde ilustrovati, je upotreba akcionih parametara.

```
module gm2::clkgen (two_t inout y) {
  conversion { a2d = "cmos_a2d"; d2a = "cmos_d2a"; }
  action (double width1, double width0, double startdelay=0.0) {
    process {
      double delay;
      two_t newy;
      int init = 1;

      newy = ~y;
      if (init && startdelay) {
        y <- newy after startdelay;
        init = 0;
      }
      else {
        delay = (y=='1') ? width1 : width0;
        y <- newy after delay;
      }
      wait y;
    }
  }
}
```

Akcioni parametar `width1` predstavlja trajanje stanja logičke jedinice, a `width0` trajanje logičke nule, dok je parametar `startdelay` trajanje početnog perioda neaktivnosti generatora. Početno stanje izlaza generatora definiše se spolja, zadavanjem inicijalnog stanja signala za koji je generator vezan. Na primer, talasni oblik kao na slici 3.9 može se dobiti upotrebom takt generatora `clkgen` na sledeći način.

```
clkgen clockgenerator;
signal two_t a='0';                  // set initial state for generator
clockgenerator (a) action (10ns, 20ns, 50ns);
```



Slika 3.9 *Talasni oblik na izlazu takt generatora*

Taktni generator `clkgen` definisan je kao objekat klase `gm2` da bi se omogućilo automatsko ubacivanje D/A konvertora ukoliko je za njegov izlaz vezana neka analogna komponenta. Inače, on ne koristi modelske parametre klase `gm2` `tplh` i `tphl`. Ako konverzija signala nije potrebna, može se pobudni generator definisati i kao nezavisna komponenta, koja ne prima modelsku karticu. U sledećem primeru definisan je pobudni generator sa tri izlaza koji na izlazima generiše sve pobudne vektore.

```
module gen3 (two_t out y[]) {
  action (double delay) {
    process initial {
      y <- "000" after delay,
           "001" after 2 * delay,
           "010" after 3 * delay,
           "011" after 4 * delay,
           "100" after 5 * delay,
```

```
            "101" after 6 * delay,
            "110" after 7 * delay,
            "111" after 8 * delay;
    }
  }
}
```

Generator `gen3` vezuje se u simuliranom kolu za vektorski signal tipa `two_t` dužine 3. Zadaje mu se akcioni parametar `delay` - kašnjenje izmedju svaka dva pobudna vektora.

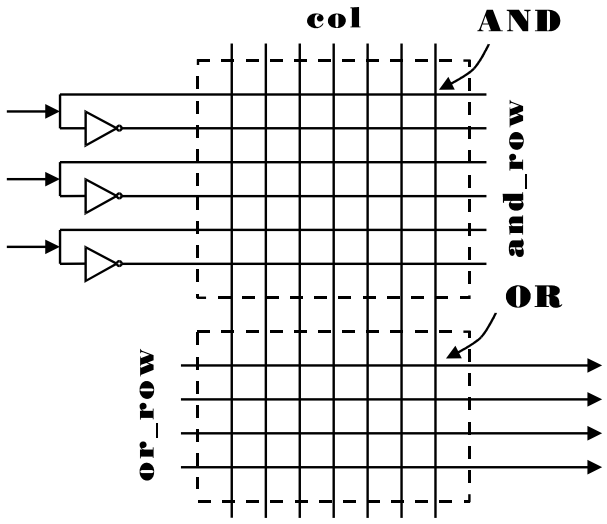
```
module gen3 g;
signal two_t global_inputs[3];
g (global_inputs) action (10ns);
```

Ovakav generator može biti vrlo koristan pri testiranju funkcije projektovanog troulaznog kola. Pogodno je u simulacionoj biblioteci definisati slične generatore sa različitim brojem izlaznih terminala, jer pobudjivanje kola sa više ulaza svim pobudnim vektorima u procesu sihnronizovanom signalom `initial` pri samoj simulaciji (u `root` modulu) može biti zamoran posao, posebno u sistemima sa više od dva logička stanja.

3.8 Modeliranje programabilnih logičkih struktura

Biblioteka za sistem sa dva stanja sadrži i modele programabilnih logičkih polja (PLA). Da bi se implementirao model PLA uvedena je nova modelska klasa `pla2` koja sadrži parametre kašnjenja, dimenzije polja i ime datoteke iz koje se učitava konfiguracija osigurača u polju. Ova klasa definisana je u datoteci **ss2.h** na sledeći način.

```
class pla2 : public io2 {
      double tp;                    // pla propagation time
      double tplh, tphl;            // output buffers propagation times
      char *filename;               // switch matrices setup file
      int and_row, or_row, col;     // dimensions
  public:
      pla2();
      ~pla2();
      friend module spla;           // simple time model pla
      friend module pla;            // tplh/tphl time model pla
};
```



Slika 3.10 *Programabilno logičko polje (PLA)*

Konstruktor klase `pla2` definisan je u funkcijskoj datoteci **op2.hi** i u njemu se alocira memotrijski prostor za promenljivu `filename` i zadaju difoltne vrednosti parametara kašnjenja. Difoltne vrednosti parametara koji odredjuju dimenzije polja nema smisla zadavati. Ove dimenzije će zadati korisnik kad opisuje neku konkretnu PLA komponentu programiranu da obavlja odredjenu funkciju. Slika 3.10 prikazuje uopštenu PLA strukturu koja se sastoji od dve matrice osigurača, I matrice i ILI matrice. Parametar `and_row` predstavlja broj vrsta u I matrici. Taj broj mora biti dva puta veći od dimenzija vektora ulaznih signala `inputs`, jer se na polje vode ulazni signali i njihovi komplementi. Komplementiranje ulaza se obavlja unutar modela komponente. Parametar `or_row` predstavlja broj izlaznih linija polja, odnosno broj vrsta u ILI matrici. Parametar `col` predstavlja broj kolona u I, odnosno ILI matrici. Kako se na ovim linijama generišu potrebni proizvodi (produkti) ulaznih promenljivih, zvaćemo ih i produktnim linijama. Parametar `filaname` predstavlja ime datoteke u kojoj je zadat raspored osigurača u matricama I i ILI (takozvana *setup* datoteka). Ova datoteka se posebno generiše za svaki pojedinačni PLA modul i opisuje njegovo programiranje.

```
# define FILENAME_LENGTH      32
pla2::pla2 () {
  if (!(filename = (char *)calloc(FILENAME_LENGTH, sizeof(char *))))
    warning("no room for filename in pla2", 1);
  tp = 0.0;
  tplh = 0.0;
  tphl = 0.0;
}
```

Kao što vidimo, ime *setup* datoteke sme biti dugo do 32 karaktera (makro `FILENAME_LENGTH`), a kašnjenja su postavljena na vrednost 0.0.

Destruktor klase `pla2` oslobadja prostor koji je zauzimala promenljiva `filename`. Pri tome se koristi standardni C++ operator `delete`.

```
pla2::~pla2 () { delete filename; }
```

Modelska kartica jednog konkretnog PLA modula može da ima sledeći izgled.

```
model pla2::f1_pla_model {
  and_row=32;                  // 16 input variables
  or_row=10;                   // 10 output functions
  col=50;                      // 50 columns in AND and OR array
  filename="f1.pla";           // setup file
  tp = 5ns;                    // PLA propagation time
}
```

Strukture neophodne za modeliranje funkcije PLA strukture smeštene su u posebnu klasu po imenu `pld2`, definisanu u datoteci **ss2.h** na sledeći način.

```
class pld2 {
      int is_set;                          // ==1 if pld already initialized
      int and_rows, or_rows, cols;         // and, or tables dimensions
      Bool *input_flag;                    // floating product line flag
      Bool *output_flag;                   // floating sum line flag
      bit **and_matrix, **or_matrix;       // and, or matrices structure
      two_t *input_line;                   // temporraries
      two_t *output_line;
      two_t *product_line;
  public:
      pld2(int and_row, int or_row, int col, char *filename);
      ~pld2();
      int set(char *filename);             // set pld from file
      void set_flags();                    // set intput and output flags
      void show();                         // show pld structure
      two_t *produce_output(two_t *);      // the very pld function
};
```

Konstruktor klase `pld2` definisan je u datoteci **op2.hi** i ima sledeći izgled.

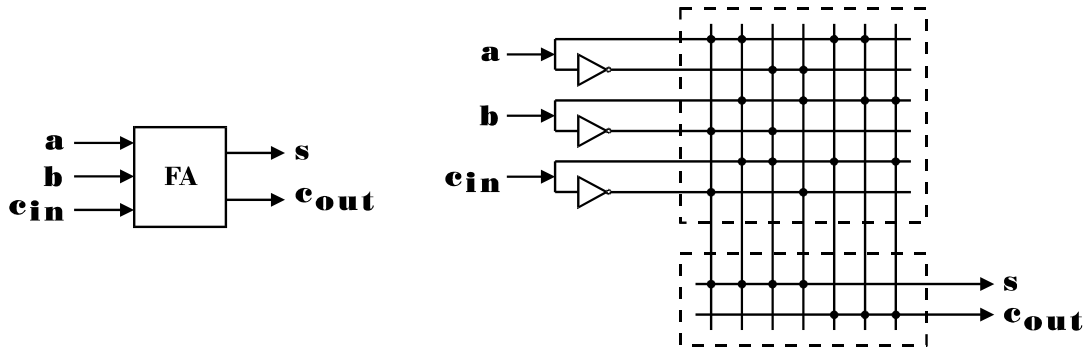
```
pld2::pld2 (int and_row, int or_row, int col, char *filename) {
  int i;

  is_set=0;                          // not set

  if ((and_rows=and_row)<=0) warning("incorrect and_rows in pld2", 1);
  if ((or_rows=or_row)<=0) warning("incorrect or_rows in pld2", 1);
  if ((cols=col)<=0) warning("incorrect cols in pld2", 1);

  if (!(and_matrix = (bit **)calloc(and_rows, sizeof(bit*))))
    warning("no room for and_matrix in pld2", 1);
  if (!(or_matrix = (bit **)calloc(or_rows, sizeof(bit*))))
    warning("no room for or_matrix in pld2", 1);

  for (i=0; i < and_rows; i++) {
    if (!(and_matrix[i] = new bit [cols]))
      warning("no room for and_matrix in pld2", 1);
  }
  for (i=0; i< or_rows; i++) {
    if (!(or_matrix[i]= new bit [cols]))
      warning("no room for or_matrix in pld2", 1);
  }

  if (!(input_flag = new Bool [cols]))
    warning("no room for input_flag in pld2", 1);
  if (!(output_flag = new Bool[or_rows]))
    warning("no room for output_flag in pld2", 1);

  if (!(input_line  = new two_t [and_rows]))
    warning("no room for input_line in pld2", 1);
  if (!(product_line = new two_t [cols]))
    warning("no room for product_line in pld2", 1);
  if (!(output_line = new two_t [or_rows]))
    warning("no room for output_line in pld2", 1);

  if (!set(filename))  warning("errors during pld2 initialization", 1);
  set_flags();
}
```

Odgovarajući destruktor `~pld2` koji oslobadja rezervisanu memoriju korišćenjem operatora `delete` ima sledeći izgled.

```
pld2::~pld2 () {
  int i;
  for (i=0; i<and_rows; i++) delete and_matrix[i];
  for (i=0; i<or_rows; i++) delete or_matrix[i];
  delete and_matrix;
  delete or_matrix;
  delete input_flag;
  delete output_flag;
  delete input_line;
  delete product_line;
  delete output_line;
}
```

Treba uočiti da konstruktor `pld2` posle alociranja memorije poziva funkciju `set` predajući joj ime *setup* datoteke. Ova funkcija može da se realizuje na različite načine, zavisno od izabranog formata u kome će podaci biti zadati. Njen zadatak je da otvori datoteku sa imenom `filename`, pročita iz nje raspored osigurača u matricama, upiše ovu informaciju u matrice `and_matrix` i `or_matrix` i zatvori datoteku `filename`. Pri učitavanju potrebno je izvršiti niz provera kako bi se obezbedila korektnost učitanih podataka. U našoj biblioteci usvojeno je da se matrice zadaju tako

da nula (`0`) označava da na odredjenom mestu u matrici nema kontakta (izgoreo osigurač), a jedinica (`1`) označava da na odredjenom mestu postoji kontakt (osigurač nije pregoreo). Zato su promenljive `and_matrix` i `or_matrix` dinamički alocirane matrice koje sadrže podatke tipa `bit` (tip `bit` je u jeziku AleC++ sistemski definisan i sadrži konstante `'0'` i `'1'`). Funkcija `set` je realizovana tako da ostavlja korisniku simulatora izvesne slobode u izgledu *setup* datoteke. Dozvoljeni su prazni redovi, prazna mesta izmedju znakova i linijski komentari koji počinju zvezdicom (`*`) i traju do kraja linije. Zbog njene dužine nećemo je ovde navoditi.



(a) (b)

Slika 3.11 *a) Blok šema potpunog sabirača b) Realizacija potpunog sabirača PLA strukturom*

Kao primer *setup* datoteke navešćemo datoteku **fa.pla** u kojoj je opisan raspored osigurača u PLA komponenti dimenzija 6x2x7 programiranoj da realizuje funkciju potpunog sabirača. Potpuni sabirač ima tri ulaza: a, b, i c_{in} i dva izlaza: s i c_{out}. Funkcija potpunog sabirača opisana je na sledeći način:

$$s = a \oplus b \oplus c_{in} = a\bar{b}\bar{c}_{in} + abc_{in} + \bar{a}\bar{b}c_{in} + \bar{a}b\bar{c}_{in}$$

$$c_{out} = ac_{in} + bc_{in} + ab$$

Blok šema potpunog sabirača i raspored osigurača za PLA realizaciju prikazan je na slici 3.11. Datoteka **fa.pla** ima sledeći izgled.

```
* PLA : setup file
* Function : full adder
* Date : Sun Mar 27 16:56:30 MDT 1995

* AND matrix :

      1100110           * a
      0011000           * ~a
      0101011           * b
      1010000           * ~b
      0110101           * cin
      1001000           * ~cin

* OR matrix :

      1111000           * sum
      0000111           * cout
```

Usvojena je konvencija da *setup* datoteke imaju ekstenziju **pla**, što nije obavezno.

Konstruktor `pld2` takodje poziva i funkciju `set_flags` koja pretražuje matrice `and_matrix` i `or_matrix` i proverava da li postoje kolone I matrice koje "vise", odnosno nisu povezane ni sa jednom ulaznom linijom, kao i vrste ILI matrice koje nisu povezane ni sa jednom produktnom linijom. Posle izvršenja ove funkcije vektor `input_flag` (parametar modelske klase `pld2`) sadrži logičku vrednost `false` u onim produktnim linijama koje nisu povezane sa ulazima, a vektor `output_flag` sadrži logičku vrednost `false` u onim izlaznim linijama koje nisu povezane sa

produktnim linijama. Vrednost `true` označava da makar jedna veza (nepregoreo osigurač) postoji. Ove informacije značajne su kasnije pri odredjivanju logičke funkcije polja.

Na bazi ovako definisanih struktura razvijen je model progamabilnog logičkog polja proizvoljnih dimenzija sa dva različita modela kašnjenja. Deklaracije odgovarajućih modula imaju sledeći oblik.

```
module pla2::spla (two_t in inputs[]; two_t out outputs[]);
module pla2::pla  (two_t in inputs[]; two_t out outputs[]);
```

Moduli `spla` i `pla` su objekti modelske klase `pla2`, te im sleduje odgovarajuća modelska kartica sa neophodnim parametrima. Modul `spla` ima jednostavan model kašnjenja kod koga je kašnjenje prednje ivice jednako kašnjenju zadnje ivice (paramerat `tp`). Da bi se implementirao rise/fall model kašnjenja, kod modula `pla` uvedeni su baferi na izlaznim priključcima logičkog polja. U datoteci **gates2.hi** definisani su PLA moduli sledećim kodom.

```
// Simple asignable delay model tplh=tphl=tp. Note: tplh/tphl not used.
module pla2::spla (signal two_t in inputs[]; signal two_t out outputs[]) {
  action {
    process (inputs) {
                              // construct pla:
      pld2 p(this.and_row, this.or_row, this.col, this.filename);
                              // execute function:
      outputs <- p.produce_output(inputs) after this.tp;
    }
  }
}

// Rise/fall delay model. PLA propagation time tp can be ommitted from
// model card (default is tp=0s).
module pla2::pla (two_t in inputs[]; two_t out outputs[]) {
  module sbuf outbuf;               // output buffers
  signal two_t pla_out[auto];       // internal signals, buffers' inputs

  action {
    process structural {            // introduce rise/fall delay model
      signal two_t pla_out(this.or_row);

      for (int i=0; i < this.or_row; i++) {
        clone outbuf[i] (pla_out[i], outputs[i]) action (this.tplh, this.tphl);
      }
    }

    spla_model : process (inputs) {
                                          // construct pla:
      pld2 p(this.and_row, this.or_row, this.col, this.filename);
                                          // execute function:
      pla_out <- p.produce_output(inputs) after this.tp;
    }
  }
}
```

Modul `spla` sadrži samo jedan proces osetljiv na promenu stanja bilo kog ulaznog signala (ulazni signali su elementi vektora `inputs`), a proces sadrži samo dve naredbe. Prva je deklaracija objekta `p` tipa `pld2`. Deklaracija izaziva automatsko pozivanje konstruktora klase `pld2`. Parametri koji se prosledjuju konstruktoru uvedeni su u modul preko modelske kartice modula (tipa `pla2`). Pri pozivu konstruktora učitava se i kontroliše konfiguracija matrice osigurača. Drugom naredbom definišu se nova stanja na izlazima PLA strukture pomoću funkcije `produce_output` i prosledjuju se na listu budućih dogadjaja, sa kašnjenjem `tp`. Ovo je, očigledno, prilično jednostavan model kašnjenja. On je morao biti upotrebljen, jer AleC++ ne dozvoljava razdvajanje vektora signala usmerenja `out` pri upotrebi operatora `<-` na pojedinačne skalarne signale. Signal `outputs` mora se

pojaviti kao celina sa leve strane operatora `<-`, tako da nije moguće razdvojiti prednje od zadnjih ivica na pojedinim signalima koji su elementi ovog vektora.

Ovaj problem je razrešen kod modula `pla` time što su na izlaze PLA strukture vezani baferi (bafer je komponenta koja prosledjuje na izlaz stanje sa ulaza posle odredjenog kašnjenja), moduli tipa `sbuf` kojima se preko akcionih parametara prosledjuju kašnjenja prednje i zadnje ivice. Kako PLA struktura (objekat `p` tipa `pld2`) ima promenljiv broj izlaznih linija (parametar `or_row` u modelskoj kartici tipa `pla2`), moralo se upotrebiti dinamičko strukturisanje kola (komanda `clone`). Modul `pla` sadrži proces sinhronisan implicitnim signalom `structural` u kome se korišćenjem `for` petlje dodaje `or_row` bafera na izlaze PLA strukture. Kod ovog modula drugi proces je isti kao kod modula `spla`, mada je kašnjenje same PLA strukture `tp` moglo da se izostavi. Ako je zadat samo rise/fall model kašnjenja celog PLA modula, onda je potrebno parametar `tp` postaviti na nulu, što je automatski postignuto njegovim izostavljanjem pri definiciji modelske kartice, jer mu je difoltna vrednost nula.

Funkcija `produce_output` kojom se odredjuju stanja na izlazu PLA strukture na osnovu stanja na ulazima definisana je u funkcijskoj datoteci **op2.hi** i ima sledeći izgled.

```
two_t * pld2::produce_output (two_t *inputs) {
  int i;                        // input   lines index
  int p;                        // product lines index
  int o;                        // output  lines index

  for (i=0; i < and_rows/2; i++) {  // decode pld2 inputs
    input_line[2*i  ] =  inputs[i];
    input_line[2*i+1] = ~inputs[i];
  }
                                // AND operation :
  for (p=0; p < cols; p++) product_line[p]='1';
  for (p=0; p < cols; p++) {
    if (input_flag[p] == false)
      continue;                 // disconnected product line keeps state '1'
    else {                      // product connected to some inputs
      for (i=0; i < and_rows; i++) {
      if (and_matrix[i][p] == '1')
        product_line[p] = product_line[p] & input_line[i];
      }
    }
  }
                                      // OR operation :
  for (o=0; o < or_rows; o++) output_line[o]='0';
  for (o=0; o < or_rows; o++) {
    if (output_flag[o] == false)
      continue;                 // disconnected sum line keeps state '0'
    else {                      // output connected to products
      for (p=0; p < cols; p++) {
        if (or_matrix[o][p] == '1')
          output_line[o] = output_line[o] | product_line[p];
      }
    }
  }
  return output_line;
}
```

Funkcija `produce_output` prihvata stanja na ulazima PLA strukture i konvertuje ih u stanja ulaznih linija (svaki ulazni signal generiše dve linije - jednu sa istim, a drugu sa komplementiranim stanjem). Zatim se na osnovu stanja na ulaznim linijama odredjuju parcijalni proizvodi - stanja produktnih linija (logička I operacija) i na kraju se sumiraju odgovarajući parcijalni proizvodi (logička ILI operacija), to jest odredjuju se stanja izlaza.

U funkciji `produce_output` podrazumeva se da je produktnoj liniji koja nije povezana ni sa jednom ulaznom linijom dodeljeno stanje logičke jedinice, kao nedominantno za I logičku operaciju. Slično ovome, "visećoj" izlaznoj liniji dodeljeno je stanje logičke nule. Ovo, medjutim,

može lako da se modifikuje uvodjenjem dve globalne promenljive tipa `two_t` koje bi korisnik mogao da postavi na željene vrednosti, recimo u inicijalnom procesu `root` modula.

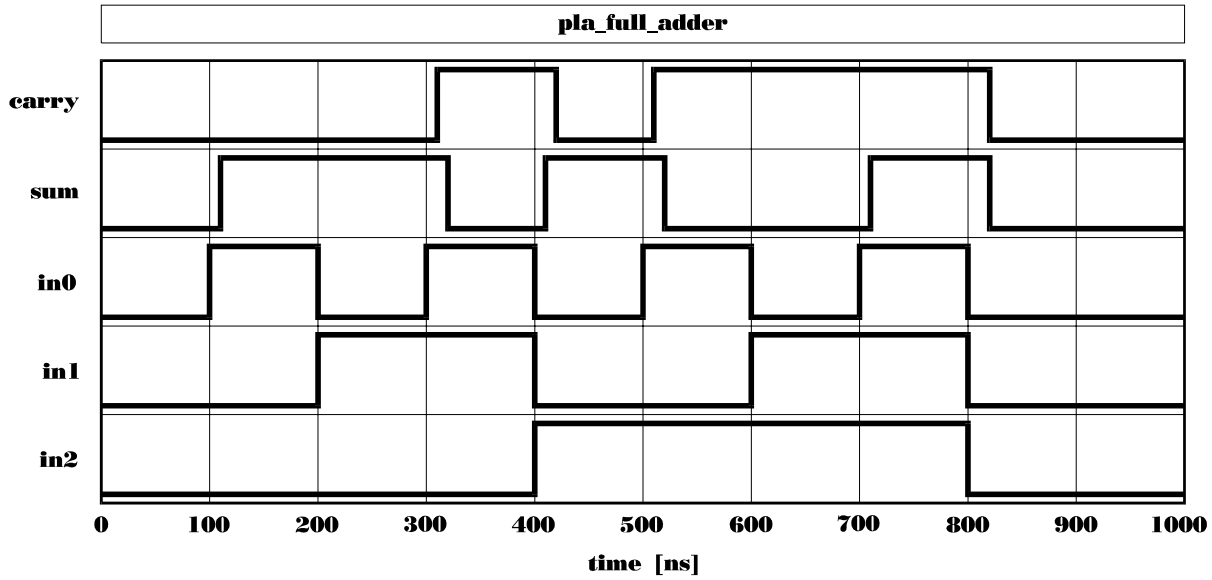
Kao ilustracija upotrebe modula `pla` neka posluži primer simulacije PLA strukturom realizovane funkcije potpunog sabirača o kojoj je već bilo reči. Upotrebljeno programabilno logičko polje ima kašnjenje prednje ivice 10ns i kašnjenje zadnje ivice 20ns, kao i dimenzije 6x2x7, što je definisano modelskom karticom po imenu `fa_pla_model u` modelskoj datoteci **model2.hi**. Sadržaj datoteke **fa.pla** definiše konfiguraciju osigurača u I i ILI matrici i već je ranije naveden.

```
model pla2::fa_pla_model {
  and_row=6;
  or_row=2;
  col=7;
  filename="fa.pla";
  tp = 0ns;
  tplh = 10ns;
  tphl = 20ns;
}
```

Opis simuliranog kola ima sledeći izgled.

```
# include "ss2.h"
library model2, gates2, op2;

root module pla_full_adder () {
  module pla2::pla pla_chip;
  signal two_t x[3]="000", y[2]="01";

  pla_chip (x, y) { model = fa_pla_model; };

  timing { tstop = 1000ns; }

  out {
    signal two_t in2 { return x[0]; };
    signal two_t in1 { return x[1]; };
    signal two_t in0 { return x[2]; };
    signal two_t sum { return y[0]; };
    signal two_t carry { return y[1]; };
  }

  action {
    process initial {
      x <- "001" after 100ns,
           "010" after 200ns,
           "011" after 300ns,
           "100" after 400ns,
           "101" after 500ns,
           "110" after 600ns,
           "111" after 700ns,
           "000" after 800ns;
    }
  }
}
```

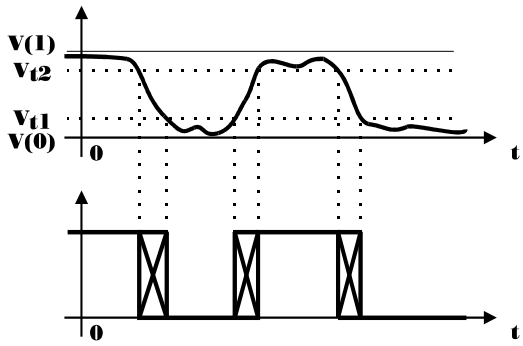
Rezultati simulacije prikazani su na slici 3.12. Jasno se uočava funkcija potpunog sabirača, kao i funkcionisanje rise/fall modela kašnjenja. Na primer, u trenutku 300ns na ulaze `in0`, `in1` i `in2` dovodi se pubudni vektor `"110"`. Izlaz `sum` reaguje prelaskom u stanje `'0'` posle kašnjenja `tphl`=20ns, a izlaz `carry` prelazi u stanje `'1'` posle `tplh`=10ns.



Slika 3.12 *Rezultati simulacije potpunog sabirača realizovanog pomoću PLA strukture*

4 Biblioteka za sistem sa tri stanja

Sistem sa tri stanja osim logičke nule i jedinice sadrži još jedno stanje koje obuhvata kako neodredjeno stanje (napon izmedju pragova logičke nule i jedinice, "plivajuće stanje"), tako i nepoznato (ali odredjeno) stanje (recimo pri inicijalizaciji kada ne znamo početna stanja u kolu). Na slici 3 prikazana je konverzija napona u sistem sa tri logička stanja poredjenjem sa vrednostima dva praga: V_{t1} i V_{t2}.



Slika 4.1 *Konverzija analogne veličine u sistem sa tri stanja*

Biblioteka je organizovana u četiri datoteke:

ss3.h	- deklaraciona datoteka
op3.h	- funkcijska datoteka
gates3.hi	- modulska datoteka
model3.hi	- biblioteka modela

4.1 Definisanje sitema stanja i preopterećenje operatora

Sistem sa tri stanja definisan je u datoteci **ss3.h** na sledeći način.

```
typedef enum { 'X', 'x'='X', '0', '1', '_'=void, ' '=void } three_t;
```

Neodredjenom stanju `'X'` odgovara celobrojni indeks 0, a definisano je i stanje `'x'` kome odgovara taj isti indeks, kao i separatori `'_'` i `' '`. Stanje `'X'` je namerno izabrano kao prvo u

definiciji enumerisanog tipa `three_t` da bi se sve promenljive ovog tipa njime automatski inicijalizirale. Sa ovako definisanim sistemom stanja korektne su deklaracije

```
three_t a='x', b='0', c, vec[8]="01 x1 1X X0", tab[2][2]={{'1', '0'},
                                                         {'x', '1'} };
```

ali ne i deklaracija:

```
three_t d[]="001x";     // vector length must be defined!
```

Dužina vektora enumerisanog tipa ne može se automatski odrediti pri inicijalizaciji stringom, ali je zato moguće izvršiti samo delimičnu inicijalizaciju deklaracijom

```
three_t d[4]="001";     // d[0]='0', d[1]='0', d[2]='1', d[3]='x'
```

pri čemu sve neinicijalizovane komponente vektora dobijaju automatski difoltnu vrednost (prvu vrednost u sistemu stanja).

Kao i kod sistema sa dva stanja, korisno je definisati vektor sa imenima stanja. Njegova deklaracija nalazi se u datoteci **ss3.h**, a definicija u datoteci **op3.hi**.

```
extern const char *names3[];                    // declaration
const char *names3[] = { "X", "0", "1" };       // definition
```

Predefinisaćemo osnovne logičke operatore za rad u ovom sistemu stanja. Odgovarajuće deklaracije u datoteci **ss3.h** imaju sledeći izgled.

```
extern three_t operator ~  (three_t op);
extern three_t operator &  (three_t op1, three_t op2);
extern three_t operator |  (three_t op1, three_t op2);
extern three_t operator ~& (three_t op1, three_t op2);
extern three_t operator ~| (three_t op1, three_t op2);
extern three_t operator ^  (three_t op1, three_t op2);
extern three_t operator ~^ (three_t op1, three_t op2);
```

Operator ~^ (ekskluzivno NILI, ekvivalencija) definisan je u funkcijskoj datoteci **op3.h** na sledeći način.

```
three_t const nxor_tab[][3]= { 'x','x','x',
                               'x','1','0',
                               'x','0','1'  };
three_t operator ~^ (three_t op1, three_t op2)  { return nxor_tab[op1][op2]; }
```

4.2 Parazitne kapacitivnosti - atributi signala

Usložnićemo model u odnosu na onaj korišćen u biblioteci sa dva stanja na taj način što ćemo uzeti u razmatranje uticaj parazitnih kapacitivnosti u kolu na kašnjenja logičkih komponenti. Naime, poznato je da se kašnjenje komponente povećava sa kapacitivnim opterećenjem izlaza. Obično se koristi linearna aproksimacija promene propagacionog kašnjenja sa kapacitivnim opterećenjem [Lito91b], pa ćemo i mi koristiti ovaj model. U takvom modelu svaki logički element karakteriše se ulaznim i izlaznim kapacitivnostima i nagibom prave porasta kašnjenja sa opterećenjem izlaza. Potrebno je, dakle, ugraditi dve novine u već korišćeni model: u modelsku karticu gejta treba uvrstiti neophodne parametre (kapacitivnosti i nagib) i pronaći način da se gejtu dostavi informacija o ukupnoj parazitnoj kapacitivnosti vezanoj za njegov izlazni terminal. Prvi zahtev je lako ostvariti definisanjem odgovarajuće modelske klase. Drugi zahtev moguće je ostvariti zahvaljujući tome što sintaksa jezika AleC++ podržava definiciju korisničkih atributa signala. Naime, osim atributa koje sam simulator generiše za svaki signal tokom simulacije (`active`, `event`, `quiet`, `stable`, `fanin`, `fanout` i `hybrid`), korisnik ima mogućnost da dodefiniše

nove atribute signala. Sledećom deklaracijom kreiran je novi tip signala `three_full` kao signal tipa `three_t` uz koji je vezan atribut tipa `double` koji predstavlja totalnu kapacitivnost signala.

```
typedef three_t@double three_full;  // attribute: total capacitance
```

Korisnički atribut može biti proizvoljnog tipa, a ako ih ima više potrebno ih je udružiti u strukturu ili klasu. Operator `@` služi za pridruživanje korisničkog atributa signalu odredjenog tipa. Isti operator koristi se za pristup korisničkom atributu, za razliku od pristupa automatski generisanim atributima koji se vrši operatorom `->` kao u sledećim primerima.

```
signal three_full a;
*@a = 2pF;
if (a->event) { ... }
```

Kao što se uočava iz gornjih primera, samo ime signala `a` predstavlja pokazivač na memoriju u kojoj se nalaze podaci o tom signalu od kojih je jedan automatski atribut `event`, dok se pokazivač na memoriju u kojoj se smeštaju korisnički atributi signala dobija kao `@a`, a samoj vrednosti atributa se pristupa sa `*@a`.

Upotreba novog atributa signala za obradu kapacitivnosti biće objašnjena pri definiciji gejtova.

4.3 Modeliranje osnovnih logičkih elemenata

Kao i kod sistema sa dva stanja, definisaćemo baznu klasu koja će sadržati parametre vezane za konverziju pri hibridnoj simulaciji, a iz nje ćemo izvesti modelske klase za gejtove. Bazna klasa definisana je u deklaracijskoj datoteci **ss3.h** na sledeći način.

```
typedef struct {        // data structure of break points for D/A converter
    double time;
    double res;         // output resistance at the moment
    double level;       // output voltage at the moment
} Point;
enum Status { Rising, Falling, Steady };// D/A converter output voltage status

class io3 {
  protected:
                  // parameters for D/A converter
      double rise_time;       // rising ramp duration
      double fall_time;       // falling ramp duration
      double zero_level;      // voltage level of logic 0
      double one_level;       // voltage level of logic 1
      double x_level;         // voltage level of logic X
      double zero_res;        // resistance of logic 0
      double one_res;         // resistance of logic 1
      double x_res;           // resistance of logic X
      double Cout;            // logic gate output capacitance
      double ResTol;          // resistance tolerance
      double LevelTol;        // voltage tolerance
                  // parameters for A/D converter
      double Cin;             // analog circuit strey capacitance
      double one_threshold;   // logic level 1 threshold voltage
      double zero_threshold;  // logic level 0 threshold voltage
  public:
      io3  ();
      >io3 ();
      double state2level (three_t);
      double state2res (three_t);
      void set_breakpoints (double, three_t, three_t, Point*, Point*,
                                                      Status*, Status*);
      double evaluate_vramp (double, Point*, Point*);
```

```
      double evaluate_rramp (double, Point*, Point*);
      friend module cmos_a2d, cmos_d2a;
};
```

Konstruktor i procesor klase `io3` imaju sledeći oblik.

```
io3::io3 () {
  double zero_margin=1.5v, one_margin=1.5v;
                              // default parameter values:
  rise_time = fall_time = 2ns;
  zero_level = 0V;
  one_level  = 5V;
  x_level    = 2.5V;
  zero_res   = 1kohm;
  one_res    = 1kohm;
  x_res      = 10kohm;
  Cout       = 10fF;
  ResTol     = 1ohm;
  LevelTol   = 10nV;

  Cin = 100fF;
  one_threshold = one_level - one_margin;
  zero_threshold = zero_level + zero_margin;
}

io3::>io3 () {
  if (rise_time<=0.1ps) warning ("too short rise time", 1);
  if (fall_time<=0.1ps) warning ("too short fall time", 1);
  if (x_level<=zero_level || x_level>=one_level)
      warning ("x_level must lay between zero_level and one_level", 1);
  if (zero_res<=0) warning ("zero_res must be positive", 1);
  if (one_res<=0)  warning ("one_res must be positive", 1);
  if (x_res<=0)    warning ("x_res must be positive", 1);
  if (zero_threshold<=zero_level || zero_threshold>=one_level)
      warning ("zero_threshold must be between zero_level and one_level", 1);
  if (one_threshold<=zero_level || one_threshold>=one_level)
      warning ("one_threshold must be between zero_level and one_level", 1);
}
```

Kao objekti klase `io3` definisani su A/D i D/A konvertori `cmos_a2d` i `cmos_d2a`. Njihove deklaracije nalaze se u datoteci **ss3.h** i imaju sledeći oblik. O konvertorima će biti više reči kasnije.

```
module io3::cmos_a2d (node analog; signal three_full out digital);
module io3::cmos_d2a (signal three_full in digital; node analog);
```

U modelskoj datoteci **model3.hi** nalazi se definicija difoltne modelske kartice za klasu `io3`.

```
model io3::io3_default_model { }
```

Iz klase `io3` izvedena je klasa `gm3` kojoj pripadaju modelirani gejtovi.

```
class gm3 : public io3 {            // gate models for 3-state system
  private:
      double skew;
      double incap, outcap;
      double tplh, tphl;
  public:
      gm3 ();
      double delay3 (three_t ns, three_t os, double c, int hyb);
      friend module     inv, and, or, xor, nand, nor, nxor, mux, mux41,
                        clkgen, andx, orx, buf, full_add, shift,
                        ffrs, ffd, fft, cffrs, cffd, cffdr, cffjk, cffjkr, cfft;
};
```

Parametar `skew` predstavlja nagib porasta kašnjenja gejta sa kapacitivnim opterećenjem izlaza (jedinica s/F - sekund po faradu). Parametri `incap` i `outcap` predstavljaju ulazne i izlazne kapacitivnosti gejta. Konstruktor klase `gm3` definiše sledeće difoltne vrednosti parametara klase.

```
gm3::gm3 () {
  skew = 0.0;
  incap = 1pF;
  outcap = 1pF;
  tplh = tphl = 10ns;
}
```

Kao objekti klase `gm3` (i njeni "prijatelji" da bi mogli da pristupaju parametrima modela) definisani su sledeći gejtovi.

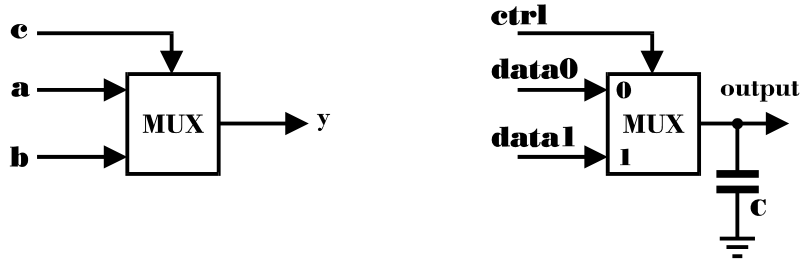
```
/* ------------------------------------------------- Standard gates :  */
module gm3::buf  (three_full out y; three_full in a);
module gm3::inv  (three_full out y; signal three_full in a);
module gm3::and  (three_full out y; signal three_full in a,b);
module gm3::or   (three_full out y; signal three_full in a,b);
module gm3::andx (three_full out y; three_full in a[]);
module gm3::orx  (three_full out y; three_full in a[]);
module gm3::xor  (three_full out y; signal three_full in a,b);
module gm3::nand (three_full out y; signal three_full in a,b);
module gm3::nor  (three_full out y; signal three_full in a,b);
module gm3::nxor (three_full out y; signal three_full in a,b);
module gm3::mux  (three_full out y; signal three_full in a, b, c);
module gm3::mux41(three_full out y; signal three_full in a, b, c, d, c1, c2);
module gm3::full_add (three_full in a, b, pi; three_full out c, po);
/* ---------------------------------------- Asynchronous flip-flops :  */
module gm3::ffrs  (three_full in r, s; three_full out q, nq);
module gm3::ffd   (three_full in d; three_full out q, nq);
module gm3::fft   (three_full in t; three_full out q, nq);
/* -------------------------------- Synchronous (clocked) flip-flops :  */
module gm3::cffdr (three_full in d, resetb, c; three_full out q, nq);
module gm3::cffd  (three_full in d, c; three_full out q, nq);
module gm3::cffrs (three_full in r, s, c; three_full out q, nq);
module gm3::cffjk (three_full in j, k, c; three_full out q, nq);
module gm3::cffjkr(three_full in j, k, c, reset; three_full out q, nq);
module gm3::cfft  (three_full in t, c; three_full out q, nq);
```

U modelskoj datoteci **model3.hi** nalazi se definicija difoltne modelske kartice za klasu `gm3`.

```
model gm3::gm3_default_model { }
```

Modul kojim je modeliran generator taktnih impulsa nije uvršten u klasu `gm3`, jer se on vezuje na ulaze kola i pretpostavlja se da radi kao idealan ulaz, tako da nije potrebno modelirati uticaj kapacitivnosti na njegovo ponašanje. Takodje ćemo pretpostaviti da će taktni generator biti korišćen samo za pobudjivanje digitalnog dela kola, tako da uz njega nije potrebno deklarisati D/A konvertor.

```
module clkgen (signal three_full inout y)
            action (double width1, double width0, double startdelay);
```



a) *b)*

Slika 4.2 *a) Multiplekser 2 u 1 b) Multiplekser 2 u 1 sa kapacitivno opterećenim izlazom*

Jedan od modula "prijatelja" klase `gm3` je i modul `mux` koji predstavlja multiplekser 2 u 1 (slika 4.2a). Njegov model nalazi se u strukturnoj datoteci **gates3.hi** i dat je sledećim kodom.

```
module gm3::mux (three_full out y; three_full in a, b, c) {
  conversion { a2d = "cmos_a2d"; d2a = "cmos_d2a"; }
  action {
    process post_structural {
      *@a += incap;
      *@b += incap;
      *@c += incap;
      *@y += outcap;
    }
    process (a, b, c) {
      if (c=='0')      { y <- a after this.delay3 (a, y, *@y, y->hybrid); }
      else if (c=='1') { y <- b after this.delay3(b, y, *@y, y->hybrid); }
      else {
        if (a!=b) {
          warning ("mux control input unknown (different inputs)");
          y <- 'x' after this.delay3('x', y, *@y, y->hybrid);
        }
        else {
          warning ("mux control input unknown (equal inputs)");
          y <- a after this.delay3(a, y, *@y, y->hybrid);
        }
      }
    }
  }
}
```

Model multipleksera sadrži dva procesa od kojih privi služi za dodavanje terminalnih kapacitivnosti atributu signala za koji su terminali vezani. Sličan proces sadrže svi moduli koji predstavljaju osnovne logičke elemente - gradivne blokove simuliranog kola. Proces je sinhronizovan internim signalom `post_structural` koji je aktivan samo jednom, neposredno posle uspostavljanja strukture hijerarhijskog stabla opisa kola. Korisnički atribut signala definisan u okviru tipa `three_full` pre početka simulacije sadržavaće informaciju o ukupnoj kapacitivnosti vezanoj za signal. Na ovaj način obezbedjeno je da procesi koji su aktivni u toku simulacije imaju neophodnu informaciju o kapacitivnostima. Recimo, drugi proces u modelu multipleksera prosledjuje totalnu kapacitivnost izlaza (`*@y`) funkciji `delay3` koja na osnovu nje odredjuje kašnjenje multipleksera.

Logično je pretpostaviti da će korisnik poželeti da modelira pojavu parazitne kapacitivnosti na dugim žicama u simuliranom kolu, tako da je potrebno obezbediti mehanizam za direktno dodavanje kapacitivnosti pojedinim signalima. Jedan od načina da se simulira ova kapacitivnost je da se ona veže u kolu kao analogna kapacitivnost. Ovakav način nepotrebno komplikuje simulaciju. Drugi način je definisanje posebnog modula koji ima funkciju "digitalne kapacitivnosti". On dodaje zadatu vrednost kapacitivnosti atributu signala za koji je vezan. Ovaj modul definisan je u datoteci **gates3.hi** na sledeći način.

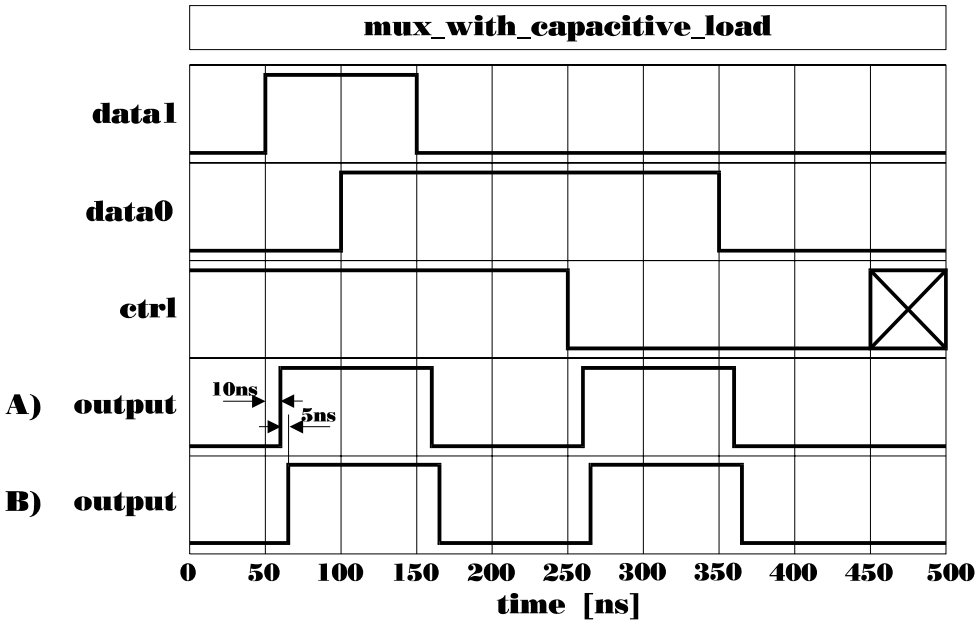
```
module digcap (three_full out y) {
   action (double cap) {
    process post_structural { *@y += cap; }
  }
}
```

Kao ilustraciju upotrebe modula `digcap` simuliraćemo kolo sa slike 4.2b. Opis kola za simulaciju ima sledeći izgled.

```
# include "ss3.h"
library gates3, op3;
```

```
model gm3::model_10ns_1000s_per_farad {
  tplh=10ns; tphl=10ns;
  skew=1000s_per_farad;
  incap=0.0; outcap=0.0;
}

root mux_with_capacitive_load () {
  module gm3::mux multiplexor;
  module digcap load;
  signal three_full data0, data1, ctrl, output='0';

  multiplexor (output, data0, data1, ctrl) model=model_10ns_1000s_per_farad;
  load (output) action (5pF);
  //load (output) action (0pF);

  timing { tstop = 500ns; }
  out { signal three_full output, ctrl, data0, data1; }
  action {
    process initial {
        data0 <- '0' after 0ns, '1' after 100ns, '0' after 350ns;
        data1 <- '0' after 0ns, '1' after  50ns, '0' after 150ns;
        ctrl  <- '1' after 0ns, '0' after 250ns, 'x' after 450ns ;
    }
  }
}
```

Za izlaz multipleksera 2 u 1 (modul `mux`) vezano je kapacitivno opterećenje `C=5pF`. U modelskoj kartici `model_10ns_1000s_per_farad` definisani su parametri modela multipleksera. Multiplekser ima jednaka kašnjenja prednje i zadnje ivice `tplh=tphl=10ns` i nagib porasta kašnjenja sa kapacitivnim opterećenjem izlaza `skew=1000s/F=1ns/pF`, a terminalne kapacitivnosti su zanemarivo male (`incap=outcap=0`). Dodatno kašnjenje koje je posledica kapacitivnog opterećenja u ovom slučaju se izračunava kao proizvod `skew*C=1000s/F*5pF=5ns`. Kolo je simulirano bez kapacitivnog opterećenja (akcioni parametar komponente `load` postavljen na `0`) i sa kapacitivnim opterećenjem `5pF`. Rezultati ove dve simulacije prikazani su na slici 4.3. Sa A) je označen talasni oblik na izlazu kada je vrednost kapacitivnosti `0`, a sa B) talasni oblik na izlazu kada je vrednost kapacitivnosti `5pF`.



Slika 4.3 *Uticaj kapacitivnog opterećenja na kašnjenje*

Kada je ulaz `ctrl` na logičkoj jedinici (od početka simulacije do `250ns`), na izlaz `output` se preslikava stanje sa ulaznog terminala `data1` posle kašnjenja multipleksera. Kada je ulaz `ctrl` na logičkoj nuli (od `250ns` do `450ns`), na izlaz se preslikava stanje sa ulaza `data0`. Kad kontrolni ulaz ode u neodredjeno stanje, izlaz zavisi od stanja oba ulaza - ako su stanja ista, izlaz je definisan (posle trenutka `450ns`). Na slici 4.3 jasno se uočava kašnjenje talasnog oblika na izlazu u slučaju pod B) za `5ns`.

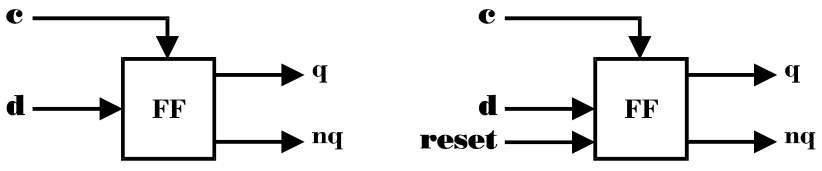
4.4 Model kašnjenja

Modul `mux` koji smo izabrali za primer modeliranja osnovnih gejtova koristi funkciju `delay3` za odredjivanje kašnjenja. Njoj se prosledjuje naredno i prethodno stanje izlaza, totalna kapacitivnost vezana za izlaz i informacija o aspektu izlaznog signala. Ako u sistemu sa tri stanja koristimo iste parametre kašnjenja kao u sistemu sa dva stanja, funkcija za kašnjenje se modifikuje na sledeći način.

```
# define RISING         0
# define FALLING        1
# define WHO_KNOWS      2
double gm3::delay3 (three_t ns, three_t os, double c, int hyb) {
  double result;
  int tmp;
  if      (ns == '0') { result = tphl+skew*c; tmp = FALLING; }
  else if (ns == '1') { result = tplh+skew*c; tmp = RISING;  }
  else {
    if (os=='0') { result = tplh+skew*c; tmp = RISING;  }
    if (os=='1') { result = tphl+skew*c; tmp = FALLING; }
    else {
      result (((tplh+skew*c) > (tphl+skew*c)) ? (tplh+skew*c) : (tphl+skew*c));
      tmp = WHO_KNOWS;
    }
  }
  if (hyb) {
    return ( (tmp==FALLING) ? result-fall_time/2.0 :
             (tmp==RISING)  ? result-rise_time/2.0 :
                              result-MAX(rise_time, fall_time)/2.0 );
  }
  else return result;
}
```

4.5 Modeliranje flip-flopova

U ovom odeljku daćemo pregled modela flip-flopova koje sadrži biblioteka za sistem sa tri stanja. Vremenske provere koje se obično ugradjuju u modele sekvencijalnih kola neće ovde biti uzete u razmatranje, jer ih je relativno lako dodati osnovnom modelu funkcije.



a) *b)*

Slika 4.4 *a) D flip-flop b) D flip-flop sa direktnim reset ulazom*

Razmotrićemo najpre D flip-flop kao najjednostavniji medju flip-flopovima. D flip-flop se odlikuje svojstvom pamćenja prethodnog stanja dovedenog na ulaz. Osnovna struktura sinhronog

(taktovanog) D flip-flopa prikazana je na slici 4.4a. Na ulaz `d` dovode se podaci, a `c` je ulaz za takt. Model ovog kola u simulatoru Alecsis2.1 dat je sledećim kodom.

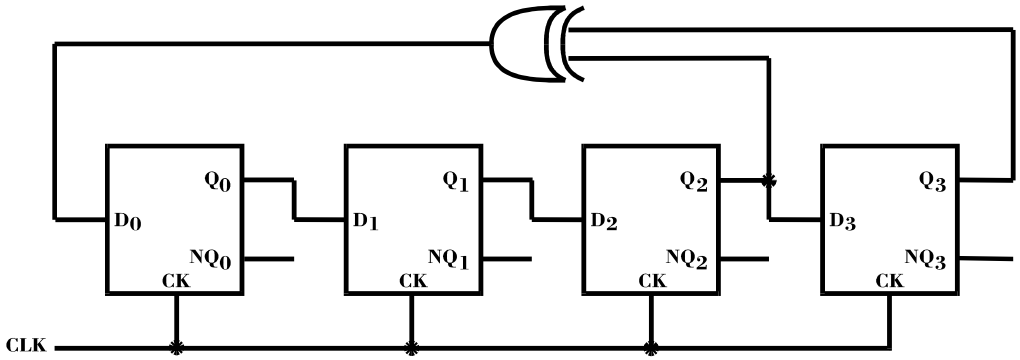
```
// Rising edge triggered d flip-flop.
module gm3::cffd (three_full in d, c; three_full out q, nq) {
  conversion { a2d = "cmos_a2d"; d2a = "cmos_d2a"; }
  action {
    add_capacitances : process post_structural {
      *@d  += incap;
      *@c  += incap;
      *@q  += outcap;
      *@nq += outcap;
    }
    behaviour : process (c) {
      if (c == 'x') {
        warning ("clock unknown");
        q  <- 'x' after delay3('x',  q,  *@q,  q->hybrid);
        nq <- 'x' after delay3('x', nq, *@nq, nq->hybrid);
      }
      else if (c->event && c == '1') {    // rising edge
        q  <-  d after delay3( d,  q,  *@q,  q->hybrid);
        nq <- ~d after delay3(~d, nq, *@nq, nq->hybrid);
      }
    }
  }
}
```

S obzirom da je proces `behaviour`, koji modelira ponašanje flip-flopa, sinhronizovan samo signalom takta `c`, za detekciju prednje ivice dovoljan je uslov `c=='1'`. Da ima još signala na listi osetljivosti procesa, morala bi da se detektuje i aktivnost signala `c`, kao što je to, ilustracije radi, uradjeno u gornjem slučaju.

Usložnjavanje osnovnog modela D flip-flopa nije teško, što ćemo pokazati na primeru razvoja modela D flip-flopa sa direktnim reset ulazom (slika 4.4b). Ulaz za reset dejstvuje asinhrono i ima dominantnu ulogu u odnosu na ostatak logičke funkcije. Zato ga je vrlo jednostavno dodati već razvijenom modelu.

```
// Rising edge triggered d flip-flop with low active direct reset.
module gm3::cffdr (three_full in d, resetb, c; three_full out q, nq) {
  conversion { a2d = "cmos_a2d"; d2a = "cmos_d2a"; }
  action {
    add_capacitances : process post_structural {
      *@d      += incap;
      *@c      += incap;
      *@resetb += incap;
      *@q      += outcap;
      *@nq     += outcap;
    }
    behaviour : process (resetb, c) {
      signal three_full tmp;

      if (resetb == '0') {
          tmp='0';
          q  <-  tmp after delay3( tmp,  q,  *@q,  q->hybrid);
          nq <- ~tmp after delay3(~tmp, nq, *@nq, nq->hybrid);
      }
      else if (resetb == 'x') {
        tmp='x';
        warning ("reset unknown");
        q  <- tmp after delay3(tmp,  q,  *@q,  q->hybrid);
        nq <- tmp after delay3(tmp, nq, *@nq, nq->hybrid);
      }
      else if (c->event && c=='x') { // reset pasive, standard operation mode:
        tmp='x';
        warning ("clock gone unknown");
```

```
        q  <- tmp after delay3(tmp,  q,  *@q,  q->hybrid);
        nq <- tmp after delay3(tmp, nq, *@nq, nq->hybrid);
      }
      else if (c->event && c=='1') {      // rising edge
        tmp=d;
        q  <-  tmp after delay3( tmp,  q,  *@q,  q->hybrid);
        nq <- ~tmp after delay3(~tmp, nq, *@nq, nq->hybrid);
      }
    }
  }
}
```



Slika 4.5 *Primer upotrebe flip-flopova: generator sekvence 100010011010111*

Testirajmo model sinhronog D flip-flopa `cffd` jednom jednostavnom simulacijom. Kolo sa slike 4.5 predstavlja generator sekvence 100010011010111. Svaki generator sekvence ima jedno stanje izlaza flip-flopova koje blokira rad. U ovom slučaju to je stanje `"0000"`. Iz njega je izlazak nemoguć. Opis kola sa slike 4.5 za simulaciju je sledeći.

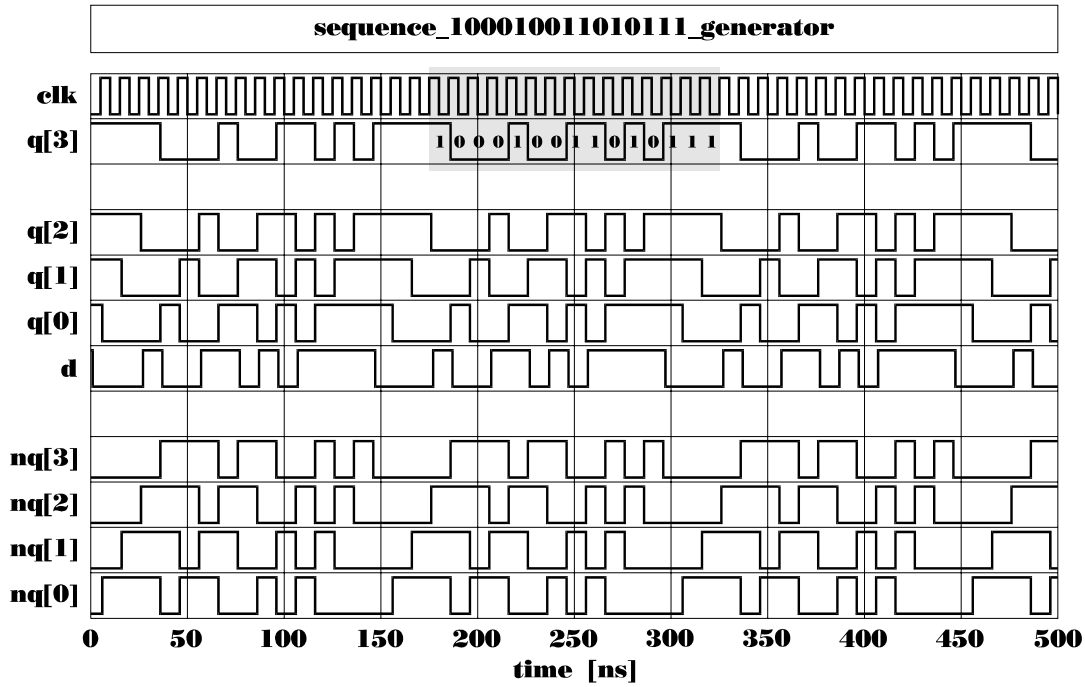
```
# include "ss3.h"
library model3, gates3, op3;
root module sequence_100010011010111_generator () {
  module gm3::cffd g;
  module gm3::xor x;
  module clkgen cc;
  signal three_full d='1', clk='0', q[4]="1111", nq[4]="0000";

  cc (clk) { width1=5ns; width0=5ns; startdelay=5ns; }
  x  (d, q[3], q[2]) model=gm3_1ns_model;

  timing { tstop = 500ns; }
  out { signal three_full nq, d, q[0], q[1], q[2], q[3], clk; }
  action {
    process structural {
      clone g[0] (d, clk, q[0], nq[0]) model=gm3_1ns_model;
      for (int i=1; i < 4; i++)
        clone g[i] (q[i-1], clk, q[i], nq[i]) model=gm3_1ns_model;
    }
  }
}
```

Simuliraćemo kolo sa jediničnim modelom kašnjenja, tako da modelska kartica `gm3_1ns_model` ima sledeći izgled.

```
model gm3::gm3_1ns_model {
      tplh=1ns;
      tphl=1ns;
}
```

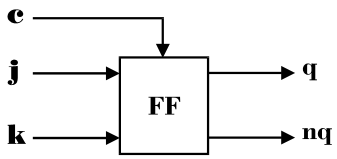
U strukturnom delu opisa kola naveden je generator taktnih impulsa `cc` i kolo ekskluzivno ILI `x`, dok su D flip-flopovi klonirani u procesu sinhronizovanom signalom `structural`. Kako kolo

ima povratnu spregu sa izlaza na ulaz, pobuda nije neophodna. Jedino je neophodno sprečiti da simulacija startuje sa stanjima flip-flopova koja blokiraju rad. Zato su stanja `q` izlaza inicijalizirana logičkom jedinicom. Ovo odgovara realnoj situaciji u kolu, s obzirom da se ovaj generator sekvence obavezno realizuje od D flip-flopova sa direktnim ulazom za setovanje, što na slici 4.5 nije posebno istaknuto. Slika 4.6 prikazuje rezultat simulacije. Jasno se uočava generisana sekvenca koja se neprekidno ponavlja. Ako se signal `q` inicijalizuje vrednošću `"0000"`, kolo ulazi u stabilno stanje i ne generiše željeni talasni oblik, što se lako potvrdjuje simulacijom.



Slika 4.6 *Rezultat simulacije generatora sekvence sa slike 4.5*

U osnovne tipove flip-flopova koje mora da sadrži biblioteka logičkih primitiva spadaju još T, RS i JK flip-flop. Osnovni model JK flip-flopa sa slike 4.7 dajemo bez daljih objašnjenja.



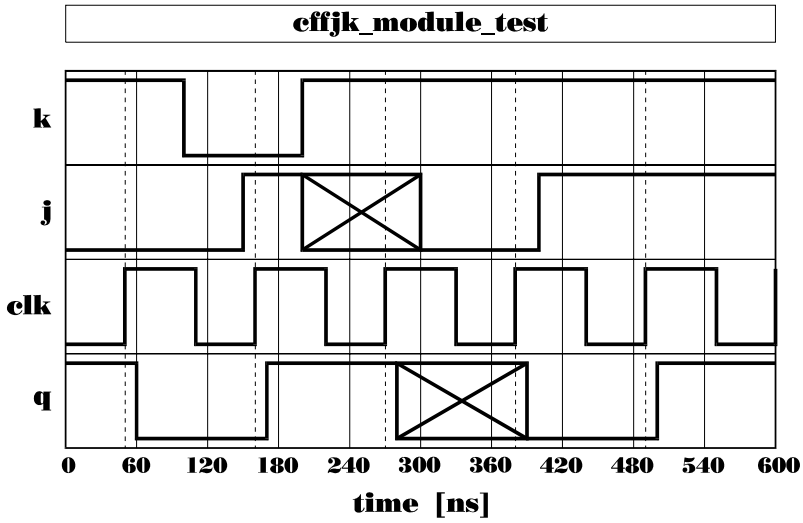
Slika 4.7 *Sinhroni JK flip-flop*

```
module gm3::cffjk (three_full in j, k, c; three_full out q, nq) {
  conversion { a2d = "cmos_a2d"; d2a = "cmos_d2a"; }
  action  {
    process post_structural {
      *@j     += incap;
      *@k     += incap;
      *@c     += incap;
      *@q     += outcap;
      *@nq    += outcap;
    }
    process (c) {
      three_full tmp;

      if (c =='x') {                // clock unknown
        tmp='x';
        warning ("clk input gone unknown, outputs set to 'x'");
        q  <- tmp after delay3(tmp,  q,  *@q,  q->hybrid);
        nq <- tmp after delay3(tmp, nq, *@nq, nq->hybrid);
      }
```

```
      else if (c=='1') {            // rising edge
        if (j =='0' && k=='1') {    // reset state
          tmp='0';
          q  <-  tmp after delay3( tmp,  q,  *@q,  q->hybrid);
          nq <- ~tmp after delay3(~tmp, nq, *@nq, nq->hybrid);
        }
        else if (j=='1' && k=='0') {// set state
          tmp='1';
          q  <-  tmp after delay3( tmp,  q,  *@q,  q->hybrid);
          nq <- ~tmp after delay3(~tmp, nq, *@nq, nq->hybrid);
        }
        else if (j=='1' && k=='1') {// complement state
          tmp = ~q;
          q  <-  tmp after delay3( tmp,  q,  *@q,  q->hybrid);
          nq <- ~tmp after delay3(~tmp, nq, *@nq, nq->hybrid);
        }
        else if (j=='x' || k=='x') {// mailfunction state
          tmp='x';
          warning ("j/k unknown");
          q  <- tmp after delay3(tmp,  q,  *@q,  q->hybrid);
          nq <- tmp after delay3(tmp, nq, *@nq, nq->hybrid);
        }
      }
    }
  }
}
```

Proveru funkcionisanja modula `gm3::cffjk` obavićemo sledećim test primerom.

```
# include "ss3.h"
library model3, gates3, op3;
root module cffjk_module_test () {
  module gm3::cffjk g;
  module clkgen cg;
  signal three_full j, k, clk='0', q='1', nq ;
  g (j, k, clk, q, nq) model=gm3_default_model;
  cg (clk) { width1=60ns; width0 = 50ns; startdelay=0ns; }

  timing { tstop = 600ns; }
  out { signal three_full q, clk, j, k; }
  action {
    process initial {
      j <-  '0' after 0ns, '1' after 150ns, 'x' after 200ns,
            '0' after 300ns, '1' after 400ns;
      k <-  '1' after 0ns, '0' after 100ns, '1' after 200ns;
    }
  }
}
```



Slika 4.8 *Rezultat testiranja JK flip-flopa*

Rezultat simulacije ovog kola dat je na slici 4.8. JK flip-flop je sinhronizovan prednjom ivicom `clk` signala. Izlaz `q` menja stanje u trenucima kada `clk` signal ima uzlaznu ivicu: 50ns, 150ns, 250ns, ... Kašnjenje kako prednje, tako i zadnje ivice je 10ns, kao što je zadato modelom `gm3_default_model` (vrednosti parametara modela definisane su u konstruktoru klase `gm3`). Stanje na izlazu zavisi od stanja ulaza `j` i `k`. Na primer, u trenutku 50ns, pri pojavi prednje ivice taktnog signala stanje na ulazima je `j='0'`, `k='1'`, što prouzrokuje resetovanje flip-flopa u trenutku 60ns (izlaz `q` prelazi u stanje `'0'`). Kad jedan od ulaza `j` i `k` dodje u neodredjeno stanje `'X'`, na izlaz se prosledjuje stanje `'X'` (recimo u trenutku 360ns), ali naravno tek na prednju ivicu taktnog signala.

4.6 Žičana logika

Kao i kod sistema sa dva stanja, u sistemu sa tri stanja moguće je modelirati žičanu logiku. Funkcije rezolucije koje je potrebno definisati ostaju vrlo slične, osim što treba voditi računa i o `'X'` stanju. Definisane su tri funkcije rezolucije. Njihove deklaracije iz datoteke **ss3.h** slede.

```
extern three_t wired_same (const three_t *, int *);
extern three_t wired_and  (const three_t *, int *);
extern three_t wired_or   (const three_t *, int *);
```

Definicije ovih funkcija nalaze se u datoteci **op3.hi**. Ovde ćemo navesti funkcije `wired_and` i `wired_same`. Funkcija `wired_and` uspostavlja dominaciju stanja `'0'` na signalu, a funkcija `wired_same` vraća stanje `'X'` ako vrednosti svih drajvera signala nisu iste.

```
three_t wired_and (const three_t *drivers, int *rprt) {
  three_t result;
  int i;

  for (i=1, result=drivers[0]; i<lengthof(drivers); i++) {
    result = result & drivers[i];
    if (result=='0') break;
  }
  if (result=='x') *rprt=2;
  return result;
}

three_t wired_same (const three_t *drivers, int *rprt) {
  three_t result;
  int i, countX=0;
```

```
  if ((result=drivers[0])=='x') countX++;;
  if (result!='x') {
    for (i=1; i<lengthof(drivers); i++) {
      if (drivers[i]=='x') countX++;
      if (result!=drivers[i]) { result = 'x'; break; }
    }
  }
  if (countX==0 && result=='x') *rprt=1;        // report conflict
  else if (countX!=0) *rprt=2;                  // report possible conflict
  return result;
}
```

Kao što je već rečeno u drugom poglavlju, kod funkcija za rezoluciju postoje dve mogućnosti za automatsko izveštavanje o konfliktima. Ako se vrednost `*rprt` postavi na 1, simulator će javiti o konfliktu, a ako se postavi na 2, simulator će prijaviti potencijalni konflikt (recimo kada je jedan od drajvera u stanju `'X'`).

4.7 A/D i D/A konverzija

Konverzija signala i sprega analognog i digitalnog dela kola su u sistemu sa tri stanja zasnovane na istim principima kao kod sistema sa dva stanja. Smatraćemo da se stanje `'X'` javlja uvek pri promeni stanja sa logičke nule na logičku jedinicu i obrnuto. To omogućava da se razviju modeli A/D i D/A konvertora. Ostaćemo pri konvertorima za CMOS logička kola. Konvertori će biti objekti bazne modelske klase `io3`.

4.7.1 A/D konverzija

Ulaz CMOS logičkog kola može se modelirati kapacitivnošću gejta. Pri A/D konverziji signala vrši se poredjenje sa dva praga promene. Kada je vrednost analogne veličine manja od praga V_{t1}, smatramo da je u pitanju stanje logičke nule `'0'`, a kada je vrednost analogne veličine veća od praga V_{t2}, u pitanju je stanje logičke jedinice `'1'`. Sve vrednosti izmedju pragova V_{t1} i V_{t2} rezultuju u neodredjenom stanju `'X'`. Modul `io3::cmos_a2d` definisan je u strukturnoj datoteci **gates3.hi** i ima sledeći izgled.

```
module io3::cmos_a2d (node analog; signal three_full out digital) {
  capacitor Capin;
  Capin (analog, 0);
  action {
    assign: process post_structural { Capin->value = Cin; }
    convert: process post_moment {
      three_full new_event, last_event='x';

      if (analog >= one_threshold) {            // definitely '1'
        new_event = '1';
      }
      else if (analog <= zero_threshold) {      // definitely '0'
        new_event = '0';
      }
      else {                                 // unknown intermediate state
        new_event = 'x';
      }
      if (new_event != last_event) {
        last_event = new_event;
        digital <- new_event;
      }
    } // convert: process
  }
}
```

4.7.2 D/A konverzija

D/A konvertor zahteva definisanje izlazne otpornosti gejta u stanju `'X'` (ako stanje `'X'` smatramo za prelazno stanje izmedju `'0'` i `'1'`, onda je u tom stanju otpornost velika, jer ni N-kanalni ni P-kanalni tranzistor na izlazu u jednom trenutku ne vode). Kod kojim je modeliran konvertor `io3::cmos_d2a` ima sledeći izgled.

```
module io3::cmos_d2a (signal three_full in digital; node analog) {
  capacitor Capout;
  resistor Rout;
  cgen Iout;

  Capout (analog, 0);
  Rout   (analog, 0);
  Iout   (0, analog);

  action {
    static Status istat=Steady, rstat=Steady;
    static three_full old_state='0';      // guessed initial input state
    static three_full new_state='0';
    static Point start={0,1k,0}, stop={0,1k,0};

    assign: process post_structural {     // assign component values
      Capout->value = this.Cout;          // *Capout = Cout; <- this is O.K. too
      Rout->value = this.zero_res;        // *Rout = zero_res;
    }

    watch: process (digital) {            // monitor digital node
      // when an event occur...
      old_state = new_state;
      new_state = digital;
      //printf ("new_state = %s\n", names3[new_state]);
      set_breakpoints(now, old_state, new_state, &start, &stop, &istat, &rstat);
    }

    control: process per_moment {         // control analog devices
      double rvalue, new_level, level;

      rvalue = Rout->value;
      new_level = level = Iout->value * rvalue;
      if (rstat != Steady) {
        if (fabs(rvalue - stop.res) <= ResTol) rstat = Steady;
        else {                            // resistance still in transition
          Rout->value = evaluate_rramp(now, &start, &stop);
          Iout->value = level / Rout->value;
        }
      }
      if (istat != Steady) {
        if (fabs(level - stop.level) <= LevelTol) istat = Steady;
        else {                            // level still in transition
          new_level = evaluate_vramp(now, &start, &stop);
          Iout->value = new_level / Rout->value;
        }
      }
    }
  }
}
```

Upotrebljen je isti model konvertora kao kod sistema sa dva stanja, s tom razlikom da je vodjeno računa o trećem, `'X'` stanju. U tu svrhu definisani su odgovarajući parametri u modelskoj klasi `io3`. Parametar `x_level` definiše napon na izlazu D/A konvertora kada je ulaz u `'X'` stanju, a parametar `x_res` definiše izlaznu otpornost kada je ulaz konvertora u `'X'` stanju. Izlaz digitalnog

elementa modeliran je konstantnom izlaznom kapacitivnošću, promenljivom izlaznom otpornošću i kontrolisanim strujnim generatorom. Modul `cmos_d2a` ostvaruje svoju funkciju kroz tri procesa. Prvi proces, označen labelom `assign`, dodeljuje vrednosti parametrima izlaznog kola D/A konvertora - izlaznoj kapacitivnosti i otpornosti. Drugi proces, označen labelom `watch`, prati vrednost ulaznog digitalnog signala i na svaku njenu promenu postavlja prelomne tačke za talasni oblik koji D/A konvertor treba da kreira na izlazu. Pri tome se koriste dve promenljive tipa `Point` - `start` i `stop`. Posle poziva funkcije `set_breakpoints` one sadrže vremenske trenutke, kao i vrednost izlazne otpornosti i napona na izlazu na početku i na kraju prelaznog procesa koji treba modelirati na izlazu konvertora. Takodje se koriste i dve promenljive tipa `Status` - `istat` i `rstat` koje posle poziva funkcije `set_breakpoints` sadrže informaciju o trenutnom stanju izlazne otpornosti i kontrolisanog izlaznog strujnog generatora. I konačno, treći proces, označen labelom `control`, a sinhronisan signalom `per_moment`, kontroliše vrednosti izlazne otpornosti i napona (odnosno izlaznog strujnog generatora).

Korišćene funkcije smeštene su u funkcijsku datoteku **op3.hi** i imaju sledeći oblik.

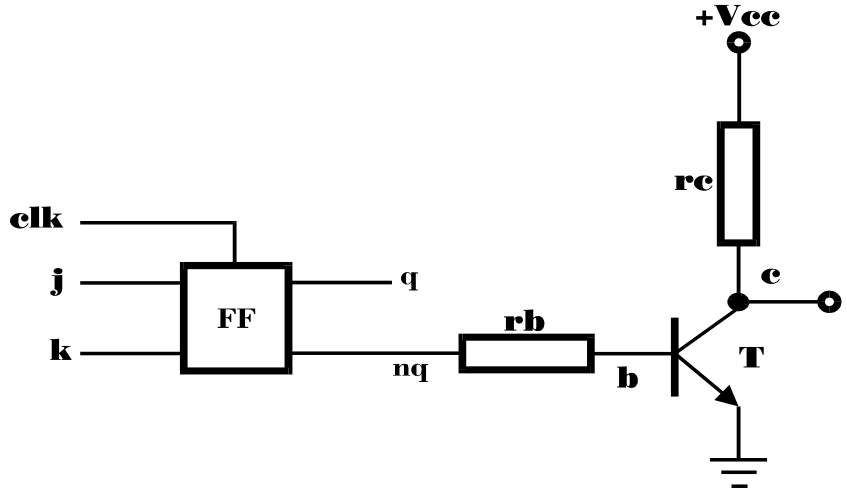
```
inline double io3::state2level (three_t state) {
  if (state == '1') return one_level;
  if (state == '0') return zero_level;
  return x_level;
}


inline double io3::state2res (three_t state) {
  if (state == '1') return one_res;
  if (state == '0') return zero_res;
  return x_res;
}

void io3::set_breakpoints (double time, three_t old_s, three_t new_s,
                           Point *start, Point *stop, Status *istat,
                                                      Status *rstat) {
  // Determine output voltage level status:
  if (old_s == new_s) *istat = Steady;
  else if (new_s=='1' || (new_s=='x' && old_s=='0')) *istat = Rising;
  else *istat = Falling;

  // Determine output resistance status:
  double old_res = state2res(old_s);
  double new_res = state2res(new_s);
  if (old_res > new_res) *rstat = Falling;
  else if (old_res < new_res) *rstat = Rising;
  else *rstat = Steady;

  // Set breakpoints:
  start->time = time;
  start->res = stop->res;
  start->level = stop->level;
  stop->res = new_res;
  stop->level = state2level(new_s);
  stop->time = time + (*istat == Rising ? rise_time : fall_time);
}
```

Funkcije `evaluate_vramp` i `evaluate_rramp` odredjuju novu vrednost napona, odnosno izlazne otpornosti na osnovu trenutka simulacionog vremena i dve zadate prelomne tačke. Opredelili smo se za jednostavni model linearne promene na izlazu. Izmenom ove dve funkcije moguće je implementirati i drugačije prelazne procese na izlazu D/A konvertora pri promeni stanja digitalnog ulaza.

```
double io3::evaluate_vramp (double time, Point *start, Point *stop) {
  // linear voltage ramp
  double slope = (stop->level - start->level)/(stop->time - start->time);
  double new_level = start->level + slope * (time - start->time);
```

```
  if (slope < 0.0) new_level = MAX(new_level, stop->level);
  else new_level = MIN(new_level, stop->level);
  return new_level;
}

double io3::evaluate_rramp (double time, Point *start, Point *stop) {
    // linear resistance ramp
    double slope = (stop->res - start->res)/(stop->time - start->time);
    double rlevel = start->res + slope * (time - start->time);
    if (slope < 0.0) rlevel = MAX(rlevel, stop->res);
    else rlevel = MIN(rlevel, stop->res);
    return rlevel;
}
```

Ilustrovaćemo funkcionisanje D/A konvertora na primeru sa slike 4.9. To je JK flip-flop (o njemu je već bilo reči) kome je dodat tranzistorski izlazni stepen na `nq` izlazu. Simulator će automatski ubaciti D/A konvertor izmedju `nq` izlaza JK flip-flopa i analognog dela kola (otpornik `rb`).



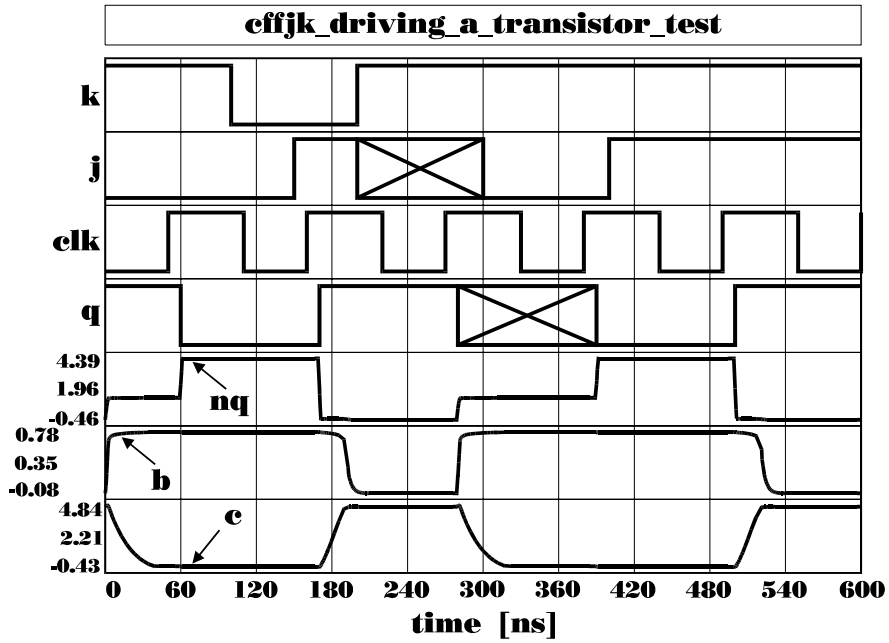
Slika 4.9 *Hibridno kolo - JK flip-flop sa tranzistorskim izlaznim stepenom*

Opis kola sa slike 4.9 za simulaciju je sledeći.

```
spice {                                    // SPICE syntax region
.MODEL MODEL1 NPN (BF=100,CJC=0.5PF)
}
# include "ss3.h"
library model3, gates3, op3;
root module cffjk_driving_a_transistor_test () {
  module gm3::cffjk g;
  module clkgen cg;
  signal three_full j, k, clk='0', q='1', nq='0';

  bjt tran;
  vgen Vcc;
  resistor rb, rc;
  node c, b, vcc;

  g (j, k, clk, q, nq) model=gm3_default_model;
  cg (clk) { width1=60ns; width0 = 50ns; startdelay=0ns; }
  tran (c, b, 0) model = MODEL1;
  rb (b, nq) 10k;
  rc (c, vcc) 1k;
  Vcc (vcc, 0) 5v;

  timing { tstop = 600ns; a_step=0.1ns; }
  out {
    node c;
```

```
    node b;
    signal three_full nq;     // treated as a node
    signal three_full q, clk, j, k;
  }
  action {
    process initial {
      j <- '0' after 0ns, '1' after 150ns, 'x' after 200ns,
           '0' after 300ns, '1' after 400ns;
      k <- '1' after 0ns, '0' after 100ns, '1' after 200ns;
    }
  }
}
```

Slika 4.10 prikazuje rezultat simulacije. Vidimo da je napon stanja `'X'` tako izabran (2.5V, postavljen u konstruktoru klase `io3`) da je i za stanje `'X'` na izlazu `nq` tranzistor u provodnom stanju. Na slici se može proveriti pravilno funkcionisanje JK flip-flopa (posmatranjem digitalnih promenljivih), kao i funkcionisanje tranzistorskog stepena.



Slika 4.10 *Rezultat simulacije kola sa slike 4.9*

5 Biblioteka za sistem sa četiri stanja

Sistem sa četiri stanja sadrži neodredjeno stanje `'X'`, stanje logičke nule `'0'`, stanje logičke jedinice `'1'` i stanje visoke impedanse `'Z'`. Stanje visoke impedanse javlja se na izlazu jedne grupe logičkih kola koja nazivamo trostatička kola (imaju tri validna stanja na izlazu - `'0'`, `'1'` i `'Z'`). U sistemu sa četiri stanja potrebno je modelirati pojave povezane sa ovim stanjem, pa ćemo tome posvetiti ovo poglavlje.

Razvijena je biblioteka gejtova za sistem sa četiri stanja koja obezbedjuje modeliranje trostatičkih digitalnih kola. Biblioteka je organizovana u četiri datoteke:

ss4.h - deklaraciona datoteka
op4.h - funkcijska datoteka
gates4.hi - modulska datoteka
model4.hi - biblioteka modela

5.1 Definisanje sistema stanja i preopterećenje operatora

Sistem sa četiri stanja formiraćemo na sledeći način.

```
typedef enum {
      'X', 'x'='X',
      '0',
      '1',
      'Z', 'z'='Z',
      '_'=void, ' '=void
} four_t;
```

Stanje `'X'` je namerno izabrano kao prvo u definiciji enumerisanog tipa `four_t` da bi se sve promenljive ovog tipa automatski inicijalizirale ovim stanjem. Kao i kod drugih sistema stanja, definisana je tabela `names4` koja sadrži imena stanja.

```
const char *names4[] = { "X", "0", "1", "Z" };
```

Sledeći logički operatori preopterećeni su za rad u ovom sistemu stanja.

```
extern four_t operator ~  (four_t op);
extern four_t operator &  (four_t op1, four_t op2);
extern four_t operator |  (four_t op1, four_t op2);
extern four_t operator ~& (four_t op1, four_t op2);
```

```
extern four_t operator ~| (four_t op1, four_t op2);
extern four_t operator ^  (four_t op1, four_t op2);
extern four_t operator ~^ (four_t op1, four_t op2);
```

Kao primer neka posluži operator ~| (logička NI operacija) koji je definisan u funkcijskoj datoteci **op4.hi** na sledeći način.

```
four_t const nand_tab[4][4]= { //X    0    1    Z     - op2
                                 'X', '1', 'X', 'X',   // X  - op1
                                 '1', '1', '1', '1',   // 0  - op1
                                 'X', '1', '0', 'X',   // 1  - op1
                                 'X', '1', 'X', 'X'    // Z  - op1
                               };
four_t operator ~& (four_t op1,four_t op2) { return nand_tab[op1][op2]; }
```

5.2 Parazitne kapacitivnosti - atributi signala

I u sistemu sa četiri stanja modeliraćemo uticaj parazitnih kapacitivnosti na propagaciono kašnjenje logičkih kola korišćenjem mehanizma korisničkih atributa signala. Ovde ćemo, medjutim, korisničke atribute definisati primenom strukture tipa klase.

```
class four_att {
      double tcap;                  // total signal capacitance
  public :
      four_att() { tcap=0; }
      void add_cap (double c) { tcap += c;   }
      double get_tcap ()      { return tcap; }
};
typedef four_t @ four_att four_full;
```

Ukupna parazitna kapacitivnost signala (prema masi) je privatni parametar klase `four_att` i da bi joj se pristupilo definisane su funkcije `add_cap` (dodaje vrednost `c` atributu signala), i `get_tcap` (vraća vrednost ukupne kapacitivnosti). Operatorom `@` atributska klasa `four_att` pridružena je signalima tipa `four_t` i tako je definisan novi tip signala sa imenom `four_full`. Pre početka simulacije mora biti sračunata ukupna kapacitivnost svakog signala. Zato svaki modul sadrži proces sinhronizovan signalom `post_structural` u kome se korišćenjem funkcije `add_cap` predaju terminalne kapacitivnosti signalima za koje je modul vezan. Na primer, neka modul ima samo jedan ulaz označen sa `a` i jedan izlaz označen sa `y`, a ulazna i izlazna kapacitivnost imaju vrednost 1ns i 2ns respektivno. Odgovarajući proces imao bi sledeći izgled.

```
process post_structural {
  (@a)->add_cap(1ns);
  (@y)->add_cap(2ns);
}
```

Takodje je potrebno pri izračunavanju kašnjenja pročitati vrednost ukupne parazitne kapacitivnosti, što se može postići korišćenjem operatora `@` (da bi se dobio pokazivač na atributsku klasu) i funkcije `get_tcap`. Izraz `(@y)->get_tcap()` predstavlja kapacitivnost signala y.

5.3 Modeliranje osnovnih logičkih elemenata

Bazna klasa u biblioteci za sistem sa četiri stanja nosi ime `io4` i sadrži parametre modela A/D i D/A konvertora za logičke elemente. Definisani su konvertori `cmos_a2d` i `cmos_d2a`, kao i kod ranije predstavljenih logičkih biblioteka. Iz klase `io4` izvedeno je više modelskih klasa, za različite grupe gejtova. Logički elementi u sistemu sa četiri stanja svrstani su u sledeće grupe: standardni gejtovi, flip-flopovi, trostatički gejtovi, memorije, programabilne logičke strukture i

ostale logičke komponente. Za modeliranje flip-flopova izabran je isti model kašnjenja kao za standardne gejtove, tako da im je modelska klasa zajednička. Trostatički gejtovi imaju drugačije parametre kašnjenja, pa je za njih definisana posebna modelska klasa. Isto važi i za programabilne logičke strukture i memorije. Osim ovih logičkih komponenti, modelirana je i digitalna kapacitivnost, taktni generatori, pullup i pulldown otpornik. Nabrojane komponente nemaju posebne modelske klase, već primaju parametre modela preko akcionih parametara.

5.4 Standardni gejtovi

Standardni gejtovi i flip-flopovi grupisani su u modelsku klasu `gm4` sledećeg izgleda.

```
class gm4 : public io4 {            // gate models for 4-state system
      double skew;
      double incap, outcap;
      double tplh, tphl;
  public:
      gm4 ();
      double delay4(four_t ns, four_t os, double c, int hyb);
      friend module buf, inv, and, or, xor, nand, nor, nxor, mux, mux41, fa,
                         ffrs, ffd, ffjk, fft,
                         cffrs, cffd, cffdr, cffjk, cffjkr, cfft;
};
```

Konstruktor klase `gm4` ima sledeći izgled.

```
gm4::gm4 () {
  skew = 1nF_per_sec;
  incap = 1pF;
  outcap = 1pF;
  tplh = tphl = 10ns;
}
```

Modelska datoteka **model4.hi** sadrži difoltni model ove klase `gm4_default_model`.

```
model gm4::gm4_default_model { }
```

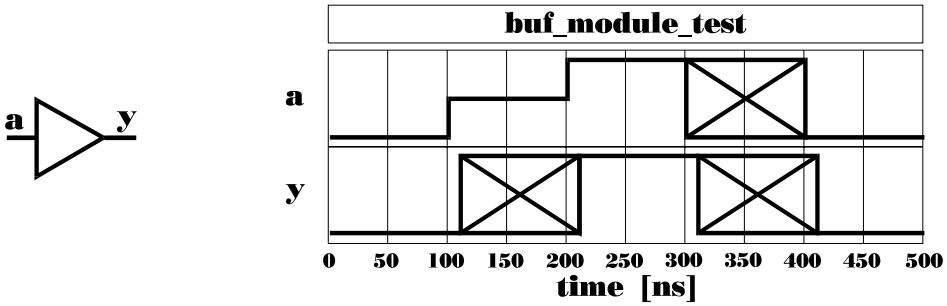
Stanje visoke impedanse se tumači kao neodredjeno kada se javi na ulazu logičkog elementa, tako da je korisno imati mehanizam konverzije `'Z'` stanja u `'X'` stanje. Odgovarajuća konverziona tabela definisana je u datoteci **op4.hi** na sledeći način.

```
                          //  X    0    1    Z
four_t const ZtoX_tab[4] = { 'X', '0', '1', 'X' };
```

Modul `buf` (logički element koji prosledjuje na izlaz stanje sa ulaza, uz izuzetak da se stanje `'Z'` konvertuje u `'X'`) opisan je u datoteci **gates4.hi** sledećim kodom.

```
module gm4::buf (four_full out y; four_full in a) {
  conversion { a2d = "cmos_a2d"; d2a = "cmos_d2a"; }
  action {
    process post_structural {
      (@a)->add_cap(this.incap);
      (@y)->add_cap(this.outcap);
    }
    process (a) {
      four_t tmp;
      tmp=ZtoX_tab[a];
      y <- tmp after this.delay4 (tmp, y, (@y)->get_tcap(), y->hybrid);
    }
  }
}
```

Modul `buf` prikazan je šematski na slici 5.1a. Talasni oblici sa slike 5.1b dobijeni su simulacijom tako što su na ulaz `a` dovedena logička stanja `'0'`, `'Z'`, `'1'` i `'X'`. Talasni oblik označen sa `y` predstavlja odziv kola. Korišćen je model `gm4_default_model`, definisan u datoteci **model4.hi**, kod koga su kašnjenja prednje i zadnje ivice jednaka i iznose 10ns.



a) *b)*

Slika 5.1 *Modul* `buf` *a) šematski prikaz b) rezultat simulacije*

5.5 Trostatički gejtovi

Osnovna karakteristika trostatičkih logičkih kola je da imaju (bar jedan) kontrolni ulaz kojim se kontroliše aktivnost izlaza kola. Ako je kolo neaktivno, izlaz kola je u stanju visoke impedanse, a u aktivnom stanju kolo obavlja svoju logičku funkciju.

Za trostatička kola potrebno je uvesti novi model kašnjenja u kome će se definisati kašnjenje od kontrolnog priključka do izlaza. Zato ova kola imaju drugačije parametre kašnjenja od standardnih gejtova, pa je za njihovo modeliranje definisana nova modelska klasa `tgm4`.

```
class tgm4 : public io4 {           // tri-state gates models
  private:
      double skew;
      double datacap, ctrlcap, outcap;
      double tp01, tp10, tp0Z, tpZ0, tp1Z, tpZ1;
  public:
      tgm4 ();
      double delay4z (four_t ns, four_t os, double c, int hyb);
      friend module     bufif1, bufif0, invif1, invif0, moveif1, moveif0,
                        tranif1, tranif0;
};
```

Parametri modelske klase `tgm4` su sledeći: `skew` je nagib porasta propagacionog kašnjenja sa kapacitivnim opterećenjem izlaza, `datacap`, `ctrlcap` i `outcap` su ulazne kapacitivnosti ulaznog, kontrolnog i izlaznog terminala, a parametri `tp01`, `tp10`, `tp0Z`, `tpZ0`, `tp1Z` i `tpZ1` predstavljaju kašnjenja gejta pri promeni stanja izlaza izmedju dva stanja sadržana u imenu parametra.

Konstruktor klase `tgm4` je definisan na sledeći način.

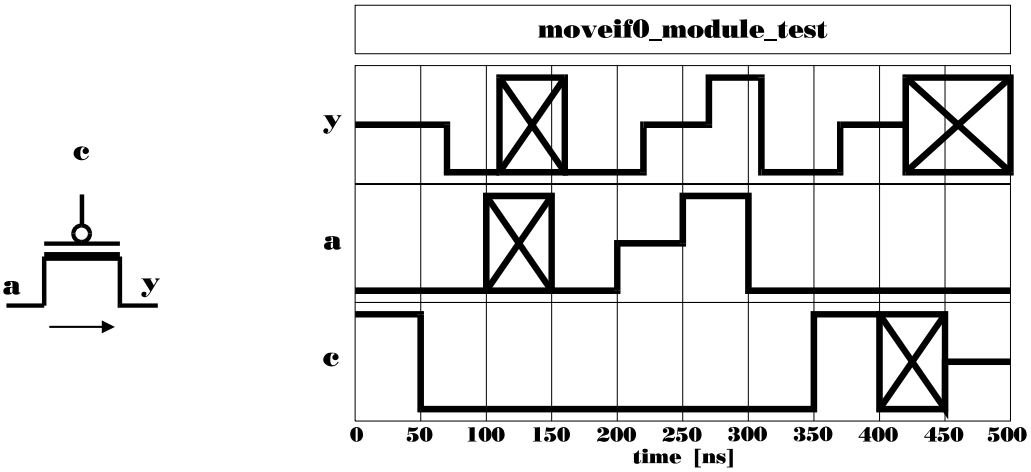
```
tgm4::tgm4 () {
  skew = 1nF_per_sec;
  datacap = 1pF;
  ctrlcap = 1pF;
  outcap = 1pF;
  tp01 = tp10 = 10ns;
  tp0Z = tpZ0 = tp1Z = tpZ1 = 20ns;
}
```

Modelska datoteka **model4.hi** sadrži difoltni model `tgm4_defualt_model` ove klase definisan na sledeći način.

```
model tgm4::tgm4_default_model { }
```

Modelirani su sledeći trostatički gejtovi: bafer sa invertujućim kontronim ulazom `bufif0`, bafer sa neinvertujućim kontrolnim ulazom `bufif1`, invertor sa invertujućim kontrolnim ulazom `invif0`, invertor sa neinvertujućim kontrolnim ulazom `invif1`, usmereni transmisioni gejt sa invertujućim kontrolnim ulazom `moveif0` i usmereni transmisioni gejt sa neinvertujućim kontrolnim ulazom `moveif1` (imena gejtova su usvojena po ugledu na slične komponente kod logičkog simulatora HILO). Za primer ćemo ovde navesti kod kojim je modeliran usmereni transmisioni gejt sa invertovanim kontrolnim ulazom. U pitanju je MOS tranzistor kod koga su na jednoj strani vezani samo izlazi logičkih elemenata, a na drugoj strani samo ulazi logičkih elemenata. Kod ovakve konfiguracije smer protoka signala je poznat, tako da nije neophodno modelirati dvosmerni prenos signala. Model ovakve komponente, modul `moveif0`, definisan je u datoteci **gates4.hi** na sledeći način.

```
module tgm4::moveif0 (four_full out y; four_full in a, c) {
  conversion { a2d = "cmos_a2d"; d2a = "cmos_d2a"; }
  action {
    process post_structural {
      (@a)->add_cap(this.datacap);
      (@c)->add_cap(this.ctrlcap);
      (@y)->add_cap(this.outcap);
    }
    process (a, c) {
      if (c=='0') {                  // enabled
        y <- a after this.delay4z (a, y, (@y)->get_tcap(), y->hybrid);
      }
      else if (c=='1') {             // dissabled
        y <- 'Z' after this.delay4z ('Z', y, (@y)->get_tcap(), y->hybrid);
      }
      else {                         // control input unknown ('X' or 'Z')
        y <- 'X' after this.delay4z ('X', y, (@y)->get_tcap(), y->hybrid);
      }
    }
  }
} // module tgm4::moveif0 ()
```



a) *b)*

Slika 5.2 *Modul* `moveif0` *a) šematski prikaz b) rezultat simulacije*

Treba uočiti da usmereni transmisioni gejt prenosi stanje visoke impedanse sa ulaza na izlaz, za razliku od baferskih i invertorskih trostatičkih drajvera. Šematski prikaz modula `moveif0` dat je na slici 5.2a, a logička funkcija se može analizirati sa slike 5.2b koja predstavlja rezultat simulacije istog modula. Na ulaz su dovedena sva stanja iz sistema sa četiri stanja, a korišćen je model `tgm4_default_model` definisan u datoteci **model4.hi** koji koristi difoltne vrednosti parametara kašnjenja postavljene u konstruktoru.

```
model tgm4::tgm4_default_model { }
```

5.6 Funkcije za kašnjenje

Modelska klasa `gm4` ima definisanu funkciju za kašnjenje `delay4` koja se ne razlikuje od slične funkcije u sistemu sa tri stanja, s obzirom da je koriste samo flip-flopovi i standardni gejtovi, a na njihovim izlazima ne može se pojaviti stanje visoke impedanse. Kod modeliranja trostatičkih gejtova, javlja se potreba za definisanjem nove funkcije za kašnjenje. Ova funkcija je označena sa `delay4z`, a definisana je u datoteci **op4.hi** na sledeći način.

```
double tgm4::delay4z (four_t ns, four_t os, double c, int hyb) {
  double tmp;

  if(os=='Z' && ns=='Z') {
    tmp = MAX(tp0Z+skew*c, tpZ0+skew*c);
    tmp = MAX(tmp, tpZ1+skew*c);
    tmp = MAX(tmp, tp1Z+skew*c);
    if (hyb) return tmp - MAX(rise_time, fall_time)/2.0;
    else return (tmp);
  }
  else if (os == 'Z')
    switch (ns) {
      case '0':
        if (hyb) return ((tpZ0+skew*c) - fall_time/2.0);
        else return (tpZ0+skew*c);
      case '1':
        if (hyb) return ((tpZ1+skew*c) - rise_time/2.0);
        else return (tpZ1+skew*c);
      case 'X':
        if (hyb) return ( MAX(tpZ1+skew*c, tpZ0+skew*c) -
                                          MAX(rise_time, fall_time)/2.0 );
        else return ( MAX(tpZ1+skew*c, tpZ0+skew*c) );
      default : warning("tgm4::delayZ() : This cannot happen!",1);
    }
  else if (ns == 'Z') {
    switch (os) {
      case '0':
        if (hyb) return ((tp0Z+skew*c) - rise_time/2.0);
        else return (tp0Z+skew*c);
      case '1':
        if (hyb) return (tp1Z+skew*c) - fall_time/2.0;
        else return (tp1Z+skew*c);
      case 'X':
        if (hyb)
          return (MAX(tp1Z+skew*c, tp0Z+skew*c) -
                                          MAX(rise_time, fall_time)/2.0);
        else return (MAX(tp1Z+skew*c, tp0Z+skew*c));
      default : warning("tgm4::delayZ() : This cannot happen! No way!",1);
    }
  }
  else {                // clasical delay function for 3-state system
    if(ns=='0') {
      if (hyb) return (tp10+skew*c - fall_time/2.0);
      else return (tp10+skew*c);
    }
    if(ns=='1') {
      if (hyb) return (tp01+skew*c - rise_time/2.0);
      else return (tp01+skew*c);
    }
                        // therefore, remains ns=='X'
    if(os=='1') {
      if (hyb) return (tp10+skew*c - fall_time/2.0);
      else return (tp10+skew*c);
    }
```

```
    if(os=='0') {
      if (hyb) return (tp01+skew*c - rise_time/2.0);
      else return (tp01+skew*c);
    }
    if(os=='X') {             // take the worst case
      if (hyb) return ( MAX(tp10+skew*c, tp01+skew*c) -
                                        MAX(rise_time, fall_time)/2.0 );
      else return ( MAX(tp10+skew*c, tp01+skew*c) );
    }
  }
}
```

Funkcija `delay4z` prima dve pomenljive tipa `four_t`: `ns` je novo stanje signala, a `os` je staro stanje signala. Preko parametra c prenosi se kapacitivno opterećenje izlaza, a parametar `hyb` donosi informaciju o tome da li signal ima hibridni ili čisto digitalni aspekt. Pri neodredjenim prelazima funkcija vraća maksimalnu vrednost kašnjenja, što predstavlja najgori slučaj.

5.7 Rezolucione funkcije

U biblioteci za sistem sa četiri stanja definisano je pet rezolucionih funkcija. Njihove deklaracije iz datoteke **ss4.h** su sledeće.

```
extern four_t wiredAND     (const four_t*, int*);
extern four_t wiredOR      (const four_t*, int*);
extern four_t resolution4  (const four_t*, int*);
extern four_t pullUP       (const four_t*, int*);
extern four_t pullDOWN     (const four_t*, int*);
```

Funkcije `wiredAND` i `wiredOR` koriste se za razrešenje konflikta pri modeliranju žičane logike u digitalnim kolima. Funkcija `resolution4` je osnovna funkcija za razrešenje konflikta na magistrali, a funkcije `pullUP` i `pullDOWN` odredjuju stanje magistrale sa pullup i pulldown otpornikom.

5.7.1 Rezolucija žičane logike

Funkcije rezolucije žičane logike `wiredAND` i `wiredOR` realizovane su slično kao i za sisteme sa dva, odnosno tri stanja korišćenjem predefinisanih operatora `&` i `|`. Ovi operatori uključuju u sebi manipulaciju stanjem visoke impedanse koje se javlja kao razlika u odnosu na sisteme sa dva i tri stanja. Funkcija za vezu žičano I ima sledeći izgled.

```
four_t wiredAND (const four_t *drv, int *flag) {
  int n = lengthof drv;
  four_t result;

  result = drv[0];
  for (int i=1; i<n; i++) {
    result = result & drv[i];
    if (result=='0') break;
  }
  return result;
}
```

Funkcije `wiredAND` i `wiredOR` ne treba koristiti za modeliranje žičane veze izlaza trostatičkih kola, jer one stanje visoke impedanse tumače kao neodredjeno. Ako neki od drajvera signala može da dodje u stanje `'Z'`, treba koristiti funkcije rezolucije na magistrali `resolution4`, `pullUP` i `pullDOWN`.

5.7.2 Funkcija rezolucije na magistrali

Najopštija funkcija rezolucije na magistrali `resolution4` opisana je sledećim kodom u datoteci **op4.hi**.

```
four_t resolution4 (const four_t *drv, int *flag) {
  int n = lengthof drv;
  four_t result;

  result = drv[0];
  for (int i=1; i<n; i++) {
    result = res_tab[result][drv[i]];
    if (result=='X') break;
  }
  if (result=='X') {
    int counter=0;
    for (int i=0; i<n; i++) { if (drv[i]=='X') counter++; }
    if (counter) *flag=2;           // potential conflict
    else *flag=1;                   // conflict for sure
  }
  return (result);
}
```

Tabela označena sa `res_tab` koju koristi funkcija `resolution4` definisana je na sledeći način.

```
four_t const res_tab[4][4] = { //X    0    1    Z   - op2
                                  'X', 'X', 'X', 'X',   // X  - op1
                                  'X', '0', 'X', '0',   // 0  - op1
                                  'X', 'X', '1', '1',   // 1  - op1
                                  'X', '0', '1', 'Z'    // Z  - op1
                               };
```

Neodredjeno stanje na magistrali može da se javi kao rezultat konflikta (bar jedan drajver u stanju `'1'` i bar jedan drajver u stanju `'0'`) ili kao rezultat neodredjenog stanja na izlazu nekog od drajvera. Neodredjeno stanje je dominantno u odnosu na sva ostala stanja i jedan drajver u `'X'` stanju nameće `'X'` stanje magistrali. U slučaju konflikta, vrednost `*flag` treba da se postavi na 1, što je znak simulacionoj mašini da javi poruku o konfliktu. U drugom slučaju postoji potencijalna mogućnost da je došlo do konfliktne situacije, pa se vrednost `*flag` postavlja na 2, tako da simulaciona mašina prijavljuje korisniku pojavu potencijalnog konflikta. Vrednost `*flag` je difoltno 0 pri pozivu rezolucione funkcije, pa je nije potrebno menjati u slučaju da nema konflikta.

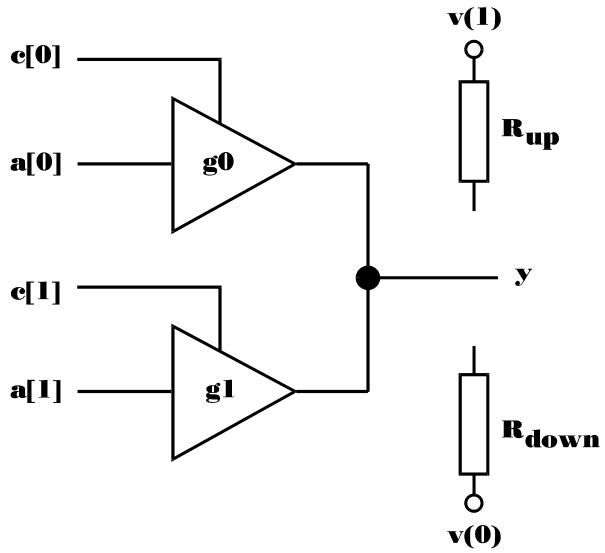
5.7.3 Funkcija rezolucije sa pullup i pulldown otpornikom

Pri projektovanju digitalnih kola vodi se računa da stanje u svim tačkama kola bude u svakom trenutku odredjeno. U normalnom radnom režimu, uvek je jedan od drajvera magistrale aktivan. Ako je potrebno da u nekim uslovima svi drajveri magistrale odu u stanje visoke impedanse, neophodno je osigurati da magistrala predje u neko odredjeno stanje. Naime, zbog velike parazitne kapacitivnosti magistrala obično neko vreme drži prethodno stanje, a zatim prelazi u neodredjeno stanje. Neodredjeno stanje na magistrali može stvoriti probleme u kolu ako nije predvidjeno blokiranje svih logičkih elemenata čiji su ulazi vezani za magistralu. Zato se obično magistrala spreže na napon napajanja (logička jedinica) ili masu (logička nula) pomoću otpornika relativno velike vrednosti (pullup, odnosno pulldown otpornik). Na taj način je obezbedjeno da magistrala predje u odredjeno stanje posle konačnog vremena koje zavisi od vrednosti ekvivalentne kapacitivnosti i ekvivalentne otpornosti. Ako je ovo vreme relativno kratko u odnosu na propagaciona kašnjenja logičkih elemenata kola, moguće ga je zanemariti i prelaz modelirati kao

trenutan pomoću rezolucione funkcije. Funkcije `pullUP` i `pullDOWN` obezbedjuju trenutni prelazak magistrale u stanje `'1'`, odnosno `'0'`, kada svi drajveri magistrale odu u stanje visoke impedanse. Funkcija `pullUP` ima sledeći izgled.

```
four_t pullUP (const four_t *drv, int *flag) {
  int n = lengthof drv;
  four_t result;
  int active=0;

  for (int i=0; i<n; i++) { if (drv[i]!='Z') active=1; break; }

  if (active) {                      // standard resolution procedure
    result = drv[0];
    for (int i=1; i<n; i++) {
      result = res_tab[result][drv[i]];
      if (result=='X') break;
    }
    if (result=='X') {
      int counter=0;
      for (int i=0; i<n; i++) { if (drv[i]=='X') counter++; }
      if (counter) *flag=2;          // potential conflict
      else *flag=1;                  // conflict for sure
    }
  }
  else {                             // all drivers in 'Z' state
    result = '1';                    // pull up the bus
  }
  return (result);
}
```

Na sličan način realizovana je i funkcija `pullDOWN` koja postavlja magistralu u stanje logičke nule kada svi drajveri predju u stanje visoke impedanse.



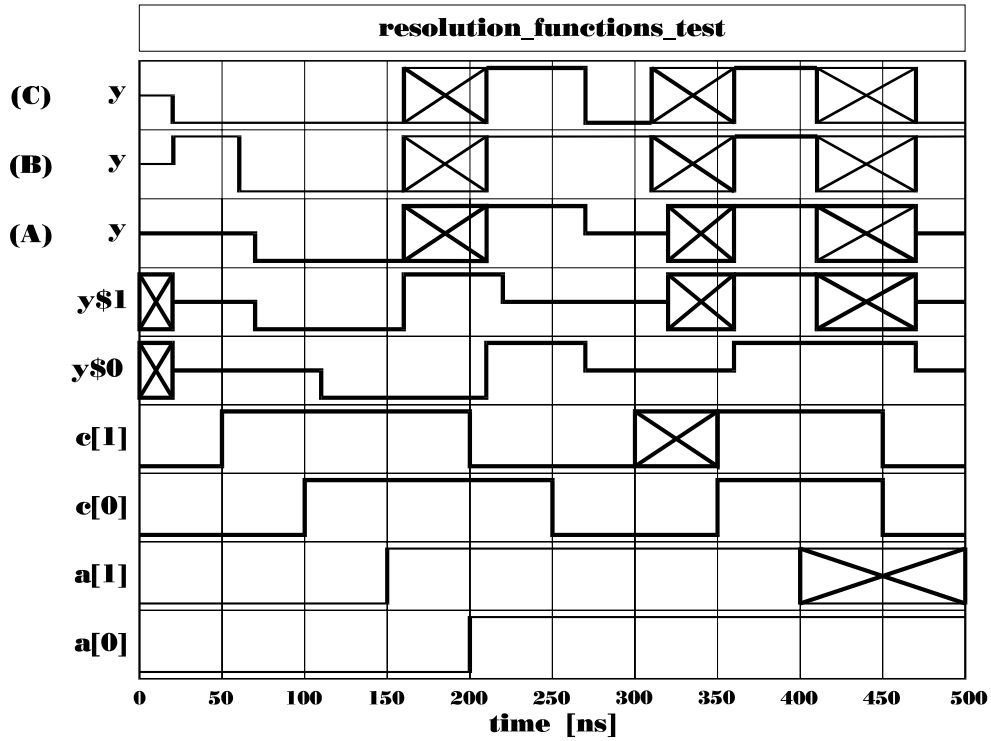
Slika 5.3 *Magistrala - sprezanje izlaza trostatičkih kola*

Kao ilustracija upotrebe ovih funkcija poslužiće nam simulacija kola sa slike 5.3. Izlazi dva trostatička bafera spojeni su u istu tačku koja time postaje magistrala (često se koristi i drugi naziv - bus). Ako magistrala nije otporno spregnuta sa logičkom jedinicom ili logičkom nulom, koristićemo rezolucionu funkciju `resolution4` za razrešenje konflikta. U suprotnom ćemo koristiti funkciju `pullUP` ili `pullDOWN`. Opis kola za simulaciju je sledeći.

```
# include "ss4.h"
library model4, gates4, op4;
root resolution_functions_test() {
  module tgm4::bufif1 g0, g1;
```

```
  signal four_full a[2]="00", c[2]="00";
  signal four_full:resolution4 y='Z';           // A
  //signal four_full:pullUP y='Z';              // B
  //signal four_full:pullDOWN y='Z';            // C

  g0 (y, a[0], c[0]) model=tgm4_default_model;
  g1 (y, a[1], c[1]) model=tgm4_default_model;

  timing { tstop = 500ns; }
  out {
    signal four_full a, c;
    signal four_full inout y;
  }
  action {
    process initial {
      c <- "01" after  50ns, "11" after 100ns, "10" after 200ns,
           "00" after 250ns, "0X" after 300ns, "11" after 350ns,
           "00" after 450ns;
      a <- "01" after 150ns, "11" after 200ns, "1X" after 400ns;
    }
  }
}
```

Slika 5.4 predstavlja rezultat simulacije. Na slici su nacrtani talasni oblici napona na ulaznim terminalima bafera (`a[0]` i `a[1]`), na kontrolnim terminalima bafera (`c[0]` i `c[1]`) i na magistrali (signal `y`). Sa `y$0` i `y$1` označeni su talasni oblici koje daju baferi na izlazu (dobijeni na grafiku specificiranjem usmerenja `inout` za signal `y` u `out` naredbi), a `y` predstavlja rezultat koji daje funkcija rezolucije njihovim kombinovanjem. Sa A je označen rezultujući talasni oblik na magistrali kada je korišćena rezoluciona funkcija `resolution4`, sa B rezultat koji daje funkcija `pullUP`, a sa C rezultat koji daje funkcija `pullDOWN`. Vidimo da funkcije `pullUP` i `pullDOWN` pretvaraju stanje visoke impedanse iz slučaja označenog sa A u stanje `'1'`, odnosno `'0'`.



Slika 5.4 *Rezultat simulacije kola sa slike 5.3 (A - funkcija* `resolution4`*, B - funkcija* `pullUP`*, C - funkcija* `pullDOWN`*)*

5.8 Modeliranje pullup i pulldown otpornika

Ako je vremenska konstanta prelaznog procesa kojim magistrala odlazi u odredjeno stanje velika u odnosu na propagaciona vremena logičkih komponenti kola, mora se primeniti drugačiji pristup u modeliranju. Tada se na magistrali mora zadržati prethodno stanje odredjeno vreme koje je parametar modela magistrale, a zatim treba magistrali nametnuti stanje `'1'` u slučaju pullup otpornika, odnosno stanje `'0'` u slučaju pulldown otpornika. Za ovu namenu razvijeni su moduli `pull_up` i `pull_down` koji predstavljaju pullup i pulldown otpornike. Deklaracije modula `pull_up` i `pull_down` imaju sledeći izgled.

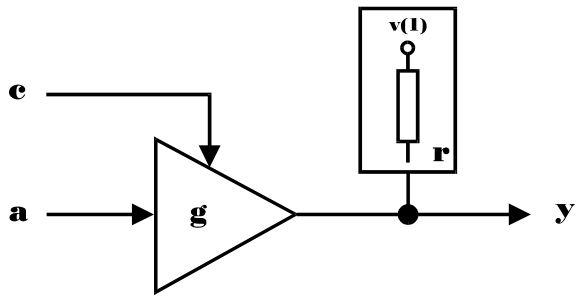
```
module pull_up   (signal four_full inout y) action (double delay);
module pull_down (signal four_full inout y) action (double delay);
```

Akcioni parametar `delay` predstavlja vreme držanja stanja na magistrali i srazmeran je vremenskoj konstanti prelaznog procesa. Moduli imaju samo jedan pristupni priključak `y` koji je usmerenja `inout`, što znači da se preko njega očitava stanje magistrale i nameće stanje magistrali.

Modul `pull_up` definisan je u datoteci **gates4.hi** sledećim kodom.

```
module pull_up (signal four_full inout y) {
  action (double delay) {
    pullup: process {
      int initialized=0;
      four_full last_y='Z';

      if (!initialized) { y <- 'Z'; initialized=1; }

      wait (y);
      if (y != 'Z') last_y=y;             // memorize active bus state
      if (y=='Z') {                       // bus went inactive
        y <- transport last_y;            // hold last state until delay expires
        wait for delay;
        y <- (four_t)'1';                 // pullup the line
      }
      else {
        y <- 'Z';                         // disconnect from line
      }
    }
  }
}
```

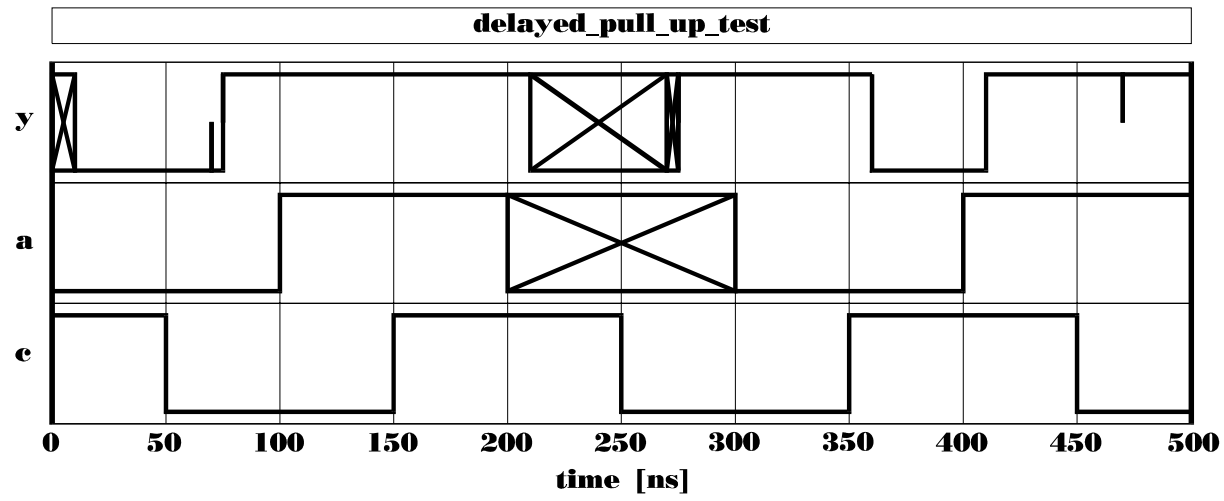
Inicijalno stanje drajvera signala `y` je `'X'`, pa je neophodno inicijalizirati njegovu vrednost na neaktivno `'Z'` stanje. Modul `pull_up` je neaktivan sve dok je magistrala (odnosno signal `y`) u nekom od aktivnih stanja (`'0'`, `'1'` ili `'X'`). Da ne bi uticao na stanje magistrale za vreme neaktivnosti, njegov izlaz `y` mora se održavati u stanju `'Z'`. Za vreme neaktivnosti, medjutim, modul `pull_up` "osluškuje" magistralu i u promenljivoj `last_y` pamti poslednje aktivno stanje magistrale. Kada svi drajveri magistrale odu u stanje `'Z'`, funkcija rezolucije postavlja magistralu u stanje `'Z'`, na šta se modul `pull_up` aktivira i vraća magistralu u prethodno aktivno stanje. Zatim proces čeka da istekne zadato kašnjenje `delay`, i šalje stanje logičke jedinice na izlaz.



Slika 5.5 *Kolo sa pullup otpornikom*

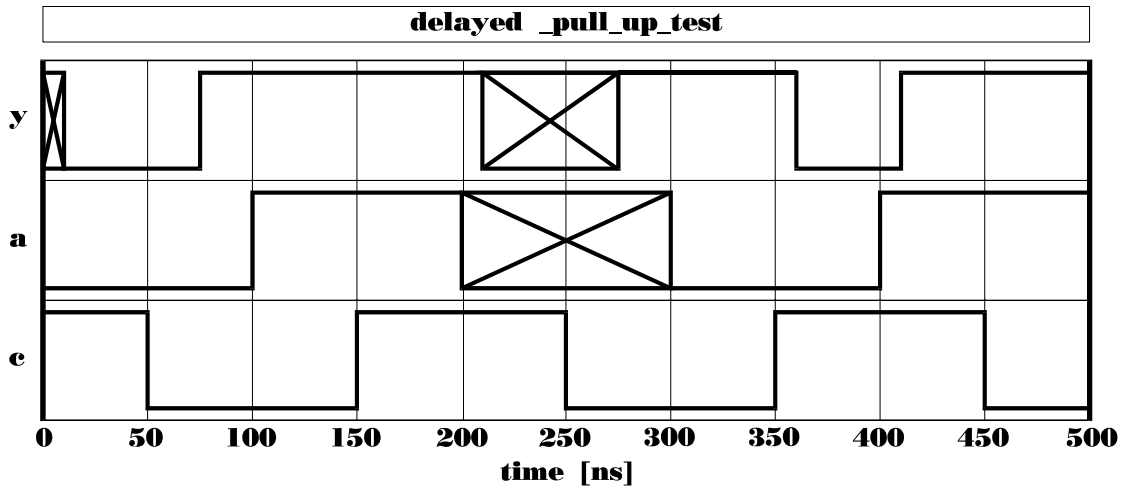
Verifikovaćemo razvijeni model simulacijom kola sa slike 5.5. Za izlaz trostatičkog bafera vezan je pullup otpornik. U ovom kolu izlaz ide u stanje logičke jedinice ako kontrolni ulaz bafera dodje u stanje logičke nule (na izlazu bafera pojavljuje se stanje visoke impedanse, tako da pullup otpornik preuzima kontrolu stanja izlaza `y`). Neka je bafer opisan modelskom karticom `tgm4_default_model`, a prelazak izlaznog signala u stanje `'1'` pod uticajem pullup otpornika neka kasni 5ns. Opis kola za simulaciju je sledeći.

```
# include "ss4.h"
library model4, gates4, op4;

root delayed_pull_up_test () {
  module tgm4::bufif1 g;
  module pull_up r;

  signal four_full a='0', c='1';
  signal four_full:resolution4 y='0';

  r (y) action (5ns);
  g (y, a, c) model=tgm4_default_model;

  timing { tstop = 500ns; }
  out { signal four_full c, a, y; }

  action {
    process initial {
      c    <- '0' after  50ns, '1' after 150ns,
              '0' after 250ns, '1' after 350ns,
              '0' after 450ns;
      a    <- '0' after   0ns, '1' after 100ns,
              'x' after 200ns, '0' after 300ns, '1' after 400ns;
    }
  }
}
```



Slika 5.6 *Rezultat simulacije kola sa slike 5.5*

Rezultat simulacije prikazan je na slici 5.6. Uočavamo pikove nultog trajanja u trenucima kada magistrala pokušava da predje u stanje visoke impedanse, ali modul `pull_up` to ne dozvoljava (u istom trenutku vraća magistralu u prethodno aktivno stanje). Ovi kvazi-dogadjaji mogu se eliminisati postprocesiranjem izlazne datoteke pomoću programa NZD (No Zero Delay) koji uklanja sve dogadjaje koji se dešavaju sa nultim kašnjenjem osim poslednjeg. Rezultat postprocesiranja prikazan je na slici 5.7.



Slika 5.7 *Rezultat simulacije kola sa slike 5.5 posle postprocesiranja programom NZD*

5.9 A/D i D/A konverzija

O A/D i D/A konverziji u sistemu sa četiri stanja može se naći u [Gloz94b], tako da ovde nećemo posebno navoditi modele pomenutih konvertora `io4::cmos_a2d` i `io4::cmos_d2a`. Pomenimo samo da je promenu izlazne otpornosti modela D/A konvertora potrebno modelirati u linearnom obliku za prelaze izmedju stanja `'0'`, `'1'` i `'X'`, a u eksponencijalnom obliku za prelaze sa i na stanje `'Z'`. Ovo je neophodno zato što je vrednost izlazne otpornosti u stanju `'Z'` za više redova veličine veća od vrednosti izlazne otpornosti u drugim stanjima. Odgovarajući model jednostavno je razviti modifikacijom funkcije `evaluate_rramp`. Takodje je moguće od konvertora zahtevati i neki drugi osim linearnog oblika promene vrednosti izlaznog napona, što se postiže modifikacijom funkcije `evaluate_vramp`.

5.10 Modeliranje memorija

Memorije su sekvencijalni logički elementi relativno visokog stepena složenosti koji "pamte" digitalne informacije. Dele se u dve grupe: ROM (read only memory) i RAM (random access memory). ROM dozvoljava samo čitanje uskladištene informacije, a RAM se odlikuje slobodnim pristupom skladišnom prostoru - upis i čitanje informacija.

Memorije se mogu modelirati i u jednostavnijim sistemima stanja (dva i tri stanja), ali smo odabrali sistem sa četiri stanja za prezentaciju njihovih modela s obzirom da se memorijski čipovi selektuju posebnim kontrolnim ulazom, dok su u neselektovanom stanju izlazi u stanju visoke impedanse. Ovo omogućuje da se kratkospoje izlazi više memorijskih čipova, čime se dobijaju memorije većeg kapaciteta.

Da bi smo modelirali memorijske module deklarisali smo prvo klasu `mem4` koja sadrži neophodne strukture podataka. Deklaracija ove klase nalazi se u datoteci **ss4.h** i ima sledeći izgled.

```
class mem4 {                          // data for memory structures
      int is_set;
      int n_addr;                     // number of address lines
      int word_len, word_No;          // word length and memory capacity
      four_t **memory_matrix;         // storage space
      four_t *outword;                // temporary register for output word
  public :
      mem4 (int, int, int, const char *);
      ~mem4 ();
      int set (const char *);         // read setup file
      void show ();                   // print rom contents to the screen
```

```
      four_t *pick_word (four_t *); // rom4/ram4 function: read word
      four_t *set_outword(four_t);  // ram4 function: ram4 dissabled
      void load_unknown(four_t*);   // ram4 function: write 'x' at address
      void load_word(four_t*,four_t*);// ram4 function: write new word to ram4
      int four_t2int (four_t *);
};
```

Konstruktor klase `mem4` rezerviše potreban memorijski prostor i postavlja inicijalne vrednosti promenljivih. Neophodno je poznavati željenu veličinu memorije pre alociranja prostora za promenljive iz klase `mem4`, tako da konstruktor klase prima parametre `n_addresses` (broj adresnih linija), `w_No` (broj reči koje sačinjavaju memoriju), `wlen` (broj bitova u jednoj memorijskoj reči) i `fname` (ime datoteke iz koje se učitava početni sadržaj memorijske matrice). Posle alociranja promenljive `memory_matrix`, konstruktor poziva funkciju `set` koja čita *setup* datoteku i zatim štampa na ekranu učitani inicijalni sadržaj memorije korišćenjem funkcije `show`. Usvojena je konvencija za označavanje *setup* datoteka, one nose ekstenziju **.mem**, ali to nije pravilo. Funkcija `set` realizovana je tako da u *setup* datoteci podržava komentare tipa `/* ... */` kao i linijske komentare koji počinju znakom `//`, a završavaju se krajem linije. Takodje, prelazak u novi red, tabulatori i prazan prostor se ignorišu. Konstruktor je definisan u datoteci **op4.hi** na sledeći način.

```
mem4::mem4 (int n_addresses, int w_No, int wlen, const char *fname) {
  int i;

  is_set = 0;                          // still not set
  if ((word_No = w_No) < 1) warning ("incorrect number of words in mem4", 1);
  if ((word_len = wlen) < 1) warning ("incorrect word length in mem4", 1);
  if ((n_addr = n_addresses) < 1)
    warning ("incorrect number of address lines in mem4", 1);

  if (!(outword = (four_t *) calloc (word_len, sizeof(four_t)) ) )
    warning ("no room for outword in mem4", 1);

  if (!(memory_matrix = (four_t **)calloc(word_No, sizeof(four_t *))))
      warning ("no room for memory_matrix in mem4", 1);
  for (i = 0; i < word_No; i++) {
    if (!(memory_matrix[i] = (four_t *)calloc(word_len, sizeof(four_t))))
      warning ("no room for memory_matrix[i] in mem4", 1);
  }

  if (set (fname)) warning ("errors during mem4 initialization", 1);
  show();
}
```

Klasa `mem4` ima, naravno, i odgovarajući destruktor `~mem4` koji nećemo posebno navoditi. Interesantnije su funkcije `pick_word`, `set_outword`, `load_word` i `load_unknown`. Funkcija `pick_word` vraća sadržaj odredjene memorijske lokacije u obliku vektora dužine `word_len`. Njen izgled je sledeći.

```
four_t * mem4::pick_word (four_t *addr) {
  int i, int_addr;

  int_addr = four_t2int(addr);
  if (int_addr >= 0) {
    for (i=0; i < word_len; i++) { outword[i] = memory_matrix[int_addr][i]; }
  }
  else {                               // 'x'es in addr
    warning ("address lines contain 'x', don't know what word to pick");
    for (i=0; i < word_len; i++) {
      outword[i] = 'x';
    }
  }
  return outword;
}
```

Funkcija `four_t2int` pretvara vektor tipa `four_t` u ceo broj. Da bi konverzija bila moguća neophodno je da vektor sadrži samo stanja `'0'` i `'1'`, jer stanja `'X'` i `'Z'` nemaju binarno tumačenje. U slučaju da vektor sadrži neko od ovih stanja, funkcija `four_t2int` vraća `-1`. Funkcija `four_t2int` definisana je sledećim kodom u datoteci **op4.hi**.

```
int mem4::four_t2int (four_t ad[]) {
  int i, tmp, unknown;

  if (lengthof ad != word_len)
    warning ("incorrect address length in four_t2int", 1);
  tmp = 0;
  unknown = 0;
  for (i = 0; i<lengthof(ad); i++) {
    tmp *= 2;
    if (ad[i] == '1') tmp++;
    if (ad[i] == 'x' || ad[i] == 'z') { unknown=1; break; }
  }
  if (unknown) return -1; else return tmp;
}
```

Funkcija `set_outword` vraća vektor dužine `word_len` koji na svim lokacijama sadrži zadato logičko stanje. Ova funkcija je korisna kada je potrebno izlazne linije podataka postaviti u stanje visoke impedanse `'Z'` ili u neodredjeno stanje `'X'`.

```
four_t * mem4::set_outword (four_t f_s) {
  int i;

  for (i=0; i < word_len; i++) { outword[i] = f_s; }
  return outword;
}
```

Slična je namena funkcije `load_unknown` koja u reč na zadatoj adresi učitava neodredjeno logičko stanje `'X'` (recimo kada read/write kontrolna linija RAM-a dodje u neodredjeno stanje).

```
void mem4::load_unknown(four_t ad[]) {
  int i, int_ad;

  int_ad = four_t2int(ad);
  if (int_ad >= 0) {
    for (i=0; i < word_len; i++) { memory_matrix[int_ad][i] = 'x'; }
  }
  else warning ("address lines contain 'x', don't now how to load word", 1);
}
```

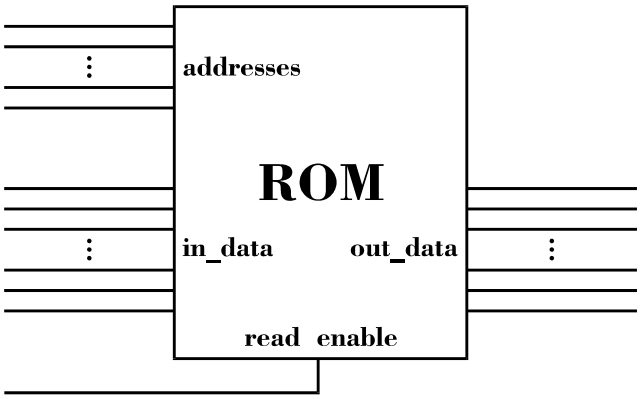
I na kraju, funkcija `load_word` služi za upis novog sadržaja u memorijsku reč na zadatoj adresi.

```
void mem4::load_word(four_t ad[], four_t inl[]) {
  int i, int_ad;

  if (lengthof inl != word_len)
    warning ("length of input word doesn't match memory word length", 1);
  int_ad = four_t2int(ad);
  if (int_ad >= 0) {
    for (i=0; i<word_len; i++) { memory_matrix[int_ad][i] = inl[i]; }
  }
  else warning ("address lines contain 'x', don't now how to load word", 1);
}
```

Kad imamo ovako definisanu klasu `mem4`, možemo pristupiti modeliranju ROM i RAM komponenti.

5.10.1 Modeliranje ROM-a

Šematski simbol memorije tipa ROM prikazan je na slici 5.8. ROM ima adresne linije, ulazne linije podataka i izlazne linije podataka, kao i kontrolni ulaz za dozvolu čitanja. Eventualno, ROM može da ima i druge kontrolne ulaze (recimo chip select), ali ćemo se mi zadržati na strukturi sa slike 5.8, uz napomenu da je obogaćivanje modela ROM-a koji će biti razvijen novim ulaznim priključcima relativno jednostavno.



Slika 5.8 *ROM*

Razne vrste ROM struktura grupisaćemo u jednu modelsku klasu sa imenom `rom4`. Ova klasa deklarisana je u datoteci **ss4.h** na sledeći način.

```
class rom4 : public io4 {           // read only memory model class
      double tp;
      double tplh, tphl;
      double skew;
      double incap, outcap;
      int addr_lines_No;            // number of address lines
      int w_length;                 // memory word length
      int capacity;                 // number of memory words
      char *filename;               // rom setup file
  public :
      rom4 ();
      ~rom4 ();
      double get_tr() { return io4::rise_time; }
      double get_tf() { return io4::fall_time; }
      friend module     srom, rom;
};
```

U okviru biblioteke sa četiri stanja definisana su dva ROM modula: `srom` i `rom`. Njihove deklaracije iz datoteke **ss4.h** imaju sledeći izgled.

```
module rom4::srom (four_full in read_en, addr[]; four_full out out_lines[]);
module rom4::rom  (four_full in read_en, addr[]; four_full out out_lines[]);
```

Modul `srom` predstavlja ROM sa jednostavnim modelom kašnjenja kod koga je kašnjenje prednje i kašnjenje zadnje ivice signala na izlazu jednako. Parametar `tp` u modelskoj klasi `mem4` predstavlja vreme pristupa memoriji. Kod kojim se definiše modul `srom` nalazi se u datoteci **gates4.hi**.

```
module rom4::srom(four_full in read_en, addr[]; four_full out out_lines[]) {
  action {
    process (read_en, addr) {
      mem4 r(this.addr_lines_No, this.capacity, this.w_length, this.filename);

      if (read_en->event && (read_en=='x' || read_en=='z')) {
      warning ("read enable went unknown, out_lines go 'X'\n", now);
        out_lines <- r.set_outword('X') after this.tp;
      }
```

```
      if (read_en->event && read_en=='0') {
      warning ("rom dissabled, out_lines go 'Z'\n", now);
        out_lines <- r.set_outword('Z') after this.tp;
      }
      else if (read_en == '1') {
        //printf ("t=%g : rom read enabled\n", now);
        out_lines <- r.pick_word(addr) after this.tp;
      }
      else {
        //printf ("t=%g : rom address changes, but read dissabled or "
                  //"read_en unknown\n", now);
      }
    }
  }
} // module rom4::srom
```

Modul `srom` ima samo jedan proces osetljiv na promene na adresnim linijama `addr` i na kontrolnom ulazu za dozvolu čitanja `read_en`. U okviru ovog procesa najpre se deklariše objekat `r` tipa `mem4`, pri čemu se konstruktoru klase `mem4` prosledjuju dimenzije potrebnog memorijskog polja (parametri modelske klase `rom4`). U nastavku se definiše logička funkcija ROM-a. Čitanje je dozvoljeno ako je na ulazu `read_en` logička jedinica i tada se na izlazne linije podataka `out_lines` postavlja adresirana memorijska reč (koristi se funkcija `pick_word`). Ako na ulaz `read_en` dospe neodredjeno stanje `'X'` ili stanje visoke impedanse `'Z'`, izlazne linije podataka se postavljaju u neodredjeno stanje (koristi se funkcija `set_outword`), jer izlazi mogu biti ili u `'Z'` stanju (ako bi `read_en` bio u stanju `'0'`) ili u nekim odredjenim stanjima koja zavise od adresirane memorijske reči (ako bi `read_en` bio u stanju `'1'`). Dok je ulaz `read_en` u stanju `'0'`, čitanje je zabranjeno i izlazne linije se postavljaju u stanje visoke impedanse `'Z'`.

Na sličan način modeliran je i modul `rom`, s tom razlikom da je uvodjenjem baferskih stepena na izlazu omogućeno korišćenje rise/fall modela kašnjenja (parametri `tplh` i `tphl` u modelskoj klasi `mem4`). Treba napomenuti da baferski moduli nisu klasični baferi, s obzirom da oni treba da prenesu i stanje visoke impedanse sa svog ulaza na izlaz. To su, dakle, jednosmerni transmisioni gejtovi koji su uvek u stanju provodjenja. Model kašnjenja bi se, dalje, mogao obogatiti dodavanjem parametara kašnjenja karakterističnih za trostatička kola (kašnjenja koja uzimaju u obzir i prelaz u stanje visoke impedanse i sa njega).

```
module rom4::rom(four_full in read_en, addr[]; four_full out out_lines[]) {
  module sbuf outbuf;
  signal four_full srom_out[auto]="z";

  action {
    delay: process structural {           // output buffers introduce delays
      signal four_full srom_out(this.w_length);

      for (int i=0; i < this.w_length; i++) {
        clone outbuf[i] (out_lines[i], srom_out[i]) action(this.tplh,
                        this.tphl, this.skew, this->get_tr(), this->get_tf());
      }
    }

    srom_function: process (read_en, addr) {
      mem4 r(this.addr_lines_No, this.capacity, this.w_length, this.filename);

      if (read_en->event && (read_en=='x' || read_en=='z')) {
      warning ("read enable went unknown, out_lines go 'X'\n", now);
        srom_out <- r.set_outword('X') after this.tp;
      }
      if (read_en->event && read_en == '0') {
      warning ("rom dissabled, out_lines go 'Z'\n", now);
        srom_out <- r.set_outword('Z') after this.tp;
      }
      else if (read_en== '1') {
        srom_out <- r.pick_word(addr) after this.tp;
```

```
      }
    }
  }
} // module rom4::rom
```

U razvijenim modulima nisu korišćeni parametri `incap` i `outcap`. Ulazne i izlazne kapacitivnosti ne mogu se ovde jednostavno dodati u atribut ulaznih i izlaznih signala, jer su ulazni i izlazni signali vektori. AleC++ ne podržava atribute pojedinih skalarnih signala u okviru vektorskih signala, već je atribut jedinstven za ceo vektor. Ovo znači da nije moguće primeniti generalni pristup koji je ranije korišćen, ali je moguće za konkretne memorijske module dopuniti postojeće modele `rom` i `srom` tako da se vektorski ulazni i izlazni signali zamene skalarnim. Tada je moguće iskoristiti parametre `incap` i `outcap` iz modelske klase `rom4` da bi se ugradio uticaj parazitnih kapacitivnosti na kašnjenja u kolu.

5.10.2 Primer simulacije sa ROM-om

Testiraćemo validnost razvijenih modela ROM-a na primeru množača dva dvobitna broja. Ovakav množač projektuje se tako što brojeve koje množimo dovodimo na adresne ulaze ROM-a, a na izlaznim linijama dobijamo vrednost proizvoda. Proizvod dva dvobitna broja je četvorobitni broj, tako da zahtevana memorija treba da sadrži reči dužine 4 bita. Takodje, potrebne su 4 adresne linije (dva puta po dva bita). Tablicu množenja upisaćemo u sadržaj ROM-a koji mora sadržati najmanje 16 memorijskih reči. Izgled opisa kola za simulaciju je sledeći.

```
# include "ss4.h"
library model4, gates4, op4;
root module rom_based_simple_multiplier_test () {
  module rom4::rom rom_chip;
  signal four_full addr[4]="0111", data[4], readenable='0';

  rom_chip (readenable, addr, data) model = rom4_multiplier_model;

  timing { tstop = 2000ns; }
  out {
      signal four_full p_3_msb      { return data[0]; };
      signal four_full p_2          { return data[1]; };
      signal four_full p_1          { return data[2]; };
      signal four_full p_0_lsb      { return data[3]; };

      signal four_full x2_lsb       { return addr[3]; };
      signal four_full x2_msb       { return addr[2]; };
      signal four_full x1_lsb       { return addr[1]; };
      signal four_full x1_msb       { return addr[0]; };

      signal four_full readenable;
  }
  action {
    process initial {
      readenable <- '1' after  50ns, '0' after 1800ns, 'x' after 1900ns;
      addr <-     "0000" after 100ns,
                  "0001" after 200ns,
                  "0010" after 300ns,
                  "0011" after 400ns,
                  "0100" after 500ns,
                  "0101" after 600ns,
                  "0110" after 700ns,
                  "0111" after 800ns,
                  "1000" after 900ns,
                  "1001" after 1000ns,
                  "1010" after 1100ns,
                  "1011" after 1200ns,
                  "1100" after 1300ns,
```

```
                    "1101" after 1400ns,
                    "1110" after 1500ns,
                    "1111" after 1600ns,
                    "0000" after 1700ns;
    }
  }
}
```

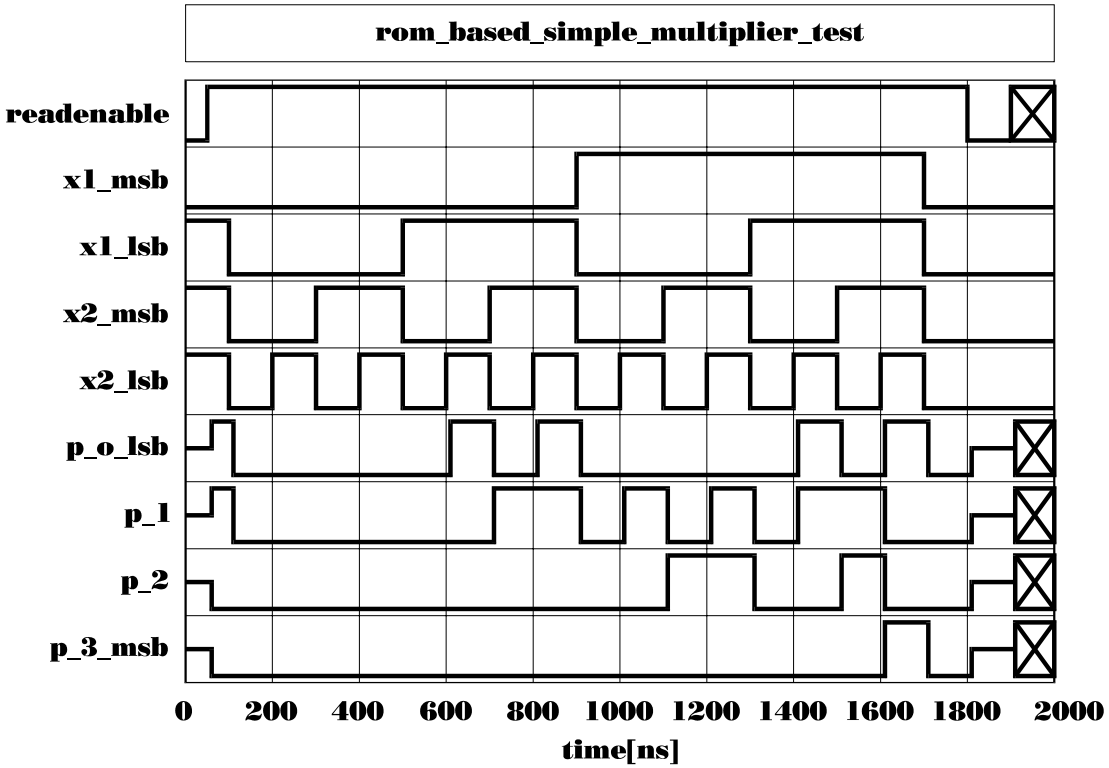
Model `rom4_multiplier_model` definisan je u modelskoj datoteci **model2.hi** na sledeći način.

```
model rom4::rom4_multiplier_model {
  tp=10ns;
  addr_lines_No=4;
  w_length=4;
  capacity=16;
  filename="mult.mem";
}
```

Vreme pristupa memoriji je 10ns. Kako vremena `tplh` i `tphl` nisu definisana (konstruktor klase `rom4` im zadaje nulte vrednosti), svejedno je da li koristimo modul `rom` ili `srom`. Iz *setup* datoteke **mult.mem** učitava se u memoriju tablica množenja. Sadržaj ove datoteke je sledeći.

```
//    Setup file for rom containing multiplying table
//    for integers from 0 to 3.

//    addr_lines_No=4
//    w_length=4 bit
//    capacity=16 memory words
// memory content:      address:          meaning:
      0000              // 0000           0x0=0
      0000              // 0001           0x1=0
      0000              // 0010           0x2=0
      0000              // 0011           0x3=0
      0000              // 0100           1x0=0
      0001              // 0101           1x1=1
      0010              // 0110           1x2=2
      0011              // 0111           1x3=3
      0000              // 1000           2x0=0
      0010              // 1001           2x1=2
      0100              // 1010           2x2=4
      0110              // 1011           2x3=6
      0000              // 1100           3x0=0
      0011              // 1101           3x1=3
      0110              // 1110           3x2=6
      1001              // 1111           3x3=9
```

Rezultat simulacije prikazan je na slici 5.9. Na ulaze ROM-a dovedene su sve raspoložive ulazne reči (kompletna tablica množenja). Na izlaznim linijama ROM-a dobija se proizvod ulaznih dvobitnih reči. Bit veće težine prvog činioca (označimo ga sa x1) dovodi se na ulaz `addr[0]`, njegov drugi bit (bit manje težine) dovodi se na ulaz `addr[1]`. Bit veće težine drugog činioca množenja (x2) dovodi se na adresnu liniju `addr[2]`, a njegov drugi bit (bit manje težine) dovodi se na `addr[3]`. Proizvod brojeva x1 i x2 dobija se posle vremena pristupa memoriji `tp`=10ns na linijama podataka `data[0]`, ..., `data[3]`, s tim da je `data[0]` bit najveće, a `data[3]` bit najmanje težine proizvoda. Recimo, u trenutku 1500ns na ulaz se dovodi x1=3 (`"11"`) i x2=2 (`"10"`), a u trenutku 1510ns se na izlaznim linijama ROM-a dobija proizvod p=6 (`"0110"`). U trenutku 1800ns ulaz za dozvolu čitanja `readenable` postavlja se u stanje logičke nule, na šta izlazne linije ROM-a reaguju prelaskom u stanje visoke impedanse `'Z'`. Kad `readenable` ode u neodredjeno stanje `'X'`, i izlazne linije prelaze u neodredjeno stanje, s obzirom da na njima može da se javi ili stanje visoke impedanse ili adresirana memorijska reč.



Slika 5.9 *Rezultat simulacije množača dva dvobitna broja realizovanog pomoću ROM-a*

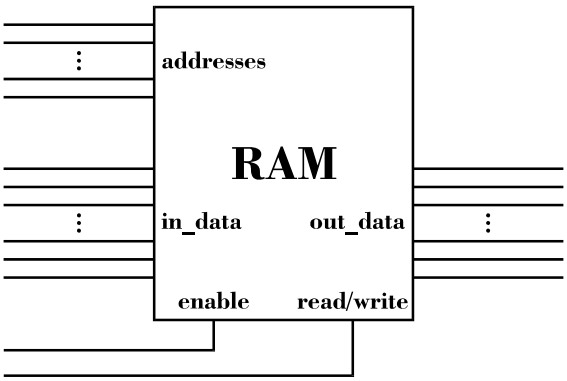
Tokom rada simulator štampa na ekranu sledeća upozorenja.

```
*** device "rom_chip", t=1.8e-06s:
- Warning #1: rom dissabled, out_lines go 'Z'

*** device "rom_chip", t=1.9e-06s:
- Warning #2: read enable went unknown, out_lines go 'X'
```

5.10.3 Modeliranje RAM-a

Najjednostavnija opšta šematska predstava RAM-a data je na slici 5.10. Pored adresnih linija i linija podataka, RAM ima liniju za dozvolu rada i liniju za izbor jednog od dva moguća radna režima: čitanja sadržaja RAM-a i upisivanja u RAM. Naravno, moguće su i mnoge druge konfiguracije (recimo RAM sa dve odvojene linije za dozvolu čitanja i upisa). Za modeliranje različitih RAM struktura kreirana je modelska klasa `ram4` sledećeg izgleda.

```
class ram4 : public io4 {           // random access memory model class
      double tp;
      double tplh, tphl;
      double skew;
      //double incap, outcap;
      int addr_lines_No;            // number of address lines
      int w_length;                 // memory word length
      int capacity;                 // number of memory words
      char *filename;               // ram setup file
  public :
      ram4 ();
      ~ram4 ();
      double get_tr() { return io4::rise_time; }
      double get_tf() { return io4::fall_time; }
      friend module     sram, ram;
};
```



***Slika 5.10** RAM*

Definisana su dva modula, kao i kod ROM-a, sa sledećim deklaracijama u datoteci **ss4.h**.

```
module ram4::sram ( signal four_full in en, r_w, addr[], in_lines[];
                                        signal four_full out out_lines[] );
module ram4::ram  ( signal four_full in en, r_w, addr[], in_lines[];
                                        signal four_full out out_lines[] );
```

Modul `sram` odgovara slici 5.10. To je RAM sa jednostavnim modelom kašnjenja, vreme pristupa memoriji je `tp`. Kod kojim je definisan modul `sram` u datoteci **gates4.hi** je sledeći.

```
module ram4::sram(four_full in en, r_w, addr[], in_lines[];
                                                four_full out out_lines[]) {
  action {
    process (en, r_w, addr, in_lines) {
      mem4 r(this.addr_lines_No, this.capacity, this.w_length, this.filename);
      int i;

      if (en=='x' || en=='z') {
        if (r_w=='1') {                    // read enabled
          warning("enable unknown, read enabled, out_lines go 'X'\n");
          out_lines <- r.set_outword('x') after this.tp;
        }
        else if (r_w=='0') {               // write enabled
          warning("enable unknown, write enabled, addressed word go 'X'\n");
          r.load_unknown(addr);
          out_lines <- r.set_outword('z') after this.tp;
        }
        else {
          warning("enable unknown, r_w unknown too, everything go 'X'\n");
          r.load_unknown(addr);
          out_lines <- r.set_outword('x') after this.tp;
        }
      }
      else if (en->event && en=='0') {     // ram4 just dissabled
        printf("ram4 dissabled, output lines go 'z'\n");
        out_lines <- r.set_outword('z') after this.tp;
      }
      else if (en=='1') {                  // ram4 enabled
        if (r_w=='x' || r_w=='z') {
          warning("r_w unknown, addressed word & output lines go 'X'\n");
          printf ("t=%g: r_w=%s, r_w unknown\n", now, names4[r_w]);
          r.load_unknown(addr);
          out_lines <- r.set_outword('x') after this.tp;
        }
        else if (r_w=='1') {               // read addressed word
          printf ("read word from ram4\n");
          out_lines <- r.pick_word(addr) after this.tp;
        }
        else {                             // load word to specified address
          printf ("t=%g: r_w=%s, load word to ram4\n", now, names4[r_w]);
```

```
          r.load_word(addr, in_lines);
          out_lines <- r.set_outword('z') after this.tp;
        }
      }
    }
  }
} // module ram4::sram
```

Na sličan način realizovan je i modul `ram`, s tom razlikom što su ugradjeni izlazni baferi koji omogućavaju rise/fall model kašnjanja.

```
module ram4::ram(four_full en, r_w, addr[], in_lines[];
                                                four_full out out_lines[]) {
  module sbuf outbuf;
  signal four_full sram_out[auto]="z";

  action {
    delay: process structural {           // output buffers introduce delays
      signal four_full sram_out(this.w_length);

      for (int i=0; i < this.w_length; i++) {
        clone outbuf[i] (out_lines[i], sram_out[i]) action(this.tplh, this.tphl,
                                  this.skew, this->get_tr(), this->get_tf());
      }
    }
    sram_function: process (en, r_w, addr, in_lines) {
      mem4 r(this.addr_lines_No, this.capacity, this.w_length, this.filename);
      int i;

      if (en->event && (en=='x' || en=='z')) {
        if (r_w=='1') {                   // read enabled
          warning("enable unknown, read enabled, out_lines go 'X'\n");
          sram_out <- r.set_outword('x') after this.tp;
        }
        else if (r_w=='0') {              // write enabled
          warning("enable unknown, write enabled, addressed word go 'X'\n");
          r.load_unknown(addr);
          sram_out <- r.set_outword('z') after this.tp;
        }
        else {
          warning("enable unknown, r_w unknown too, everything go 'X'\n");
          r.load_unknown(addr);
          sram_out <- r.set_outword('x') after this.tp;
        }
      }
      else if (en->event && en=='0') {    // ram4 just dissabled
        printf("ram4 dissabled, output lines go 'z'\n");
        sram_out <- r.set_outword('z') after this.tp;
      }
      else if (en=='1') {                 // ram4 enabled
        if (r_w=='x' || r_w=='z') {
          warning("r_w unknown, addressed word & output lines go 'X'\n");
          r.load_unknown(addr);
          sram_out <- r.set_outword('x') after this.tp;
        }
        else if (r_w=='1') {              // read addressed word
          sram_out <- r.pick_word(addr) after this.tp;
        }
        else {                            // load word to specified address
          r.load_word(addr, in_lines);
          sram_out <- r.set_outword('z') after this.tp;
        }
      }
    } // sram_function: process ()
  }
} // module ram4::ram
```

Kao i kod ROM modula, parametri `incap` i `outcap` se ne koriste, ali su stavljeni u modelsku karticu zbog konzistencije modela. Njihovo korišćenje je moguće i potrebno pri modeliranju konkretnih memorijskih komponenti, kod kojih je poznat broj ulaza i izlaza, tako da ih je moguće realizovati sa skalarnim signalima.

5.10.4 Primer simulacije sa RAM-om

Sledećom simulacijom testiraćemo korektnost razvijenog modela RAM-a.

```
# include "ss4.h"
library model4, gates4, op4;
root module sram_module_test () {
  module ram4::sram ram_chip;
  signal four_full addr[4]="0000";        // address lines
  signal four_full in_d[4]="1111";        // input data lines
  signal four_full out_d[4]="zzzz";       // output data lines
  signal four_full en='0';                // chip enable
  signal four_full r_w='1';               // read/write

  ram_chip (en, r_w, addr, in_d, out_d) model = ram4_zero_model;
  timing { tstop = 1000ns; }
  out {
      signal four_full out_d;
      signal four_full in_d;
      signal four_full addr;
      signal four_full r_w;
      signal four_full en;
  }
  action {
    process initial {
      en    <-    '1' after  50ns, '0' after 700ns, 'x' after 800ns;
      r_w   <-    '0' after 200ns, '1' after 300ns, 'x' after 600ns,
                  '0' after 900ns;
      in_d  <-    "0101" after 100ns;
      addr  <-    "0100" after 100ns,
                  "0000" after 400ns,
                  "0100" after 500ns;
    }
  }
}
```

U primeru je korišćen modul `sram`, s tim da je moguće jednostavnom zamenom umesto njega koristiti i modul `ram`. Model `ram4_zero_model` definisan je u modelskoj datoteci **model4.hi** na sledeći način.

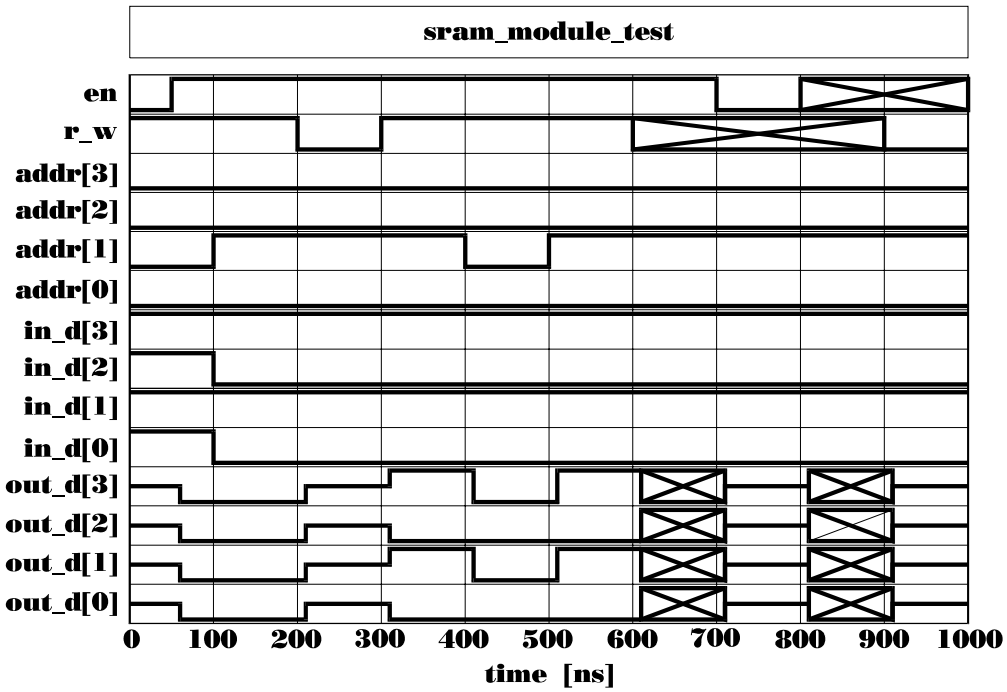
```
model ram4::ram4_zero_model {
  tp=10ns;
  addr_lines_No=4;
  w_length=4;
  capacity=16;
  filename="zero.mem";
}
```

Kašnjenje (vreme pristupa) memorije je 10ns, memorija ima 16 četvorobitnih reči i, naravno, 4 adresne linije za njihovo adresiranje. Početni sadržaj memorije učitava se iz datoteke **zero.mem**. Datoteka **zero.mem** mora da postoji, a ako u njoj nema podataka, onda se memorijska matrica `memory_matrix` inicijalizira vrednošću `'X'` (prvo stanje iz sistema stanja `four_t`) na svim pozicijama. Medjutim, mi ćemo u memoriju upisati logičke nule na sve pozicije, tako da datoteka **zero.mem** ima sledeći sadržaj.

```
//    Setup file for ram containing all zeros
```

```
//    addr_lines_No=4
//    w_length=4 bit
//    capacity=16 memory words
      0000  0000  0000  0000  0000  0000  0000  0000
      0000  0000  0000  0000  0000  0000  0000  0000
```

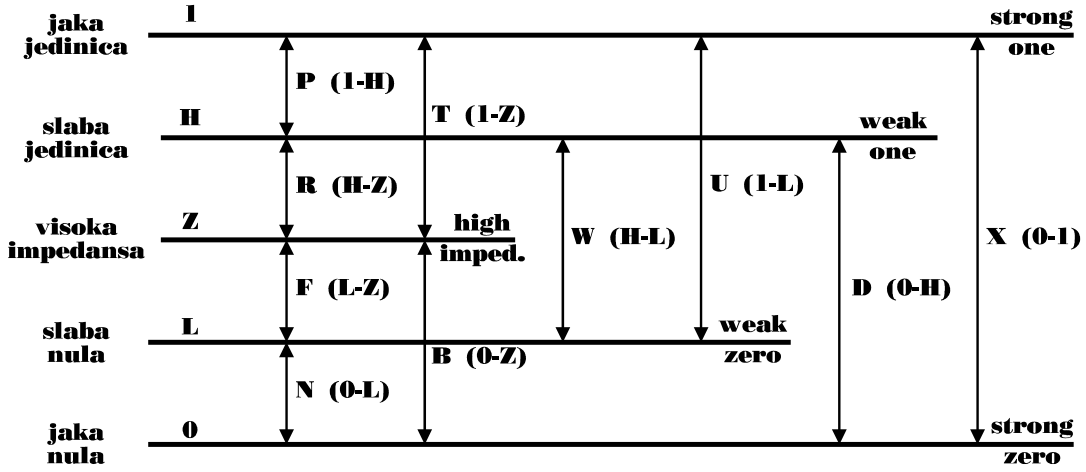
Rezultat simulacije prikazan je na slici 5.11. Na početku simulacije dozvoljeno je čitanje memorije (`r_w` je na logičkoj jedinici). Ulaz `en` je na logičkoj nuli, pa je na izlaznim linijama stanje visoke impedanse `'Z'`. Adresne linije postavljene su na `"0111"`, a kako je sadržaj svih memorijskih reči isti (`"0000"`), kada `en` ulaz u trenutku 50ns ode na logičku jedinicu, izlazne linije podataka `out_d` uzimaju stanje `"0000"`. Ulazne linije podataka se u trenutku 100ns postavljaju na stanje `"0101"`, ali nemaju uticaja na rad RAM-a sve dok se u trenutku 200ns kontrolna linija `r_w` ne prebaci u stanje logičke nule (dozvoljen upis u memoriju). Tada se upisuje ulazna reč `"0101"` u memorijsku lokaciju odredjenu stanjem adresnih linija `"0100"` (četvrta memorijska reč). Za vreme upisivanja, izlazne linije podataka idu u stanje visoke impedanse. U trenutku 300ns linija `r_w` vraća se u stanje logičke jedinice, a kako se stanje adresnih linija u medjuvremenu nije menjalo, na izlazne linije podataka se postavlja upisana reč `"0101"`. Sada je dozvoljena promena na adresnim linijama. U trenucima 400ns i 500ns adresne linije menjaju stanje, što se odražava promenom stanja izlaznih linija (sadržaj memorijske lokacije `"0000"` je `"0000"`, a sadržaj memorijske lokacije `"0100"` je, kao i ranije, `"0101"`). Na kraju je testirano ponašanje modela RAM-a kada se na kontrolne ulaze dovedu neodredjeni signali.



Slika 5.11 *Rezultat testiranja RAM modula* `sram`

6 Biblioteka za HILO sistem stanja

Logički simulator HILO je jedan od najšire korišćenih simulatora digitalnih kola danas. Ovo se može objasniti ako se ima u vidu da simulator HILO podržava vrlo sofisticirane menanizme za modeliranje digitalnih kola [Maks94c]. Tu treba pomenuti dobro izabran sistem stanja, solidan (za vreme u koje je nastao) jezik za funkcionalno i strukturno modeliranje digitalnih kola, mogućnost simulacije različitih tehnologija, kao i veoma složen, ali prilagodljiv model kašnjenja. Simulator HILO ima ugradjen sistem sa petnaest logičkih stanja kako to prikazuje slika 6.1.



Slika 6.1 *HILO sistem stanja*

Stanje `'H'` predstavlja takozvanu slabu logičku jedinicu, a stanje `'L'` prestavlja slabu logičku nulu. Slabo logičko stanje razlikuje se od jakog po tome što je izlazna otpornost kola koje ga generiše na izlazu značajno veća, tako da u slučaju rezolucije na magistrali jako stanje dominira nad slabim. Stanje `'P'` predstavlja mogućnost da se na izlazu pojavi ili jaka ili slaba logička jedinica. Slično, stanje `'T'` predstavlja nesigurnost izmedju stanja `'1'` i stanja `'Z'` na izlazu nekog logičkog elementa.

Biblioteka za simulaciju u HILO sistemu stanja organizovana je u četiri datoteke:

ss15.h - deklaraciona datoteka
op15.hi - funkcijska datoteka
gates15.hi - strukturna (modulska) datoteka
model15.hi - modelska datoteka

Deklaraciona datoteka **ss15.h** sadrži deklaracije korišćenih struktura, globalnih podataka, funkcija i modula. Funkcijska datoteka **op15.hi** sadrži funkcije koje se koriste u modeliranju (funkcije kašnjenja, funkcije rezolucije, funkcije za propterećenje operatora i druge), kao i definicije globalnih podataka (promenljivih, tabela i slično). Datoteka **gates15.hi** sadrži definicije digitalnih komponenti. Datoteka **model15.hi** sadrži neke osnovne modelske kartice.

6.1 Definisanje sistema stanja i konverzionih tabela

Sistem sa petnaest stanja koji koristi simulator HILO je definisan na sledeći način.

```
typedef enum {
             'X', 'x'='X',          // unknown
             'D', 'd'='D',          // '0' - 'H'
             'U', 'u'='U',          // '1' - 'L'
             'W', 'w'='W',          // 'H' - 'L'
             'B', 'b'='B',          // '0' - 'Z'
             'T', 't'='T',          // '1' - 'Z'
             'N', 'n'='N',          // '0' - 'L'
             'F', 'f'='F',          // 'L' - 'Z'
             'R', 'r'='R',          // 'H' - 'Z'
             'P', 'p'='P',          // '1' - 'H'
             '0',                   // logic zero
             'L', 'l'='L',          // weak logic zero
             'Z', 'z'='Z',          // high impedance
             'H', 'h'='H',          // weak logic one
             '1',                   // logic one
             '_' = void,            // separator
             ' ' = void             // alternative separator - blank
} fift_t;
```

Na sreću, nije potrebno da se logičke operacije obavljaju u sistemu sa petnaest stanja (to bi zahtevalo da se definišu tabele istinitosti operatora veličine 15x15=225 elemenata). Efikasnije je konvertovati stanja nad kojima treba izvršiti logičku operaciju u sistem sa tri ili četiri stanja, u ovom redukovanom sistemu izvršiti logičku operaciju, a zatim ponovo izvršiti konverziju rezultata u sistem sa petnaest stanja. Iz ovog razloga, u biblioteci treba definisati i sisteme sa tri i četiri stanja.

```
typedef enum {'X', 'x'='X', '0', '1', ' ' = void, '_' = void} three_t;
typedef enum {'X', 'x'='X', 'Z', 'z'='Z', '0', '1', ' ' =void, '_'=void} four_t;
```

Naravno, kao i kod drugih logičkih biblioteka definisane su tabele `names15, names3 i names4` da bi se omogućilo jednostavno štampanje stanja na ekranu.

```
char *names3[] ={"X", "0", "1"};
char *names4[] ={"X", "Z", "0", "1"};
char *names15[]={"X","D","U","W","B","T","N","F","R","P","0","L","Z","H","1"};
```

Konverzija iz jednog u drugi sistem stanja obavlja se pomoću konverzionih tabela `con15to3`, `con3to15`, `con15to4`, `con4to15`, `con3to4` i `con4to3`. Sva stanja koja nisu jednoznačno odredjena u sistemu sa petnaest stanja prevode se u sisteme sa tri i četiri stanja kao stanje `'X'`. Slaba logička nula smatra se stanjem `'0'`, a slaba logička jedinica smatra se stanjem `'1'`. Takodje, stanje `'N'` koje predstavlja neodredjenost izmedju jake i slabe logičke nule konvertuje se u stanje `'0'`. Slično, stanje `'P'` se tumači kao stanje `'1'` u sistemima sa tri i četiri stanja. Konverzione tabele definisane su na sledeći način.

```
three_t con15to3[15] = {
  'X',          // 'X',  unknown
  'X',          // 'D',  '0' - 'H'
```

```
  'X',          // 'U',  '1' - 'L'
  'X',          // 'W',  'H' - 'L'
  'X',          // 'B',  '0' - 'Z'
  'X',          // 'T',  '1' - 'Z'
  '0',          // 'N',  '0' - 'L'
  'X',          // 'F',  'L' - 'Z'
  'X',          // 'R',  'H' - 'Z'
  '1',          // 'P',  '1' - 'H'
  '0',          // '0',  logic zero
  '0',          // 'L',  weak logic zero
  'X',          // 'Z',  high impedance
  '1',          // 'H',  weak logic one
  '1'           // '1'   logic one
};
fift_t con3to15[3] = {'X', '0', '1'};
four_t  con15to4[15] = {
  'X',          // 'X',  unknown
  'X',          // 'D',  '0' - 'H'
  'X',          // 'U',  '1' - 'L'
  'X',          // 'W',  'H' - 'L'
  'X',          // 'B',  '0' - 'Z'
  'X',          // 'T',  '1' - 'Z'
  '0',          // 'N',  '0' - 'L'
  'X',          // 'F',  'L' - 'Z'
  'X',          // 'R',  'H' - 'Z'
  '1',          // 'P',  '1' - 'H'
  '0',          // '0',  logic zero
  '0',          // 'L',  weak logic zero
  'Z',          // 'Z',  high impedance
  '1',          // 'H',  weak logic one
  '1'           // '1',  logic one
};
fift_t con4to15[4] = {'X', 'Z', '0', '1'};
three_t con4to3[4] = {'X', 'X', '0', '1'};
four_t con3to4[3] = {'X', '0', '1'};
```

Logičke operatore ćemo sada preopteretiti za sistem sa tri stanja. Recimo, operatori `|` i `|=` (logička ILI operacija) definisani su na sledeći način.

```
three_t or_tab[3][3] = { 'X', 'X', '1',
                         'X', '0', '1',
                         '1', '1', '1'  };
three_t operator |  (three_t l, three_t r) { return or_tab[l][r]; }
three_t operator |= (three_t &l, three_t r) { return ( l = or_tab[l][r] ); }
```

Kod operatora `|=` levi operand označen sa `l` prenosi se po adresi, pa je moguće da se u funkciji promeni njegova vrednost (desni operand `r` se prenosi po vrednosti). Sledeća dva logička izraza u kojima učestvuju dve promenljive tipa `three_t`, `a` i `b`, su ekvivalentna.

```
a = a | b;  // this expression is equivalent to:
a |= b;
```

6.2 Parazitne kapacitivnosti - atributi signala

Model kašnjenja koji koristi simulator HILO uključuje uticaj kapacitivnog opterećenja. Zato je potrebno definisati kapacitivnost signala kao korisnički atribut. Učinićemo to korišćenjem konstrukcije `class`, definisaćemo atributsku klasu `fift_att`.

```
class fift_att {
      double tcap;                  // total signal capacitance
      >fift_att ();
  public :
      fift_att ();
```

```
      void add_cap (double c) { tcap += c;   }
      double get_tcap ()      { return tcap; }
};
```

Novi tip signala definisaćemo pridruživanjem atributske klase `fift_att` postojećem tipu `fift_t`.

```
typedef fift_t @ fift_att fift_full;
```

U simulaciji ćemo koristiti tip `fift_full`, a rezolucione funkcije ćemo mu pridruživati po potrebi.

6.3 Modovi simulacije

Svi parametri modela komponenti kojima se opisuje simulirani sistem mogu se kod simulatora HILO zadati sa tri vrednosti: minimalnom, tipičnom i maksimalnom. Pri simulaciji je zatim moguće odabrati koja će od ovih vrednosti biti korišćena. Ovaj izbor se u našoj biblioteci vrši pomoću globalne promenljive `simulationmode` i tri makroa za izbor moda simulacije: `MIN_MODE`, `TYP_MODE` i `MAX_MODE`. U datoteci **ss15.h** nalaze se njihove deklaracije.

```
extern int simulationmode;
# define    MIN_MODE    0
# define    TYP_MODE    1
# define    MAX_MODE    2
```

U datoteci **op15.hi** postavljena je difoltna vrednost promenljive `simulationmode` na `TYP_MODE`, što odgovara situaciji kod simulatora HILO (pri simulaciji se koristi tipična vrednost parametara modela osim ako je izabran neki drugi mod simulacije).

```
int simulationmode = TYP_MODE;
```

Promena moda simulacije obavlja se u `root` modulu korišćenjem procesa sinhronisanog signalom `initial`. Naravno, da bi globalna promenljiva `simulationmode` bila vidljiva iz `root` modula, potrebno je priključiti datoteku **ss15.h**, a u naredbi `library` navesti datoteku **op15.ao**.

```
# include "ss15.h"
library op15;
root ... {
  ...
  process initial {
    simulationmode=MAX_MODE;
    ...
  }
}
```

Da bi smo omogućili zadavanje tri vrednosti parametrima modela definisali smo novi tip podataka `param` koji predstavlja vektor od tri vrednosti tipa `double`.

```
# define TOTAL_TOLERANCE_MODES_NUMBER     3
typedef double param[TOTAL_TOLERANCE_MODES_NUMBER];
```

Svi parametri modela koji se definišu minimalnom, tipičnom i maksimalnom vrednošću biće tipa `param`. Na primer, kašnjenje `tplh` zadato kao 10ns:20ns:30ns (u HILO HDL notaciji) definisalo bi se izrazom

```
param tplh = { 10ns, 20ns, 30ns };
```

ili bez inicijalizacije vrednosti u deklaraciji

```
param tplh;
tplh[MIN_MODE] = 10ns; tplh[TYP_MODE] = 20ns; tplh[MAX_MODE] = 30ns;
```

Funkcija `pick` je zadužena za izbor jedne od tri zadate vrednosti koja je potrebna.

```
inline double pick (const param vec, int mode) { return (vec[mode]); }
```

6.4 Modeliranje različitih tehnologija izrade gejtova

Simulator HILO duguje svoju popularnost delimično i tome što omogućava da se u jedinstvenom sistemu stanja modeliraju različite tehnologije izrade logičkih kola. Naime, od tehnologije izrade gejta zavisi izlazna otpornost u stanju logičke nule i izlazna otpornost u stanju logičke jedinice. Kada izlaz ima malu izlaznu otpornost kaže se da je stanje jako, a kada izlaz ima veliku izlaznu otpornost kaže se da je stanje slabo. Recimo, u NMOS tehnologiji stanje logičke nule je jače od stanja logičke jedinice, jer je izlazna otpornost u stanju logičke nule manja nego u stanju logičke jedinice. Kratkospajanjem izlaza dva NMOS logička kola dobiće se žičana logika u kojoj dominira stanje logičke nule, odnosno veza žičano I.

Sistem sa petnaest stanja koji koristi simulator HILO sadrži u sebi informaciju o jačini stanja, koja je potrebna za modeliranje različitih tehnologija. Osnovne tehnologije (TTL, TTLOC, ECL, NMOS, PMOS, CMOS) izrade digitalnih kola definisaćemo makroima kako bi se po potrebi dodavale nove.

```
# define    TECH_NUMBER_MIN              0
# define    TTL               0
# define    TTLOC             1
# define    ECL               2
# define    NMOS              3
# define    PMOS              4
# define    CMOS              5
# define    WEAK_LOW_WEAK_HIGH      6
# define    STRONG_LOW_STRONG_HIGH  7
# define    WEAK_LOW_STRONG_HIGH    8
# define    STRONG_LOW_WEAK_HIGH    9
# define    TECH_NUMBER_MAX              9
```

Specijalnim postupkom projektovanja nezavisno od tehnologije moguće je ostvariti i druge kombinacije jačina izlaza. Zato su osnovnim tehnologijama dodate još četiri moguće kombinacije jačina stanja izlaza: `WEAK_LOW_WEAK_HIGH`, `WEAK_LOW_STRONG_HIGH`, `STRONG_LOW_WEAK_HIGH` i `STRONG_LOW_STONG_HIGH`.

TECHNOLOGY	Output state	
	Logic one	Logic zero
TTL	1	0
TTLOC	Z	0
ECL	H	0
NMOS	H	0
PMOS	1	L
CMOS	1	0
STRONG_LOW_STRONG_HIGH	1	0
WEAK_LOW_STRONG_HIGH	1	L
STRONG_LOW_WEAK_HIGH	H	0
WEAK_LOW_WEAK_HIGH	H	L

Tabela 6.1 *Konverzija stanja na izlazu gejta u zavisnosti od tehnologije izrade*

Konverzija izlaznih stanja zasniva se na tabeli 6.1. Da bi se odredilo stanje na izlazu gejta u zavisnosti od tehnologije izrade, definisali smo konverzionu tabelu `tech_tab`.

```
fift_t tech_tab [][15] = {
/*** States:
  'X'  'D'  'U'  'W'  'B'  'T'  'N'  'F'  'R'  'P'  '0'  'L'  'Z'  'H'  '1'  ***/

  // 0 (TTL): from {'X','Z','0','1'} to {'X','Z','0','1'} (no change)
  'X', 'D', 'U', 'W', 'B', 'T', 'N', 'F', 'R', 'P', '0', 'L', 'Z', 'H', '1',

  // 1 (TTLOC): from {'X','Z','0','1'} to {'B','Z','0','Z'}
  'B', 'B', 'F', 'F', 'B', 'Z', 'N', 'F', 'R', 'Z', '0', 'L', 'Z', 'Z', 'Z',

  // 2 (ECL): from {'X','Z','0','1'} to {'D','Z','0','H'}
  'D', 'D', 'W', 'W', 'B', 'R', 'N', 'F', 'R', 'H', '0', 'L', 'Z', 'H', 'H',

  // 3 (NMOS): from {'X','Z','0','1'} to {'D','Z','0','H'}
  'D', 'D', 'W', 'W', 'B', 'R', 'N', 'F', 'R', 'H', '0', 'L', 'Z', 'H', 'H',

  // 4 (PMOS): from {'X','Z','0','1'} to {'U','Z','L','1'}
  'U', 'W', 'U', 'W', 'F', 'T', 'L', 'F', 'R', 'P', 'L', 'L', 'Z', 'H', '1',

  // 5 (CMOS): from {'X','Z','0','1'} to {'X','Z','0','1'}(no change)
  'X', 'D', 'U', 'W', 'B', 'T', 'N', 'F', 'R', 'P', '0', 'L', 'Z', 'H', '1',

  // 6 (WEAK_LOW_WEAK_HIGH): from {'X','Z','0','1'} to {'W','Z','L','H'}
  'W', 'W', 'W', 'W', 'F', 'R', 'L', 'F', 'R', 'H', 'L', 'L', 'Z', 'H', 'H',

  // 7 (STRONG_LOW_STRONG_HIGH): from {'X','Z','0','1'} to {'X','Z','0','1'}
  // (no change)
  'X', 'D', 'U', 'W', 'B', 'T', 'N', 'F', 'R', 'P', '0', 'L', 'Z', 'H', '1',

  // 8 (WEAK_LOW_STRONG_HIGH): from {'X','Z','0','1'} to {'U','Z','L','1'}
  'U', 'W', 'U', 'W', 'F', 'T', 'L', 'F', 'R', 'P', 'L', 'L', 'Z', 'H', '1',

  // 9 (STRONG_LOW_WEAK_HIGH): from {'X','Z','0','1'} to {'D','Z','0','H'}
  'D', 'D', 'W', 'W', 'B', 'R', 'N', 'F', 'R', 'H', '0', 'L', 'Z', 'H', 'H'
};
```

Prva dimenzija tabele `tech_tab` je tehnologija izrade gejta, a druga je stanje u sistemu sa petnaest stanja. Na primer, posle izvršenja komande

```
fift_t a = tech_tab [TTLOC]['1'];
```

promenljiva `a` ima vrednost `'Z'`.

6.5 Modeliranje osnovnih digitalnih komponenti

Biblioteka o kojoj govorimo nije namenjena za simulaciju hibridnih kola, već samo za logičku simulaciju, tako da nije predvidjena bazna modelska klasa za modeliranje A/D i D/A konvertora kao kod drugih biblioteka. U slučaju potrebe, ovaj nedostatak se može relativno lako ispraviti dodavanjem bazne klase ili dodavanjem parametara modela konvertora u korišćenu modelsku klasu `m15` o kojoj će biti reči u daljem tekstu.

Radi parametrizovanja modela osnovnih logičkih elemenata definisana je modelska klasa `m15`. Ova klasa sadrži sledeće parametre modela gejtova: kašnjenje prednje i zadnje ivice na izlazu, kašnjenja trostatičkih kola vezana za pojavu stanja visoke impedanse na izlazu, nagib porasta kašnjenja prednje i zadnje ivice sa kapacitivnim opterećenjem izlaza i parazitna terminalna kapacitivnost. Takodje, parametar modela gejta je i tehnologija izrade, s obzirom da od nje zavise jačine stanja na izlazu.

```
class m15 {
      param       delay01, delay10, delay0Z, delayZ0, delay1Z, delayZ1;
      param       skew01, skew10;
      param       termcap;
      int         tech;
      m15   () { }
      >m15  ();
  public :
      double      delay  (three_t, three_t, double);
      double      Zdelay (fift_t, fift_t, double);
            // modules permitted to access class m15 private members
      friend module inv, buf, and2, and3, and4, and5, or2, or3, or4, or5;
      friend module nand2, nand3, nand4, nand5, nor2, nor3, nor4, nor5;
      friend module xor, nxor, and, andm, or, nand, nor, mux21, moveif1;
      friend module moveif0, bufif1, bufif0, notif1, notif0, balr;
};
```

Kako se pri rezervisanju memorijskog prostora za podatke njihova vrednost automatski anulira, a drugačije vrednosti nisu zadate u konstruktoru, to su svi podaci klase `m15` automatski postavljeni na nulu. Dakle, zadate su nulte vrednosti kašnjenja, nagiba porasta kašnjenja i terminalne kapacitivnosti, kao i tehnologija TTL. Korisnik zadaje vrednosti parametara modela konkretne komponente definisanjem modelske kartice, a procesor `>m15` je zadužen da proveri logičnost zadatih vrednosti.

Modelskoj klasi `m15` odgovara difoltni model `m15_default` definisan u modelskoj datoteci **model15.hi**.

```
model m15::m15_default_model { }
```

Kao prijatelji (`friend`) klase `m15` deklarisani su modelirani gejtovi. Definisana su i dva metoda klase `m15`, funkcije za kašnjenje `delay` i `Zdelay`. Modeliranje osnovnih logičkih elementa ilustrovaćemo na dvoulaznom ILI kolu. Ovaj gejt opisan je sledećim kodom u datoteci **gates15.hi**.

```
module m15::or2 (fift_full out y; fift_full in l, r) {
  action {
    process post_structural {
      (@l)->add_cap(this.termcap[simulationmode]);
      (@r)->add_cap(this.termcap[simulationmode]);
      (@y)->add_cap(this.termcap[simulationmode]);
    }
    process (l, r) {
      three_t tmp;

      tmp = con15to3[l] | con15to3[r];
      y <- (fift_full) ( tech_tab[this.tech][con3to15[tmp]] ) after
           this.delay(tmp, con15to3[y],
                           (@y)->get_tcap()-this.termcap[simulationmode]);
    }
  }
}
```

Modul `m15::or2` ima dva ulazna, `l` i `r`, i jedan izlazni terminal, `y`. Preko modelske kartice tipa `m15` modulu se prosledjuju parametri modela. Modul sadrži dva procesa. Prvi proces snabdeva signale vezane za terminale gejta informacijom o terminalnim kapacitivnostima. Drugi proces, osetljiv na promene signala `l` i `r`, modelira logičku funkciju gejta. Da bi se upotrebio ranije definisani operator `&` neophodna je konverzija vrednosti ulaznih signala u tip `three_t` upotrebom konverzione tabele `con15to3`. Izračunato stanje se zatim pretvara ponovo u tip `fift_t` (korišćenjem tabele `con3to15`) i uskladjuje sa tehnologijom izrade gejta (iščitavanjem tabele `tech_tab`). Posle kašnjenja koje je izračunato upotrebom funkcije `delay`, novo stanje se šalje na izlaz gejta, pri čemu se vrši prilagodjenje tipa (tabela `tech_tab` vraća `fift_t`, a signal `y` je tipa `fift_full`) korišćenjem *cast* operatora. Treba uočiti da se funkciji kašnjenja šalje vrednost totalne

kapacitivnosti izlaznog terminala (atribut signala `y`) umanjena za terminalnu kapacitivnost gejta. Naime, kod simulatora HILO model kašnjenja gejta je nešto drugačiji nego do sada opisivani modeli. Smatra se da izlazna kapacitivnost gejta već učestvuje u osnovnom kašnjenju gejta (nije ni moguće izmeriti kašnjenje gejta bez ove kapacitivnosti), tako da je treba isključiti iz kapacitivnog opterećenja izlaza gejta. Opterećenje izlaza su izlazne kapacitivnosti drugih gejtova (u slučaju žičane veze izlaza više gejtova) i ulazne kapacitivnosti gejtova vezanih za signal `y`.

6.6 Funkcije za kašnjenje

Funkcija za kašnjenje `delay` deklarisana kao metod klase `m15` odredjuje kašnjenje gejta u zavisnosti od novog stanja izlaza, starog stanja na izlazu i kapacitivnog opterećenja izlaza. Koriste je samo standardni gejtovi (na njihovom izlazu mogu se pojaviti samo tri stanja: `'0'`, `'1'` i `'X'`). Analiza hibridnog aspekta izlaznog terminala nije uzeta u obzir. Pre poziva funkcije `delay` potrebno je novo i staro stanje izlaza konvertovati u sistem sa tri stanja, s obzirom da je funkcija tako definisana da prima promenljive tipa `three_t`.

```
double m15::delay (three_t ns, three_t os, double c=0.0) {
  double help1, help2;

  if(ns=='0')
    return (pick(delay10, simulationmode) + pick(skew10, simulationmode) * c);
  if(ns=='1')
    return (pick(delay01, simulationmode) + pick(skew01, simulationmode) * c);
                                          // therefore, ns == 'X'
  if(os == '1')
    return (pick(delay10, simulationmode) + pick(skew10, simulationmode) * c);
  if(os == '0')
    return (pick(delay01, simulationmode) + pick(skew01, simulationmode) * c);
  if(os == 'X') {                         // take the worst case
    help1 = pick(delay01, simulationmode) + pick(skew01, simulationmode) * c;
    help2 = pick(delay10, simulationmode) + pick(skew10, simulationmode) * c;
    return (MAX(help1, help2));
  }
}
```

Funkcija `Zdelay` namenjena je odredjivanju kašnjenja trostatičkih gejtova, na čijem se izlazu može pojaviti svih petnaest logičkih stanja. Funkcija prima stanja u sistemu sa petnaest stanja, a potrebne konverzije se vrše unutar funkcije.

```
double m15::Zdelay (fift_t ns, fift_t os, double c=0.0) {
  four_t ns4, os4;
  double tmp, help1, help2, help3, help4;

  ns4 = con15to4[ns];  os4 = con15to4[os];
  if( (os4 == 'Z') && (ns4 == 'Z') ) {
    help1 = pick(delay0Z, simulationmode) + pick(skew01, simulationmode) * c;
    help2 = pick(delayZ0, simulationmode) + pick(skew10, simulationmode) * c;
    help3 = pick(delayZ1, simulationmode) + pick(skew01, simulationmode) * c;
    help4 = pick(delay1Z, simulationmode) + pick(skew10, simulationmode) * c;
    tmp = MAX(help1,help2); tmp = MAX(tmp,help3); tmp = MAX(tmp,help4);
    return (tmp);
  }
  else if (os4=='Z') {
    switch (ns4) {
      case '0':
            return (pick(delayZ0,simulationmode)+pick(skew10,simulationmode)*c);
      case '1':
            return (pick(delayZ1,simulationmode)+pick(skew01,simulationmode)*c);
      case 'X':
            help1 = pick(delayZ1,simulationmode)+pick(skew01,simulationmode)*c;
```

```
            help2 = pick(delayZ0,simulationmode)+pick(skew10,simulationmode)*c;
            return (MAX(help1,help2));
      default : warning("Zdelay() : This cannot happen!",1);
    }
  }
  else if (ns4=='Z') {
    switch (os4) {
      case '0':
            return (pick(delay0Z,simulationmode)+pick(skew01,simulationmode)*c);
      case '1':
            return (pick(delay1Z,simulationmode)+pick(skew10,simulationmode)*c);
      case 'X':
            help1 = pick(delay1Z,simulationmode)+pick(skew10,simulationmode)*c;
            help2 = pick(delay0Z,simulationmode)+pick(skew01,simulationmode)*c;
            return (MAX(help1,help2));
      default : warning("Zdelay() : This cannot happen! No way!",1);
    }
  }
  else {                              // like m15::delay()
    if(ns4=='0')
      return (pick(delay10, simulationmode) + pick(skew10, simulationmode) * c);
    if(ns4=='1')
      return (pick(delay01, simulationmode) + pick(skew01, simulationmode) * c);
    if(os4=='1')
      return (pick(delay10, simulationmode) + pick(skew10, simulationmode) * c);
    if(os4=='0')
      return (pick(delay01, simulationmode) + pick(skew01, simulationmode) * c);
    if(os4=='X') {                    // take the worst case
      help1 = pick(delay10, simulationmode) + pick(skew10, simulationmode) * c;
      help2 = pick(delay01, simulationmode) + pick(skew01, simulationmode) * c;
      return (MAX(help1,help2));
    }
  }
} // double m15::Zdelay()
```

Obe opisane funkcije mogu se koristiti i bez zadavanja parametra `c`, pri čemu se podrazumeva da je njegova vrednost `0.0`, kao što je difinisano u listi formalnih parametara. Tako su legalni svi pozivi funkcija `delay` i `Zdelay` koji slede.

```
double d1 = delay (a, b, 1nF);
double d2 = delay (a, '1');
double d3 = Zdelay ('1', 'T', 2.2p);
double d4 = Zdelay ((fift_t)a, (fift_t)b);
```

6.7 Rezolucione funkcije

Kod simulatora HILO rezoluciona funkcija se bira specificiranjem tipa veze (WIRE, INPUT, WAND, WOR, ...). U našem slučaju, to se radi navodjenjem odgovarajuće rezolucione funkcije iza tipa signala pomoću operatora `:`. Definisane su rezolucione funkcije za žičanu logiku i za rezoluciju na magistrali sa pullup ili pulldown otpornikom i bez njega. Njihove deklaracije iz datoteke **ss15.h** imaju sledeći izgled.

```
extern fift_t bus15res   (fift_t *, int *);
extern fift_t PullUp15   (fift_t *, int *);
extern fift_t PullDown15 (fift_t *, int *);
extern fift_t WiredAnd   (fift_t *, int *);
extern fift_t WiredOr    (fift_t *, int *);
```

Kao primer ćemo navesti funkciju za rezoluciju na magistrali `bus15res`. Ovo je osnovni tip rezolucione funkcije i odgovara tipu veze WIRE kod simulatora HILO.

```
fift_full bus15res (const fift_full *drivers, int *flag) {
```

```
  int countX=0, n = lengthof(drivers);
  fift_full result;

  result = drivers[0];
  if ((result=='X') || (result=='W')) countX++;
  for (int i = 1; i < n; i++) {
    result = (fift_full) ( bus15_tab[result][drivers[i]] );
    if ((drivers[i]=='X') || (drivers[i]=='W')) countX++;
  }
  if (countX)             *flag=2;   // repport possible conflict
  else if(result == 'X') *flag=1;   // repport conflict for sure
  else                    *flag=0;
  return (result);
}
```

Očigledno je iskorišćena osobina asocijativnosti funkcije rezolucije. Rezultujuće stanje više od dva drajvera odredjuje se tako što se primenom tabele `bus15_tab` na dva drajvera odredi početna vrednost rezultata, a zatim se, za sve ostale drajvere, rezultat odredjuje primenom tabele `bus15_tab` na jedan drajver i prethodno odredjenu vrednost rezultata. Takodje je moguće uzeti za početnu vrednost rezultata stanje `'Z'` kao nedominantno. Tabela `bus15_tab` koja odredjuje rezultujuće stanje dva drajvera magistrale ima sledeći izgled.

```
fift_t bus15_tab[15][15] = {
  //    X   D   U   W   B   T   N   F   R   P   0   L   Z   H   1
     { 'X','X','X','X','X','X','X','X','X','X','X','X','X','X','X' },   // 'X'
     { 'X','D','X','D','D','X','D','D','D','X','0','D','D','D','X' },   // 'D'
     { 'X','X','U','U','X','U','X','U','U','U','X','U','U','U','1' },   // 'U'
     { 'X','D','U','W','D','U','D','W','W','U','0','W','W','W','1' },   // 'W'
     { 'X','D','X','D','B','X','B','B','D','X','0','B','B','D','X' },   // 'B'
     { 'X','X','U','U','X','T','X','U','T','T','X','U','T','T','1' },   // 'T'
     { 'X','D','X','D','B','X','N','B','D','X','0','N','N','D','X' },   // 'N'
     { 'X','D','U','W','B','U','B','F','W','U','0','F','F','W','1' },   // 'F'
     { 'X','D','U','W','D','T','D','W','R','T','0','W','R','R','1' },   // 'R'
     { 'X','X','U','U','X','T','X','U','T','P','X','U','P','P','1' },   // 'P'
     { 'X','0','X','0','0','X','0','0','0','X','0','0','0','0','X' },   // '0'
     { 'X','D','U','W','B','U','N','F','W','U','0','L','L','W','1' },   // 'L'
     { 'X','D','U','W','B','T','N','F','R','P','0','L','Z','H','1' },   // 'Z'
     { 'X','D','U','W','D','T','D','W','R','P','0','W','H','H','1' },   // 'H'
     { 'X','X','1','1','X','1','X','1','1','1','X','1','1','1','1' }    // '1'
};
```

Konflikt na magistrali rezultuje pojavom `'X'` stanja. Kako je `'X'` stanje dominantno pri rezoluciji, ono može da se javi i ako je neki od drajvera u `'X'` stanju, što predstavlja potencijalni, ali ne i sigurni konflikt. U skladu sa ovim, celobrojna vrednost na koju ukazuje pokazivač `flag` postavlja se na 1 (u slučaju sigurnog konflikta), 2 (potencijalni konflikt) ili 0 (nema konflikta), kako bi u slučaju konflikta ili potencijalnog konflikta simulator štampao na ekranu odgovarajuću poruku korisniku.

Kao što je ranije pomenuto, rezoluciona funkcija se pridružuje signalu korišćenjem operatora `:`. Na primer, ako želimo da signal `bus` bude žičana veza izlaza više gejtova kod koje se rezultujuće stanje odredjuje funkcijom `bus15res`, signal `bus` ćemo deklarisati na sledeći način.

```
signal fift_full:bus15res bus;
```

Karakteristika rezolucione funkcije je da ona deluje trenutno. Ako je potrebno rezultat koji daje rezoluciona funkcija dodeliti signalu posle odredjenog kašnjenja, mora se koristiti drugi mehanizam modeliranja. Recimo, ako je kapacitivnost magistrale velika, kad svi drajveri odu u stanje visoke impedanse prethodno stanje se održava na ovoj kapacitivnosti još izvesno vreme. Kod simulatora HILO ova pojava se modelira tako što se specificira da je magistrala tipa TRI, TRI0, TRI1 ili TRIREG. Magistrala tipa TRI (tri state wire) zadržava prethodno stanje u toku konačnog, zadatog vremenskog perioda, a zatim prelazi u stanje `'Z'`. Slično je sa magistralama tipa TRI0 i TRI1, s tim što je njima modelirano prisustvo pulldown i pullup otpornika, pa magistrala ne odlazi

u stanje 'Z', već u stanje 'L', odnosno 'H'. Magistrala tipa TRIREG (tri state register) drži poslednje aktivno stanje beskonačno dugo po deaktiviranju drajvera. Ove pojave nazvaćemo memorijskim svojstvima signala, a način njihovog modeliranja biće tema naredne glave

6.8 Modeliranje memorijskih svojstava signala

Memorijska svojstva signala modeliraćemo vezivanjem specijalnih modula za te signale. U biblioteci za HILO sistem stanja to su moduli: pull_down, pull_up, pull_Z i capacitive_storage. Njihove deklaracije u datoteci **ss15.h** imaju sledeći izgled.

```
module pull_down (fift_full inout y) action (double, double, double);
module pull_up   (fift_full inout y) action (double, double, double);
module pull_Z    (fift_full inout y) action (double, double, double);
module capacitive_storage (fift_full inout y);
```

U pitanju su jednostavni moduli bez modelske klase, sa samo jednim ulazno/izlaznim terminalom y. Moduli pull_down, pull_up i pull_Z imaju akcione parametre: minimalno, tipično i maksimalno vreme držanja stanja. Modul capacitive_storage drži prethodno stanje beskonačno dugo, odnosno do aktiviranja nekog drugog drajvera, tako da ne prima akcione parametre.

U biblioteci za sistem sa četiri stanja objašnjeno je modeliranje pulldown i pullup otpornika. U HILO sistemu stanja kod je vrlo sličan, treba samo uzeti u obzir da moduli pull_down i pull_up nisu drajveri, odnosno, stanje koje oni šalju na izlazni terminal y mora biti oslabljeno. Za slabljenje stanja može se iskoristiti tabela tech_tab tako što se specificira tehnologija WEAK_LOW_WEAK_HIGH, a definisana je i tabela waken_tab sledećeg izgleda.

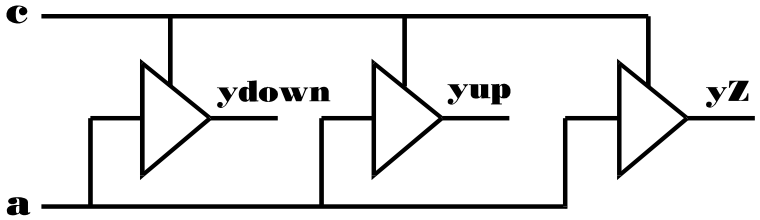
```
fift_t weaken_tab[15] = {
//   X    D    U    W    B    T    N    F    R    P    0    L    Z    H    1
    'W', 'W', 'W', 'W', 'F', 'R', 'L', 'F', 'R', 'H', 'L', 'L', 'Z', 'H', 'H'
};
```

Sledećim kodom definisan je modul pull_Z u datoteci **gates.hi**.

```
module pull_Z (signal fift_full inout y) {
  action (double mindelay, double typdelay, double maxdelay) {
    delay_Z_state: process {
      int initialized=0; fift_full last_y='Z';
      param delay;

      delay[0]=mindelay; delay[1]=typdelay; delay[2]=maxdelay;
      if (!initialized) { y <- 'Z'; initialized=1; }

      wait y;
      if (y=='Z') {
        y <- weaken_tab[last_y];    // hold last state until delay expires
        wait for delay[simulationmode];
        last_y='Z';
        y <- (fift_full)'Z';
      }
      else {
        last_y=y;
        y <- 'Z';                   // line active, disconnect
      }
    }
  }
}
```

Kad signal y ode u stanje 'Z' modul pull_Z ga vraća u oslabljeno prethodno stanje (koristi se tabela weaken_tab). Po isteku vremena delay[simulationmode] modul pull_Z na izlazni terminal y postavlja stanje 'Z'. Kako se stanje 'Z' na ulazu svih logičkih elemenata osim

transmisionog gejta tumači kao neodredjeno stanje `'X'`, njegova pojava na signalu vezanom za terminal `y` znači da je akumulirana količina naelektrisanja istekla sa parazitne kapacitivnosti ovog signala, tako da signal više nema definisano stanje.



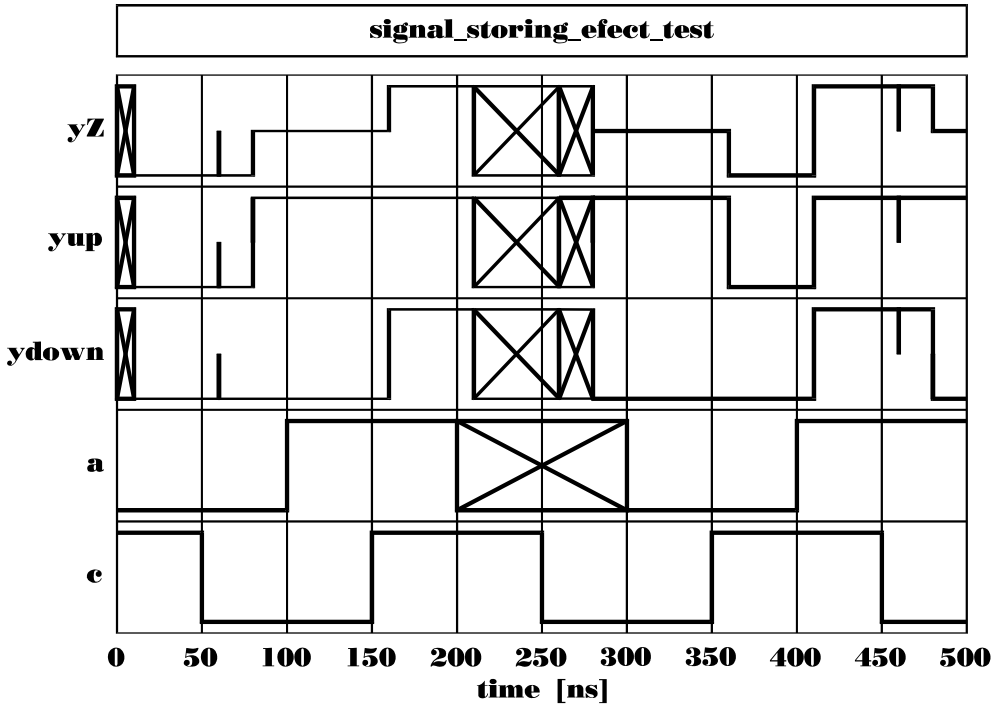
Slika 6.2 *Kolo za testiranje modula* `pull_down`, `pull_up` *i* `pull_Z`

Za ilustraciju funkcionisanja modula `pull_down`, `pull_up` i `pull_Z` poslužićemo se kolom sa slike 6.2. Za isti ulazni priključak `a` i kontrolni priključak `c` vezana su tri trostatička bafera. O modeliranju trostatičkih kola biće reči u narednim glavama, a ovde ćemo samo reći da baferi sa slike 6.2 prenose stanje sa ulaza na izlaz (stanje `'Z'` pretvara se u stanje `'X'`) kad je kontrolni ulaz u stanju `'1'`, a izlaz im ide u stanje visoke impedanse kad je kontrolni ulaz u stanju `'0'`. Za izlaz prvog bafera vezan je modul `pull_down`, za izlaz drugog bafera modul `pull_up`, a izlaz trećeg bafera modul `pull_Z`. Opis kola sa slike 6.2 za simulaciju je sledeći.

```
# include "ss15.h"
library model15, gates15, op15;
root signal_storing_efect_test () {
  module m15::bufif1 gdown, gup, gZ;
  module pull_down rdown;
  module pull_up rup;
  module pull_Z rZ;

  signal fift_full a='0', c='1';
  signal fift_full:bus15res ydown, yup, yZ;

  rdown (ydown) action (10ns, 15ns, 20ns);
  gdown (ydown, a, c) model=model_10ns;

  rup (yup) action (10ns, 15ns, 20ns);
  gup (yup, a, c) model=model_10ns;

  rZ (yZ) action (10ns, 15ns, 20ns);
  gZ (yZ, a, c) model=model_10ns;

  timing { tstop = 500ns; }
  out {
    signal four_t c     { return con15to4[c];     };
    signal four_t a     { return con15to4[a];     };
    signal four_t ydown { return con15to4[ydown]; };
    signal four_t yup   { return con15to4[yup];   };
    signal four_t yZ    { return con15to4[yZ];    };
  }
  action {
    process initial {
      simulationmode = MAX_MODE;
      c <- '0' after  50ns, '1' after 150ns,
           '0' after 250ns, '1' after 350ns, '0' after 450ns;
      a <- '0' after   0ns, '1' after 100ns,
           'x' after 200ns, '0' after 300ns, '1' after 400ns;
    }
  }
}
```

Model `model_10ns` pridružen baferima definisan je u modelskoj datoteci **model15.hi** i kod njega su sva kašnjenja postavljena na vrednost `10ns`, a parametar `skew` anuliran. Izabran je mod

simulacije MAX_MODE, što znači da se koriste maksimalne vrednosti parametara kašnjenja (10ns za bafere, 20ns za module pull_down, pull_up i pull_Z). Rezultat simulacije prikazan je na slici 6.3. Izlazna datoteka namerno nije procesirana procesorom NZD, da bi se videli impulsi nultog trajanja u trenucima kad izlazi bafera pokušavaju da predju u stanje 'Z', ali se odmah uspostavlja prethodno stanje i drži narednih 20ns. Posle 20ns na izlazu ydown uspostavlja se stanje slabe logičke nule 'L', na izlazu yup uspostavlja se stanje slabe logičke jedinice 'H', a na izlazu yZ stanje 'Z'. U naredbi out vrši se konverzija iz sistema sa petnaest u sistem sa četiri stanja, jer grafički postprocesor Art2.1 prepoznaje samo četiri logička stanja.



Slika 6.3 *Rezultat simulacije kola sa slike 6.2*

Modul capacitive_storage definisan je u datoteci **gates15.hi** na sledeći način.

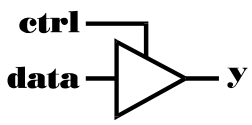
```
# define VERY_SHORT 1fs
module capacitive_storage (signal fift_full inout y) {
  action {
    store: process {
      int initialized=0; fift_full last_y='Z';

      if (!initialized) { y <- 'Z'; initialized=1; }

      wait (y);
      if (y=='Z') {
        y <- weaken_tab[last_y];    // hold last state during drivers inactivity
        wait for VERY_SHORT;        // avoid autotriggering
      }
      else {
        y <- last_y;                // memorize last active state
        y <- 'Z';                   // disconnect storage element from line
      }
    }
  }
}
```

Makroom VERY_SHORT definisan je vremenski interval posle koga se proces store ponovo aktivira i čeka dogadjaj na signalu y. Ovaj vremenski interval treba da bude dovoljno kratak da obezbedi da u medjuvremenu ne bude dogadjaja u simuliranom kolu. Ilustracija upotrebe modula capacitive_storage biće data u narednoj glavi.

6.9 Modeliranje trostatičkih kola u HILO sistemu stanja

Specifičnost sistema sa petnaest stanja koji ovde koristimo najbolje se ogleda kod modeliranja trostatičkih kola. Zahvaljujući mogućnostima ovog sistema stanja, postiže se smanjenje neodredjenosti na izlazu trostatičkog gejta kada mu je kontrolni ulaz u neodredjenom stanju.

Konkretno, posmatrajmo trostatički bafer sa slike 6.4. Kada je kontrolni ulaz `ctrl` u stanju `'X'`, u sistemu sa četiri stanja izlaz bafera se takodje postavlja u neodredjeno stanje `'X'`. Medjutim, stanje na izlazu zapravo nije tako pesimistički odredjeno. Neka je na ulazu `data` stanje logičke jedinice `'1'`. Na izlazu se onda može pojaviti ili stanje sa ulaza (`'1'`), ako je kontrolni ulaz u stanju `'1'`, ili stanje visoke impedanse `'Z'`, ako je kontrolni ulaz u stanju `'0'`. Dakle, na izlazu se nikako ne može pojaviti stanje logičke nule. U slučaju da je izlaz bafera vezan na magistralu sa još jenim drajverom, ukoliko je drugi drajver magistrale u stanju logičke jedinice, magistrala treba da bude u stanju logičke jedinice, a ne u neodredjenom stanju `'X'`, s obzirom da nikako ne može doći do konflikta na magistrali. U HILO sistemu stanja postoje odgovarajuća stanja koja opisuju smanjeni stepen neodredjenosti. Našem baferu iz gornje analize odgovaralo bi stanje `'T'` koje predstavlja mogućnost da je na izlazu ili stanje `'1'` ili stanje `'Z'`. Primenom funkcije rezolucije `bus15res` na drajver u stanju `'T'` i drajver u stanju `'1'`, dobija se rezultujuće stanje magistrale `'1'`. U sistemu sa četiri stanja rezultujuće stanje magistrale bilo bi `'X'`, što je suviše pesimistično.



data	ctrl	y
0		Z
1	0	Z
X/Z		Z
0		0
1	1	1
X/Z		X
0		X
1	X/Z	X
X/Z		X

a) *b)*

data	ctrl	y
0		Z
1	0	Z
X/Z		Z
0		0
1	1	1
X/Z		X
0		B
1	X/Z	T
X/Z		X

c)

Slika 6.4 *Trostatički bafer a) simbol b) tablica istinitosti u sistemu sa četiri stanja c) tablica istinitosti u HILO sistemu stanja*

U nastavku je dat kod kojim je modeliran trostatički bafer sa slike 6.4.

```
module m15::bufif1 (fift_full out y; fift_full in data, ctrl) {
  action {
    process post_structural {
      (@data)->add_cap(this.termcap[simulationmode]);
      (@ctrl)->add_cap(this.termcap[simulationmode]);
      (@y)->add_cap(this.termcap[simulationmode]);
    }
    process (data, ctrl) {
      fift_t result; three_t d3, c3;

      d3 = con15to3[data]; c3 = con15to3[ctrl];
      switch (c3) {
      case 'X':
        switch (d3) {
          case 'X': result = 'X'; break;
          case '0':
            result = 'B';           // '0' - 'Z'
            break;
          case '1':
            result = 'T';           // '1' - 'Z'
            break;
          default : warning("module bufif1 : This cannot happen!",1);
```

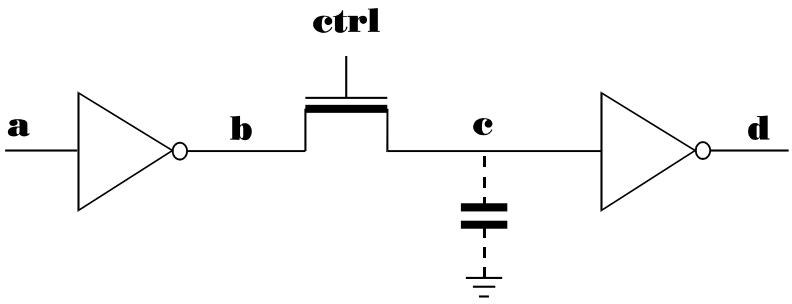
```
        }                           // switch (d3)
        break;
      case '0': result = 'Z'; break;
      case '1': result = con3to15[d3]; break;
      default : warning("module bufif1 : This cannot happen! No way!",1);
      }                             // switch (c3)
      result = tech_tab[this.tech][result];
      y <- result after Zdelay (result, y,
                                (@y)->get_tcap()-this.termcap[simulationmode]);
    } // process (data, ctrl)
  }
} // module bufif1
```

Treba uočiti stanja `'T'` i `'B'` koja se šalju na izlaz bafera kada je ulaz `ctrl` u neodredjenom stanju `'X'`. Takodje je bitno uočiti da modul `bufif1` stanje `'Z'` na ulazu pretvara u stanje `'X'` na izlazu.

Za razliku od bafera, jednosmerni transmisioni gejt, modul po imenu `moveif1`, prenosi stanje `'Z'` sa ulaza na izlaz neizmenjeno. Pri prenosu signala može se eventualno zahtevati slabljenje, recimo specificiranjem tehnologije `WEAK_LOW_WEAK_HIGH`. Navešćemo ovde definiciju modula `moveif1_init` kod koga je ugradjeno nulto kašnjenje pri inicijalizaciji, a zatim ćemo ilustrovati njegovu upotrebu simulacijom kola sa slike 6.5.

```
module m15::moveif1_init (fift_full out y; fift_full in data, ctrl) {
  action {
    process post_structural {
      (@data)->add_cap(this.termcap[simulationmode]);
      (@ctrl)->add_cap(this.termcap[simulationmode]);
      (@y)->add_cap(this.termcap[simulationmode]);
    }
    process (data, ctrl) {
      fift_t result; three_t c3; four_t d4;

      d4 = con15to4[data]; c3 = con15to3[ctrl];
      switch (c3) {
        case 'X':
          switch (d4) {
            case 'X': result = 'X'; break;
            case 'Z': result = 'Z'; break;
            case '0':
              result = 'B';         // '0' - 'Z'
              break;
            case '1':
              result = 'T';         // '1' - 'Z'
              break;
            default :
              warning("module moveif1_init : This cannot happen!",1);
          } // switch (d4)
          break;
        case '0': result = 'Z'; break;
        case '1': result = data; break;
        default :
          warning("module moveif1_init : This cannot happen! No way!",1);
      } // switch (c3)
      result = tech_tab[this.tech][result];
      if (now==0.0)
        y <- (fift_full) result;          // zero delay during initialization
      else
        y <- (fift_full) result after Zdelay(result, y, (@y)->get_tcap());
    }
  }
} // module moveif1_init ()
```

Pri korišćenju modula `moveif1_init` treba imati u vidu da se svi dogadjaji koji se dešavaju u trenutku `0.0s` smatraju inicijalnom aktivnošću i da modul na njih reaguje sa nultim kašnjenjem. Dakle, ne sme se dovesti pobuda u trenutku `0.0s`.



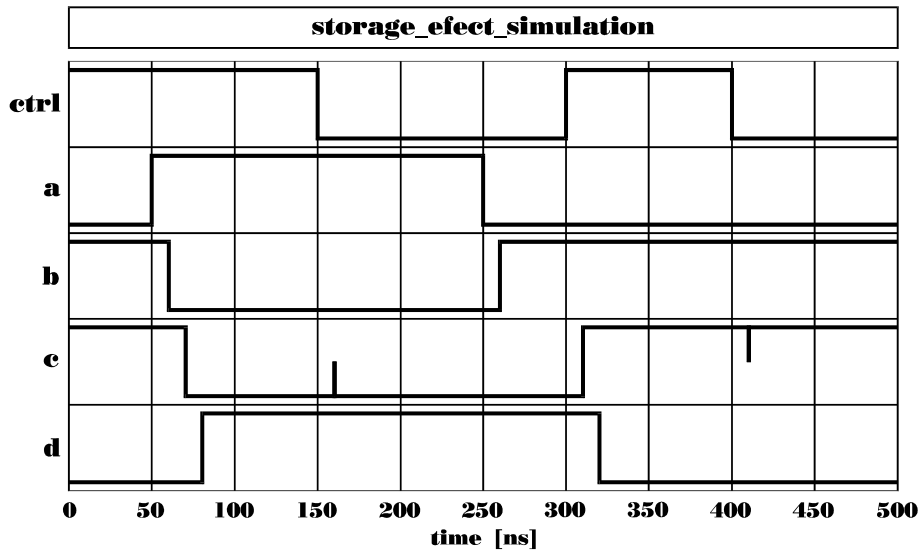
Slika 6.5 *Parazitna kapacitivnost kao memorijski elemenat*

U kolu sa slike 6.5 prisutna je parazitna kapacitivnost u tački `c`. Kad transmisioni gejt prekine vezu sa tačkom `b`, ova kapacitivnost ostaje praktično izolovana i memoriše prethodno stanje u tački `c` teorijski beskonačno dugo. Za modeliranje ove pojave iskoristićemo modul `capacitive_storage` definisan u prethodnoj glavi. Opis kola sa slike 6.5 za simulaciju dat je u nastavku.

```
# include "ss15.h"
library model15, gates15, op15;

root storage_efect_simulation () {
  module m15::inv g1, g2;                 // inverters
  module capacitive_storage st;           // storage element
  module m15::moveif1_init tg;            // unidirectional transmission gate

  signal fift_full a='0', b='1', d='0', ctrl='1';
  signal fift_full:bus15res c='1';

  g1 (b, a) model = model_10ns;
  g2 (d, c) model = model_10ns;
  st (c);
  tg (c, b, ctrl) model = model_10ns;

  timing { tstop = 500ns; }
  out {
    signal four_t d { return con15to4[d]; };
    signal four_t c { return con15to4[c]; };
    signal four_t b { return con15to4[b]; };
    signal four_t a { return con15to4[a]; };
    signal four_t ctrl { return con15to4[ctrl]; };
  }

  action {
    process initial {
      a <- '1' after 50ns, '0' after 250ns;
      ctrl <- '0' after 150ns, '1' after 300ns, '0' after 400ns;
    }
  }
}
```

Invertori i transmisioni gejt imaju istu vrednost svih parametara kašnjenja - `10ns` (model `model_10ns` je tako definisan u datoteci **model15.hi**). Za signal `c` koristi se najopštija rezoluciona funkcija `bus15res`. Zahvaljujući tome što modul `capacitive_storage` slabi memorisano stanje (videti definiciju modula `capacitive_storage` u prethodnoj glavi), ne dolazi do konflikta pri ponovnom otvaranju transmisionog gejta. Na slici 6.6 koja predstavlja rezultat simulacije, jasno se

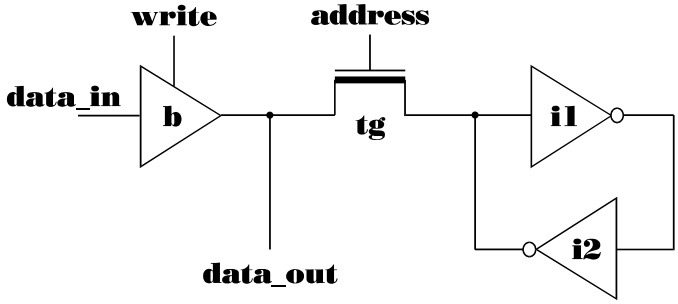
uočavaju pikovi nultog trajanja u trenucima kada funkcija rezolucije daje stanje `'Z'` u tački `c`, a modul `capacitive_storage` obnavlja prethodno aktivno stanje signala.



Slika 6.6 *Rezultat simulacije kola sa slike 6.5*

U svakom simulatoru čiji se rad bazira na sinhronizaciji paralelnih procesa, poput simulatora Alecsis2.1, modeliranje dvosmernog transmisionog gejta predstavlja veliki problem. Kod VHDL-a je taj problem rešen [Coel90] upotrebom mehanizama modeliranja koji u AleC++ nisu ugradjeni. Konkretno, za ovaj model neophodno je da proces može da se aktivira pri aktivnosti (a ne pri dogadjaju) na signalu, što kod simulatora Alecsis2.1 nije moguće. Zato biblioteka za HILO sistem stanja ne sadrži model dvosmernog transmisionog gejta.

Ovo nije velika mana, s obzirom da se osobina dvosmernosti transmisionog gejta vrlo retko koristi pri projektovanju digitalnih kola. Jedan od poznatih primera upotrebe dvosmernog transmisionog gejta je statička RAM ćelija prikazana na slici 6.7. Invertori `i1` i `i2` memorišu stanje. Transmisioni gejt `tg` dozvoljava pristup pri upisu i čitanju stanja ćelije. Bafer `b` koristi se za upis novog stanja u ćeliju. Da bi upis bio moguć, drajver `b` mora biti jači od invertora `i2`, pa se pri projektovanju layout-a o tome vodi računa. Novo stasnje upisuje se u ćeliju kada su obe linije `address` i `write` u stanju logičke jedinice, a memorisano stanje čita se iz ćelije kada je adresna linija u stanju logičke jedinice, a linija `write` u stanju logičke nule.



Slika 6.7 *Statička RAM ćelija*

Simuliraćemo ovo kolo na taj način što ćemo razviti specijalni model transmisionog gejta. Naime, uočava se da transmisioni gejt pri čitanju prenosi podatke zdesna u levo, a pri upisu sleva u desno. Dakle, smer prenosa podataka je poznat i zavisi od stanja na `write` ulazu. Treba još obratiti pažnju na to da se sleva u desno prenose jaka logička stanja (bafer `b` je jak), a zdesna ulevo slaba logička stanja (invertor `i2` je slab). Ovakva komponenta može se modelirati na sledeći način.

```
module m15::static_RAM_cell_tg (fift_full inout left, right;
                                fift_full in direction, enable) {
```

```
  action {
    process post_structural {
      (@left)->add_cap(this.termcap[simulationmode]);
      (@right)->add_cap(this.termcap[simulationmode]);
      (@enable)->add_cap(this.termcap[simulationmode]);
    }
    process (direction, enable, left, right) {
      three_t dir, en;

      en = con15to3[enable];  dir = con15to3[direction];
      if (now==0.0) {                          // model initializaiton
        right <- 'Z';  left  <- 'Z';
      }
      else {
        if (dir == '1') {                 // ->
          if (en=='1') {                       // address enabled
            right <- left after this.Zdelay(left, right, (@right)->get_tcap());
          }
          else if (en=='0') {                  // address dissabled
            right <- 'Z' after this.Zdelay('Z', right, (@right)->get_tcap());
          }
          else {                               // address unknown
            right <- 'X' after this.Zdelay('X', right, (@right)->get_tcap());
          }
        }
        else if (dir == '0') {      // <-
          if (en=='1') {                   // address enabled
            left <- (fift_full) (tech_tab[WEAK_LOW_WEAK_HIGH][right])
                        after this.Zdelay(right, left, (@left)->get_tcap());
          }
          else if (en=='0') {              // address dissabled
            left <- 'Z' after this.Zdelay('Z', left, (@left)->get_tcap());
          }
          else {                           // address unknown
            left <- 'W' after this.Zdelay('W', left, (@left)->get_tcap());
          }
        }
        else {                        // <->, direction not determined
          left  <- 'W' after this.Zdelay('W', left,  (@left)->get_tcap());
          right <- 'X' after this.Zdelay('X', right, (@right)->get_tcap());
        }
      }
    }
  }
} // static_RAM_cell_tg
```

Na ulazni terminal `direction` dovodi se signal `write` radi detekcije smera protoka signala. Stanje na ulazu `direction` konvertuje se u sistem sa tri stanja, tako da promenljiva `dir` može imati vrednost `'1'`, `'0'` ili `'X'`. Vrednost `'1'` znači smer sleva u desno (upis podatka u ćeliju). Stanje se prenosi bez slabljenja. Vrednost `'0'` promenljive `dir` znači smer zdesna u levo (čitanje memorisanog podatka). Stanje se oslabljuje korišćenjem tabele `tech_tab` i specificiranjem tehnologije `WEAK_LOW_WEAK_HIGH`. Ako je vrednost promenljive `dir` neodredjena (`'X'`), na levu stranu se šalje jako neodredjeno stanje `'X'`, a na desnu stranu slabo neodredjeno stanje `'W'`. Na ulazni terminal `enable` vezuje se signal `address`. Ovo je ulaz za dozvolu pristupa ćeliji. Promenljiva `en` sadrži vrednost ulaza `enable` posle konverzije u sistem sa tri stanja. Vrednost `'1'` promenljive `en` dozvoljava prenos signala u smeru odredjenom promenljivom `dir`, vrednost `'0'` prekida prenos signala (na izlaz odabran promenljivom `dir` šalje se stanje `'Z'`). U slučaju da je vrednost `en` neodredjena, na izlaz se šalje neodredjeno stanje (na desnu stranu stanje `'X'`, a na levu stanje `'W'`).

U kolu sa slike 6.7 invertori `i1` i `i2` formiraju zatvorenu konturu sa povratnom spregom. Prema ugradjenom mehanizmu inicijalizacije, modul `static_RAM_cell_tg` u trenutku `0.0s` šalje na obe strane stanje `'Z'`. Kako se ovo stanje tumači kao neodredjeno na ulazu invertora, stanje `'X'`

će se javiti na izlazu invertora `i1`, a zatim i stanje `'W'` na izlazu invertora `i2`. Prvi upis stanja u ćeliju poništiće slabo neodredjeno stanje na izlazu invertora `i2`. Ako želimo da već u početnom trenutku neko stanje bude upisano u memorijsku ćeliju, potreban nam je model invertora kome je moguće zadati početno stanje na izlazu. Zato je razvijen modul `inv_init` sledećeg izgleda.

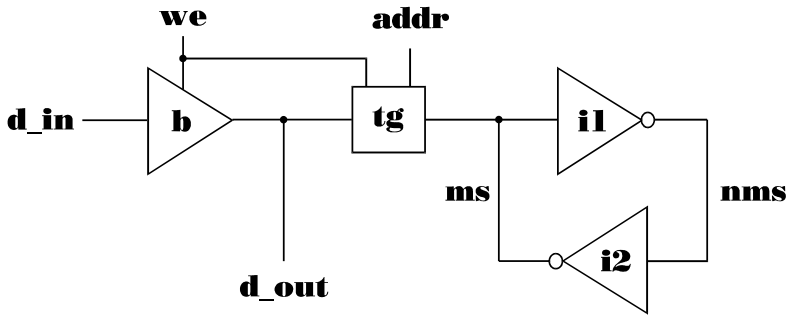
```
module m15::inv_init (fift_full out y; fift_full in a) {
  action {
    process post_structural { (@a)->add_cap(this.termcap[simulationmode]); }
    process (a) {
      three_t tmp;

      tmp = ~ con15to3[a];
      if (now==0.0) {  y <- this.init_state;  }
      else {
        y <- (fift_full) ( tech_tab[this.tech][con3to15[tmp]] )
                       after this.delay (tmp, con15to3[y], (@y)->get_tcap());
      }
    }
  }
}
```

Za potrebe inicijalizacije stanja na izlazu invertora (i drugih modula objekata klase `m15`), u modelsku klasu `m15` dodata je promenljiva `init_state`.

```
class m15 {
  ...
  fift_t init_state;
  ...
}
```

Difoltna vrednost promenljive `init_state` je `'X'`, a u modelskoj kartici se može zadati neka druga. Iskorišćen je isti mehanizam inicijalizacije kao i kod modela transmisionog gejta: trenutak `0.0s` je rezervisan za inicijalizaciju. Naravno, modul `inv_init` je deklarisan kao prijatelj klase `m15` (naredba `friend`), da bi mu se omogućio slobodan pristup promenljivoj `init_state` i parametrima kašnjenja.

Na sličan način inicijalizacija je ugradjena u model trostatičkog bafera. Odgovarajući modul nazvan je `bufif1_init`. I ovaj modul biće nam potreban pri simulaciji kola sa slike 6.7.



Slika 6.8 *Simulacioni model statičke RAM ćelije*

Kolo sa slike 6.7 modifikovano je saglasno razvijenim modelima sastavnih komponenata kako to prikazuje slika 6.8. Opis kola sa slike 6.8 za simulaciju je sledeći.

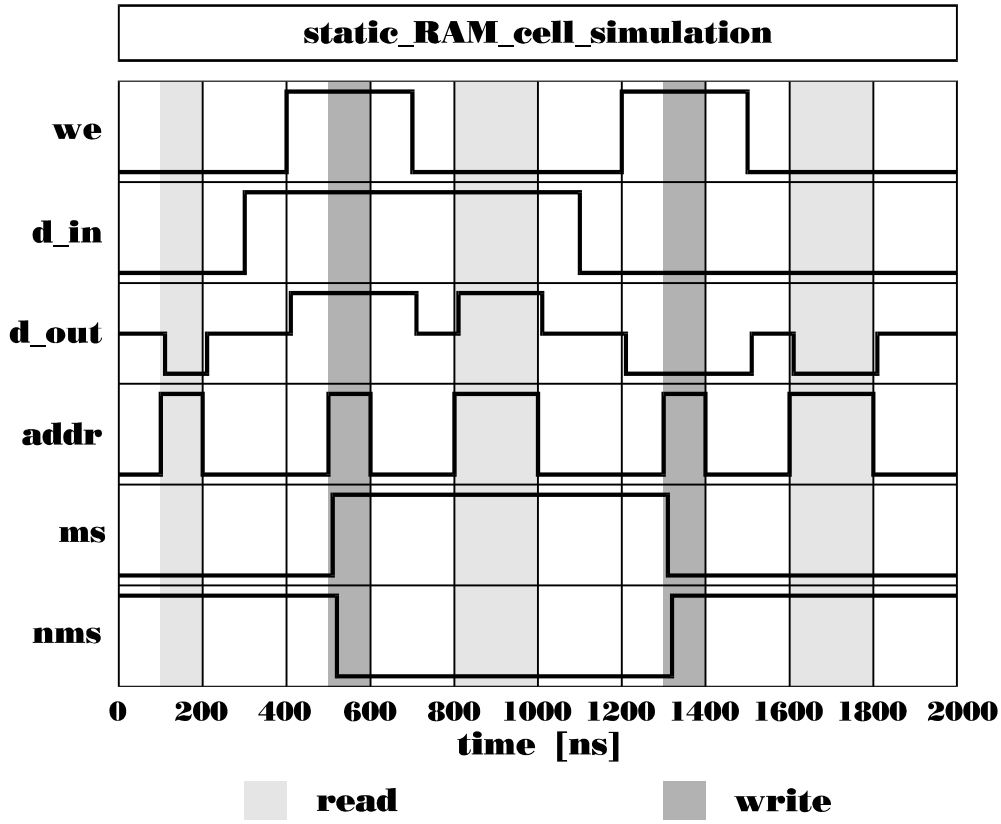
```
# include "ss15.h"
library model15, gates15, op15;
root static_RAM_cell_simulation () {
  module m15::bufif1_init b;                    // input buffer
  module m15::static_RAM_cell_tg tg;            // transmission gate
  module m15::inv_init i1, i2;                  // inverters
  signal fift_full d_in='0';                    // data in
  signal fift_full:bus15res d_out='0';          // data out
```

```
  signal fift_full:bus15res ms='0';             // memorized state
  signal fift_full nms='1';                     // inverted memorized state
  signal fift_full addr='0';                    // address line
  signal fift_full we='0';                      // write enable

  b  (d_out, d_in, we) model=strong_driver;
  tg (d_out, ms, we, addr) model=tg_model;
  i1 (nms, ms) model=strong_driver_initially_1;
  i2 (ms, nms) model=weak_driver_initially_L;

  timing { tstop = 2000ns; }
  out {
    signal four_t nms   { return con15to4[nms];   };
    signal four_t ms    { return con15to4[ms];    };
    signal four_t addr  { return con15to4[addr];  };
    signal four_t d_out { return con15to4[d_out]; };
    signal four_t d_in  { return con15to4[d_in];  };
    signal four_t we    { return con15to4[we];    };
  }
  action {
    process initial {
      addr <- '1' after 100ns, '0' after 200ns,       // read
              '1' after 500ns, '0' after 600ns,       // write
              '1' after 800ns, '0' after 1000ns,      // read
              '1' after 1300ns, '0' after 1400ns,     // write
              '1' after 1600ns, '0' after 1800ns;     // read
      we   <- '1' after 400ns, '0' after 700ns,       // write
              '1' after 1200ns, '0' after 1500ns;     // write
      d_in <- '1' after 300ns,                        // set new data
              '0' after 1100ns;                       // set new data
    }
  }
}
```

Korišćeni modeli definisani su u datoteci **model15**.hi na sledeći način.

```
model m15::strong_driver {
  delay01={10ns, 10ns, 10ns};  delay10={10ns, 10ns, 10ns};
  delayZ0={10ns, 10ns, 10ns};  delay0Z={10ns, 10ns, 10ns};
  delay1Z={10ns, 10ns, 10ns};  delayZ1={10ns, 10ns, 10ns};
  tech = STRONG_LOW_STRONG_HIGH;
}

model m15::tg_model {
  delay01={10ns, 10ns, 10ns};  delay10={10ns, 10ns, 10ns};
  delayZ0={10ns, 10ns, 10ns};  delay0Z={10ns, 10ns, 10ns};
  delay1Z={10ns, 10ns, 10ns};  delayZ1={10ns, 10ns, 10ns};
  tech = STRONG_LOW_STRONG_HIGH;
}

model m15::strong_driver_initially_1 {
  init_state='1';
  delay01={10ns, 10ns, 10ns};  delay10={10ns, 10ns, 10ns};
  delayZ0={10ns, 10ns, 10ns};  delay0Z={10ns, 10ns, 10ns};
  delay1Z={10ns, 10ns, 10ns};  delayZ1={10ns, 10ns, 10ns};
  tech = STRONG_LOW_STRONG_HIGH;
}

model m15::weak_driver_initially_L {
  init_state='L';
  delay01={10ns, 10ns, 10ns};  delay10={10ns, 10ns, 10ns};
  delayZ0={10ns, 10ns, 10ns};  delay0Z={10ns, 10ns, 10ns};
  delay1Z={10ns, 10ns, 10ns};  delayZ1={10ns, 10ns, 10ns};
  tech = WEAK_LOW_WEAK_HIGH;
}
```

Rezultat simulacije prikazan je na slici 6.9. U memorijsku ćeliju inicijalno je upisano stanje logičke nule (modelska kartica `weak_driver_initially_L` definiše početno stanje slabe logičke nule na izlazu invertora `i2`, a modelska kartica `strong_driver_initially_1` definiše stanje jake logičke jedinice na izlazu invertora `i1`). Stanje na izlazu bafera `b` inicijalizovano je kao `'Z'`, jer mu je kontrolni ulaz `we` (write enable) u stanju `'0'` na početku simulacije. Adresna linija `addr` je takodje neaktivna na početku, tako da je ćelija izolovana od izlaza `d_out` i na izlazu je stanje visoke impedanse. Prvi dogadjaj pobude dešava se u trenutku `100ns` - adresna linija `addr` se aktivira, tako da je dozvoljeno čitanje memorisanog stanja. Sa slike 6.9 vidimo da se stanje logičke nule pojavljuje na izlaznoj liniji `d_out`. Kad se adresna linija vrati u stanje `'0'`, izlaz ponovo prelazi u stanje `'Z'`. Novo stanje upisuje se u memorijsku ćeliju tako što se prvo aktivira linija `we` (u trenutku `400ns` ide u stanje `'1'`), čime se stanje sa ulazne linije podataka `d_in` prebacuje na izlaz bafera `b` (na signal `d_out`), a zatim se aktivira i adresna linija (u trenutku `500ns` ide u stanje `'1'`). Stanje sa izlaza bafera (linija `d_out`) prenosi se na signal `ms` pri čemu se poništava prethodno memorisano stanje koje održava slabi invertor `i2`. Čim invertor `i1` promeni stanje u tački `nms` može se smatrati da je novo stanje upisano u ćeliju, tako da se najpre deaktivira adresna linija, a zatim i linija `we`. Ovim je završen upis novog stanja u memorijsku ćeliju. Na slici 6.9 uočavamo dva ciklusa upisa podatka u ćeliju i tri ciklusa čitanja memorisanog podatka.

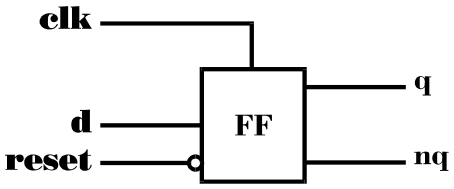


Slika 6.9 *Rezultat simulacije statičke RAM ćelije*

6.10 Modeliranje flip-flopova u HILO sistemu stanja

Daćemo ovde kompletan model sinhronog D flip-flopa iz biblioteke za HILO sistem stanja. Ova komponenta nije iz standardnog skupa komponenti koje prepoznaje simulator HILO, ali je neophodna za simulaciju sekvencijalnih kola. D flip-flop se okida prednjom ivicom taktnog signala `clk` i ima asinhroni ulaz za resetovanje `reset`. Simbol flip-flopa prikazan je na slici 6.10. Vremenski model flip-flopa je nešto složeniji nego do sada korišćeni modeli. Naime, kašnjenja prednje i zadnje ivice na izlazima `q` i `nq` razlikuju se u zavisnosti od ulaza na kome se desio

dogadjaj koji je izazvao promenu na izlazu. Recimo, nije isto kašnjenje prednje ivice na `nq` izlazu kad je ova nastala kao rezultat dejstva `reset` ulaza i kad je nastala kao rezultat memorisanja logičke nule sa `d` ulaza pri pojavi prednje ivice `clk` signala. Sva su kašnjenja zadata minimalnom, tipičnom i maksimalnom vrednošću. Zadate su i kapacitivnosti ulaznih terminala `clk`, `d` i `reset`. Kapacitivnosti izlaznih terminala nisu zadate, već se smatra da je njihov uticaj ugradjen u propagaciono kašnjenje flip-flopa. Flip-flop ima zadate vremenske uslove postavljanja i držanja signala na ulazima `d` i `reset` u odnosu na aktivnu ivicu taktnog signala.



Slika 6.10 *D flip-flop sa asinhronim ulazom za resetovanje*

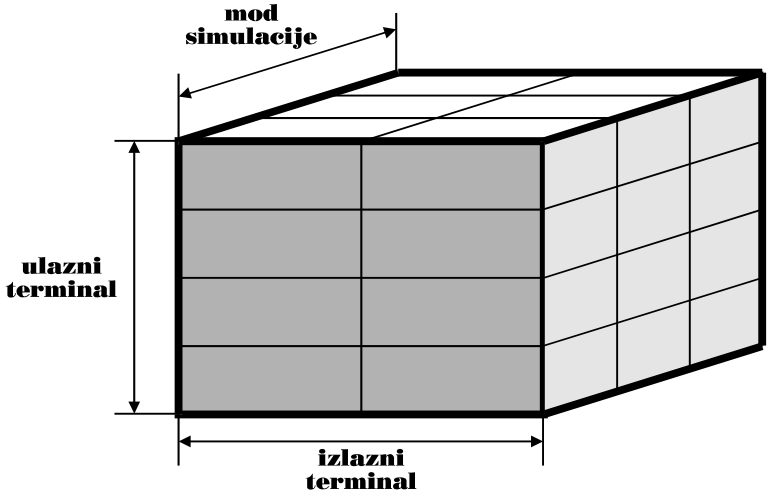
Da bi bilo moguće modelirati sve ove osobine flip-flopa, definisana je posebna modelska klasa `dff15`. Ova klasa sadrži parametre modela D flip-flopova: kašnjenja, nagib porasta kašnjenja sa kapacitivnim opterećenjem izlaza, parazitne kapacitivnosti ulaznih terminala, tehnologiju izrade, vremena držanja i postavljanja signala. Da bi se modeliranje učinilo što opštijim, definisani su makroi koji pojašnjavaju značenja odredjenih parametara.

```
# define    FF_INPUTS_MIN_NUMBER    0
# define    D_INPUT           0
# define    RESET_INPUT 1
# define    CLK_INPUT   2
# define    SET_INPUT   3
# define    FF_INPUTS_MAX_NUMBER    3
# define    FF_INPUTS_NUMBER        4

# define    FF_OUTPUTS_MIN_NUMBER   0
# define    Q_OUTPUT    0
# define    NQ_OUTPUT   1
# define    FF_OUTPUTS_MAX_NUMBER   1
# define    FF_OUTPUTS_NUMBER       2

class dff15 {                       // model class for D-type flip-flops
      param       delay01[FF_INPUTS_NUMBER][FF_OUTPUTS_NUMBER],
                  delay10[FF_INPUTS_NUMBER][FF_OUTPUTS_NUMBER];
      param       skew01[FF_OUTPUTS_NUMBER], skew10[FF_OUTPUTS_NUMBER];
      param       termcap[FF_INPUTS_NUMBER];
      int         tech;
      double      dsetup,     dhold;
      double      resetsetup, resethold;
      double      setsetup,   sethold;
      dff15       ();
      >dff15 ();
  public :
      double delayffdr   (three_t, three_t, int, int, double);
      double delayffdrds (three_t, three_t, int, int, double);
            // modules permitted to access class ff15 private members
      friend module dffdr, dffdrds;
};
```

Konstruktor klase `dff15` postavlja diflotne vrednosti parametara modela, a procesor proverava logičnost parametara koje je zadao korisnik pri definisanju modelske kartice tipa `dff15`. Trodimenzionalna struktura koja je korišćena za podatke `delay01` i `delay10` prikazana je na slici 6.11. Prva dimenzija predstavlja ulazni terminal na kome se javio dogadjaj koji je izazvao promenu na izlazu (koriste se definisani makroi). Druga dimenzija predstavlja odgovarajući izlazni terminal. Treća dimenzija je mod simulacije, s obzirom da je tip `param` sastavljen od tri podatka tipa `double`.

Recimo, sledeći izraz predstavlja tipično kašnjenje zadnje ivice signala na izlazu `q` usled resetovanja flip-flopa.

```
delay10 [RESET_INPUT][Q_OUTPUT][TYP_MODE]
```



Slika 6.11 *Struktura podataka za parametre kašnjenja D flip--flopa*

Kao prijatelji klase `dff15` deklarisani su D flip-flop sa direktnim ulazom za resetovanje `dffdr` i D flip-flop sa direktnim ulazima za resetovanje i setovanje `dffdrds`. Ovde ćemo se zadržati samo na prvom od njih. Za svaki od pomenutih flip-flopova definisana je posebna funkcija za kašnjenje. Našem flip-flopu odgovara funkcija za kašnjenje `delayffdr`.

```
double dff15::delayffdr (three_t ns, three_t os, int from, int to, double c) {
  double help1, help2;

  if (ns == '0') return (pick(delay10[from][to], simulationmode) +
                                    pick(skew10[to], simulationmode) * c);
  if (ns == '1') return (pick(delay01[from][to], simulationmode) +
                                    pick(skew01[to], simulationmode) * c);
                                    // therefore, ns == 'X'
  if (os == '1') return (pick(delay10[from][to], simulationmode) +
                                    pick(skew10[to], simulationmode) * c);
  if (os == '0') return (pick(delay01[from][to], simulationmode) +
                                    pick(skew01[to], simulationmode) * c);
  if (os == 'X') {                  // take the worst case
    help1 = pick(delay01[from][to], simulationmode) +
                                    pick(skew01[to], simulationmode) * c;
    help2 = pick(delay10[from][to], simulationmode) +
                                    pick(skew10[to], simulationmode) * c;
    return  (help1 > help2 ? help1 : help2);
  }
}
```

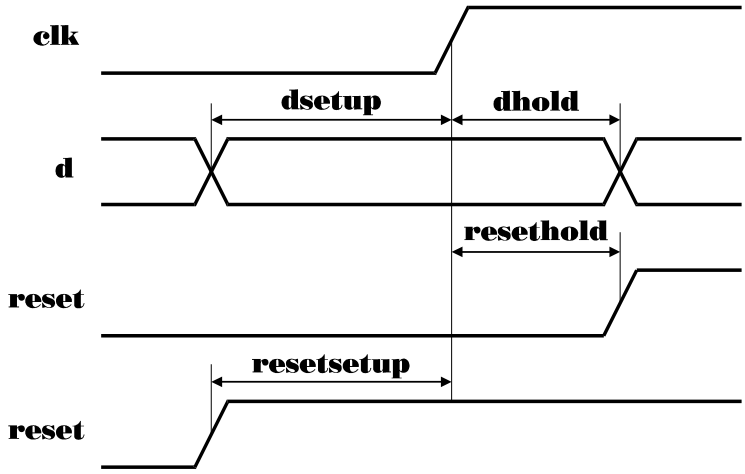
Modul `dffdr` definisan je tako da obuhvata sve pomenute zahteve modeliranja.

```
module ff15::dffdr (fift_full out q, nq; fift_full in d, clk, reset) {
  action {
    reset_setup_check: process {
       double last_reset=0.0;
                                        // inactive setup
       wait reset while reset != '1';   // wait for inactive reset
       last_reset = now;
       wait clk while clk != '1';       // wait on clock's rising edge
       if (now-last_reset < resetsetup) warning("reset setup violation", 0);
    }
    d_setup_check: process {
```

```
   double last_d=0.0;
   wait d;                              // wait on an event on data input
   last_d = now;
   wait clk while clk != '1';           // wait on clock's rising edge
   if (now-last_d < dsetup) warning("data setup violation", 0);
}
reset_hold_check: process {
   double last_clk=0.0;
                                        // active hold
   wait clk while clk != '1';           // wait on clock's rising edge
   last_clk = now;
   if (reset == '0') {
     wait reset;                        // wait on reset to become inactive
     if (now - last_clk < resethold) warning ("reset hold violation", 0);
   }
}
d_hold_check: process {
   double last_clk=0.0;

   wait clk while clk != '1';           // wait on clock's rising edge
   last_clk = now;
   wait d;                              // wait on an event on data input
   if (now - last_clk < dhold) warning ("data hold violation", 0);
}
input_capacitances: process post_structural {
  (@d)->add_cap(this.termcap[D_INPUT][simulationmode]);
  (@clk)->add_cap(this.termcap[CLK_INPUT][simulationmode]);
  (@reset)->add_cap(this.termcap[RESET_INPUT][simulationmode]);
}
clk_unknown_detect: process (clk) {
  if (con15to3[clk]=='X') warning ("clk gone unknown", 0);
}
reset_unknown_detect: process (reset) {
  if (con15to3[reset]=='X') warning ("reset gone unknown", 0);
}
ff_behaviour: process (reset, clk) {
  three_t d3, nd3, reset3, clk3;
  int active_input;

  d3=con15to3[d]; reset3=con15to3[reset]; clk3=con15to3[clk]; nd3=~d3;
  if (reset->active) active_input = RESET_INPUT;
  else               active_input = CLK_INPUT;

  if(reset3 == '0') {
    d3 = '0', nd3 = '1';
    q <- (fift_full)( tech_tab[tech][(fift_t)'0'] )
                  after this.delayffdr (d3, con15to3[q], active_input,
                                        Q_OUTPUT, (@q)->get_tcap());
    nq <- (fift_full)( tech_tab[this.tech][con3to15[nd3]] )
                  after this.delayffdr (nd3, con15to3[nq], active_input,
                                        NQ_OUTPUT, (@nq)->get_tcap());
  }
  else if(reset3 == 'X') {
    d3 = nd3 = 'X';
    q <- (fift_full)( tech_tab[this.tech][con3to15[d3]] )
                  after this.delayffdr (d3, con15to3[q], active_input,
                                        Q_OUTPUT, (@q)->get_tcap());
    nq <- (fift_full)( tech_tab[this.tech][con3to15[nd3]] )
                  after this.delayffdr (nd3, con15to3[nq], active_input,
                                        NQ_OUTPUT, (@nq)->get_tcap());
  }
  else if(clk == 'X') {
    d3 = nd3 = 'X';
    q <- (fift_full)( tech_tab[this.tech][con3to15[d3]] )
                  after this.delayffdr (d3, con15to3[q], active_input,
                                        Q_OUTPUT, (@q)->get_tcap());
    nq <- (fift_full)( tech_tab[this.tech][con3to15[nd3]] )
```

```
                        after this.delayffdr (nd3, con15to3[nq], active_input,
                                                NQ_OUTPUT, (@nq)->get_tcap());
      }
      else if(clk == '1') {
        q <- (fift_full)( tech_tab[this.tech][con3to15[d3]] )
                        after this.delayffdr (d3, con15to3[q], active_input,
                                                Q_OUTPUT, (@q)->get_tcap());
        nq <- (fift_full)( tech_tab[this.tech][con3to15[nd3]] )
                        after this.delayffdr (nd3, con15to3[nq], active_input,
                                                NQ_OUTPUT, (@q)->get_tcap());
      }
    }
  }
} // module dff15::dffdr ()
```

Modul ima izlaze `q` i `nq` i ulaze `d` (ulaz za podatke), `clk` (taktni signal) i `reset` (asinhroni reset ulaz). Modeliran je pomoću osam procesa. Prva četiri procesa vrše proveru vremenskih ograničenja koje model nameće. To podrazumeva vremena postavljanja i držanja signala na `d` i `reset` ulazu u odnosu na prednju ivicu signala `clk`. Ova vremena su prikazana na slici 6.12.



Slika 6.12 *Vremena postavljanja i držanja D flip-flopa sa direktnim reset ulazom*

Sledeća dva procesa samo prate stanja na ulazima `clk` i `reset` i upozoravaju korisnika o pojavi neodredjenog stanja. Proces označen labelom `ff_behaviour` modelira logičku funkciju D flip-flopa. Kašnjenje flip-flopa odredjuje se funkcijom `delayffdr`. Funkciji se, osim novog i starog stanja na izlazu, prosledjuje informacija o aktivnom ulazu (uzročniku promene na izlazu), odgovarajućem izlazu i kapacitivnom opterećenju izlaza. U slučaju pojave neodredjenog stanja na ulazu `clk` ili `reset`, izlazi idu u neodredjeno stanje. Kako su u model flip-flopa ugradjene samo ulazne kapacitivnosti, nije potrebno oduzimanje izlazne kapacitivnosti od atributa izlaznog signala.

Navešćemo ovde još samo jednu od konkretnih modelskih kartica opisanog D flip-flopa pod imenom `dffdr_model_1`, kako bi se stekao uvid u način definisanja modelskih kartica tipa `dff15`.

```
model ff15::dffdr_model_1 {
  delay10[RESET_INPUT][Q_OUTPUT ] = { 5ns, 7ns, 11.5ns };
  delay01[RESET_INPUT][NQ_OUTPUT] = { 4.5ns, 8.5ns, 15ns };
  delay01[CLK_INPUT][Q_OUTPUT] = {15ns, 22ns, 28ns};
  delay10[CLK_INPUT][Q_OUTPUT] = {10ns, 12ns, 15ns};
  delay01[CLK_INPUT][NQ_OUTPUT] = {15ns, 20ns, 25ns};
  delay10[CLK_INPUT][NQ_OUTPUT] = {9ns, 15ns, 20.5ns};
  skew01[Q_OUTPUT] = {75., 90., 120.};   skew10[Q_OUTPUT] = {80., 100., 110.};
  skew01[NQ_OUTPUT] = {70., 90., 120.};  skew10[NQ_OUTPUT] = {85., 115., 150.};
  termcap[D_INPUT    ] = {1pF,  1.2pF,  1.7pF };
  termcap[RESET_INPUT] = {1.5pF,  2pF,  2.2pF };
  termcap[CLK_INPUT  ] = {2.5pF,  4.3pF,  5.3pF };
  tech = CMOS;
  dsetup = 5ns;      dhold = 4ns;
```

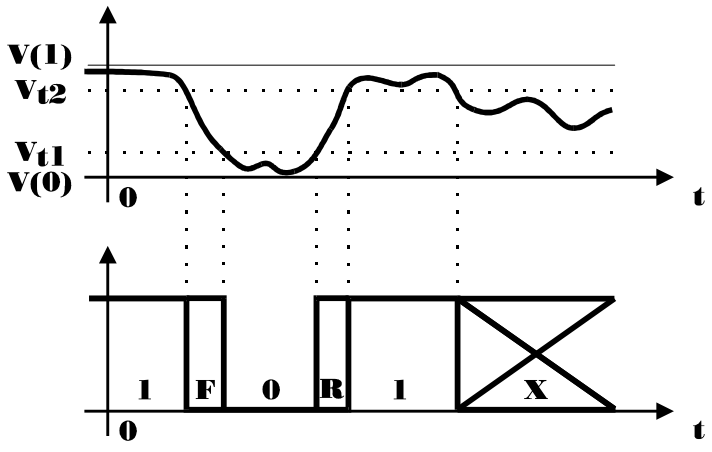
```
  resetsetup = 3ns;  resethold = 2ns;
}
```

Validnost razvijene biblioteke za HILO sistem stanja potvrdjena je simulacijama pri projektovanju cifarsko-serijskog množača [Maks93b, Mile93a, Mile93b]. Detalji se mogu naći u [Gloz94b].

7 Biblioteka za PSpice sistem stanja

Simulator PSpice je prilagodjen radu na PC računarima, te je stoga, uz MICRO-CAP i IsSpice, jedan od najšire rasprostranjenih simulatora elektronskih kola danas [Rowe91]. PSpice je integrisani hibridni simulator. Njegovu osnovu, kao i kod mnogih drugih hibridnih simulatora, čini algoritam za električnu analizu u jednosmernom, frekventnom i vremenskom domenu. Simulator ima nepromenljiv skup elektronskih komponenti koje prepoznaje. Logička simulacija je ugradjena u ovakav, u osnovi električni simulator, uvodjenjem novog tipa komponente (označena sa U) i svedena na analizu u vremenskom domenu. Kako nije učinjen napor da se sintaksa ulaznog jezika unapredi u odnosu na onu koju koristi SPICE, nema mehanizama za funkcionalno modeliranje.

Sistem stanja za logičku simulaciju je ugrađen i nema mogućnosti njegovog menjanja ili predefinisanja. PSpice podržava sistem od pet stanja, signali mogu imati sledeće vrednosti: 1 (stanje logičke jedinice), 0 (stanje logičke nule), X (neodređeno stanje), R (prelazak sa 0 na 1) i F (prelazak sa 1 na 0), kao što prikazuje slika 7.1.



Slika 7.1 *PSpice sistem stanja*

Potencijalno konfliktne situacije koje u elektronskom kolu nastaju pri kratkospajanju izlaza više logičkih elemenata (drajveri magistrale ili žičana logika) u simulatoru PSpice se razrešavaju analizom izlaznih otpornosti ovih kola. U komandi .OPTIONS korisnik zadaje raspon izlaznih otpornosti u kolu. Kontrolna promenljiva DIGDRVZ definiše maksimalnu otpornost (izlaznu otpornost drajvera u stanju visoke impedanse), a promenljiva DIGDRVF definiše minimalnu otpornost. Na osnovu ove dve vrednosti formira se logaritamska skala sa 64 nivoa otpornosti.

Jačina drajvera obrnuto je srazmerna njegovoj izlaznoj otpornosti - najjači je drajver sa izlaznom otpornošću DIGDRVF. Dva drajvera mogu imati približno iste jačine, tako da je korisniku ostavljena mogućnost da definiše koliko mora biti drajver jači od svih ostalih aktivnih drajvera da bi se mogao smatrati dominantnim. Kontrolna promenljiva DIGOVRDRV služi za ovu svrhu. Tokom simulacije drajver koji ima jačinu najmanje za DIGOVRDRV veću od ostalih drajvera istog čvora, određuje stanje tog čvora. U suprotnom može doći do konflikta ukoliko dva ili više drajvera približno iste izlazne otpornosti imaju na izlazima različita logička stanja.

Vremenske karakteristike digitalnih komponenti definišu se sa dve modelske kartice. Jedna modelska kartica (vremenski model) specificira propagaciono kašnjenje, vremena postavljanja i držanja signala. PSpice podržava model preciznog dodeljivog kašnjenja. Prema parametrima kašnjenja kojima se opisuju, logičke komponente koje prepoznaje PSpice podeljene su u više grupa. Na primer, za grupu standardnih gejtova (bafer, invertor, I kolo, NI kolo, ILI kolo, NILI kolo, ekskluzivno ILI kolo i kolo ekvivalencije) zadaju se sledeći parametri: TPLHMN, TPLHTY i TPLHMX (minimalno, tipično i maksimalno kašnjenje prednje ivice), TPHLMN, TPHLTY i TPHLMX (minimalno, tipično i maksimalno kašnjenje zadnje ivice). Korisniku je ostavljena mogućnost da izabere da li će se pri simulaciji koristiti minimalna, tipična ili maksimalna kašnjenja zadavanjem vrednosti kontrolne promenljive DIGMNTYMX.

Druga modelska kartica (U/I model) specificira ulazne i izlazne kapacitivnosti i otpornosti. Parametri modela su: INLD, OUTLD (ulazna i izlazna kapacitivnost prema masi), DRVH (izlazna otpornost kola kada je na izlazu stanje logičke jedinice) i DRVL (izlazna otpornost kola kada je na izlazu stanje logičke nule). Ukupno propagaciono kašnjenje kroz logičko kolo određuje se kao zbir propagacionog kašnjenja iz vremenskog modela i dodatnog kašnjenja koje je posledica konačnog vremena punjenja odnosno pražnjenja kapacitivnosti na izlazu.

PSpice prepoznaje sledeće grupe digitalnih komponenti: standardni gejtovi (primaju vremenski model tipa UGATE), trostatički gejtovi (vremenski model tipa UTGATE), ivicom postavljani flip-flopovi (UEFF), nivoom postavljani flip-flopovi (UGFF), višebitni A/D i D/A konvertori (UADC i UDAC), memorije tipa RAM i ROM (URAM, UROM), programabilna logička polja (UPLD), pullup i pulldown otpornik, linija za kašnjenje (UDLY) i kola za proveru vremenskih ograničenja (USUHD, UWDTH).

Razvijena biblioteka za simulaciju u PSpice sistemu stanja organizovana je u četiri datoteke:

PSpicedef.h - deklaraciona datoteka
PSpicefun.hi - funkcijska datoteka
PSpicestr.hi - strukturna (modulska) datoteka
PSpicemod.hi - modelska datoteka

Deklaraciona datoteka **PSpicedef.h** sadrži deklaracije korišćenih struktura, globalnih podataka, funkcija i modula. Funkcijska datoteka **PSpicefun.hi** sadrži funkcije koje se koriste u modeliranju (funkcije kašnjenja, funkcije rezolucije, funkcije za propterećenje operatora i druge). Datoteka **PSpicestr.hi** sadrži definicije digitalnih modula. Datoteka **PSpicemod.hi** sadrži modelske kartice definisane za konkretnu simulaciju.

7.1 Definisanje sistema stanja, atributa signala i rezolucione funkcije

Da bi se hibridni simulator Alecsis2.1 osposobio za simulaciju u PSpice stilu, neophodno je prvo definisati sistem stanja kakav koristi PSpice. Sistem stanja se definiše na sledeći način.

```
typedef enum {
       'X', 'x'='X',          // unknown
       '0',                   // logic zero
       '1',                   // logic one
       'R', 'r'='R',          // rising edge
       'F', 'f'='F',          // falling edge
       'Z', 'z'='Z'           // high impedance
```

```
} ps_t;
extern const char * const namesps[];
```

Kao što vidimo, novi sistem stanja formira se definisanjem enumerisanog tipa podataka `ps_t`. Stanje `'Z'` je dodato PSpice sistemu stanja iako ne predstavlja pravo stanje (pod `'Z'` stanjem podrazumevaćemo proizvoljno stanje sa jačinom drajvera DIGDRVZ), jer ga PSpice prihvata kao stanje na izlazu pobudnog generatora. Interna prezentacija pobrojanih stanja je u obliku celobrojnih indeksa. Stanju `'X'` odgovara celobrojni indeks 0, stanju `'0'` odgovara indeks 1 i tako dalje. Stanju `'x'` odgovara isti indeks kao stanju `'X'`, što je specificirano konstrukcijom `'x'='X'`. Ovime smo omogućili korisniku da koristi kako mala, tako i velika slova za označavanje stanja.

Tabela `namesps` definisana je u datoteci **PSpicefun.hi** na sledeći način.

```
const char * const namesps[] = { "X", "0", "1", "R", "F", "Z" };
```

Sa ovako definisanom tabelom `namesps` može se štampati stanje na ekranu, recimo na sledeći način.

```
signal ps_t a='R';
printf ("a=%s\n", namesps[a]);
```

Sledeći korak u razvoju biblioteke je predefinisanje logičkih operatora za rad u novom sistemu stanja. Naime, Alecsis2.1 ima ugradjen tip podataka `bit` i operatore za logičke operacije u ovom sistemu (`~`, `&`, `~&`, `|`, ...). Za potrebe modeliranja logičkih funkcija osnovnih logičkih kola pogodno je predefinisati (preopteretiti) sve ove operatore tako da se mogu koristiti u novom sistemu stanja. Preopterećenje operatora ilustrovaćemo na primeru operatora negacije (komplementiranje) `~` i operatora konjunkcije (logička I operacija) `&`. Sledeće deklaracije se nalaze u fajlu **PSpicedef.h**.

```
extern const ps_t not_tab[];
extern const ps_t and_tab[][6];
extern ps_t operator ~ (ps_t op);
extern ps_t operator & (ps_t op1, ps_t op2);
```

Sama definicija tabela i funkcija za preopterećenje operatora nalazi se u funkcionom fajlu **PSpicefun.hi** i ima sledeći izgled.

```
ps_t not_tab[6] = { 'X', '1', '0', 'F', 'R', 'X' };
ps_t and_tab[][6] = {
  'X', '0', 'X', 'X', 'X', 'X',
  '0', '0', '0', '0', '0', '0',
  'X', '0', '1', 'R', 'F', 'X',
  'X', '0', 'R', 'R', 'X', 'X',
  'X', '0', 'F', 'X', 'F', 'X',
  'X', '0', 'X', 'X', 'X', 'X'
};
ps_t operator ~ (ps_t op) { return not_tab[op]; }
ps_t operator & (ps_t op1, ps_t op2) { return and_tab[op1][op2]; }
```

S obzirom da simulator PSpice problem rezolucije rešava analizom izlaznih otpornosti drajvera, to je neophodno uz informaciju o stanju pridružiti i informaciju o izlaznoj otpornosti drajvera. Zato ćemo definisati novi tip podataka koji sadrži obe informacije.

```
class ps_class {
      double resistance;
      ps_t state;
  public:
      ps_class (double res=10kohm, ps_t sta='x')
                                  { resistance=res; state=sta; }
      void set_state(ps_t sta)    { state = sta;               }
      void set_res(double res)    { resistance = res;          }
      ps_t get_state()            { return state;              }
      char *get_st_name()         { return namesps[state];     }
```

```
      double get_res()              { return resistance;         }
};
```

Poznato je da kašnjenje logičkog kola raste sa porastom kapacitivnog opterećenja izlaza. Ova kapacitivnost je karakteristika signala, te ćemo je vezati uz signal korišćenjem mehanizma definisanja korisničkih atributa signala. Definisaćemo dodatni atribut signala `tcap` koji predstavlja totalnu parazitnu kapacitivnost od signala do mase. Nju čini paralelna veza svih izlaznih i ulaznih kapacitivnosti odgovarajućih komponenti.

```
class ps_att {
      double tcap;            // total signal stray capacitance
  public:
      ps_att();
      void add_cap(double c) { tcap += c;   }
      double get_tcap ()     { return tcap; }
};
```

Ovako definisan atribut pridružuje se signalu operatorom `@`, a pristupa mu se istim operatorom na sledeći način.

```
signal ps_class @ ps_att a='R';
(@a)->add_cap(2pF);
```

Rezoluciona funkcija PSpice tipa definisana je u datoteci **PSpicefun.hi**. Implementiran je jednostavan algoritam koji medju aktivnim drajverima pronalazi onaj koji ima najmanju izlaznu otpornost (najjači) da bi se njegovo stanje vratilo kao rezultujuće stanje magistrale. U slučaju aktivnosti više drajvera približno istih izlaznih otpornosti, ako oni ne daju isto stanje na izlazu, funkcija rezolucije detektuje i prijavljuje konflikt na magistrali i postavlja magistralu u neodredjeno stanje `'X'`. Potrebne kontrolne promenljive `digdrvz`, `digdrvf` i `digovrdrv` definisane su kao globalni podaci, tako da korisnik može da im menja vrednost pri simulaciji. Deklaracija funkcije rezolucije smeštena je u datoteku **PSpicedef.h** i ima sledeći izgled.

```
extern ps_class ps_resolution (const class ps_class *, int, int *);
```

Prvi argument funkcije rezolucije `ps_resolution` (tipa `const class ps_class *`) je vektor vrednosti drajvera (stanja i izlazne otpornosti), drugi (`int`) je dužina ovog vektora, odnosno broj drajvera magistrale, a preko trećeg argumenta (`int *`) funkcija rezolucije vraća informaciju o konfliktu na magistrali.

Kada smo definisali tip signala, atribute signala i rezolucionu funkciju, moguće je konačno definisati novi tip, nazovimo ga `ps_sig`, koji će sadržati celokupnu potrebnu informaciju.

```
typedef ps_class @ ps_att : ps_resolution ps_sig;
```

Signali tipa `ps_sig` nose, dakle, informaciju o stanju i izlaznoj otpornosti drajvera, imaju kao dodatni atribut totalnu parazitnu kapacitivnost prema masi, a u slučaju potrebe za razrešenjem konflikta koriste definisanu rezolucionu funkciju.

7.2 Modeliranje osnovnih digitalnih komponenti

Kao što je već rečeno, kod simulatora PSpice digitalne komponente su svrstane u više grupa koje se razlikuju po tome što imaju različite modelske parametre. Pri pozivu digitalne komponente zahteva se definisanje U/I modela i vremenskog modela. Parametri U/I modela isti su za sve digitalne komponente, a vremenski modeli se razlikuju kod raznih grupa komponenti.

Da bi smo podržali ovakvu organizaciju, definisaćemo najpre modelsku klasu `uio` koja će sadržati parametre U/I modela.

```
class uio {
  protected:
      double inld, outld, drvh, drvl;
  public:
      uio  ();
      >uio ();
};
```

Konstruktor klase zadaje difoltne vrednosti ovih parametara. Kreiranjem modelske kartice korisnik može definisati druge vrednosti parametara. Procesor `>uio` poziva se posle definisanja vrednosti parametara i ima zadatak da proveri njihovu logičnost.

Iz klase `uio` korišćenjem mehanizma nasledjivanja razvićemo niz klasa vremenskih modela (`ugate`, `utgate`, `ueff` i druge) koje prepoznaje simulator PSpice. Ovaj postupak ilustrovaćemo na primeru modelske klase `ugate`.

```
class ugate : public uio {
      double tplhmn, tplhty, tplhmx, tphlmn, tphlty, tphlmx;
  public :
      ugate();
      double delay (ps_t, ps_t, double);
      friend module buf, inv, and, nand, or, nor, xor, nxor,
                    bufa, inva,     anda, nanda, ora, nora, xora, nxora,
                    ao, oa, aoi, oai;
};
```

Klasa `ugate` sadrži parametre vremenskog modela, kao i funkciju za kašnjenje `delay`. Funkcija `delay`, kao metod klase, ima pristup privatnim podacima (kašnjenja). Ona prima prethodno stanje izlaza komponente, novo stanje izlaza i parazitnu kapacitivnost izlaza i na osnovu ovih podataka odredjuje propagaciono kašnjenje komponente. Kao prijatelji (`friend`) klase `ugate` deklarisani su standardni gejtovi koji primaju vremensku modelsku karticu tipa `ugate`, tako da imaju pravo pristupa privatnim podacima klase.

U ovako definisanom okruženju mogu se modelirati nabrojani standardni gejtovi. Ilustrujmo ovaj proces na primeru dvoulaznog NI kola.

```
module ugate::nand (signal ps_sig in a, b; signal ps_sig out y) {
  action {
    process post_structural {
      (@a)->add_cap(inld); (@b)->add_cap(inld); (@y)->add_cap(outld);
    }
    process (a, b) {
      ps_t rs;                // resultant state
      double dr;              // driving resistance
      double d;               // delay
      ps_class tmp(20k, '0');

      rs = ~( a.get_state() & b.get_state() );

      if      (rs=='0' || rs=='F') { dr = drvl; }
      else if (rs=='1' || rs=='R') { dr = drvh; }
      else { dr = MIN(drvl, drvh); }      // worst case

      d = this.delay(y.get_state(), rs, (@y)->get_tcap());

      tmp.set_res(dr);
      tmp.set_state(rs);
      y <- tmp after d;
    }
  }
}
```

Modul `nand` ima ulazne terminale `a` i `b` i izlazni terminal `y`. Svi su tipa `ps_sig`, što znači da mogu da se spregnu samo sa signalima (vezama) ovog tipa. Modul je modeliran sa dva procesa. Prvi od njih aktivira se interno generisanim signalom `post_structural` neposredno posle

podizanja strukture kola. Ovim procesom obezbedjuje se dodavanje parazitnih ulaznih i izlaznih kapacitivnosti kola signalima za koje su odgovarajući terminali vezani. Proces se izvršava samo jednom, pre početka simulacije i zatim se suspenduje. Unutar drugog procesa, koji se aktivira pri svakoj promeni stanja na ulazima `a` i `b`, korišćenjem predefinisanih operatora `~` i `&` određuje se stanje koje se posle vremena izračunatog funkcijom kašnjenja `delay` šalje na izlaz kola `y`.

Kao što smo rekli, svakom modulu koji je objekat modelske klase obavezno se mora pri korišćenju pridružiti odgovarajuća modelska kartica. Modelska kartica koja postavlja parametre vremenskog i U/I modela za ovako definisano NI kolo može imati sledeću strukturu.

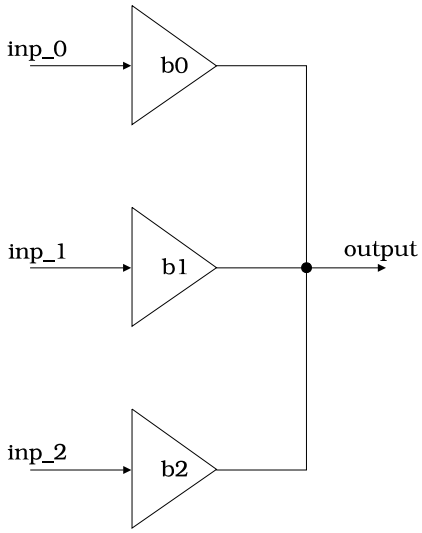
```
model ugate::nand_gate_model {
    tplhmn=1ns; tplhty=2ns; tplhmx=3ns;
    tphlmn=2ns; tphlty=3ns; tphlmx=4ns;
    inld = 120fF;
    outld = 90fF;
    uio::drvh=25ohm;
}
```

Vidimo da se jednom modelskom karticom definišu parametri oba modela, s tim da se može po potrebi (ako ima istoimenih parametara u obe klase, što u našem primeru nije slučaj) koristiti operator rezolucije `::` (parametar `drvh` u gornjem primeru). Parametri kojima se ne zada vrednost uzimaju difoltnu vrednost koju postavlja konstruktor.

Ovako definisano NI kolo može se koristiti u opisu strukture kola u kojima se ova komponenta javlja kao gradivna jedinica. Na sličan način modeliraju se i druge osnovne digitalne komponente koje prepoznaje simulator PSpice.

7.3 Primer simulacije

Ilustrovaćemo korišćenje razvijenog okruženja simulacijom kola sa slike 7.2. Kolo se sastoji od tri bafera (logička komponenta koja prenosi stanje sa ulaza na izlaz posle definisanog propagacionog kašnjenja) čiji su izlazi vezani za istu tačku. Ovakvo kolo zasniva rad na principu žičane logike, gde rezultujuće stanje izlaza zavisi od izlaznih otpornosti bafera. Variraćemo vrednosti izlaznih otpornosti bafera da bi smo testirali razvijenu rezolucionu funkciju.



Slika 7.2 *Žičana logika*

Opis kola sa slike 7.2 za simulator Alecsis2.1 je sledeći.

```
# include "PSpicedef.h"                          // header file
library PSpicemod, PSpicestr, PSpicefun;         // library files
```

```
root module wired_logic_test() {
      // Declarations:
  module ugate::buf b0, b1, b2;
  signal ps_sig   input0={2ohm, '0'}, input1={2ohm, '0'},
                  input2={2ohm, '0'}, output={20kohm, '0'};

      // Structure:
  /* A) The strongest driver b0 determines output state. */
        b0 (input0, output) model=model_20ns_20ohm; // the strongest driver
        b1 (input1, output) model=model_20ns_200ohm;
        b2 (input2, output) model=model_20ns_20kohm;

  /* B) Drivers b0 and b1 both have output resistances 20ohm.
        b0 (input0, output) model=model_20ns_20ohm;
        b1 (input1, output) model=model_20ns_20ohm; // as strong as b0
        b2 (input2, output) model=model_20ns_20kohm; // weak, no influence
  */

  /* C) All drivers have drvh=200ohm (strong one) and drvl=20k (weak zero).
        b0 (input0, output) model=model_wired_or;
        b1 (input1, output) model=model_wired_or;
        b2 (input2, output) model=model_wired_or;
  */

      // Definition of output list and simulation duration:
  timing { tstop = 1000ns; }
  out {
    signal ps_t inp_0  { return input0.get_state(); };
    signal ps_t inp_1  { return input1.get_state(); };
    signal ps_t inp_2  { return input2.get_state(); };
    signal ps_t output { return output.get_state(); };
  }

      // Circuit excitation:
  action {
    process initial {
      digdrvz=20kohm; digdrvf=2ohm; digovrdrv=2; dlymntymx=2;
      input0 <- ps_class(2ohm, '1') after 500ns,
                ps_class(2ohm, '0') after 600ns,
                ps_class(2ohm, '1') after 800ns;

      input1 <- ps_class(2ohm, '1') after 300ns,
                ps_class(2ohm, '0') after 400ns,
                ps_class(2ohm, '1') after 700ns;

      input2 <- ps_class(2ohm, '1') after 100ns,
                ps_class(2ohm, '0') after 200ns;
    }
  }
}
```

Glavni (`root`) modul predstavlja celo simulirano kolo, a definiše se na isti način kao i ostali moduli, osim što može imati i neke komande za kontrolu simulacije (`out`, `timing`). U deklaracijskom delu `root` modula navedeni su signali i moduli koji će biti korišćeni u opisu strukture kola, dok je u akcionom delu definisana pobuda kola. Baferi `b0`, `b1` i `b2` koriste modele koji su definisani u modelskoj datoteci **PSpicemod.hi**, dok je struktura samih bafera definisana u modulskoj datoteci **PSpicestr.hi**. Komentarima su izdvojene različite konfiguracije (obeležene sa A, B i C) koje će biti simulirane. Za definisanje pobude iskorišćen je proces koji se aktivira implicitnim signalom `initial` i izvršava se samo jednom, pred početak simulacije.

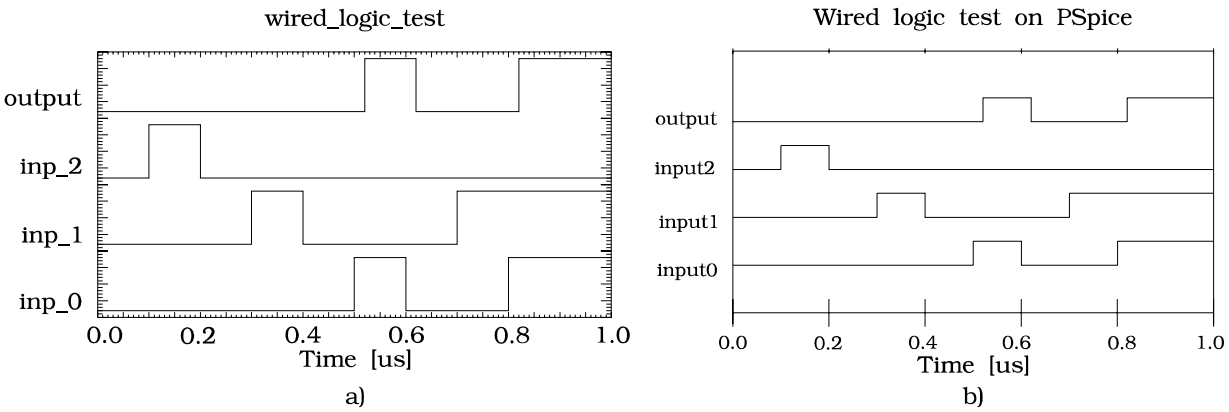
Pri pozivu komponente `buf` koja je definisana kao objekat klase `ugate` definiše se modelska kartica koja odgovara modelskoj klasi `ugate`, a u kojoj su zadate konkretne vrednosti parametara modela komponente. Više modula može koristiti istu modelsku karticu, a kako je modelska kartica statički objekat, time se ne duplira korišćena memorija. Modelske kartice smo smestili u datoteku

PSpicemod.hi, što ne znači da korisnik ne sme da ih grupiše i smešta u druge datoteke koje će navesti u naredbi `library`. Korišćeni modeli definisani su na sledeći način.

```
model ugate::model_20ns_200ohm {
      tplhty = 20ns;    tphlty = 20ns;
      uio::drvh=200.0ohm;
      uio::drvl=200.0ohm;
}

model ugate::model_20ns_20kohm {
      tplhty = 20ns;    tphlty = 20ns;
      uio::drvh=20kohm;
      uio::drvl=20kohm;
}

model ugate::model_20ns_20ohm {
      tplhty = 20ns;    tphlty = 20ns;
      uio::drvh=20.0ohm;
      uio::drvl=20.0ohm;
}

model ugate::model_wired_or {
      tplhty = 20ns;    tphlty = 20ns;
      drvh=200ohm;      // small, strong logic one
      drvl=20kohm;      // big, weak logic zero
}
```

Opis kola sa slike 7.2 za simulaciju programom PSpice je sledeći.

```
Wired logic test on PSpice
*
.options digovrdrv=2 digdrvf=2ohm digdrvz=20kohm digmntymx=2
*
*** A) The strongest driver b0 determines output state
Ub0     buf     $g_dpwr, $g_dgnd, input0, output  model_20ns io_20ohm
Ub1     buf     $g_dpwr, $g_dgnd, input1, output  model_20ns io_200ohm
Ub2     buf     $g_dpwr, $g_dgnd, input2, output  model_20ns io_20kohm
*
*** B) Drivers b0 and b1 both have output resistance 20ohm => conflict
*Ub0     buf     $g_dpwr, $g_dgnd, input0, output  model_20ns io_20ohm
*Ub1     buf     $g_dpwr, $g_dgnd, input1, output  model_20ns io_20ohm
*Ub2     buf     $g_dpwr, $g_dgnd, input2, output  model_20ns io_20kohm
*
*** C) All drivers have drvh=200ohm (strong one) and drvl=10k (weak zero)
*Ub0     buf     $g_dpwr, $g_dgnd, input0, output  model_20ns io_wired_or
*Ub1     buf     $g_dpwr, $g_dgnd, input1, output  model_20ns io_wired_or
*Ub2     buf     $g_dpwr, $g_dgnd, input2, output  model_20ns io_wired_or
*
.model io_20ohm    uio   (drvl=20ohm,  drvh=20ohm )
.model io_200ohm   uio   (drvl=200ohm, drvh=200ohm)
.model io_20kohm   uio   (drvl=20k,    drvh=20k   )
.model io_wired_or uio   (drvl=10k,    drvh=200ohm)
.model model_20ns  ugate (tplhty=20ns, tphlty=20ns)
*
Uinput0 stim (1,1) $g_dpwr, $g_dgnd input0 io_20ohm (0s, 0) (500ns, 1)
+ (600ns, 0) (800ns, 1)
Uinput1 stim (1,1) $g_dpwr, $g_dgnd input1 io_20ohm (0s, 0) (300ns, 1)
+ (400ns, 0) (700ns, 1)
Uinput2 stim (1,1) $g_dpwr, $g_dgnd input2 io_20ohm (0s, 0) (100ns, 1)
+ (200ns, 0)
*
.tran 100ns 1000ns
.probe
*.print/dgtlchg input0, input1, input2, output
.end
```



Slika 7.3 *"Najjači" drajver* `b0` *odredjuje stanje izlaza*
a) Alecsis2.1 b) PSpice

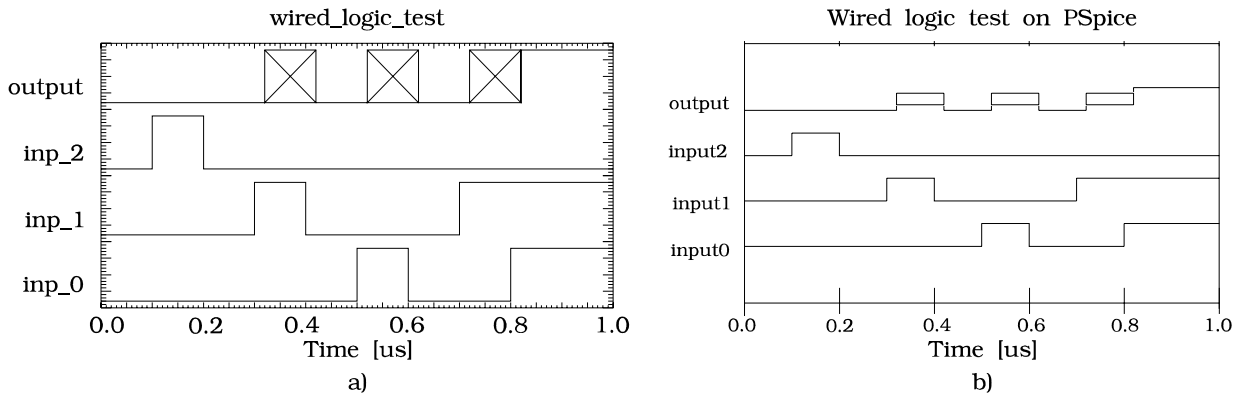
Slika 7.3a predstavlja rezultat simulacije kola koje ima strukturu označenu sa A simulatorom Alecsis2.1, a slika 7.3b simulatorom PSpice. Rezultati simulacije simulatorom Alecsis2.1 obradjeni su grafičkim postprocesorom Art2.1, a rezultati simulacije simulatorom PSpice grafičkim postprocesorom Probe [-91]. Baferi imaju različite izlazne otpornosti i to se odražava na rezultat simulacuje tako što najjači drajver dominira stanjem na izlazu. Vidimo da izlaz kola prati promenu stanja drajvera sa najmanjom izlaznom otpornošću (bafer `b0`).

Sadržaj tekstualnog izlaznog fajla simulatora Alecsis2.1 ima sledeći izgled.

```
wired_logic_test

OutputVariables:

  TIME   inp_0 inp_1 inp_2 output

OutputValues:

  0.0000e-08  0  0  0  0
  1.0000e-07  0  0  1  0
  2.0000e-07  0  0  0  0
  3.0000e-07  0  1  0  0
  4.0000e-07  0  0  0  0
  5.0000e-07  1  0  0  0
  5.2000e-07  1  0  0  1
  6.0000e-07  0  0  0  1
  6.2000e-07  0  0  0  0
  7.0000e-07  0  1  0  0
  8.0000e-07  1  1  0  0
  8.2000e-07  1  1  0  1
  1.0000e-06  1  1  0  1
```

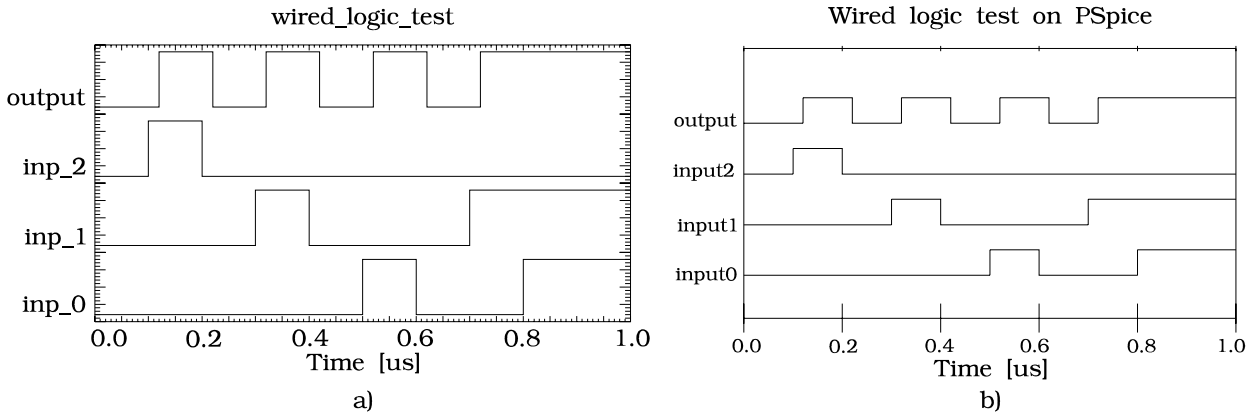
Slična izlazna lista dobija se kao rezultat simulacije simulatorom PSpice.

Ako dva bafera imaju istu izlaznu otpornost (jačinu), javiće se konflikt na zajedničkom izlazu, što rezultuje pojavom neodređenog `'X'` stanja (slučaj B u opisu kola). Rezultat simulacije kada su baferi `bo` i `b1` iste jačine dat je na slici 7.4. I ovde se postiže podudarnost rezultata simulacije sa dva različita simulatora. Na slikama uočavamo da grafički procesori Art2.1 i Probe nemaju istu grafičku predstavu neodredjenog stanja.



Slika 7.4 *Drajveri iste izlazne otpornosti prouzrokuju konflikt*
a) Alecsis2.1 b) PSpice

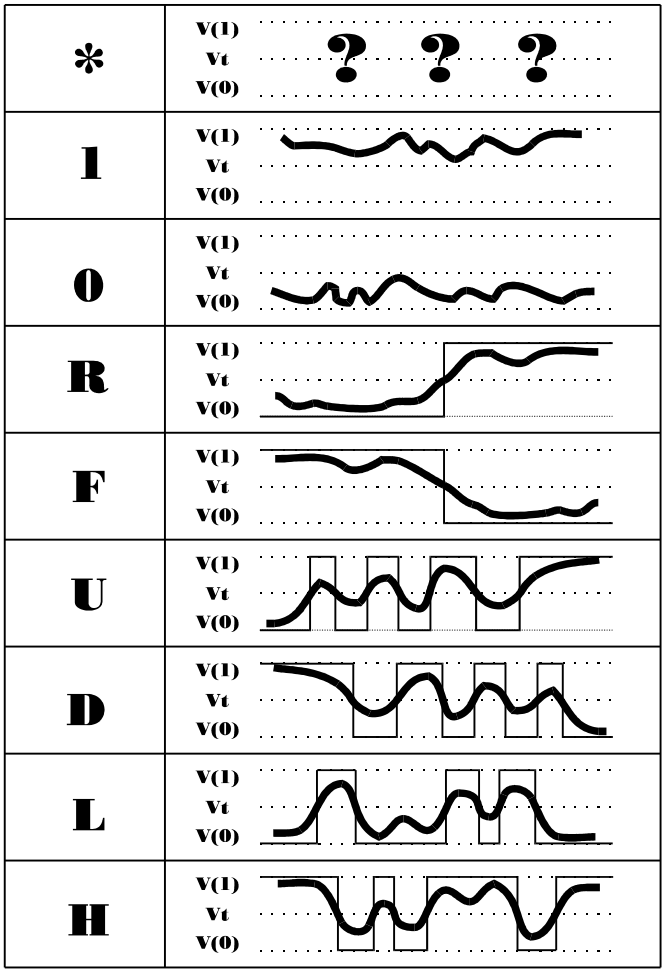
Pogodnim izborom vrednosti izlazne otpornosti bafera kada je na izlazu logička nula i kada je na izlazu logička jedinica moguće je modelirati veze žičano I i žično ILI. Ako u opis kola priključimo deo označen sa C, kolo realizuje funkciju žičano ILI. Rezultati simulacije ovakvog kola dati su na slici 7.5. Vidimo da stanje logičke jedinice dominira kad se pojavi na izlazu makar jednog bafera.



Slika 7.5 *Rezultati simulacije veze "žičano ILI"*
a) Alecsis2.1 b) PSpice

8 Biblioteka za analizu hazarda

Pri projektovanju digitalnog kola nemoguće je unapred predvideti pojavu hazardnih situacija u toku eksploatacije kola. Zato je pre proizvodnje kola neophodno izvršiti analizu hazarda i ako je moguće eliminisati njihovu pojavu u radnom režimu. Simulacija hazarda je takodje neophodna pri projektovanju specijalne klase elektronskih kola, takozvanih hazard-free kola.



Slika 8.1 *Fantauzzi-jev sistem sa 9 stanja*

Analiza hazarda u logičkim kolima može se obaviti na razne načine i u različitim sistemima stanja. Opisaćemo u ovom poglavlju sistem sa devet stanja koji je razvio Giuseppe Fantauzzi [Fant74] za analizu hazarda u kombinacionim kolima bez povratne sprege.

U ovom sistemu stanja svako stanje predstavlja trajan proces, "promenu" u čvoru koja počinje u nekom trenutku t_0 kad se stanje postavi, a završava se u nekom trenutku t_1 kad se pojavi novo stanje. Kašnjenja u ovakvom sistemu stanja nemaju smisla i smatraju se neodredjenim, a obuhvata ih sam sistem stanja. Zbog toga simulator mora da radi sa nultim kašnjenjem.

Sistem sadrži sledeća stanja: `'*'`, `'1'`, `'0'`, `'R'`, `'F'`, `'U'`, `'D'`, `'L'`, `'H'`. Na slici 8.1 prikazana je njihova grafička ilustracija. Stanje `'*'` predstavlja neodredjeno stanje. Stanja logičke nule i jedinice karakterišu se time da je vrednost napona (ili neke druge analogne promenljive) permanentno iznad napona praga V_t, odnosno ispod njega. Stanje `'R'`, koje predstavlja prelaz sa logičke nule na logičku jedinicu bez hazarda, ne treba smatrati linearnom promenom (rampom) koja počinje od napona logičke nule V(0), a završava se naponom logičke jedinice V(1). Naprotiv, ono se može smatrati trenutnim prelazom sa logičke nule na logičku jedinicu u proizvoljnom trenutku izmedju t_0 i t_1. Isto važi i za stanje `'F'`. Takodje treba napomenuti da hazardna stanja (`'U'`, `'D'`, `'L'` i `'H'`) predstavljaju *mogućnost* pojave kolebanja vrednosti signala (hazard) oko napona praga V_t, ali mogu predstavljati i "čist" prelaz (poput stanja `'R'` i `'F'`), što odgovara realnoj situaciji u digitalnom kolu.

Razvijena je biblioteka za simulaciju hazarda u kojoj postoje modeli sledećih logičkih primitiva:

- NAND, AND, OR, NOR kola sa proizvoljnim brojem ulaza
- XOR, XNOR kola
- invertor
- RS flip-flopovi postavljani nivoom (realizovani od NOR i NAND kola).

Biblioteka je organizovana u tri datoteke:

haz9.h - deklaraciona datoteka
fun.hi - funkcijska datoteka
str.hi - modulska datoteka

Biblioteka ne sadrži modelsku datoteku, jer nema parametara kašnjenja koje obično smeštamo u modelsku karticu.

8.1 Definisanje sistema stanja i preopterećenje operatora

Sistem stanja definisan je u deklaracionoj datoteci **haz9.h** na sledeći način.

```
typedef enum {
  '*', 'X'='*', 'x'='*',       // unknown
  '1',                         // always at logic one
  '0',                         // always at logic zero
  'R', 'r'='R',                // transition 0->1, no hazard, rising edge
  'F', 'f'='F',                // transition 1->0, no hazard, falling edge
  'U', 'u'='U',                // dynamic hazard 0->1, up with hazard
  'D', 'd'='D',                // dynamic hazard 1->0, down with hazard
  'L', 'l'='L',                // static hazard 0->0, low with hazard
  'H', 'h'='H',                // static hazard 1->1, high with hazard
  '_'=void
} haz_t;
extern const char * const nameshaz[];
```

U funkcionoj datoteci **fun.hi** definisan je vektor `nameshaz` koji olakšava štampanje stanja na ekranu računara kao i kod drugih, ranije opisanih biblioteka.

```
const char * const nameshaz[]={"*", "1", "0", "R", "F", "U", "D", "L", "H"};
```

Stanje `'*'` namerno je izabrano kao prvo u definiciji enumerisanog tipa `haz_t` da bi se njime automatski inicijalizirale sve promenljive i signali ovog tipa.

Osnovni logički operatori predefinisani su za rad u ovom sistemu stanja. Deklaracija odgovarajućih funkcija nalazi se u deklaracijskoj datoteci i ima sledeći izgled.

```
extern haz_t operator  ~ (haz_t op);
extern haz_t operator  & (haz_t op1, haz_t op2);
extern haz_t operator  | (haz_t op1, haz_t op2);
extern haz_t operator ~& (haz_t op1, haz_t op2);
extern haz_t operator ~| (haz_t op1, haz_t op2);
extern haz_t operator  ^ (haz_t op1, haz_t op2);
extern haz_t operator ~^ (haz_t op1, haz_t op2);
```

U deklaracionoj datoteci nalaze se takodje i deklaracije modula definisanih u ovom sistemu stanja. Navešćemo kompletnu listu raspoloživih modula.

```
module inv   (haz_t in a;   haz_t out y);
module and   (haz_t in a,b; haz_t out y);
module andx  (haz_t in a[]; haz_t out y);
module andm  (haz_t out y; haz_t in ...);
module nand  (haz_t in a,b; haz_t out y);
module nandx (haz_t in a[]; haz_t out y);
module nandm (haz_t out y; haz_t in ...);
module or    (haz_t in a,b; haz_t out y);
module orx   (haz_t in a[]; haz_t out y);
module orm   (haz_t out y; haz_t in ...);
module nor   (haz_t in a,b; haz_t out y);
module norx  (haz_t in a[]; haz_t out y);
module norm  (haz_t out y; haz_t in ...);
module xor   (haz_t in a,b; haz_t out y);
module nxor  (haz_t in a,b; haz_t out y);
module rsnandffc (haz_t in rb, sb; haz_t out q, qb; haz_t in  clear);
module rsnandffp (haz_t in rb, sb; haz_t out q, qb; haz_t in preset);
module rsnorffc  (haz_t in rb, sb; haz_t out q, qb; haz_t in  clear);
module rsnorffp  (haz_t in rb, sb; haz_t out q, qb; haz_t in preset);
```

Vidi se da je za pojedine logičke funkcije definisano više različitih modula. Recimo, za I logičko kolo postoje moduli `and`, `andx` i `andm`. Modul `and` je dvoulazno I kolo, modul `andx` je I logičko kolo sa vektorski definisanim ulazom, a modul `andm` je I logičko kolo sa proizvoljnim brojem skalarnih ulaza.

Kao ilustraciju preopterećenja logičkih operatora navešćemo operacije komplement i ekskluzivno ILI. Iako u [Fant74] nije predvidjeno modeliranje funkcije ekskluzivno ILI i funkcije ekvivalencije (ekskluzivno NILI), smatrali smo za potrebno da razvijeno modele ovih logičkih elemenata zbog njihovog značaja, s obzirom na veliko interesovanje za implementaciju logičkih funkcija u obliku razvoja Reed-Muller-a [Davi87].

```
                        //   *    1    0    R    F    U    D    L    H
haz_t const not_tab[9] = { '*', '0', '1', 'F', 'R', 'D', 'U', 'H', 'L' };
haz_t operator ~ (haz_t op1) { return not_tab[op]; }

haz_t const xor_tab[9][9]  = {
      '*', '*', '*', '*', '*', '*', '*', '*', '*',
      '*', '0', '1', 'F', 'R', 'D', 'U', 'H', 'L',
      '*', '1', '0', 'R', 'F', 'U', 'D', 'L', 'H',
      '*', 'F', 'R', 'L', 'H', 'L', 'H', 'U', 'D',
      '*', 'R', 'F', 'H', 'L', 'H', 'L', 'D', 'U',
      '*', 'D', 'U', 'L', 'H', 'L', 'H', 'U', 'D',
      '*', 'U', 'D', 'H', 'L', 'H', 'L', 'D', 'U',
      '*', 'H', 'L', 'U', 'D', 'U', 'D', 'L', 'H',
      '*', 'L', 'H', 'D', 'U', 'D', 'U', 'H', 'L'
};
haz_t operator ^ (haz_t op1, haz_t op2) { return xor_tab[op1][op2]; }
```

8.2 Modeliranje osnovnih logičkih elemenata

Predefinisanjem logičkih operatora modeliranje osnovnih logičkih kola postaje jednostavno. Na primer, model logičkog kola koje obavlja funkciju ekskluzivno ILI dat je sledećim kodom.

```
module xor (haz_t a,b; haz_t out y) {
  action { process (a,b) { y <- a^b; } }
}
```

Kao što se vidi, nova vrednost šalje se u drajver signala bez kašnjenja. Simulator, naravno, radi sa takozvanim "delta" kašnjenjem, što je zapravo kvazi-nulto kašnjenje, odnosno mehanizam simulacije sa nultim kašnjenjem po principu narednog dogadjaja.

Na sličan način modelirano je I kolo sa ulaznim terminalima grupisanim u vektor. Pri tome je iskorišćen preopterećeni operator &.

```
module andx (haz_t in a[];  haz_t out y) {
  action {
    process (a) {
      haz_t result;
      int i;
      result=a[0];
      for (i=1; result != '0' && i<lengthof a; i++) result = result & a[i];
      y <- result;
    }
  }
}
```

Nešto složenija je implementacija modela višeulaznih logičkih kola u sistemu sa 9 stanja. U [Fant74] dat je postupak modeliranja višeulaznog I kola. Model takvog kola ima sledeći izgled.

```
// AND module with arbitrary number of input terminals
module andm (haz_t out y; haz_t in ...) {
  action {
    check: process structural {
      if ($$ < 2)
        warning ("module andm has no input terminals", 1);
    }

    func: process ($2, ...) {
      int i;
      int nX, n1, n0, nR, nF, nU, nD, nL, nH;   // state counters

      nX=0; n1=0; n0=0; nR=0; nF=0; nU=0; nD=0; nL=0; nH=0;

      for (i=2; i<=$$; i++) {       // Count states:
        switch ($i) {
          case '*' : nX++; break;
          case '1' : n1++; break;
          case '0' : n0++; break;
          case 'R' : nR++; break;
          case 'F' : nF++; break;
          case 'U' : nU++; break;
          case 'D' : nD++; break;
          case 'L' : nL++; break;
          case 'H' : nH++; break;
          default  :
            printf ("state \"%s\" not recognized in andm", nameshaz[$i]);
            exit (1);
        } // switch
      } // for
                                    // Evaluate gate function:
      if (n0) {
      // 1 - at least one input at '0' state
        y <- '0';
```

```
      }
      else if (!n0 && nL) {
      // 2 - no '0'-s, but at least one input at static zero hazard
        y <- 'L';
      }
      else if (!n0 && !nL && (nU || nR) && (nD || nF)) {
      // 3 - no zeros, no static zero hazards, but at least one input
      //     rises and at least one input falls => static zero hazard
        y <- 'L';
      }
      else if (!n0 && !nL && (!nU && !nR || !nD && !nF) && nU && !nX) {
      // 4 - no zeros, no static zero hazards, there dinamic hazard
      //     on rise edge, no 'D' and no 'F'
        y <- 'U';
      }
      else if (!n0 && !nL && (!nU && !nR || !nD && !nF) && nD && !nX) {
      // 5 - no dominant states, only falling edge with hazard
        y <- 'D';
      }
      else if (!n0 && !nL && !nU && !nD && (!nR || !nF) && nH && nF && !nX) {
      // 6 - static one hazard and falling edge (hazard free) simultaneously
        y <- 'D';
      }
      else if (!n0 && !nL && !nU && !nD && (!nR || !nF) && nH && nR && !nX) {
      // 7 - static one hazard and rising edge simultaneously
        y <- 'U';
      }
      else if (!n0 && !nL && !nU && !nD && !nF && !nR && nH && !nX) {
      // 8 - only ones and static one hazard => static one hazard
        y <- 'H';
      }
      else if (!n0 && !nL && !nU && !nD && (!nF || !nR) && !nH && nF && !nX) {
      // 9 - only ones and falling edge => falling edge at output
        y <- 'F';
      }
      else if (!n0 && !nL && !nU && !nD && (!nF || !nR) && !nH && nR && !nX) {
        // 10 - only ones and rising edges => rising edge at output
        y <- 'R';
      }
      else if (!n0 && !nL && !nU && !nD && (!nF && !nR) && !nH && !nX) {
        // 11 - only ones
        y <- '1';
      }
      //else if (n0 && !nL && (!nU && !nR || !nD && !nF) && nX) {
      else if (!n0 && !nL && (!nU && !nR || !nD && !nF) && nX) {
      // no zeros, no static zero hazard and no rising and falling edge
      // simultaneously, but unknown inputs present => output goes unknown
        y <- '*';
      }
      else {
        warning("UNEXPECTED INPUT COMBINATION", 0);
        y <- '*';
      }
    }
  }
}
```

U gornjem primeru iskorišćeni su mehanizmi za formiranje modula/funkcija sa promenljivim brojem parametara u jeziku AleC++. Modul `andm` ima izlazni terminal `y` tipa `haz9` i proizvoljan broj ulaznih terminala. Oznaka `...` kaže kompajleru da na tom mestu treba očekivati listu signala tipa `haz9` i usmerenja `in`. Unutar modula definisanog na ovakav način, operator `$$` predstavlja ukupan broj terminala. Logičko stanje na prvom terminalu dobija se kao `$1`, na drugom kao `$2` i tako dalje. Prvi terminal je izlazni terminal `y`, a od drugog do `$$`-tog su ulazni terminali. Štaviše, moguće je koristiti i promenljive za indeksiranje liste terminala, recimo u okviru `for` petlje može se dobiti vrednost na `i`-tom terminalu kao `$i`. Proces označen labelom `check` sinhronizovan

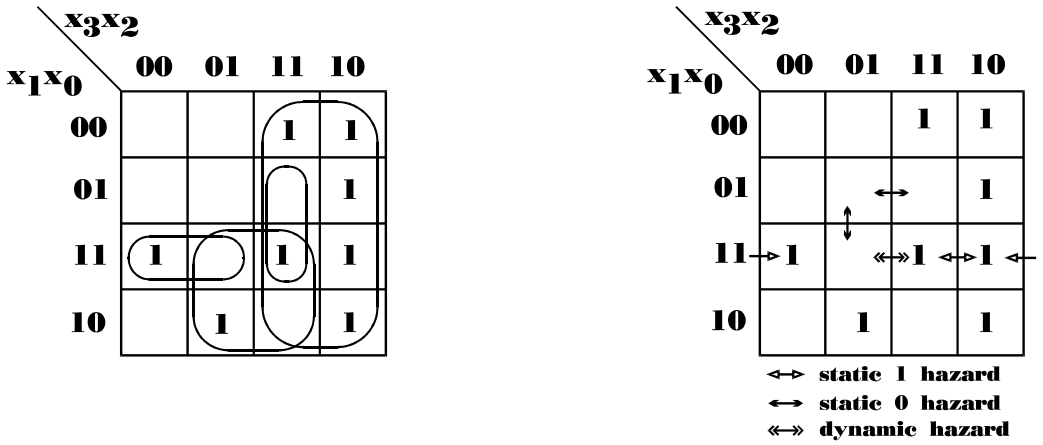
je signalom `structural` kako bi se simulacija prekinula pre početka bilo kakve aktivnosti ako modul `andm` nema ulaznih terminala. Proces označen labelom `func` osetljiv je na promene na ulaznim terminalima, dakle od `$2`, pa do poslednjeg `$$`., što je specificirano kao `$2, ...` u listi osetljivosti.

Modeli flip-flopova zahtevaju definisanje memorisanog stanja pre dovodjenja pobude na ulaz. Zato su razvijena dva modela za svaki flip-flop, jedan sa `preset` ulazom, a drugi sa `clear` ulazom. Modeli se koriste tako što se pre dovodjenja pobudnog vektora pomoću ovih ulaza definiše interno stanje flip-flopa.

```
haz_t const q1_tab[][9] = {
//reset
 /* * */          '*', '*', '*', '*', '*', '*', '*', '*', '*',
 /* 1 */          '*', '1', '0', '0', 'F', '0', 'F', '0', 'F',
 /* 0 */          '1', '1', '1', '1', '1', '1', '1', '1', '1',
 /* R */          '*', '1', 'F', '*', 'F', '*', 'F', 'F', '*',
 /* F */          '*', '1', 'R', 'R', 'H', 'R', 'H', 'R', 'H',
 /* U */          '*', '1', 'D', '*', 'D', '*', 'D', 'D', '*',
 /* D */          '*', '1', 'U', 'U', 'H', 'U', 'H', 'U', 'H',
 /* L */          'H', '1', 'H', 'H', 'H', 'H', 'H', 'H', 'H',
 /* H */          '*', '1', 'L', '*', 'D', '*', 'D', 'L', '*'
};    /*set:      '*', '1', '0', 'R', 'F', 'U', 'D', 'L', 'H'  */

haz_t const q0_tab[][9] = {
// reset
/* * */           '*', '*', '*', '*', '*', '*', '*', '*', '*',
/* 1 */           '*', '0', '0', '0', '0', '0', '0', '0', '0',
/* 0 */           '1', '1', '1', '1', '1', '1', '1', '1', '1',
/* R */           '*', '1', 'F', '*', 'F', '*', 'F', 'F', '*',
/* F */           '*', 'R', 'R', 'R', 'R', 'R', 'R', 'R', 'R',
/* U */           '*', '1', 'D', '*', 'D', '*', 'D', 'D', '*',
/* D */           '*', 'R', 'U', 'U', 'U', 'U', 'U', 'U', 'U',
/* L */           'H', '1', 'H', 'H', 'H', 'H', 'H', 'H', 'H',
/* H */           '*', '*', 'L', '*', 'L', '*', 'L', 'L', '*'
};    /*set:      '*', '1', '0', 'R', 'F', 'U', 'D', 'L', 'H'  */

/* ------------ RS flip-flop built up of two nand circuits with preset: */
module rsnandffp (haz_t in rb, sb; haz_t out q, qb; haz_t in preset) {
  action {
    process (rb, sb, preset) {
      if (preset=='1') {            // set device
        q  <- '1';
        qb <- '0';
      }
      else if (preset!='0') {
        warning ("preset state not stable, device behavior not defined", 0);
      }
      else {                        // preset == '0', normal RS operation mode
        if (q=='0') {
          q  <- q0_tab[rb][sb];
          qb <- q1_tab[sb][rb];           // commutated inputs
        }
        else if (q=='1') {
          q  <- q1_tab[rb][sb];
          qb <- q0_tab[sb][rb];           // commutated inputs
        }
        else
          warning ("rsnandffp stored state not stable", 0);
      }
    } //process
  } //action
} //module rsnandffp
```

8.3 Analiza hazarda u logičkim kolima baziranim na RM razvoju

Zadnjih godina veliki je interes za Reed-Muller polinome. RM polinomi se implementiraju korišćenjem logičkih elemenata ekskluzivno ILI. Specifičnost ovih kola sadržana je u činjenici da se stanje na izlazu menja pri svakoj promeni stanja jednog od ulaza. U [Ouya94] razvijeni su metodi za analizu hazarda u kolima sa logičkim elementima ekskluzivno ILI. Iskoristićemo jedno takvo kolo da ilustrujemo upotrebu razvijene biblioteke sa 9 stanja za analizu hazarda.



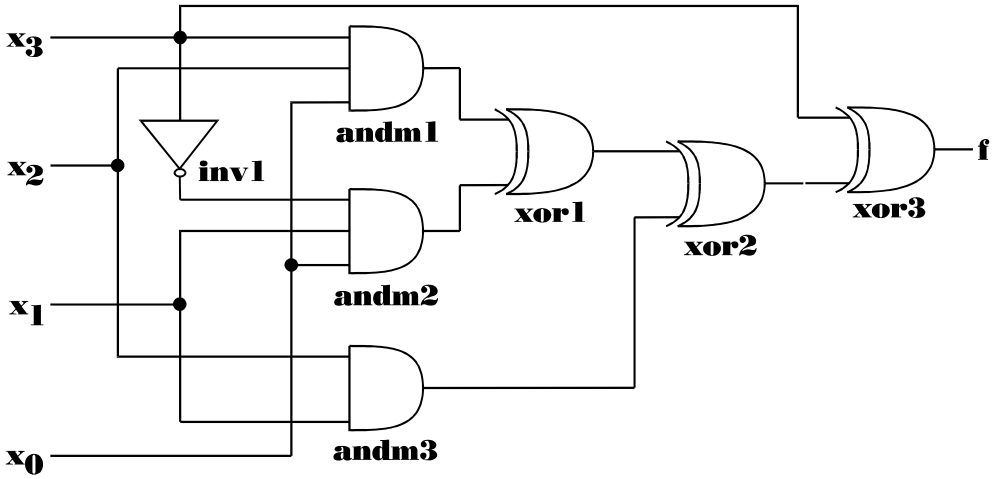
a) *b)*

Slika 8.2 *a) Predstava funkcije* $x_3 \oplus x_3x_2x_0 \oplus \overline{x_3}x_1x_0 \oplus x_2x_1$ *Karnaugh-ovom mapom*
b) Mogući hazardi pri promeni stanja samo jedne promenljive

Neka je data funkcija

$$\mathbf{f(x_3,x_2,x_1,x_0) = \oplus\sum m(3,6,8,9,10,11,12,15) = x_3 \oplus x_3x_2x_0 \oplus \overline{x_3}x_1x_0 \oplus x_2x_1}\ .$$

Ovu funkciju moguće je predstaviti Karnaugh-ovom mapom kao na slici 8.2a. Slika 8.2b prikazuje sve hazarde (statički 0 i 1 hazard i dinamički hazard) do kojih može doći u logičkom kolu koje realizuje ovu funkciju ako se menja stanje samo jednog ulaza. Do ovih rezultata dolazi se teorijskom metodom izloženom u [Ouya94], Dogadjajem na ulazu smatra se prelaz `0->1` ili prelaz `1->0`, što odgovara stanjima `'R'` i `'F'` u našem sistemu stanja.



Slika 8.3 *Realizacija funkcije* $x_3 \oplus x_3x_2x_0 \oplus \overline{x_3}x_1x_0 \oplus x_2x_1$

Slika 8.3 prikazuje realizaciju funkcije f korišćenjem neophodnih logičkih elemenata. Kad je poznata logička šema kola, otkrivanje hazarda moguće je primenom simulatora Alecsis2.1 uz korišćenje razvijene biblioteke. Potrebno je samo na ulaze kola dovesti sve kombinacije pobudnih signala kod kojih je aktivan samo jedan od ulaza. Pobudni vektori koji odgovaraju ovom zahtevu

dobijaju se tako što se jedan od ulaza postavi u `'R'` (odnosno `'F'` stanje), a od ostalih ulaza se generišu sve kombinacije, recimo `"R000"`, `"R0001"`, `"R0010"`, `"R0011"`, `"R100"` i t d. Kako za svaki pobudni signal imamo kritična stanja `'R'` i `'F'`, ukupan broj potrebnih pobudnih vektora je 64. Ovde ćemo se ograničiti na potvrdu teorijski dobijenih rezultata prikazanih na slici 8.2b, tako da ćemo kolo sa slike 8.3 pobuditi samo onim ulaznim vektorima za koje znamo da izazivaju pojavu hazarda. Na primer, dinamički hazard prikazan na slici 8.2b izazivaju pobudni vektori `"R111"` i `"F1111"`. Opis kola sa slike 8.3 za simulaciju programom Alecsis2.1 je sledeći.

```
# include "haz9.h"
library str, fun;
root module hazard_analysis (){
  module andm andm1, andm2, andm3;
  module inv  inv1;
  module xor  xor1, xor2, xor3;
  signal haz_t x[3:0]="0000", nx3, a1o, a2o, a3o, x1o, x2o, f;

  andm1 (a1o, x[3], x[2], x[0]);
  andm2 (a2o, nx3,  x[1], x[0]);
  andm3 (a3o, x[2], x[1]);
  inv1 (x[3], nx3);
  xor1 (a1o, a2o, x1o);
  xor2 (x1o, a3o, x2o);
  xor3 (x2o, x[3], f);

  timing { tstop = 100ns; }
  out { signal haz_t x[3], x[2], x[1], x[0], nx3, a1o, a2o, a3o, x1o, x2o, f; }
  action {
    process initial {
      x <- "R011" after 1ns,    // static 1 hazard
           "F011" after 2ns,    // static 1 hazard
           "01R1" after 3ns,         // static 0 hazard
           "01F1" after 4ns,         // static 0 hazard
           "R101" after 5ns,    // static 0 hazard
           "F101" after 6ns,    // static 0 hazard
           "1R11" after 7ns,         // statuc 1 hazard
           "1F11" after 8ns,         // static 1 hazard
           "R111" after 9ns,    // dynamic hazard
           "F111" after 10ns;   // dynamic hazard
    }
  }
}
```

Rezultati simulacije dobijaju se u obliku tekstualnog izlaznog fajla i imaju sledeći izgled.

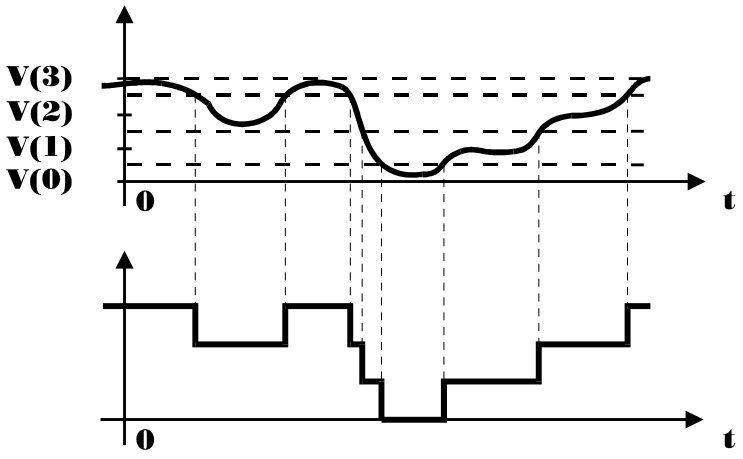
```
hazard_analysis

OutputVariables:

     TIME     x[3] x[2] x[1] x[0] nx3 a1o a2o a3o x1o x2o f

OutputValues:

  0.0000e+00   0    0    0    0    1   0   0   0   0   0  0     // initial state
  1.0000e-09   R    0    1    1    F   0   F   0   F   F  H  // st. 1 hazard
  2.0000e-09   F    0    1    1    R   0   R   0   R   R  H
  3.0000e-09   0    1    R    1    1   0   R   R   R   L  L  // st. 0 hazard
  4.0000e-09   0    1    F    1    1   0   F   F   F   L  L
  5.0000e-09   R    1    0    1    F   R   0   0   R   R  L  // st. 0 hazard
  6.0000e-09   F    1    0    1    R   F   0   0   F   F  L
  7.0000e-09   1    R    1    1    0   R   0   R   R   L  H  // st. 1 hazard
  8.0000e-09   1    F    1    1    0   F   0   F   F   L  H
  9.0000e-09   R    1    1    1    F   R   F   1   H   L  U  // dyn. hazard 0->1
  1.0000e-08   F    1    1    1    R   F   R   1   H   L  D  // dyn. hazard 1->0
  1.0000e-07   F    1    1    1    R   F   R   1   H   L  D      // tstop
```

Treba napomenuti da vremenska komponenta simulacije ovde nema značaja. Vremenski mehanizmi su iskorišćeni da se u jednoj simulaciji na ulaze kola dovede više različitih pobudnih vektora. Sama propagacija ulaznih stanja kroz kolo odvija se sa nultim kašnjenjem, tako da se odziv kola na zadati pobudni vektor dobija u istom vremenskom trenutku kada je doveden pobudni vektor. U izlaznoj datoteci zabeleženi su svi dogadjaji nastali u procesu propagacije stanja od ulaza do izlaza. Pri analizi hazarda treba razmatrati samo poslednje štampano stanje signala u kolu. Zato je potrebno iz izlazne datoteke odstraniti (odbaciti) sve dogadjaje koji su se desili u istom vremenskom trenutku pre smirivanja kola. Za tu svrhu služi konvertor **NZD**.

9 Biblioteka za simulaciju višeznačne logike

Binarni brojni sistem, koji dominira današnjom digitalnom elektronikom, ima relativno mali informacioni kapacitet. Za predstavljanje istog broja u brojnim sistemima sa više logičkih stanja potrebno je manje cifara, što podrazumeva jednostavniji hardver. U VLSI digitalnim kolima oko 70% površine čipa zauzimaju veze. Ako se izuzme korišćenje visoko regularnih struktura poput sistoličkih polja, smanjenje površine koju zauzimaju veze na čipu moguće je samo korišćenjem logike sa više od dva stanja. Ograničen broj pinova kod VLSI čipova takodje predstavlja veliki problem pri realizaciji složenih funkcija (procesori, memorije i druge), a može se prevazići korišćenjem višeznačne logike [Hurs84].

Najčešće korišćena kola sa višeznačnom logikom imaju definisana četiri logička stanja, koja se obično označavaju brojevima od 0 do 3. Zato ćemo u ovom poglavlju ilustrovati postupak razvoja biblioteke za simulaciju višeznačne logike sa četiri stanja programom Alecsis2.1. Konverzija analognog napona u sistem sa četiri stanja prikazana je na slici 9.1.



Slika 9.1 *Konverzija analogne veličine u višeznačnu logiku sa četiri stanja*

Biblioteka je organizovana u četiri datoteke:

mvl.h - deklaraciona datoteka
fun.hi - funkcijska datoteka
str.hi - modulska datoteka
mod.hi - modelska datoteka

9.1 Definisanje sistema stanja i preopterećenje operatora

Sistem višeznačne logike sa četiri stanja definisan je u datoteci **mvl.h** na sledeći način.

```
typedef enum { '0', '1', '2', '3', ' '=void, '_'=void } mvs_4;
```

U funkcijskoj datoteci **fun.hi** definisana je tabela imena stanja `names_mvs_4`.

```
char *names_mvs_4 [] = { "0", "1", "2", "3" };
```

U višeznačnim sistemima sreću se neki novi logički operatori u odnosu na standardnu logiku sa dva stanja. U datoteci **fun.hi** logički operatori su predefinisani za rad u sistemu sa četiri stanja na sledeći način.

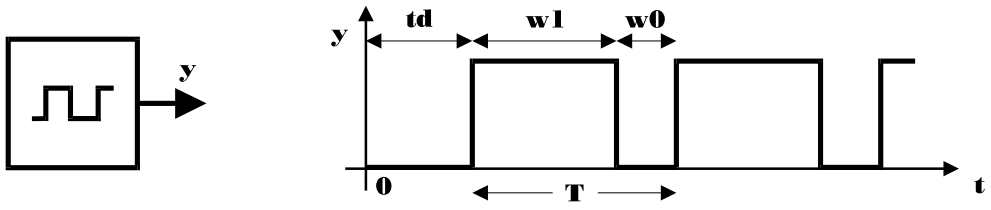
```
const mvs_4 not_table [4] = { '3', '2', '1', '0' };   // complement
mvs_4 operator ~ (mvs_4 op) { return not_table[op]; }

const mvs_4 succ_table [4] = { '1', '2', '3', '0' };  // successor
mvs_4 operator ++ (mvs_4 op) { return succ_table[op]; }

const mvs_4 pred_table [4] = { '3', '0', '1', '2' };  // predictor
mvs_4 operator -- (mvs_4 op) { return pred_table[op]; }

const mvs_4 tsum_table [4][4] = {                     // truncated sum
  '0', '1', '2', '3',
  '1', '2', '3', '3',
  '2', '3', '3', '3',
  '3', '3', '3', '3'
};
mvs_4 operator + (mvs_4 l, mvs_4 r) { return tsum_table[l][r]; }

const mvs_4 msum_table [4][4] = {                     // modulo sum
  '0', '1', '2', '3',
  '1', '2', '3', '0',
  '2', '3', '0', '1',
  '3', '0', '1', '2'
};
mvs_4 operator % (mvs_4 l, mvs_4 r) { return msum_table[l][r]; }

mvs_4 operator & (mvs_4 l, mvs_4 r) { return (l < r ? l : r); }
mvs_4 operator | (mvs_4 l, mvs_4 r) { return (l > r ? l : r); }

mvs_4 operator >> (mvs_4 op, int k) {
  mvs_4 result=op;
  for (int i=0; i<k; i++) { result = ++result; }
  return result;
}

mvs_4 operator << (mvs_4 op, int k) {
  mvs_4 result=op;
  for (int i=0; i<k; i++) result = --result;
  return result;
}
```

9.2 Modeliranje taktnog generatora u sistemu sa četiri stanja

Model taktnog generatora razlikuje se u sistemu sa četiri stanja od slične komponente u sistemu sa dva stanja. Naime, u sistemu sa četiri stanja postoji više mogućnosti za formiranje talasnog oblika taktnih impulsa. Najčešće se koristi taktni generator koji daje talasni oblik sa stanjima `'0'` i `'3'`, ali nije isključeno da korisnik poželi taktni generator koji postavlja i neka druga stanja, recimo `'1'` i `'2'`. Sledećim kodom definisan je modul `clkgen` -taktni generator u sistemu sa

četiri stanja kod koga se inicijalno stanje ulazno/izlaznog priključka `y` komplementira i tako formira talasni oblik.

```
module clkgen (mvs_4 inout y) {
  action (double w1, double w0, double td=0.0) {
    process {
      double delay;
      mvs_4 newy;
      int init=1;

      newy = ~y;
      if (init && td) {
        y <- newy after td;
        init = 0;
      }
      else {
         delay = (y=='3' || y=='2') ? w1 : w0;
         y <- newy after delay;
      }
      wait y;
    }
  }
}
```



a) *b)*

Slika 9.2 *Taktni generator a) blok šema b) talasni oblik*

Modul `clkgen` nema modelsku karticu, ali prima akcione parametre `w1` (trajanje vremenskog intervala u kome se drži stanje višeg naponskog nivoa), `w0` (trajanje vremenskog intervala u kome stanje izlaza odgovara nižoj vrednosti napona) i `td` (trajanje početnog intervala neaktivnosti generatora - vreme mirovanja pre početka generisanja talasnog oblika). Perioda oscilovanja taktnog generatora je `T=w1+w0` (videti sliku 9.2). Početnu vrednost izlaza odredjuje korisnik spolja tako što inicijalizuje vrednost signala za koji je vezan terminal `y` modula `clkgen`. Ako je ova vrednost `'0'` ili `'3'`, dobiće se taktni generator koji prebacuje izmedju ova dva stanja, a ako je početna vrednost `'1'` ili `'2'` dobiće se talasni oblik sa ove dve vrednosti signala.

Ako je potrebno realizovati taktni generator koji prebacuje izmedju dva nekomplementarna stanja, mora se definisati novi modul kod koga će se, recimo, preko akcionih parametara preneti željena stanja.

9.3 Modeliranje osnovnih logičkih komponenti u sistemu sa četiri stanja

Da bi smo obezbedili mogućnost simulacije hibridnih višeznačnih kola, definisana je osnovna (bazna) modelska klasa `io_mv_4` koja obezbedjuje mogućnost modeliranja konvertora. Trenutno nije modeliran ni jedan tip konvertora.

```
class io_mv_4 {                    // class containing converters
  public:
    io_mv_4();
};
```

Ovoj klasi odgovara hipotetički difoltni model `io_mv_4_default_model` u modelskoj datoteci **mod.hi** definisan na sledeći način.

```
model io_mv_4::io_mv_4_default_model { };
```

Iz ove klase izvedena je modelska klasa `GATES` za standardne gejtove i modelska klasa `MVPLA` za višeznačna programabilna logička polja.

9.3.1 Modeliranje gejtova

Za modeliranje osnovnih gejtova definisana je modelska klasa `GATES` na sledeći način.

```
class GATES : public io_mv_4 {
    double tp;
  public:
    GATES();
    >GATES();
    friend module Tgate4, not4, succ4, pred4, max4, min4,
                  tsum4, msum4, pcycle4, ncycle4;
};
```

Kao što se vidi, korišćen je najjednostavniji model dodeljivog kašnjenja. Parametar `tp` predstavlja propagaciono kašnjenje komponente. Konstruktor klase postavlja difoltnu vrednost ovog parametra na 10ns, a procesor klase proverava da li je korisnik prilikom definisanja modelske kartice tipa `GATES` zadao vrednost parametra `tp` koja ima logičkog smisla.

```
GATES::GATES() { tp=10ns; }
GATES::>GATES() {
  if (tp<0.0) warning ("incorect tp value in model card (GATES)", 1);
}
```

Modelskoj klasi `GATES` odgovara difoltni model `GATES_default_model` definisan u datoteci **mod.hi** na sledeći način.

```
model GATES::GATES_default_model { };
```

Kao prijatelji klase `GATES` deklarisani su osnovni gejtovi. Njihove deklaracije iz datoteke **mvl.h** prenosimo u celini.

```
module GATES::Tgate4  (mvs_4 out y; mvs_4 in x0, x1, x2, x3, sel);
module GATES::not4    (mvs_4 out y; mvs_4 in a);
module GATES::succ4   (mvs_4 out y; mvs_4 in a);
module GATES::pred4   (mvs_4 out y; mvs_4 in a);
module GATES::max4    (mvs_4 out y; mvs_4 in a, b);
module GATES::min4    (mvs_4 out y; mvs_4 in a, b);
module GATES::tsum4   (mvs_4 out y; mvs_4 in a, b);
module GATES::msum4   (mvs_4 out y; mvs_4 in a, b);
module GATES::pcycle4 (mvs_4 out y; mvs_4 in a);
module GATES::ncycle4 (mvs_4 out y; mvs_4 in a);
```

Modul `Tgate4` je multiplekser u sistemu sa četiri stanja (T-gejt) i predstavlja potpun skup funkcija u ovom sistemu stanja. Na osnovu stanja na selektorskom ulaznom priključku `sel`, T-gejt prosledjuje stanje sa jednog od četiri ulazna priključka na izlaz. Kod kojim je definisana ova komponenta je sledeći.

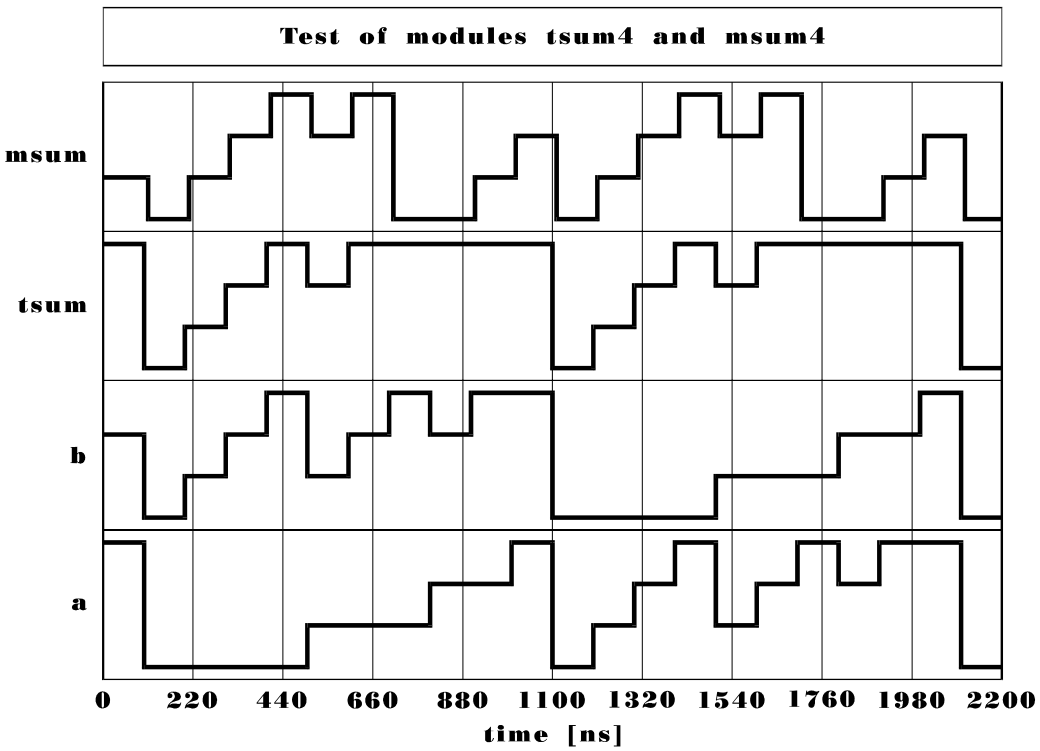
```
module GATES::Tgate4 (mvs_4 out y; mvs_4 in x0, x1, x2, x3, sel) {
  action {
    process (x0, x1, x2, x3, sel) {
      mvs_4 result;
      switch (sel) {
```

```
            case '0' : result = x0; break;
            case '1' : result = x1; break;
            case '2' : result = x2; break;
            case '3' : result = x3; break;
            default  : warning ("unrecognized constant of type mvs_4", 1);
        }
        y <- result after this.tp;
      }
    }
}
```

Modul `not` modelira ponašanje invertora u sistemu sa četiri stanja. Moduli `succ4` i `pred4` daju na izlazu naredno, odnosno prethodno stanje u odnosu na stanje ulaza (sa cikličkim pomeranjem, recimo stanju `'0'` prethodi stanje `'3'`). Pri modeliranju ovih funkcija korišćeni su odgovarajući logički operatori. Recimo, modul `succ4` opisan je sledećim kodom.

```
module GATES::succ4 (mvs_4 out y; mvs_4 in a) {
  action {
    process (a) { y <- ++a after this.tp; }
  }
}
```

Moduli `max4` i `min4` predstavljaju logička kola koja obavljaju MAX i MIN logičke funkcije u sistemu sa četiri stanja, respektivno (izlaz ima vrednost ulaznog signala kome odgovara maksimalna/minimalna analogna vrednost). Definisani su sa dva ulaza, a nije teško definisati ih sa proizvoljnim brojem ulaznih terminala.



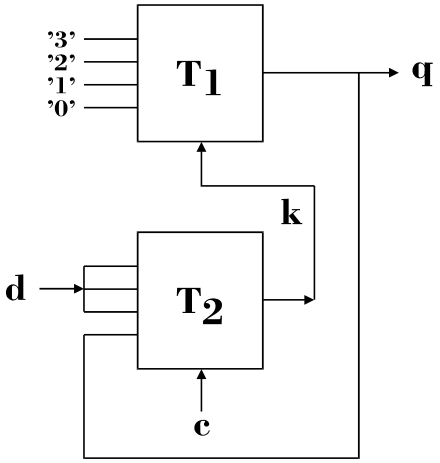
Slika 9.3 *Rezultat testiranja modula* `tsum4` *i* `msum4`

Modul `tsum4` odredjuje sumu napona, odnosno stanja na dva ulazna priključka. Pri tome se rezultat sumiranja zaokružuje odsecanjem na stanje `'3'` ako je iznad njega. Recimo, za stanje ulaza `a='1'` i `b='2'` modul `tsum4` na izlazu daje stanje `'3'`, a za `a='2'` i `b='2'` takodje daje stanje `'3'`, jer dolazi do odsecanja prenosa pri sabiranju. Modul `msum4` obavlja funkciju (a+b)%4, odnosno na njegovom izlazu dobija se ostatak celobrojnog deljenja sume ulaznih stanja sa 4. Rezultat testiranja ova dva modula prikazan je na slici 9.3. Na ulaze `a` i `b` modula `tsum4` i modula `msum4` dovedene su

sve kombinacije vrednosti dva signala u sistemu sa četiri stanja. Testirana ja komutativnost tako što su sve kombinacije dovedene u redosledu ulaza `ab` i redosledu `ba`. Recimo, u trenutku t=900ns na ulaze su dovedena stanja `a='3'` i `b='3'`. Za oba modula korišćen je model `GATES_default_model` definisan konstruktorom klase `GATES`. Posle kašnjenja 10ns (parametar `tp`) modul `tsum4` na izlazu daje stanje `'3'`, a modul `msum` daje stanje `'2'`.

9.3.2 Primeri simulacije

Na slici 9.4 prikazano je kolo koje realizuje funkciju sinhronog D flip-flopa u sistemu sa četiri stanja [Smit88]. Kolo se sastoji od dva T-gejta. Ulazni signal `d`, interni signal `k` i izlazni signal `q` mogu uzeti sva četiri stanja iz sistema stanja. T taktni signal, koji se dovodi na ulaz `c`, je dvostatički pravougaoni signal sa vrednostima `'0'` i `'3'`.

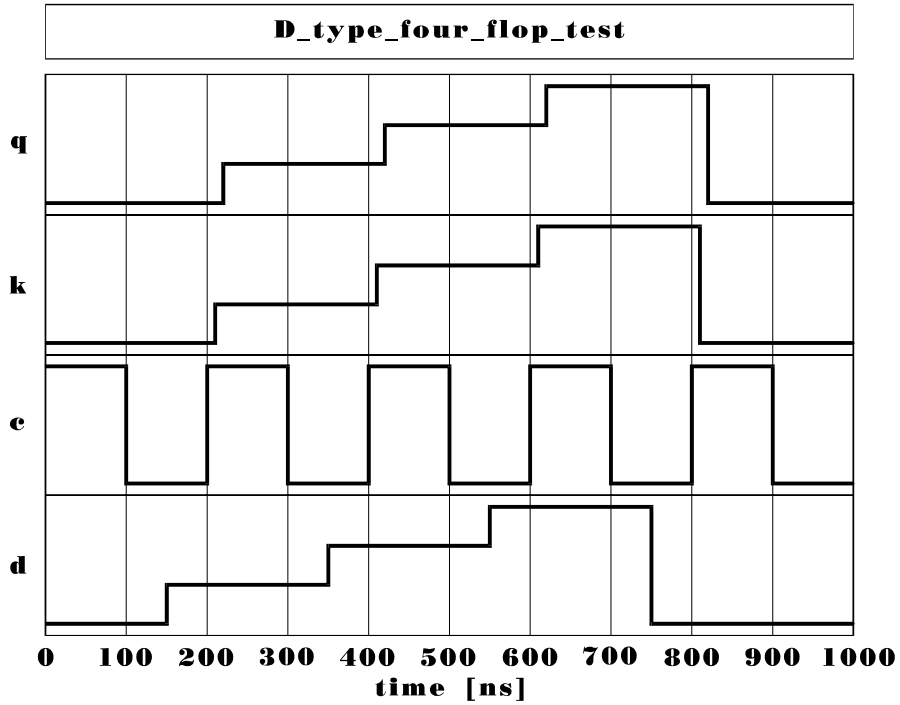


Slika 9.4 *Sinhroni D flip-flop u sistemu sa četiri stanja*

Opis kola za simulaciju je sledeći.

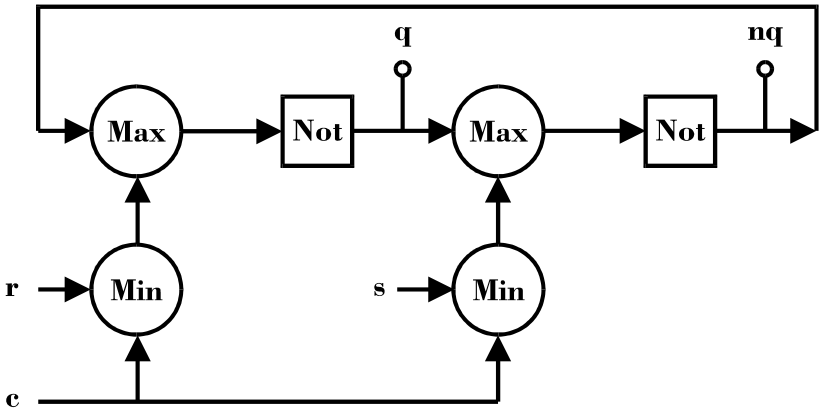
```
# include "mvl.h"
library mod, str, fun;

root module D_type_four_flop_test () {
  module GATES::Tgate4 t1, t2;
  module clkgen cg;

  signal mvs_4 c='3', d='0', k='0', q='0';
  signal mvs_4 _0='0', _1='1', _2='2', _3='3';  // constants

  t1 (q, _0, _1, _2, _3, k) { model=GATES_default_model; };
  t2 (k, q, d, d, d, c)     { model=GATES_default_model; };
  cg (c) action (100ns, 100ns, 0ns);

  timing { tstop = 1000ns; }
  out { signal mvs_4 d, c, k, q; }

  action {
    process initial {
      d <- '1' after 150ns,
           '2' after 350ns,
           '3' after 550ns,
           '0' after 750ns;
    }
  }
}
```

Moduli `t1` i `t2` pozivaju se na difoltni model klase `GATES` `GATES_default_model`, definisan u modelskoj datoteci **mod.hi**, kod koga je propagaciono kašnjenje (parametar `tp`) 10ns. Rezultat simulacije dat je na slici 9.5. Sve dok je taktni signal u stanju `'0'`, flip-flop ima vezan izlaz na ulaz i održava memorisano stanje. Za to vreme ulaz `d` može da menja stanje. Kad taktni signal `c` predje u stanje `'3'`, stanje sa ulaza `d` prenosi se posle kašnjenja multipleksera `t2` na signal `k`, a zatim posle kašnjenja multipleksera `t1` na izlaz flip-flopa `q`.



Slika 9.5 *Rezulat simulacije D flip-flopa*

Slika 9.6 prikazuje sinhroni RS flip-flop u sistemu sa četiri stanja realizovan korišćenjem osnovnih gejtova [Smit88]. Blok označen sa Max predstavlja MAX logički element, modul označen sa Min je MAX logički element, a modul označen sa Not je invertor. Taktni signal, koji se dovodi na ulaz `c`, mora biti definisan za stanja `'0'` i `'3'`. Svi ostali signali u kolu mogu prenositi sva četiri stanja. Kada je taktni signal u stanju `'0'`, MIN kola su blokirana i flip-flop zadržava memorisano stanje na izlazima. Kad je taktni signal u stanju `'3'`, ako su stanja signala `s` i `r` komplementarna $\mathbf{r = \bar{s}}$, flip-flop memoriše novo stanje $\mathbf{q = \overline{nq} = s}$. Ako stanja ulaza `r` i `s` nisu komplementarna, može doći do neregularnog rada flip-flopa.

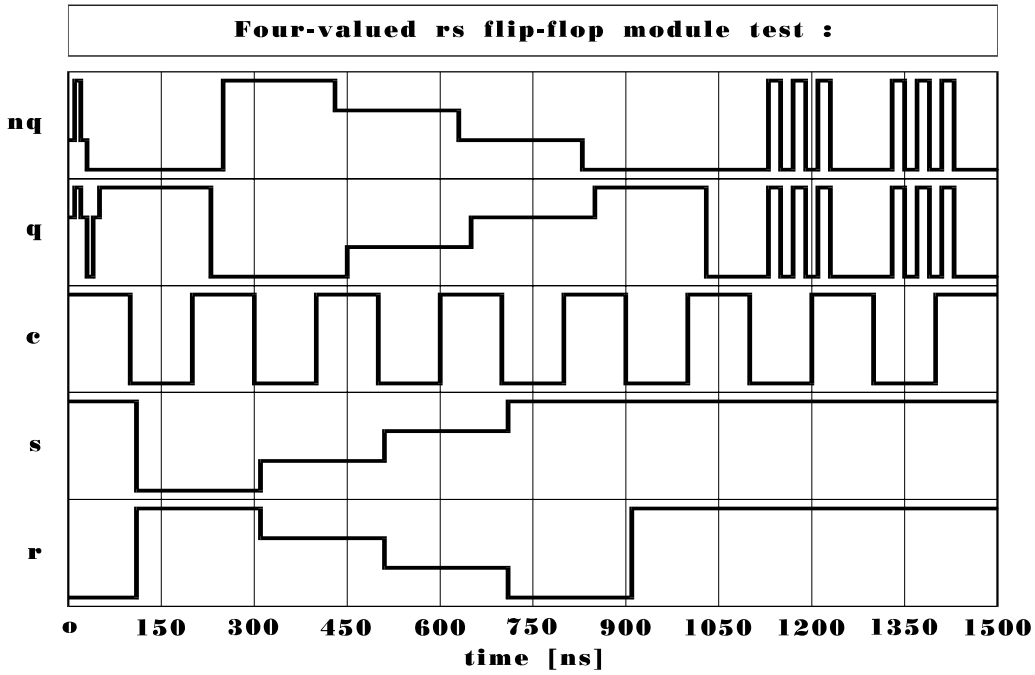


Slika 9.6 *Sinhroni RS flip-flop u sistemu sa četiri stanja*

Kolu sa slike 9.6 odgovara sledeći opis za simulaciju.

```
# include "mvl.h"
library mod, str, fun;
root rsff4_test () {
  module GATES::not4 n1, n2;
  module GATES::min4 min1, min2;
  module GATES::max4 max1, max2;
  module clkgen ck;
  signal mvs_4 min1o, min2o;
  signal mvs_4 max1o, max2o;
  signal mvs_4 r='0', s='3', q='2', nq='1', c='3';

  n1   (q,  max1o)        model=GATES_default_model;
  n2   (nq, max2o)        model=GATES_default_model;
  min1 (min1o, r, c)      model=GATES_default_model;
  min2 (min2o, s, c)      model=GATES_default_model;
  max1 (max1o, nq, min1o) model=GATES_default_model;
  max2 (max2o,  q, min2o) model=GATES_default_model;
  ck   (c) action (100ns, 100ns);

  timing {  tstop = 1500ns; }
  out {
    caption "Four-valued rs flip-flop module test :";
    signal mvs_4 r, s, c, q, nq;
  }
  action {
    process initial {
      s <- '0' after  110ns, '1' after 310ns, '2' after 510ns, '3' after 710ns,
           '3' after 910ns;
      r <- '3' after  110ns, '2' after 310ns, '1' after 510ns, '0' after 710ns,
           '3' after 910ns;
    }
  }
}
```

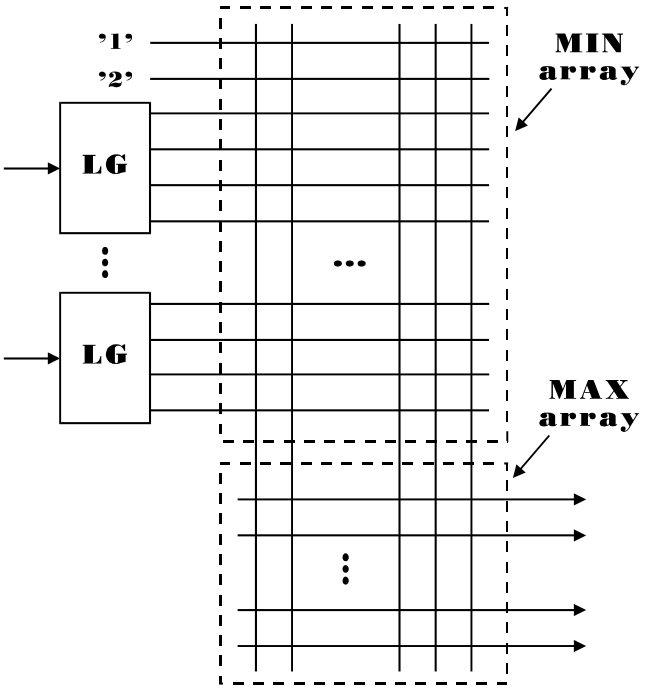
Rezultat simulacije dat je na slici 9.7. Nestabilnost kola u početnom trenutku nastaje kao posledica pogrešno memorisanog početnog stanja koje ne odgovara vrednostima ulaznih signala u početnom trenutku simulacije. Nestabilnost kola pri kraju simulacije namerno je prouzrokovana time što signali `s` i `r` nisu postavljeni na komplementarne vrednosti.



Slika 9.7 *Rezultati simulacije RS flip-flopa*

9.3.3 Modeliranje višeznačnih programabilnih logičkih polja

Strukturna blok šema programabilnog logičkog polja za realizaciju funkcija u višeznačnoj logici sa četiri stanja prikazana je na slici 9.8. Polje se sastoji od MIN matrice, MAX matrice i generatora literala (označeni sa LG).



Slika 9.8 *Blok šema višeznačne PLA strukture*

Za potrebe modeliranja višeznačnih PLA struktura definisana je modelska klasa `MVPLA`.

```
class MVPLA : public io_mv_4 {
    int up_rows, down_rows, cols;   // min table and max table dimensions
    int is_set;                     // ==1 if mvpla already initialized
    int show_only_once;             // ==0 before first print to the screen
    double tp;                      // chip propagation time
    Bool *minimum_flag;             // floating minimum line flag
    Bool *maximum_flag;             // floating maximum line flag
    bit **up_matrix, **down_matrix; // min, max matrices configuration
    char *filename;                 // setup file name
    mvs_4 *input_line;              // min matrix inputs
    mvs_4 *minimum_line;            // min matrix outputs, max matrix inputs
    mvs_4 *output_line;             // max matrix outputs
  public:
    MVPLA(int, int, int);
    ~MVPLA();
    >MVPLA();
    int set();                      // read fuse configuration from setup file
    void show();                    // print fuse configuration to the screen
    mvs_4 *produce_output(mvs_4 *);

    friend module mvpla;
};
```

Konstruktor klase `MVPLA` prima dimenzije polja, broj vrsta u MIN matrici, broj vrsta u MAX matrici i broj kolona, kao parametre i na osnovu njih rezerviše potreban memorijski prostor za podatke klase. Već sam konstruktor proverava logičnost zadatih vrednosti dimenzija matrica (što je obično posao procesora), s obzirom da nema smisla alocirati memoriju, ako su ove vrednosti nelogično zadate.

```
MVPLA::MVPLA(int u_rows, int d_rows, int c) {
   int i;

   if ((up_rows=u_rows) < 2)   warning("wrong up_rows in MVPLA", 1);
   if ((down_rows=d_rows) < 1) warning("wrong down_rows in MVPLA", 1);
   if ((cols=c) < 1)           warning("wrong cols in MVPLA", 1);
   if (!(up_matrix = (bit **)calloc(up_rows, sizeof(bit*))))
     warning("no room for up_matrix in MVPLA", 1);
   if (!(down_matrix = (bit **)calloc(down_rows, sizeof(bit*))))
     warning("no room for down_matrix in MVPLA", 1);
   for (i=0; i<up_rows; i++) {
     if (!(up_matrix[i] = (bit*)calloc(cols, sizeof(bit))))
       warning("no room for up_matrix in MVPLA", 1);
   }
   for (i=0; i<down_rows; i++) {
     if (!(down_matrix[i] = (bit*)calloc(cols, sizeof(bit))))
       warning("no room for down_matrix in MVPLA", 1);
   }
   if (!(minimum_flag = (Bool *)calloc(cols, sizeof(Bool))))
     warning("no room for minimum_flag in MVPLA", 1);
   if (!(maximum_flag = (Bool *)calloc(down_rows, sizeof(Bool))))
     warning("no room for maximum_flag in MVPLA", 1);
   input_line   = new mvs_4[up_rows];
   minimum_line = new mvs_4[cols];
   output_line  = new mvs_4[down_rows];
   is_set = 0;                        // still not set
   show_only_once = 0;                // still not printed
   tp = 10.0nsec;                     // default propagation delay
   input_line[0]='1';                 // constant '1'
   input_line[1]='2';                 // constant '2'
}
```

Destruktor klase `MVPLA` oslobadja rezervisani memorijski prostor.

```
MVPLA::~MVPLA() {
   int i;
   for (i=0; i<up_rows; i++)   delete up_matrix[i];
   for (i=0; i<down_rows; i++) delete down_matrix[i];
   delete up_matrix;
   delete down_matrix;
   delete minimum_flag;
   delete maximum_flag;
   delete input_line;
   delete minimum_line;
   delete output_line;
}
```

Definisan je i procesor klase `MVPLA`.

```
MVPLA::>MVPLA() {
  int i, m, o;                        // counters for input, minimum, output lines

  if ( !set() ) warning ("errors during MVPLA initialization", 1);
                                      // Check existency of floating lines:
  for (m=0; m<cols; m++) minimum_flag[m]=false;
  for (m=0; m<cols; m++) {            // take a minimum line
    for (i=0; i<up_rows; i++) {       // look for a connection to input line
      if (up_matrix[i][m] == '1') {
        minimum_flag[m]=true; // connection exists
        break;                        // next product line, please
      }
    }                                 // for (i=0 ...
  }                                   //for (m=0 ...

  for (o=0; o<down_rows; o++) maximum_flag[o]=false;
  for (o=0; o<down_rows; o++) {       // take an output line
    for (m=0; m<cols; m++) {          // find first connection
```

```
      if (down_matrix[o][m] == '1') {
        maximum_flag[o]=true;        // connection exists
        break;                       // connection found, stop search
      }
    }                                // for (m=0 ...
  }                                  // for (o=0 ...

  for(m=0; m<cols; m++) {
    if (minimum_flag[m]==false)      // no connection to inputs
      printf("Warning: floating minimum line %d (MIN matrix).\n", m);
  }
  for(o=0; o<down_rows; o++) {
    if (maximum_flag[o]==false)      // no connection to minimums
      printf("Warning: floating maximum line %d (MAX matrix).\n", o);
  }
}                                    // MVPLA::>MVPLA()
```

Uloga procesora je višestruka. Najpre učitava konfiguraciju osigurača iz *setup* datoteke (funkcija `set`). Zatim proverava da li u matricama postoje linije minimuma (kolone) koje nisu povezane sa ulaznim linijama (vrstama MIN matrice) i linije maksimuma (izlazne linije, vrste MAX matrice) koje nisu povezane ni sa jednom linijom minimuma. Ovakve linije obeležavaju se vrednošću `false` u vektoru `minimum_flag`, odnosno `maximum_flag`, a korisnik se obaveštava o njihovom postojanju. Korisnik pri definisanju modelske kartice konkretne PLA komponente zadaje ime *setup* datoteke pomoću parametra `filename`. Funkcija `set` otvara ovu datoteku i iz nje učitava stanja osigurača u MIN i MAX matricama. Funkcija `set` realizovana je tako da podržava linijske komentare koji počinju zvezdicom (`*`) i traju do kraja linije. Prazan prostor, tabulatori i prelazak u novi red se ignorišu. Modelskoj klasi `MVPLA` dogovara difoltni model `MVPLA_default_model` u datoteci **mod.hi** sa sledećim opisom.

```
model MVPLA::MVPLA_default_model (2, 1, 1) {
  filename = "default.pla";
};
```

Stanje osigurača zadaje se oznakom `0` za prekid i oznakom `1` za kratak spoj izmedju dve linije u matrici. Datoteka **default.pla** postoji na tekućem direktorijumu, ali je prazna. U takvom slučaju, podrazumeva se da su svi osigurači pregoreli, kao da su u datoteci sve nule, odnosno da nigde nema kontakta u matrici. Minimalna moguća PLA struktura nema ulaze, ima 2 vrste u MIN matrici za konstante `'1'` i `'2'`, jednu kolonu za liniju minimuma i jednu vrstu u MAX matrici, odnosno jedan izlaz. Ovakvim poljem moguće je ostvariti funkcije konstante `'1'` i `'2'`. Konfiguracija osigurača kojom se realizuje funkcija f=`'1'` opisana je sledećom *setup* datotekom.

```
************ Setup file for function f='1':
* MIN matrix:
      1
      0
* MAX matrix:
      1
```

Ključna funkcija u modelskoj klasi `MVPLA` je funkcija `produce_output`. Ona odredjuje stanja na izlaznim linijama polja za zadata stanja na ulazima. Opisana je sledećim kodom u funkcijskoj datoteci **fun.hi**.

```
mvs_4 * MVPLA::produce_output(mvs_4 x[]) {
    int i, m, o;                     // counters: input, minimum, output

    if( !show_only_once ) {          // Print pla configuration to screen:
      show();
      show_only_once++;
    }

                                     // Initialize input_line vector:
```

```
    for (i=0; i<(up_rows-2)/4; i++) {
      switch (x[i]) {               // literal generation:
        case '0' :
            input_line[2+4*i]='0';  // y0 = x{'1', '2', '3'}
            input_line[3+4*i]='3';  // y1 = x{'0', '2', '3'}
            input_line[4+4*i]='3';  // y2 = x{'0', '1', '3'}
            input_line[5+4*i]='3';  // y3 = x{'0', '1', '2'}
            break;
        case '1' :
            input_line[2+4*i]='3';
            input_line[3+4*i]='0';
            input_line[4+4*i]='3';
            input_line[5+4*i]='3';
            break;
        case '2' :
            input_line[2+4*i]='3';
            input_line[3+4*i]='3';
            input_line[4+4*i]='0';
            input_line[5+4*i]='3';
            break;
        case '3' :
            input_line[2+4*i]='3';
            input_line[3+4*i]='3';
            input_line[4+4*i]='3';
            input_line[5+4*i]='0';
            break;
        default :
            warning("constant of type mvs_4 not recognized", 1);
      }                             // switch (x[i])
    }                               // for (i=0 ...
                                    // MIN operation :
    for (m=0; m<cols; m++) minimum_line[m]='3'; // non-dominant state
    for (m=0; m<cols; m++) {        // for each minimum line do ...
      for (i=0; i<up_rows; i++) {
        if (minimum_flag[m] == false) { // floating line
          minimum_line[m] = '3';    // disconnected minimum line state
          break;
        }
        else if (up_matrix[i][m] == '1') { // connected, do MIN
          minimum_line[m] = minimum_line[m] & input_line[i];
          if (input_line[i] == '0') break; // dominant state, stop
        }
      }                             // for (i=0 ...
    }                               // for (m=0 ...
                                    // MAX operation :
    for (o=0; o<down_rows; o++) output_line[o]='0';   // non-dominant state
    for (o=0; o<down_rows; o++) {
      for (m=0; m<cols; m++) {
        if (maximum_flag[o] == false) { // floating line
          output_line[o] = '0';     // disconnected maximum line state
          break;                    // next output line, please
        }
        else if (down_matrix[o][m] == '1') { // connected, do MAX
          output_line[o] = output_line[o] | minimum_line[m];
          if (minimum_line[m] == '3') break; // dominant state, stop
        }                           // if ...
      }                             // for (m=0 ...
    }                               // for (o=0 ...

    return output_line;
}                                   // mvs_4 MVPLA::produce_output()
```

Ulazni parametar funkcije `produce_output` je vektor signala tipa `mvs_4` koji sadrži stanja ulaznih priključaka polja. Promenljiva `show_only_once` obezbedjuje da se konfiguracija osigurača u MIN i MAX matrici štampa korišćenjem funkcije `show` na ekranu samo jednom, pri prvom ulasku u funkciju `produce_output`. Zatim se na osnovu stanja ulaznih priključaka odredjuju stanja ulaznih

linija MIN matrice (vektor `input_line`). Ovime je modelirana funkcija generatora literala. Na prve dve linije (promenljive `input_line[0]` i `input_line[1]`) uvek se dovode konstante `'1'` i `'2'`, što je definisano u konstruktoru klase `MVPLA`. Na osnovu stanja ulaznih linija i konfiguracije osigurača, odredjuju se stanja na linijama minimuma (vektor `minimum_line`). Zatim se na osnovu stanja linija minimuma odredjuju stanja na izlaznim linijama (vektor `output_line`). Treba primetiti da linije minimuma koje nisu povezane ni sa jednom ulaznom linijom bivaju postavljene u stanje `'3'`, a da izlazne linije koje nisu povezane sa linijama minimuma bivaju postavljene u stanje `'0'`. Kako ovo može imati nepredvidive posledice po funkciju kola, a obično je rezultat greške u konfigurisanju matrice osigurača, procesor klase `MVPLA` je zadužen da prijavi korisniku pojavu nepovezanih linija.

Sa ovako definisanom funkcijom `produce_output` možemo razviti model višeznačnog programabilnog logičkog polja proizvoljnih dimenzija. Odgovarajući modul nosi ime `mvpla` i definisan je u datoteci **str.hi** sledećim kodom.

```
module MVPLA::mvpla (mvs_4 out y[]; mvs_4 in x[]) {
  action {
    process (x) { y <- this.produce_output(x) after this.tp; }
  }
}
```

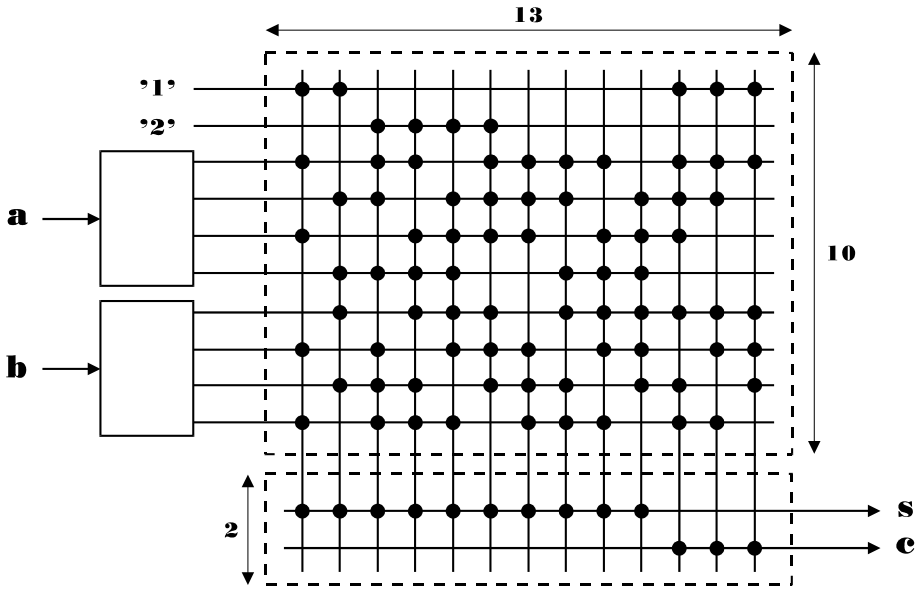
Vektor `x` predstavlja ulaze u PLA (na ulaze se dovode vrednosti promenljivih), a vektor `y` predstavlja izlaze PLA (funkcije realizovane poljem). Modulu `mvpla` odgovara modelska kartica tipa `MVPLA` koja mora da sadrži dimenzije polja i raspored osigurača. Ilustrovaćemo upotrebu modula `mvpla` jednim jednostavnim primerom u narednom odeljku.

9.3.4. Primer simulacije

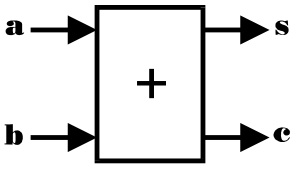
Na slici 9.9 prikazana je PLA struktura koja realizuje funkciju sabirača u sistemu sa četiri stanja [Sasa89]. Sa `a` i `b` su označeni ulazi u sabirač, a izlazi `s` i `c` predstavljaju sumu i prenos. Funkcija sabirača predstavljena je tabelarno na slici 9.10. Testiraćemo da li konfiguracija osigurača prikazana na slici 9.9 daje željene funkcije na izlazima PLA strukture. Opis kola za simulaciju je sledeći.

```
# include "mvl.h"
library mod, str, fun;

root module mvpla_realization_of_full_adder () {
  module MVPLA::mvpla mvpla_chip;
  signal mvs_4 x[2]="00", y[2]="00";      // x[0]-a;   x[1]-b;
                                          // y[0]-sum;  y[1]-carry
  mvpla_chip (y, x) model = fa_mvpla;

  timing {  tstop = 2000ns; }
  out {
    caption "Adder in 4-state logic system";
    signal mvs_4  a  { return x[0]; };
    signal mvs_4  b  { return x[1]; };
    signal mvs_4  c  { return y[1]; };
    signal mvs_4  s  { return y[0]; };
  }
  action {
    process initial {              // mvpla excitation:
      x <- "00" after  100ns,
           "01" after  200ns,
           "02" after  300ns,
           "03" after  400ns,
           "10" after  500ns,
           "11" after  600ns,
           "12" after  700ns,
           "13" after  800ns,
```

```
            "20" after  900ns,
            "21" after 1000ns,
            "22" after 1100ns,
            "23" after 1200ns,
            "30" after 1300ns,
            "31" after 1400ns,
            "32" after 1500ns,
            "33" after 1600ns;
    }
  }
}
```



Slika 9.9 *Sabirač realizovan pomoću višeznačne PLA strukture*



a	b	c	s
0	0	0	0
0	1	0	1
0	2	0	2
0	3	0	3
1	0	0	1
1	1	0	2
1	2	0	3
1	3	1	0
2	0	0	2
2	1	0	3
2	2	1	0
2	3	1	1
3	0	0	3
3	1	1	0
3	2	1	1
3	3	1	2

Slika 9.10 *Blok šema i tablica istinitosti sabirača u sistemu sa četiri stanja*

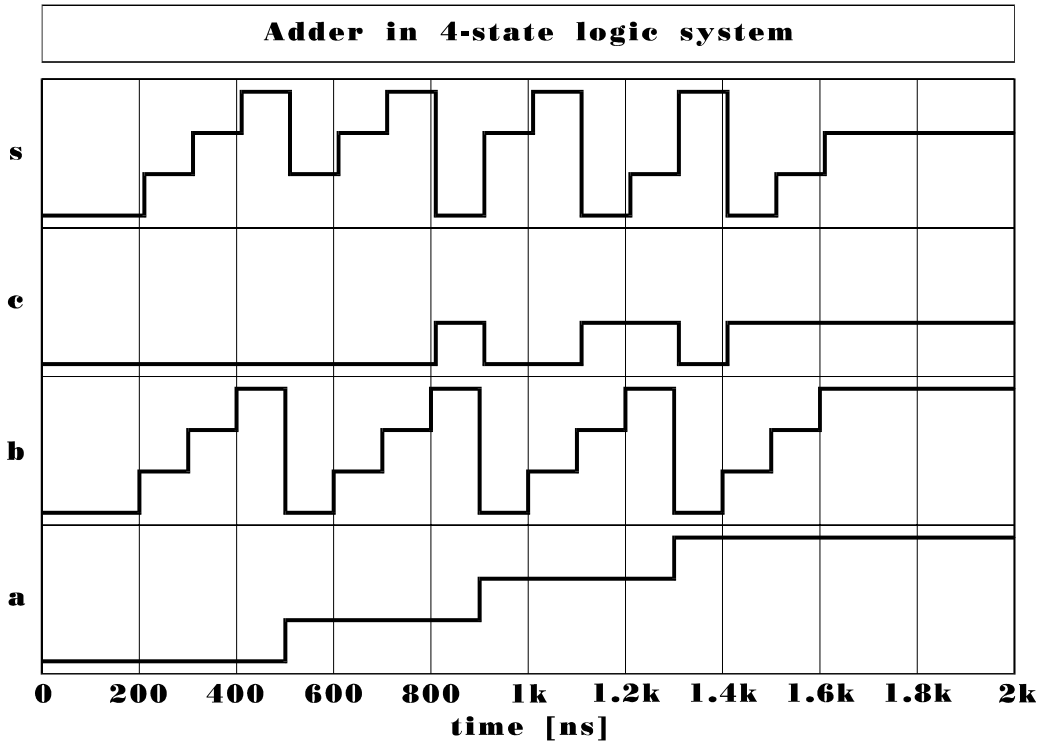
Korišćeni model `fa_mvpla` definisan je u datoteci **mod.hi** i ima sledeći izgled.

```
model MVPLA::fa_mvpla (10, 2, 13) {
    tp = 10ns;
    filename = "mvfulladd.pla";
};
```

Dimenzije matrica (10 vrsta u MIN matrici, 2 vrste u MAX matrici i 13 kolona) predate su konstruktoru klase `MVPLA`, definisano je propagaciono kašnjenje PLA strukture od 10ns, a ime *setup* datoteke je **mvfulladd.pla**. Sadržaj ove datoteke je sledeći.

```
* MVPLA setup file
* Full adder for numbers in 4-value system realized with multi-valued PLA
     * MVPLA min table
  1100000000111                * constant '1'
  0011110000000                * constant '2'
  1011011110111          * x1 {'1', '2', '3'}
  0110111101110          * x1 {'0', '2', '3'}
  1001111011100          * x1 {'0', '1', '3'}
  0111100111000          * x1 {'0', '1', '2'}
  0101110111111    * x2 {'1', '2', '3'}
  1010111011011    * x2 {'0', '2', '3'}
  0111011101101    * x2 {'0', '1', '3'}
  1011101110110    * x2 {'0', '1', '2'}
     * MVPLA max table
  1111111111000    * -> sum
  0000000000111    * -> carry
```

Na ulaze komponente `mvpla_chip` dovedene su sve kombinacije ulaznih signala. Rezultat simulacije, prikazan na slici 9.11, verifikuje ispravnost programiranja PLA strukture za ostvarivanje željenih funkcija.



Slika 9.11 *Rezultat simulacije sabirača realizovanog pomoću modula* `mvpla`

10 Zaključak

Razvoj modernih simulatora elektronskih kola predstavlja interdisciplinarnu oblast u kojoj se kao najvažniji zahtevi postavljaju funkcionalnost i fleksibilnost simulatora. Kao standard u oblasti digitalne simulacije nametnuo se VHDL. Oblast analogne simulacije zasnovana je na SPICE principima, ali se zadnjih godina pojavilo više simulatora sa znatnim unapredjenjima, posebno u pogledu automatizacije modeliranja analognih komponenti pomoću ulaznog jezika. Moderni simulatori imaju sve moćnije ulazne jezike, a u programsko jezgro se ugradjuje samo najneophodniji minimum osnovnih simulacionih mehanizama.

Simulator Alecsis2.1 predstavlja praktičnu primenu najnovijih dostignuća u oblasti analogne i digitalne simulacije, što ga svrstava u sam vrh svetskih simulatora danas. Alecsis2.1 je otvoreni sistem, sa sposobnošću da raste i usavršava se sa upotrebom. Ulazni jezik AleC++ nudi širok spektar različitih metoda za modeliranje pojava iz domena analonih i digitalnih sistema različitih nivoa apstrakcije. To je objektno-orjentisan jezik koji pored osobina jezika C++ sadrži minimum dodatnih instrukcija neophodnih za opisivanje strukture i funkcionisanja elektronskih sistema. Zbog ovoga je vrlo pristupačan i jednostavan za korišćenje, pogotovo verziranim korisnicima programskog jezika C++.

Kako simulator Alecsis2.1 sadrži samo osnovne mehanizme digitalne simulacije, praktično je neupotrebljiv za logičku simulaciju dok se ne razviju odgovarajuće digitalne biblioteke. Biblioteke se razvijaju spolja, bez intervencije na programskom kodu samog simulatora, pomoću ulaznog jezika AleC++. Razvoj biblioteke zasniva se na izboru sistema stanja, modela kašnjenja, skupa osnovnih logičkih komponenti i rezolucionih funkcija. Ovo čini kompletnu specifikaciju za jedan logički simulator, tako da svaka nova digitalna biblioteka koja se razvije pretvara simulator Alecsis2.1 u potpuno novi logički simulator. Mogućnosti ovakvog pristupa su ogromne, počev od emulacije postojećih logičkih simulatora, preko razvoja simulatora koji koriste samo najbolje osobine postojećih simulatora, pa do kreiranja potpuno novih metoda simulacije u digitalnom domenu.

U ovom radu je izložena metodologija za razvoj digitalnih biblioteka različite kompleksnosti. Krenuli smo od simulacije u sistemu sa dva stanja koja se upotrebljava za verifikaciju logičke funkcije projektovanog digitalnog kola. Razvijena biblioteka za simulaciju u sistemu sa dva stanja trenutno sadrži modele osnovnih logičkih elemenata kombinacione logike (I, ILI, ekskluzivno ILI, potpuni sabirač, ...). Takodje, ova biblioteka sadrži i model PLA. U ovu biblioteku mogu se ugraditi modeli sekvencijalnih elemenata (flip-flopovi, memorije), ali to nije učinjeno zbog problema pri inicijalizaciji. Naime, pri inicijalizaciji je neophodno opredeliti se za

odredjeno memorisano stanje sekvencijalnog elementa, što nije uvek moguće. Tada nam je potrebno neodredjeno (nepoznato) stanje koje u biblioteci sa dva stanja ne postoji.

Da bi se razrešili problemi sa inicijalizacijom uvedeno je neodredjeno stanje `'X'` i razvijena biblioteka za sistem sa tri stanja. Biblioteka sadrži modele standardnih gejtova, osnovnih flip-flopova, PLA, RAM i ROM. Moguća je dopuna ove biblioteke ugradnjom komplikovanijih modela flip-flopova (sa asinhronim ulazima za reset, set, ...), kao i PLA struktura (trostatički izlazi, izlazi sa memorijskim elementima, ...) i memorija (chip select, ...).

Sistem sa četiri stanja sadrži i stanje visoke impedanse `'Z'`. Razvijena biblioteka za simulaciju u ovom sistemu stanja omogućava simulaciju trostatičkih kola. Specifičnost ove biblioteke su mehanizmi za razrešenje problema rezolucije na magistrali i drugih problema koji nastaju uvodjenjem stanja visoke impedanse. Biblioteka za sistem sa četiri stanja može se smatrati najvažnijom bibliotekom, s obzirom da većina postojećih logičkih simulatora zasniva rad na ovom sistemu stanja. Medjutim, ovaj sistem stanja ne daje mogućnost modeliranja različitih tehnologija izrade digitalnih kola. Za modeliranje tehnologije izrade neophodno je uvodjenje informacije o jačini (izlaznoj otpornosti) kola na čijem izlazu se javilo odredjeno logičko stanje. Ponudili smo dva metoda za rešavanje ovog problema.

Prvi metod je ugradnja dodatnih stanja u sistem stanja kako bi se razlikovala jaka od slabih stanja. Ovaj metod ugradjen je u simulator HILO, jedan od najpoznatijih logičkih simulatora do danas. Razvijena je digitalna biblioteka za logičku simulaciju u stilu simulatora HILO. Implementiran je odgovarajući sistem sa petnaest logičkih stanja, odgovarajući model kašnjenja, kao i sve prateće funkcije (rezolucija signala, memorijska svojstva signala, modovi simulacije i dr.) koje simulator Alecsis2.1 pretvaraju u simulator HILO.

Drugi način modeliranja jačine signala koji smo realizovali odgovara onom ugradjenom u hibridni simulator PSpice. Kod simulatora PSpice uz informaciju o stanju (ugradjen je sistem sa pet logičkih stanja) pridružena je i informacija o vrednosti izlazne otpornosti kola na čijem izlazu je stanje generisano. Ovakav pristup zahteva da se pri simulaciji kroz kolo prenose parovi stanje-otpornost. Opisana je odgovarajuća implementacija u simulatoru Alecsis2.1. Biblioteka za simulaciju u PSpice sistemu stanja trenutno sadrži sistem stanja, model kašnjenja, osnovnu rezolucionu funkciju i standardne gejtove. Data je samo ilustracija upotrebe ove biblioteke, a da bi biblioteka bila potpuna, potrebno je implementirati ostale logičke komponente koje prepoznaje simulator PSpice: trostatičke gejtove, flip-flopove, memorije, PLA strukture i druge.

Mehanizmi za inicijalizaciju logičkog kola opisani su u drugom poglavlju i nisu (osim u specijalnim slučajevima, pri simulaciji nekih specifičnih primera) implementirani u razvijenim bibliotekama. Smatrano je da bi oni nepotrebno komplikovali modele digitalnih komponenti. Tokom izrade ovog rada, medjutim, došlo se do saznanja da je takav pristup bio pogrešan. Moguće je u modele ugraditi neki jednostavniji mehanizam inicijalizacije, tako da će biblioteke biti dopunjene u tom pogledu. Imena modula kod kojih je ugradjen mehanizam inicijalizacije nose nastavak **_init**. Na primer, ime invertora bez inicijalizacije je `inv`, a sa incijalizacijom `inv_init`.

Mehanizmi za detekciju prekršaja vremenskih ograničenja (setup, hold, visoke frekvencije i slično) i odgovarajuće intervencije simulatora opisani su u drugom poglavlju, a ugradjeni su samo u neke komponente, kada je to bilo potrebno za konkretnu simulaciju. Ovaj pristup smatramo dobrim, jer ugradnja preterano detaljnih modela osnovnih komponetata može da uspori simulaciju i da oteža korišćenje biblioteke. Objektno-orjentisani pristup modeliranju, implementiran kod jezika AleC++, omogućava modifikaciju osnovnih modela, tako da se biblioteke mogu lako proširiti detaljnim modelima.

Modeliranje hazarda obradjeno je na više mesta u radu, a razvijena je i posebna biblioteka za analizu hazarda zasnovana na sistemu sa devet logičkih stanja koji je još 1974. godine razvio Giuseppe Fantauzzi. Simulacija u ovom sistemu stanja obavlja se sa nultim kašnjenjem. Za potrebe procesiranja rezultata simulacije koji su u tekstualnom obliku, pa se ne mogu grafički prikazati, razvijen je poseban program NZD.

I, na kraju, medju osnovne primene simulatora Alecsis2.1 spada i simulacija višeznačnih digitalnih kola. Razvijena je biblioteka za simulaciju višeznačne logike sa četiri stanja i ilustrovana

njena primena za simulaciju na nivou gejtova i za simulaciju višeznačnih PLA struktura. Dalji rad na simulaciji višeznačne logike obuhvataće ugradnju neodredjenog stanja i stanja visoke impedanse u sistem logičkih stanja.

Metodi modeliranja digitalnih elektronskih kola izloženi u ovom radu standardni su, ali su inovirani objektno-orjentisanom pristupom modeliranju. Prikazan je samo deo mogućnosti simulatora Alecsis2.1, jer nam je cilj bio da opišemo osnovne, standardne biblioteke za logičku simulaciju. Usvojene konvencije za razvoj simulacionih biblioteka su karakteristika simulatora Alecsis2.1, ali mogu imati i opštiji karakter i biti dobra smernica za rad drugim projektantima. Opisane biblioteke sadrže preko 10000 linija programskog koda u jeziku AleC++, ne uzevši u obzir datoteke za testiranje. Od različitih primena simulatora Alecsis2.1 u oblasti digitalne simulacije koje nisu opisane pomenućemo simulaciju računarskog hard diska [Vasi95] i simulaciju rada računarskih mreža [Maks95].

Usavršavanje simulatora Alecsis2.1 nastaviće se i u budućnosti. Trenutno je u toku rad na postizanju potpune kompatibilnosti sa programom SPICE, kao i razvoj VHDL kompajlera koji će davati strukture podataka za simulator Alecsis2.1.

Literatura

[Abra90] Abramovici, M., Breuer, A. M., Friedman, D. A., **Digital System Testing and Testable Design**, Computer Science Press, New York, 1990.

[Acun90] Acuna, E. L., Dervenis, J. P., Pagones, A. J., Yang, F. L., Saleh, R. A., **"Simulation Techniques for Mixed Analog/Digital Circuits"**, IEEE Journal of Solid-State Circuits, Vol. SC-25, No. 2, pp. 353-362, April 1990.

[Agra80] Agrawal, V. D., Bose, A. K., Kozak, P., Nham, H. N., Pacas-Skewes, E., **"A Mixed-Mode Simulator"**, Proc. of the 17th DAC, pp. 618-625, Minneapolis, Minnesota, June 23-25, 1980.

[Bush83] Bush, M. E., **"LMOSII - A Logic Simulator for MOS Integrated Circuits"**, Proc. of IEEE Int. Conf. on Simulators, pp. 263-266, Brighton, Breat Britain, 1983.

[Chad92] Chadha, R., Visweswariah, C., Chen, C.-F., **"M^3 - A Multilevel Mixed-Mode Mixed A/D Simulator"**, IEEE Trans. on Computer-Aided Design, Vol. CAD-11, No. 5, pp 575-585, May 1992.

[Chap74] Chappel, S. G., Elmendorf, C. H., Schmidt, L. D., **"LAMP: Logic-Circuit simulators"**, The Bell System Technical Journal, Vol. 53, No. 8, pp. 1451-1476, October 1974.

[Coel88] Coelho, R. D., **"VHDL: A Call for Standards"**, Proc. of the 25th DAC, pp. 40-47, 1988.

[Coel89] Coelho, R. D., **The VHDL Handbook**, Kluwer Academic Publishers, 1989.

[Coel90] Coelho, R. D., **"A VHDL Standard Package for Logic Modeling"**, IEEE Design and Test of Computers, Vol. 7, No. 3, pp. 25-32, June 1990.

[Corm88] Corman T., Wimbrow U. M., **"Coupling a Digital Logic Simulator and an Analog Circuit Simulator"**, VLSI Systems Design, pp. 40-47, February 1988.

[Davi87] Davio, M., Deschamps, J.P., Thayse, A., **Discrete and Switching Functions**, McGraw-Hill, 1987.

[Evan78] Evans, J. D., **"Accurate Simulation of Flip-Flop Timing Characteristics"**, Proc. of the 15th DAC, pp. 398-404, Las Vegas, Nevada, June 19-21, 1978.

[Fant74] Fantauzzi, G., **"An Algebraic Model for the Analysis of Logical Circuits"**, IEEE Trans. on Computers, Vol. C-23, No. 6, pp. 576-581, June 1974.

[Faza88] Fazakerly, B. W., Smith, P. R., **"Advanced Modeling Techniques for Logic Simulation"**, VLSI Systems Design, 1988 Semicustom Design Guide, pp. 74-85, CMP Publication, New York, 1988.

[Feld92] Feldmann, P., Melville, R., Moinian, S., **"Automatic Differentiation in Circuit Simulation and Device Modeling"**, IEEE/ACM Int. Conf. on CAD Proc., pp. 248-253, Santa Clara, California, November 8-12, 1992.

[Flak74] Flake, P., Musgrave, G., White, I. J., **"HILO - A Logic System Simulator"**, IEE Int. Conf. on Computer-Aided Design Proc., pp. 130-136, Southampton, UK, April 1974.

[Flak83] Flake, P. L., Moorby, P. R., Musgrave, G., **"An Algebra for Logic Strength Simulation"**, Proc. of the 20th DAC, pp. 615-618, Miami Beach, Florida, June 27-29, 1983.

[Giam79] Giambiasi, N., Miara, A., Muriach, D., **"SILOG: A Practical Tool for Large Digital Network Simulation"**, Proc. of the 16th DAC, pp. 263-271, San Diego, California, June 25-27, 1979.

[Gloz93a] Glozić B. D., Radjenović I. J., Litovski B. V., **"Implementacija modela MOS tranzistora, modeliranog neuronskom mrežom, u hibridnom simulatoru ALECSIS"**, Zbornik XXXVII konferencije ETRAN, sveska II-E, pp. 231-236, Beograd, septembar 1993.

[Gloz93b] Glozić B. D., Litovski B. V., Mrčarica M. Ž., **"Modeling and Simulation of a Vehicle Engine Fuzzy Logic Control System in Alecsis2.0 Hybrid Simulator"**, Proc. of the 1st Balkan IFAC-type Conference on Applied Automatic Systems, Ohrid, Macedonia, 1993.

[Gloz94a] Glozić B. D., Maksimović M. D., Mrčarica M. Ž., Dimić B. Ž., Litovski B. V., **Alecsis2.0 hibridni simulator - uputstvo za upotrebu**, Laboratorija za projektovanje elektronskih kola, Elektronski fakultet, Niš, maj 1994.

[Gloz94b] Glozić B. D., **Alecsis2.1: objektno orijentisani hibridni simulator**, doktorska disertacija, Elektronski fakultet, Niš, maj 1994.

[Harr84] Harris, R. L., Davidmann, S. J., Musgrave, G., **"HILO-2 - A System to Build On"**, Proc. of the 6th International Conference and Exhibition on Computers in Design Engineering CAD84, pp. 48-60, Brighton, UK, April 3-5, 1984.

[Harr85] Harris, R. L., Omid, A. N., **"An Improved Version of the HILO Simulator"**, Electronic Engineering, pp. 159-165, September 1985.

[Holt81] Holt, D., Hutchings, D., **"A MOS/LSI Oriented Logic Simulator"**, Proc. of the 18th DAC, pp. 280-287, Nashville, Tennessee, June 29, 30, July 1, 1981.

[Hurs84] Hurst, L. S., **"Multiple-Valued Logic - Its Status and Its Future"**, IEEE Trans. on Computers, Vol. C-33, No. 12, pp. 1160-1179, December 1984.

[Hurs85] Hurst, L. S., Miller, M. D., Muzio, C. J., **Spectral Techniques in Digital Logic**, Academic Press, 1985.

[Kamb83] Kambayashi, Y., Muroga, S., **"Properties of Wired Logic"**, IEEE Trans. on Computers, Vol. C-35, No. 6, pp. 550-563, June 1983.

[Kang94] Kang, S., Szygenda, A. S., **"The Simulation Automation System (SAS); Concepts, Implementation, and Results"**, IEEE Trans. on VLSI Systems, Vol. 2, No. 1, pp. 89-99, March 1994.

[Kazm91] Kazmierski T. J., Brown A. D., Nichols K. G., Zwolinski M., **"Object-Oriented Electronic Circuit Simulation"**, Proceedings of International Conference on Concurrent Engineering & Electronic Design Automation CEEDA'91, pp. 472-474, Bournemouth International Conference Centre, UK, March 26-28, 1991.

[Kazm93] Kazmierski, T. J., Fallahi, B., **"Behavioural Modeling of Analogue Circuit Blocks"**, Research Journal, pp. 139-141, 1993.

[Kroh81] Krohn, E. H., **"Vector Coding Techniques for High Speed Digital Simulation"**, Proc. of the 18th DAC, pp. 525-529, Nashville, Tennessee, June 29, 30, July 1, 1981.

[Levi81] Levin, H., **"Enhanced Simulator Takes on Bus-Structured Logic"**, Electronic Design, pp. 153-157, October 29, 1981.

[Lips89] Lipsett R., Schaefer C., Ussery C., **VHDL: hardware description and design**, Kluwer Academic Publishers, New York, 1989.

[Lito91a] Litovski, B. V, **CADEC 1 - Analiza i optimizacija elektronskih kola**, Nauka, Beograd, 1991.

[Lito91b] Litovski, B. V, **"Logička simulacija"**, u: Litovski, B. V., urednik, **Projektovanje VLSI**, prvi deo, Nauka, Beograd, 1991.

[Maks92] Maksimović, M.D., Mrčarica, M. Ž., Glozić, B. D., Petković, M. P., Litovski, B. V., **"Arhitektura programa i strukture podataka u hibridnom simulatoru"**, Zbornik radova XXXVI Konferencije ETAN, sveska II-E, pp. 27-34, Kopaonik, 27.09-01.10. 1992.

[Maks93a] Maksimović M. D., Glozić B. D, Litovski B. V., **"Logic Simulation Modeling Techniques in Hybrid Simulator Alecsis2.0"**, Proc. of the 15-16th International Spring Seminar on Electronics Technology, pp. 133-139, Sozopol, Bulgaria, 1993.

[Maks93b] Maksimović, M.D., Milentijević, Z.I., Litovski, B.V., **"Projektovanje i simulacija cifarsko-serijskog množača"**, Zbornik radova XXXVII Konferencije ETRAN, sveska VIII-RT, pp. 221-226, Beograd, 20-23. septembar 1993.

[Maks94a] Maksimović, M. D., Glozić, B. D., **"Logička simulacija upotrebom hibridnog simulatora Alecsis 2.0"**, Zbornik radova XXXVIII Konferencije ETRAN, sveska I, pp. 55-56, Niš, 07-09. jun 1994.

[Maks94b] Maksimović, M. D., Glozić, B. D., Litovski, B. V, **"Trendovi razvoja elektronskih simulatora - Alecsis2.1 hibridni simulator"**, Zbornik radova X naučnog skupa "Čovek i radna okolina", Preventivni inženjering i informacione tehnologije PIIT'94, pp. 25-1 do 25-8, Niš, 08-10. decembar 1994.

[Maks94c] Maksimović, M. D., **"Iskustva u primeni logičkog simulatora HILO za verifikaciju projekta cifarsko-serijskog množača"**, u: Litovski, B. V., urednik, **Nove tehnologije u projektovanju integrisanih elektronskih kola**, Nauka, Beograd, 1994.

[Maks95] Maksimović, M. D., Glozić, B. D., Litovski, B. V., **"Discrete Event Modeling and Simulation in an Object-Oriented Hybrid Simulator"**, 14th IASTED Int. Conf. on Modelling, Identification and Control, biće objavljeno, Innsbruck, Austria, February 20-23, 1995.

[Menc90] Menchini, J. P., **"A Minimalist Approach to VHDL Logic Modeling"**, IEEE Design and Test of Computers, Vol. 7, No. 3, pp. 12-23, June 1990.

[Mile93a] Milentijević, Z.I., Maksimović, M.D., Stojčev, K.M., **"Cifarsko-serijski množač"**, Zbornik radova XXXVII Konferencije ETRAN-a, sveska VIII-RT, pp. 215-220, Beograd, 20-23. septembar 1993.

[Mile93b] Milentijević, I., Maksimović, D., Stojčev, M., **"Projektovanje i simulacija cifarsko-serijskog konvolvera"**, Zbornik radova konferencije TELSIKS'93, pp. 6-36 - 6-42, Niš, 07-09. oktobar 1993.

[Moon90] Moon, J. C., Sangchul, K., **"An Object-Oriented VHDL Design Environment"**, Proc. of the 27th DAC, pp. 431-435, Orlando, Florida, June 24-28, 1990.

[Mrča92] Mrčarica M. Ž., **Novi hibridni simulator elektronskih kola i programska realizacija analognog dela simulatora**, magistarska teza, Elektronski fakultet, Niš, decembar 1992.

[Mrča93] Mrčarica M. Ž, Litovski B. V., **"A New Ideal Switch Model for Time Domain Circuit Analysis"**, International Journal of Electronics, Vol. 74, No. 2, pp.241-250, February 1993.

[Mrča94] Mrčarica M. Ž, Maksimović, M. D., **"Simulacijska platforma (backplane)"**, Zbornik radova XXXVIII Konferencije ETRAN-a, sveska I, pp. 59-60, Niš, 07-09. jun 1994.

[Nage75] Nagel, L. W., **"SPICE 2: A Computer Program to Simulate Semiconductor Circuits"**, Memorandum ERL-M520, University of California, Berkley, May 1975.

[Nash80] Nash, D., Russell, IK., Silverman, P., Thiel, M., **"Functional Level Simulation at Raytheon"**, Proc. of the 17th DAC, pp. 634-639, Minneapolis, Minnesota, June 23-25, 1980.

[Nich93] Nichols, K. G., Brown, A. D., Kazmierski, T. J., Zwolinski, M., **"Event Queue Management in Logic Simulation"**, Research Journal, pp. 135-138, 1993.

[Nous93] Nousiainen, J., Nummela, A., Nurmi, J., Tenhunen, H., **"Strategies for Development and Modelling of VHDL Based Macrocell Library"**, Proc. of the European Conf. on Design Automation with the European Event in ASIC Design, pp. 478-482, Paris, France, February 20-25, 1993.

[Okaz83] Okazaki, K., Moriya, T., Yahara, T., **"A Multiple Media Delay Simulator for MOS LSI Circuits"**, Proc. of the 20th DAC, pp. 279-285, Miami Beach, Florida, June 27-29, 1983.

[Ouya94] Ouyang, C., **"Transient Analysis and Hazard-Free Design of Exclusive-OR Switching Networks"**, IEE Proc.-Comput. Digit. Tech., Vol. 141, No. 5, pp. 274-280, September 1994.

[Petk92] Petković. M. P., **"Analiza vrlo složenih elektronskih kola"**, u: grupa autora, **CADEC 2 - Simulacija i projektovanje topologije integrisanih kola**, Nauka, Beograd, 1992.

[Rowe91] Rowe, Jim, **PC-based Circuit Simulators - An Introduction**, Electronics Australia, Federal Publishing Coimpany, 1991.

[Ruan91] Ruan, G., Vlach, J., Barby, A. J., Opal, A., **"Analog Functional Simulator for Multilevel Systems"**, CAD of Integrated Circuits and Systems, Vol. 10, No. 5, pp. 565-576, May 1991.

[Sand88] Sanderson, D., P., Rose, L. L., **"Object-Oriented Modeling Using C++"**, the 21st Annual Symposium Proc., pp. 143-156, Tampa, Florida, March 16-18, 1988.

[Sasa89] Sasao, T., **"On the Optimal Design of Multiple-Valued PLA's"**, IEEE Trans. on Computers, Vol. 38, No.4, pp. 582-592, April 1989.

[Shen90] Shen, W.-Z., Jou, S.-J., Tao, Y.-S., **"EMOTA: An Event-Driven MOS Timing Simulator for VLSI Circuits"**, IEE Proceedings, Vol. 137, Pt. G, No. 4, pp. 279-290, August 1990.

[Sher81] Sherwood, W., **"A MOS Modelling Technique for 4-State True-Value Hierarchical Logic Simulation"**, Proc. of the 18th. DAC, pp. 775-785, Nashville, Tennessee, June 29, 30, July 1, 1981.

[Smit88] Smith, K.C., **"Multiple-Valued Logic: A Tutorial and Appreciation"**, Computer, Vol. 21, No. 4, pp. 17-27, April 1988.

[Stev83] Stevens, P., Arnout, G., **"BIMOS, an MOS Oriented Multi-Level Logic Simulator"**, Proc. of the 20th DAC, pp. 100-106, Miami Beach, Florida, June 27-29, 1983.

[Szyg72] Szygenda, S. A., **"TEGAS2 - Anatomy of a General Purpose Test Generation and Simulation System"**, Proc. of the 9th Design Automation Workshop, pp. 116-127. Dallas, Texas, June 1972.

[Tani94] Tanir O., Sevinc S., **"Defining Requirements for a Standard Simulation Environment"**, Computer, Vol. 27, No. 2, pp. 28-34, February 1994.

[Thel88] Thelen, D., MacDonald, J., **"Simulating Mixed Analog-Digital Circuits on a Digital Simulator"**, Proc of IEEE Int. Conf. on Computer-Aided Design ICCAD88, pp. 254-257, Santa Clara, California, November 1988.

[Toši95] Tošić, B. M., Maksimović, M. D., **"Asinhroni pristup projektovanju specijalizovanih arhitektura za obradu signala"**, prihvaćeno za izlaganje na XXXIX konferenciji ETRAN koja će se održati na Zlatiboru, 6-9 juna 1995.

[Ulri74] Ulrich, E. G., Baker, T., **"Concurrent Simulation of Nearly Identical Digital Networks"**, Computer, Vol. 7, No. 4, pp. 39-44, April 1974.

[Ulri80] Ulrich, E. G., **"Table Lookup Techniques for Fast and Flexible Digital Logic Simulation"**, Proc. of the 17th DAC, pp. 560-563, Minneapolis, Minnesota, June 23-25, 1980.

[Vasi95] Vasić, A., **"Simulacija računarskog diska pomoću hibridnog simulatora ALECSIS 2.1"**, diplomski rad, Univerzitet u Nišu, Elektronski fakultet, april 1995.

[Wilc79] Wilcox, P., **"Digital Logic Simulation at the Gate and Functional Level"**, Proc. of the 16th DAC, pp. 242-248, San Diego, California, June 25-27, 1979.

[Zwol91] Zwolinski, M., Brown, A. D., Kazmierski, T. J., Nichols, K. G., **"A Hardware Description Language for Mixed-Mode Simulation"**, Proc. of Int. Conf. on Concurrent Engineering & Electronic Design Automation CEEDA'91, pp. 211-214, Bournemouth International Conference Centre, UK, March 26-28, 1991.

[-90] **IEEE Standard 1076, VHDL, Tutorial,** CAD Language Systems, Inc., Rockville, March 1990.

[-91] **PSpice Circuit Analysis, Version 4.05**, MicroSim Corporation, Irvine, California, January 1991.