KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS
PROGRAMŲ INŽINERIJOS KATEDRA

Žygimantas Dirsė

Verilog kalbos sintezuojamų konstrukcijų atvaizdavimas *SystemC* kalboje

Magistro darbas

Darbo vadovas
Doc. V. Jusas

Kaunas, 2005

KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS
PROGRAMŲ INŽINERIJOS KATEDRA

TVIRTINU
Katedros vedėjas
doc. E. Bareiša

Verilog kalbos sintezuojamų konstrukcijų atvaizdavimas *SystemC* kalboje

Informatikos mokslo magistro baigiamasis darbas

Kalbos konsultantė
Lietuvių kalbos katedros lekt.
dr. J. Mikelionienė

Vadovas
doc. V. Jusas

Recenzentas
doc. A. Lenkevičius

Atliko
IFM 9/5 gr. stud.
Ž. Dirsė

KAUNAS, 2005

SUMMARY

This master work of the main subject: Developing and analysis of peripheral serial interface microcontroller RISC8 that is similar to PIC 16C57, goes about all phases of developing microcircuits.

There are analyzed ways of the developing system on chip, described hardware description languages.

The first phase is to develop and model a code with one of HDL (hardware description languages) like a Verilog. For that reason are used such developing tools as Cadence LDV-5.1, that is used for compiling, elaborating and simulating of the design with graphical interface. The second phase - synthesis is done using products of Synopsys Company such a design analyzer.

All phases presented in the manner like a sources, all processes, what have to be done, are described and shown with tables, pictures and other graphical tools. All results described in the same manner. All files presented electronically in compact disc, like source files, .log files, databases and other results.

The last chapter of the work describes the result of synthesis in the manner of synthesis constructs comparing two languages: Verilog and SystemC.

TURINYS

ĮŽANGA 7
1. PROJEKTO UŽDUOTIS 8
2. PROJEKTAVIMO PROCESAS IR METODIKOS 9
2.1. Projektavimo procesas 10
2.1.1. Kanoninis modelis 10
2.1.2. Sistemos projektavimo eiga 11
2.2. Projektavimo priemonės 15
2.3. Projektavimo kalbos 16
2.3.1. VHDL kalba 16
2.3.2. *Verilog* kalba 18
2.3.3. *SystemC* kalba 21
3. *RISC 8* MIKROVALDIKLIO KŪRIMAS IR ANALIZĖ 29
3.1. Pradinė specifikacija 29
3.2. Sintezuojama specifikacija 35
3.2.1. RISC8 mikrovaldiklio modeliavimas 45
3.2.2. Mikrovaldiklio modulių sintezė 48
4. *VERILOG* KALBOS SINTEZUOJAMOS KONSTRUKCIJOS *SYSTEMC* KALBOJE 52
4.1. Skirtumai tarp modeliavimo ir sintezės. 52
4.2. Sintezuojamų konstrukcijų ypatumai. 52
4.3. Sintezuojamų konstrukcijų dviprasmiškumai. 57
4.4. Nesintezuojamos *Verilog* bei *SystemC* kalbų konstrukcijos 58
5. IŠVADOS 62
LITERATŪRA 63
1 PRIEDAS. Kompaktinis diskas. 64
2 PRIEDAS. Synopsys programos skriptai 64

Lentelių sąrašas

1 lentelė. Specifikuojami projekto parametrai 29
2 lentelė. Paketo struktūra 35
3 lentelė. Suderinamumas su 16C57 mikrovaldikliu 37
4 lentelė. ALĮ operandai 39
5 lentelė.KD kontroliniai signalai komandoms 40
6 lentelė.RF loginiai adresų ryšiai 43
7 lentelė. Išplėtimo grandinės signalai 44
8 lentelė.Class bibliotekoje sintezuotų modulių parametrai 49
9 lentelė.*Tc6a_cbacore* bibliotekoje sintezuotų modulių parametrai 49
10 lentelė.Nesitezuojamos *Verilog* kalbos kosntrukcijos 59
11 lentelė.Nesitezuojamos *SystemC* kalbos konstrukcijos 60

Paveikslėlių sąrašas

1 pav. Kanoninis vienlusčių sistemų modelis 10
2 pav. Tradicinė užsakomųjų lustų (*ASIC*) diagrama 12
3 pav. Spiralinis vienlusčių sistemų projektavimo modelis 14
4 pav. *Synopsys design analyzer* programos vykdymo eiga ir galimybės 16
5 pav. *VHDL* kalba aprašomos struktūros pavyzdys 17
6 pav. Modulio paskelbimas 17
7 pav. Modulio architektūros aprašas 18
8 pav. *Verilog* kalba aprašytas *flip-flop* tipo trigeris. 20
9 pav. *SystemC* projektavimo aplinka 22
10 pav. Tipinis *SystemC* projektavimo procesas. 24
11 pav. *SystemC* kalbos architektūra 25
12 pav. *SystemC* sudaryta iš komplekso modulių 26
13 pav. *SystemC* kalba aprašytų modulių sintezės eiga 28
14 pav. Centrinio procesorinio elemento(CPE) modulis specifikacijai 30
15 pav. Mikrovaldiklio schema 31

16 pav. CPE schema 31
17 pav. Aritmetinio loginio įtaiso modulis specifikacijai 32
18 pav. Komandų dekoderio modulis specifikacijai 33
19 pav. Registrų failo modulis specifikacijai 33
20 pav. Programų atminties modulis specifikacijai 34
21 pav. Programų atminties modulis specifikacijai 34
22 pav. Architektūra 36
23 pav. Modeliavimo rezultatų atvaizdavimas banginėmis diagramomis 47
24 pav. ALĮ modulio atvaizdavimas programinės įrangos *desing_analyzer* pagalba aukščiausiame lygmenyje 50
25 pav. ALĮ atvaizdavimas ventilių lygmenyje *class* bibliotekoje, realizuojant operacijų resursų dalinimąsi. 51
26 pav. Realizuoto kodo sintezė šešiais registrais 54
27 pav. Trijų būsenų buferis 56

ĮŽANGA

Viena iš būdingiausių šiandieninės pažangos ypatybių yra platus mikroelektronikos gaminių naudojimas įvairiose ūkio srityse. Jau daugelį metų „kritinių technologijų“ sąraše elektronika tradiciškai užima pirmąją vietą kaip turinti didžiausią įtaka šalies ekonominiam ir gynybiniam potencialui komponentė.

Kiekvieną kartą pasikartojant Moor‘o dėsniui, mikroelektronikos pasaulyje vyksta vis didesni pasikeitimai ir yra didelė tikimybė, kad jau greitai gali atsitikti kaip telekomunikacijų rinkoje – technologijos kuriamos ir diegiamos daug kartų sparčiau, nei plečiasi rinka ir poreikiai. Pirmas pavyzdys būtų Intel ir AMD procesorių dvikova.

Mikrograndynų gamyba panaudojant mikroelektronikos technologijas litografiją, garinimą, ėsdinimą ir kitas apsimoka tik gaminant didelius kiekius lustų. Dėl to, kad ant vienos silicio plokštelės telpa labai daug nesudėtingų mikrograndynų, brangūs ir sudėtingi įrengimai, eksploatacinės medžiagos ir kita. Sparčiai besiplečiančioms sritims kaip mikrosistemos ar mechatronika, norint gaminti mažomis serijomis mikrograndynus, reikalingi nauji sprendimai.

Naudojant programuojamos logikos matricas, užsakomuosius lustus (*PLD, PPLD, FPGA, ASIC*) kiekviena technologija turi savo specifiką ir privalumus. Panaudojant *HDL* (*Hardware description language*) resursus kaip programavimo kalbas *VHDL, Verilog* ar *SystemC,* aprašant matricos konstrukciją galima gauti labai specifinius ir pageidaujamus rezultatus per labai trumpą laiką ir nenaudojant didelių resursų. Panaudojant *Cadence* firmos *Nclaunch* programą modeliavimo rezultatų analizei, atliekant sintezę *Synopsys* firmos produktais panaudojant bibliotekas, keičiant elementus, optimizuojant konstrukciją galima pasiekti gerų rezultatų. Taip pat yra aibė kitų gamintojų programų.

Taigi į šiuos lustus galima integruoti norimas konstrukcijas naudojant įvairias automatizuotas projektavimo priemones, kas suteikia konkurentabilumą bei projektavimo įvairiapusiškumą, suteikiant užsakovui galimybę pateikti, kad ir keletą lustų.

Šiame darbe analizuojami galimi vielusčių sistemų projektavimo metodai, aprašomos aparatūrinės kalbos. Nagrinėjamos pagrindinės mikrograndynų automatizuoto projektavimo fazės projektuojant mikrovaldiklį *RISC8* analogišką *PIC 16C57*. Programų aprašymai pateikti *Verilog* kalba. Modeliavimas vykdomas panaudojant *Cadence* firmos *Nclaunch* programą. Kaip sintezės priemonės panaudojamos firmos *Synopsys* produktas – *design analyzer*. Kiekviena projekto fazė pateikiama su šaltiniais, darbo eiga ir gautais rezultatais bei pavaizduotomis iliustracijomis.

Pasinaudojant eksperimento rezultatais, aprašomos sintezuojamos konstrukcijos *Verilog* ir *SystemC* kalbose.

1. PROJEKTO UŽDUOTIS

Verilog kalbos sintezuojamų konstrukcijų atvaizdavimas *SystemC* kalboje projektuojant RISC8 mikrovaldiklį.

Aparatūrinėmis kalbomis aprašyti moduliai kompiliuojami, testuojami bei sintezuojami *Candence* ir *Synopsys* kompanijų įrankiais.

2. PROJEKTAVIMO PROCESAS IR METODIKOS

Silicio technologijos leidžia gaminti lustus, sudarytus iš dešimčių milijonų tranzistorių. Šios technologijos nepaliaujamai plinta ir tobulėja, tačiau lygiai taip pat projektuotojams iškelia naujus iššūkius. Kaip šio proceso pasekmė atsiranda poreikis analizuoti ir ieškoti panašių sprendimų jau sukurtose technologijose norint juos atkartoti.

Vienlustės sistemos (*Systems-on-chip*) leidžia panaudoti keletą efektyvių projektavimo variantų sutaupant laiko ir energijos.

Pagal [4] galime išskirti šias integrinių mikroschemų kūrimo metodologijas:

- viskas daroma nuo pradžios iki galo;
- viskas daroma nuo pradžios iki galo struktūriškai;
- užsakomieji lustai *ASIC;*
- programuojami loginiai lustai (*FPGA*);
- vienlustės sistemos (*SoC*).

Taip pat galime išskirti programuojamus įrenginius, kuriuos galima naudoti kaip galutinius produktus:

- PLM – programuojamos loginės matricos (*PLA, SoP*);
- PLĮ – programuojami loginiai įtaisai – tai PLM su atmintimi *FSM (final state machine)* būsenai saugoti;
- EPLĮ – elektriškai programuojami loginiai įtaisai (*EPLD*);
- programuojami loginiai lustai (*FPGA*).

Pagal vyraujančias tendencijas, galime išskirti šiuos prioritetus *SoC* rinkoje:

- mažesnė projektavimo kaina ir trumpesnis produkto pateikimo laikas į rinką;
    - kuo didesnis panaudojimas atkartojimo technologijų, atkartojamų komponentų;
- dizaino specifikacijos kuo aukštesniame abstrakcijos lygmenyje;
- mišrių signalų integriniai grandynai (analoginiai, skaitmeniniai);
- lanksčių architektūrų panaudojimas per funkcinius blokus bei programavimo lankstumą.

2.1. Projektavimo procesas

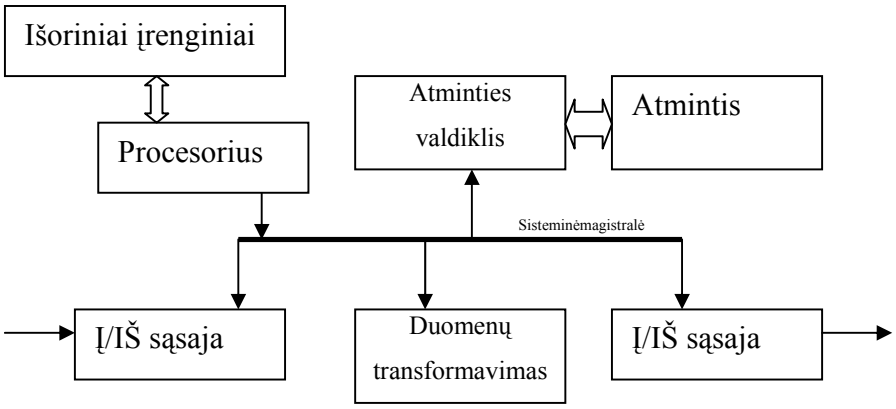
2.1.1. Kanoninis modelis

Bendrinis vienlusčių sistemų modelis parodytas 1 pav. Jis susideda iš:

- mikroprocesoriaus ir jo atminties sistemos;
- duomenų magistralės, kurioje yra sąsaja su išorinėmis sistemomis;
- transformacijos blokai, kurie transformuoja gautus išorinius duomenis į sistemos formatą;
- kitos sąsajos į išorinę sistemą.

Šis modelis apima daugumą struktūrų realiose vienlustėse sistemose. Procesoriumi gali būti bet kas, nuo 8 bitų procesoriaus iki 64 bitų sumažinto komandų skaičiaus (*RISC*) architektūros mikrovaldiklio. Atminties posistemė gali būti vieno arba kelių lygmenų, statinio (*SRAM*) arba dinaminio (*DRAM*) tipo. Komunikaciją galima realizuoti per išorinių įrenginių sąsajas: nuoseklią sąsają *PCI*, *USB*, analogas-kodas, kodas-analogas keitiklius, vidinį tinklą (*Ethernet*), elektromechaninius ar elektrooptinius keitiklius. Duomenų transformacijos blokas gali būti grafinis procesorius arba tinklo maršrutizatorius. Projektavimo procesas reikalauja specifikuoti sistemą, sukurti ir patikrinti blokus, surinkti juos į vieną lustą, kuriame yra visi pagrindiniai vienlustės sistemos elementai.

Realios konstrukcijos yra kur kas sudėtingesnės nei šis kanoninis modelis. Reali konstrukcija apima keletą komponentų sąsajų, duomenų transformatorių. Taip pat pasitaiko daug atvejų, kai realizuojama keletas procesorių, procesorių kombinacijų ir signalus apdorojančių procesorių. Vienlusčių sistemų atmintys taip pat yra pakankamai sudėtingos su įvairiomis buferių ir dalomomis atmintimis, taip pat specialiais duomenų transformacijos blokais. Taigi, šis kanoninis modelis yra tik miniatiūrinė vienlustės sistemos versija.



1 pav. Kanoninis vienlusčių sistemų modelis

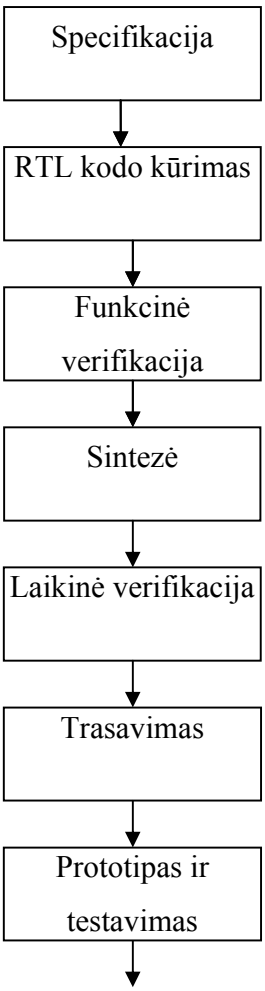
2.1.2. Sistemos projektavimo eiga

Projektuojant vienlustes sistemas pasirenkami du pagrindiniai būdai:

- nuo krioklio modelio į spiralinį;
- nuo viršus-į-apačią metodologijos į kombinacinę viršus-į-apačią arba apačia-į-viršų.

Tradicinis užsakomųjų lustų modelis, pavaizduotas 2 pav., vadinamas *krioklio* modeliu. Krioklio modelyje projektas vystomas žingsniais, nuo fazės prie fazės, niekada negrįžtant jau prie įvykusių fazių. Šis procesas prasideda nuo specifikacijos kūrimo. Sudėtingiems užsakomiesiems lustams su aukšto lygio algoritminiu kontekstu, kokį turi grafikos apdorojimo lustai, algoritmus kuria specializuoti ekspertai būtent grafinių lustų srityje. Po funkcinio verifikavimo projektas sintezuojamas į loginių ventilių lygmenį. Laikinė verifikacija atliekama norint įsitikinti ar konstrukcija atitinka užduotas laikines charakteristikas. Loginių ventilių lygmens projektas trasuojamas atlikus laikinę verifikaciją. Galiausiai lustas pagaminamas ir testuojamas su atitinkamomis programomis. Daugeliu atveju programinės įrangos kūrimas prasideda iškart po aparatūros kūrimo pradžios. Be aparatūros modelio derinimo, programinės įrangos modelio kūrimo procesas gali vykti šiek tiek greičiau, kol pristatomas prototipas. Taigi, šie du lygiagretūs procesai yra betarpiškai susiję.

Šitoks projektavimo tipas tinka tik konstrukcijoms turinčioms ne daugiau kaip 100000 ventilių bei pagamintoms pagal 0.5 μm technologiją, kadangi didesnės sistemos yra per daug sudėtingos. Didelėms, mikronų dalių konstrukcijoms krioklio metodas nėra tinkamas dėl to, kad fizinės ir kitos charakteristikos turi būti aptartos ankstyvojoje proceso stadijoje norint pasiekti užsibrėžtų rezultatų.



2 pav. Tradicinė užsakomųjų lustų (*ASIC*) diagrama

Geometrija „traukiasi“ projektui sudėtingėjant, trumpėja produkto pateikimo į rinką laikas. Projektavimo procesui taip pat naudojamas *spiralinis* metodas. Šiame modelyje visi projekto etapai yra tarpusavyje susiję.

Trečiame paveiksle parodytas spiralinis modelis, kur:

1. Projektuojama lygiagrečiai tiek aparatūra, tiek programinė įranga.
2. Lygiagrečiai sintezuojama ir atliekama verifikacija.
3. Išdėstymas ir trasavimas atliekamas sintezės metu.
4. Moduliai kuriami tik tada, jei nenaudojami jau sukurti.
5. Suplanuotas pilnas iteracijų procesas.

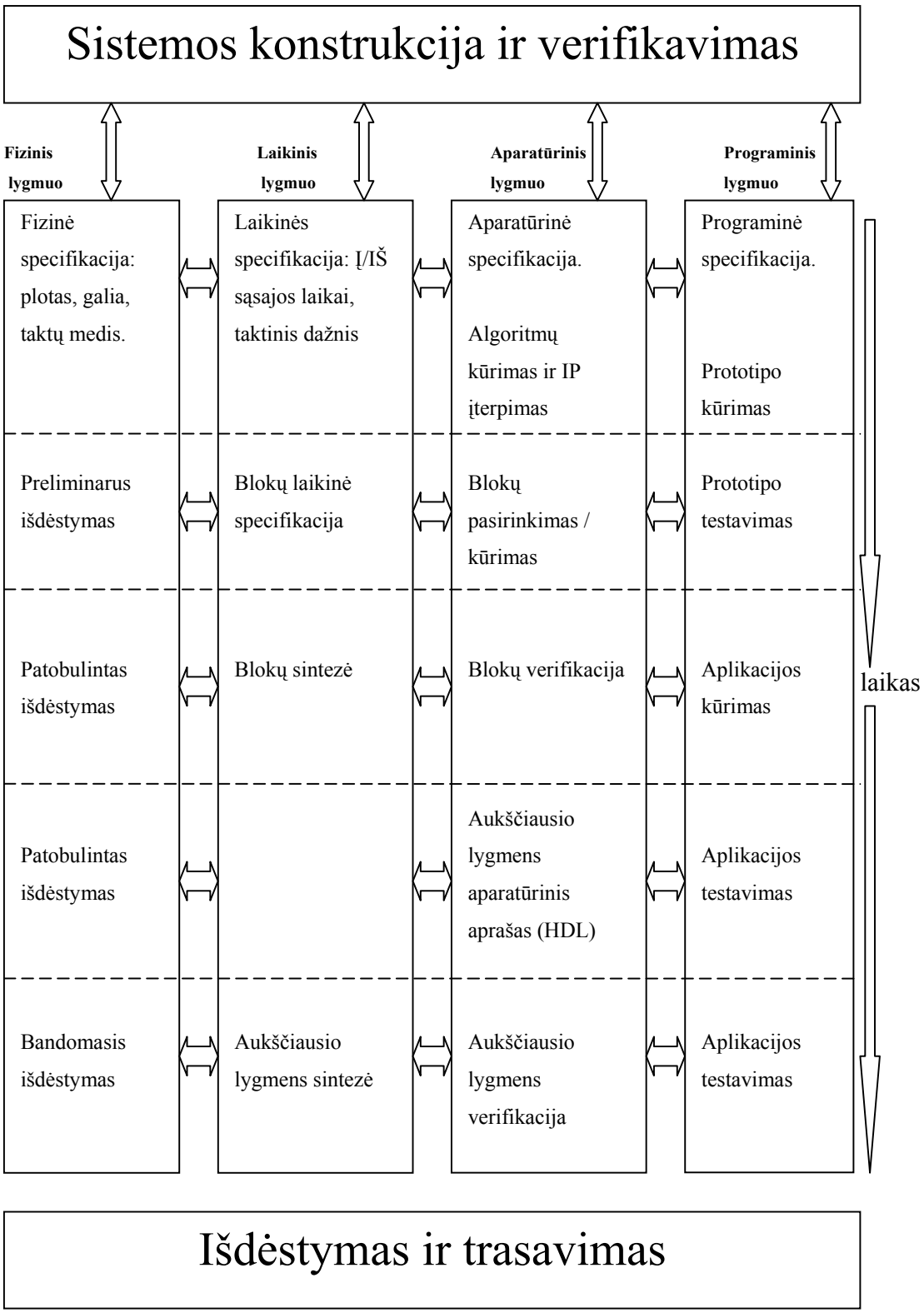
Daugumoje projektų, inžinieriai tuo pačiu metu kuria aukščiausio lygmens (*TOP level*) sistemų specifikacijas, algoritmus kritiniams blokams, sistemos lygio verifikacijos rinkinius, bei laikinius parametrus galutiniam lusto testavimui. Tai reiškia, kad paimami visi aparatūrinio lygmens ir programinės įrangos parametrai: funkcionalumas, laikinės charakteristikos, fizinė konstrukcija ir verifikavimas.

Kaip teigiama [1], klasikinis viršus-į-apačią procesas prasideda specifikacijomis ir baigiasi integracija bei verifikavimu:

1. Sudaryti pilną projektuojamos sistemos ar posistemės specifikaciją.
2. Nustatyti architektūrą, algoritmus, įskaitant programų bei aparatūros kūrimą, ir, jei reikalinga, bendrą testavimą.
3. Suskaidyti architektūrą į aiškiai apibrėžtus modulius.
4. Sukurti arba pasirinkti modulius.
5. Integruoti žemesnio lygio sukurtus arba pasirinktus modulius į aukščiausią lygmenį; verifikuoti laikines charakteristikas bei funkcionalumą.
6. Perduoti sistemą aukštesniam projekto lygmeniui, trasavimui bei išdėstymui.
7. Patikrinti visus konstrukcijos lygmenis bei aspektus (funkcionalumą, laikines charakteristikas).

Stengiantis kuo greičiau pateikti rinkai produktus ieškoma būdų kaip tai padaryti. Pakankamai galingi sintezės ir emuliacijos įrankiai duoda atitinkamų rezultatų. Taip pat kuriant atkartojamų kodų bibliotekas stipriai paspartinamas procesas.

Tačiau kaip ir krioklio modelio atveju viršus-į-apačią modelis yra idealus variantas, kurį mes galime pasiekti. Šios metodologijos procese žemiausio lygmens specifikuoti blokai, galima sakyti, gali būti iš karto pagaminti, tačiau kažką keičiant ar išimant šiuos blokus, tenka perdaryti visą projektą. Todėl realiai projektuotojai naudoja mišrų modelį, sudarytą iš viršus-į-apačią bei iš apačios-į-viršų metodų, konstruojant kritinius žemiausio lygmens modelius.



3 pav. Spiralinis vienlusčių sistemų projektavimo modelis

2.2. Projektavimo priemonės

Projektavimui galima naudoti standartais patvirtintas *VHDL*, *Verilog* arba aukštesnio lygio *SystemC*, *HandelC* kalbas, taip pat nemažai kitų. Projektavimo įrankių gausoje galima išskirti keletą grupių:

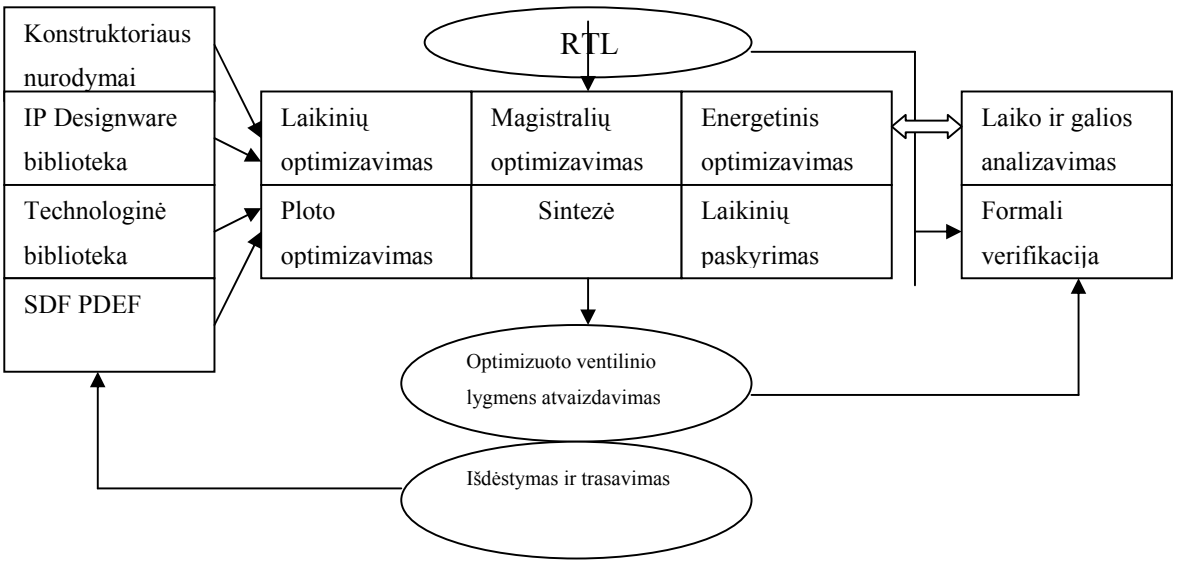
- modeliavimui skirti įrankiai (*Cadence LDV-5.0, Aldec Riviera ir t.t.*);
- sintezei skirti įrankiai (*Synopsys design analyzer, design vision, Aldec Simplify ir kiti*);
- išdėstymui ir trasavimui skirti įrankiai (*Cadence IC, Xilinx Floorplaner ir t.t.*);
- įrankiai, apimantys visas minėtas sritis (*Xilinx ISE 5.1, Activ Aldec 5.1*).

Pagrindinės projektavimo priemonės sukurtos *Cadence* ir *Synopsys* firmų. Modeliavimui naudota programa *Cadence LDV-5.1*. Tai *VHDL* ir *Verilog* kalba aprašytus failus modeliuojanti programa. Nurodytą kodą programa sukompiliuoja, detalizuoja t.y. patikrina sintaksės ir semantines klaidas, nustato bei susieja komponentus ir modulius, modulių funkcionalumą. Po to, jei parašyta testinė programa, galima modulį ar komponentą testuoti užduodant į įėjimus norimas signalų reikšmes.

Nors ši programa yra pakankamai “griežta” klaidoms, kurias kartais sunku surasti, tačiau pats modeliavimo procesas yra kokybiškesnis. Lyginant su *Aldec Riviera* ar *Aldec Activ*.

Sintezei buvo naudojami *Synopsys* kompanijos sintezės įrankiai *design vision*, *design analyzer*.

Design analyzer – tai galingas sintezės įrankis, kuris suteikia galimybę valdyti sintezės procesą, konstrukciją bei analizuoti dizainą grafinėje aplinkoje. Programa leidžia atlikti įvairius užstatymus, analizės funkcijas, besimokančias sintezės funkcijas, taip pat interaktyvią aplinką su principine schema.



4 pav. *Synopsys design analyzer* programos vykdymo eiga ir galimybės

Šiuos produktus naudoja praktiškai apie 90% *ASIC* projektuotojų.

Kalbant apie *design vision* reikia paminėti, kad ši programa kiek sunkiau įsisavinama bei turi mažiau galimybių formuojant ataskaitas.

2.3. Projektavimo kalbos

Aparatūros aprašymo kalbos (*HDL*) labai panašios į įprastas ir daugelis jų yra sukurtos įprastų kalbų pagrindu. Pvz.: *Verilog* ir *SystemC* yra sukurtos *C++* kalbos pagrindu. Pagrindinis skirtumas yra tas, kad šiose kalbose aprašomi tam tikri struktūriniai moduliai, kurie gali turėti signalus. Procesai gali vykti lygiagrečiai.

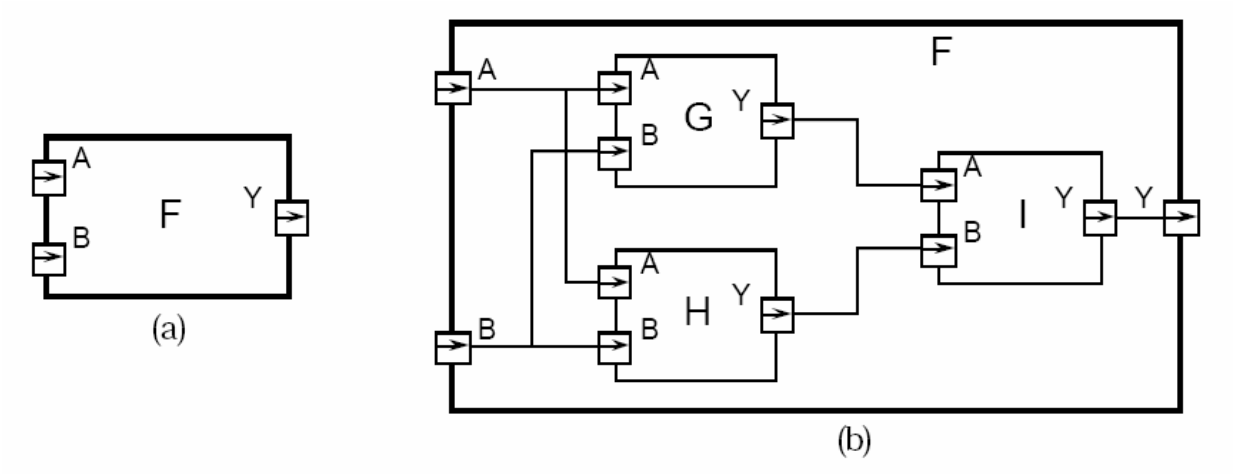
Kaip standartas pati pirmoji JAV karinių pajėgų buvo sukurta *VHDL* kalba.

2.3.1. VHDL kalba

VHDL kalba – tai skaitmeninių sistemų aprašymo kalba. Ji atsirado 1980 metais pagal JAV karinio departamento programą *„Very High Speed Integrated Circuits (VHSIC)“*. Sukūrus šią kalbą, Tarptautinis elektronikos ir elektrotechnikos institutas patvirtino ją kaip standartą.

VHDL kalba leidžia aprašyti konstrukcijos struktūrą, kaip sistema suskirstyta į atskirus modulius. Taip ši kalba leidžia funkciškai aprašyti šiuos modulius. Ir galiausiai šia kalba imituoti parašytos programos veikimą, taigi taip iš karto galima pataisyti nereikalingas klaidas ir sutaupyti daug brangaus laiko.

Skaitmeninė sistema gali būti aprašyta kaip modulis su įėjimais ir išėjimais. Elektrinės išėjimų savybės – tai tam tikros įėjimų reikšmių funkcijos.



5 pav. *VHDL* kalba aprašomos struktūros pavyzdys

Ketvirtame paveiksle a dalyje pavaizduotas skaitmeninės sistemos supratimo pavyzdys. Modulis F turi du įėjimus, A ir B, ir išėjimą Y. Pagal *VHDL* terminologiją tai būtų: F - konstrukcijos modulis *entity,* o įėjimai ir išėjimai vadinami prievadais. Vienas iš dviejų būdų funkcionalumo aprašymo yra parodymas, kaip jis sudarytas. Kiekvienas iš žemesnio lygmens modulių yra *komponentas*, kuris priklauso kažkuriam moduliui, prievadai yra sujungti *signalais*. Paveikslo b dalyje parodyta kaip F modulis gali būti sudarytas iš komponentų G, H ir I. Šio tipo aprašymas vadinamas *struktūriniu*.

```
entity count2 is
    generic (prop_delay : Time := 10 ns);
    port (clock : in bit;
          q1, q0 : out bit);
end count2;
```

6 pav. Modulio paskelbimas

Daugeliu atveju nėra racionalu aprašinėti vien tik struktūriškai. Galima atskirame modulyje aprašyti jo veikimą elgsenos lygmenyje. Pavydžiui, loginė funkcija.

Kūriant testavimui programą paprasčiausiai į modulio įėjimo prievadus įjungiami testinės programos modulio prievadai, tas pats padaroma ir su išėjimais. Taigi testinėje programoje paduodame signalus su laikinėmis pasirodymo reikšmėmis bei vykdant testuojamo modulio veikimą galima naudojant tam tikrą programinę įrangą matyti reakcijas į mūsų paduodamus signalus.

Modulio funkcionalumas aprašomas programos dalyje, kuri vadinasi *architektūra.* Modulis gali turėti keletą architektūrų, bet veikti galima tik pagal vieną. Kuri architektūra naudojama, nurodo *konfigūracijos* sakinys.

```
architecture behaviour of count2 is
begin
    count_up: process (clock)
        variable count_value : natural := 0;
    begin
        if clock = '1' then
            count_value := (count_value + 1) mod 4;
            q0 <= bit'val(count_value mod 2) after prop_delay;
            q1 <= bit'val(count_value / 2) after prop_delay;
        end if;
    end process count_up;
end behaviour;
```

7 pav. Modulio architektūros aprašas

Aišku, *VHDL* kalba turi savo sintaksę ir semantiką, su kuria artimiau susipažinti galima [2].

2.3.2. *Verilog* kalba

Verilog kalba buvo pradėta naudoti *Gateway Design Automation* kompanijoje 1984 metais, perimant struktūą iš *HiLo* ir *C* kalbų, bet oficialiai ši kalba buvo pirpažinta 1995 metais. Ši kalba daugiausiai sukurta *C++* kalbos pagrindu, kaip *VHDL* gali būti tiek struktūrinė, tiek elgsenos.

Verilog kalba galima kurti modulius įvairiuose abstrakcijos lygmenyse:

- **algoritminis** modelis, kuriame realizuojamos aukšto lygio kalbų konstrukcijos. Lygiagrečiai vykdomi nuoseklių konstrukcijų veiksmai;
- **TPL (Tarpregistrinis perdavimo lygmuo)** modelis, kuris aprašo duomenų perdavimą tarp registrų ir duomenų apdorojimą. Naudojamas išorinis taktinio dažnio genratorius (TDG);
- **ventilių modelis,** kuris aprašo loginius ventilius ir sąsajas konstrukcijoje tarp jų;
- **perjungimo lygmuo,** kuriame aprašomi tranzistorių ir atminties elementų mazgai ir sąsajos tarp jų.

Elgsenos specifikacija apibūdina skaitmeninės sistemos (modulio) elgseną naudojant įprastas programavimo kalbos kosntrukcijas, pvz.: *is, case*.

Struktūrinė specifikacija apibūdina skaitmeninės sistemos (modulio) elgseną kaip hierarchinę sistemą tarp mažesniojo lygmens modulių. Žemiausiame lygmenyje komponentai turi būti primityvūs arba specifikuota jų elgsena.

Pagrindinis sudaromasis blokas kaip ir *VHDL* kalbos atveju yra modulis. Kiekvienas modulis turi sąsają su kitais moduliais. Moduliai gali veikti lygiagrečiai, bet įprastai būna tik vienas aukščiausiojo lygmens modulis, kuriame paskelbiami visi sudarantieji sistemą. Moduliai gali atvaizduoti nuo paprasčiausių techninės įrangos dalių iki pilnų sistemų, pvz.: mikroprocesorių.

Čia taip pat įmanoma panaudoti metodologijas apačia-į-viršų bei viršus-į-apačią. Modulis gali būti visa konstrukcija arba tos konstrukcijos dalis. Modulyje taip gali būti paskelbti kiti moduliai.

Verilog elgsenos lygmuo suskirstytas ir veikia lygiai taip pat kaip *C* programavimo kalba.

Elgsenos lygmens struktūros algoritminiai bei TPL modeliai:

- struktūrinės procedūros nuosekliam arba lygiagrečiam vykdymui;
- aiški procedūrų kontrolė aktyvacijai, specifikuotai pagal vėlinimus arba reikšmių pasikeitimo įvykius;
- kontroliuojami įvykiai aktyvuojant arba panaikinant tam tikrus veiksmus procedūrose;
- sąlyginiai sakiniai, ciklai;
- procedūros, funkcijos;
- aritmetinės, loginės, bitų operacijos.

Struktūrinė *Verilog* kalbos dalis skirta ventilių ir perjungimo modeliams bei palaiko:

- kombinatorinę logiką;
- abipusį perdavimą;
- galimybę modeliuoti *MOS* technologijos modelius su apkrovimo paskirstymu ir mažinimu.

Assign sakinys naudojamas modeliuoti kombinacinei logikai, o *initial* ar *always* naudojami nuosekliai logikai aprašyti, pvz.: būsenų automatui. Svarbu nesumaišyti paskirties, kadangi procedūriniai priskyrimai *always* konstrukcijoje keičia registrų būklę. *Assign* sakinyje vykdomi pastovūs priskyrimai.

Kadangi *Verilog* kalbos paskirtis yra modeliuoti skaitmenines sistemas, tai pirminiai duomenų tipai yra registrai (*reg*) ir signalai (wire). *Reg* tipo kintamieji saugo paskutinę reikšmę, kuri buvo procedūriškai priskirta, o *wire* nesaugo. *Wire* – tai tiesiog jungiantis laidas, tačiau turintis keletą patogių „mazgų“ tipo modifikacijų: *wand (wire* ir*), wor (wire* arba*), tri (*trijų būsenų buferis*)*.

Verilog kalba palaiko šias galimas reikšmes:

- 0 – loginis nulis arba klaidinga reikšmė;
- 1 – loginis vienetas arba teisinga reikšmė;
- x – nežinoma loginė reikšmė;
- z – aukštas impedansas arba trijų būsenų ventilis.

Kintamasis *reg* gauna *x* reikšmę paskelbiant jį pirmą kartą, o *wire* gauna *x* reikšmę, kai nėra prijungtas. Šie kintamieji gali būti ir vektorinio tipo. Taip pat palaikomi *integer, real, time* tipo kintamieji.

Atmintys specifikuojamos kaip registrai.

Atminties specifikavimo pavyzdys:

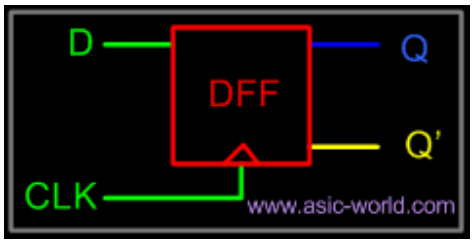
```
reg [31:0] Mem [0:1023];
```

Verilog kalba parašytos programos pavyzdys:

```
// D flip-flop aprašas
module d_ff ( d, clk, q, q_bar);
input d ,clk;
output q, q_bar;
wire d ,clk;
reg q, q_bar;

always @ (posedge clk)
begin
  q <= d;
  q_bar <= !d;
end

endmodule
```



8 pav. *Verilog* kalba aprašytas *flip-flop* tipo trigeris.

Skirtingai nuo *VHDL*, čia didžiosios ir mažosios raidės neskiriamos, todėl yra žymiai patogiau aprašinėti vardus.

Verilog aparatūros aprašymo kalba turi didelius kontrolinių kontrukcijų resursus, kurios dažniausiai naudojamos *always* arba *initial* blokuose. Dauguma jų yra labai panašios į *C* kalbos konstrukcijas. Pagrindinis skirtumas yra tas, kad vietoje *C* kalbos skliaustelių **{},** *Verilog* kalboje naudojama *begin* ir *end.*

Skirtingai nuo *C* kalbos čia nereikalinga *break* konstrukcija *case* sakinyje:

```
case (sig)
      1'bz: $display("Signalas neaiškus");
      1'bx: $display("Signal nežinomas");
      default: $display("Signalas yra %b", sig);
    endcase
```

Ciklų automatiniai didinimo mažinimo operatoriai *Verilog* kalboje nenaudojami.

Verilog kalboje naudojami keleto tipų priskyrimai:

- blokuojantys = ;
- neblokuojantys <= .

Blokuojantis priskyrimas veikia panašiai kaip ir *C* ar kitoje tradicinėje kalboje. Atliekamas pilnas priskyrimas, kol pereinama prie kito priskyrimo sakinio. Neblokuojantis priskyrimas realizuoja lygiagretų priskyrimą t.y. pirma realizuojamos visos dešinės pusės reikšmės, po to kairės.

Verilog funkcijos ir procedūros (*task*) veikia panašiai kaip ir *C* išskyrus:

- *Verilog* funkcijos vykdomos per vieną simuliacinį laiko vienetą, t.y. jokios laiką kontroliuojančios konstrukcijos #, @, *ar wait* neleidžiamos.

2.3.3. *SystemC* kalba

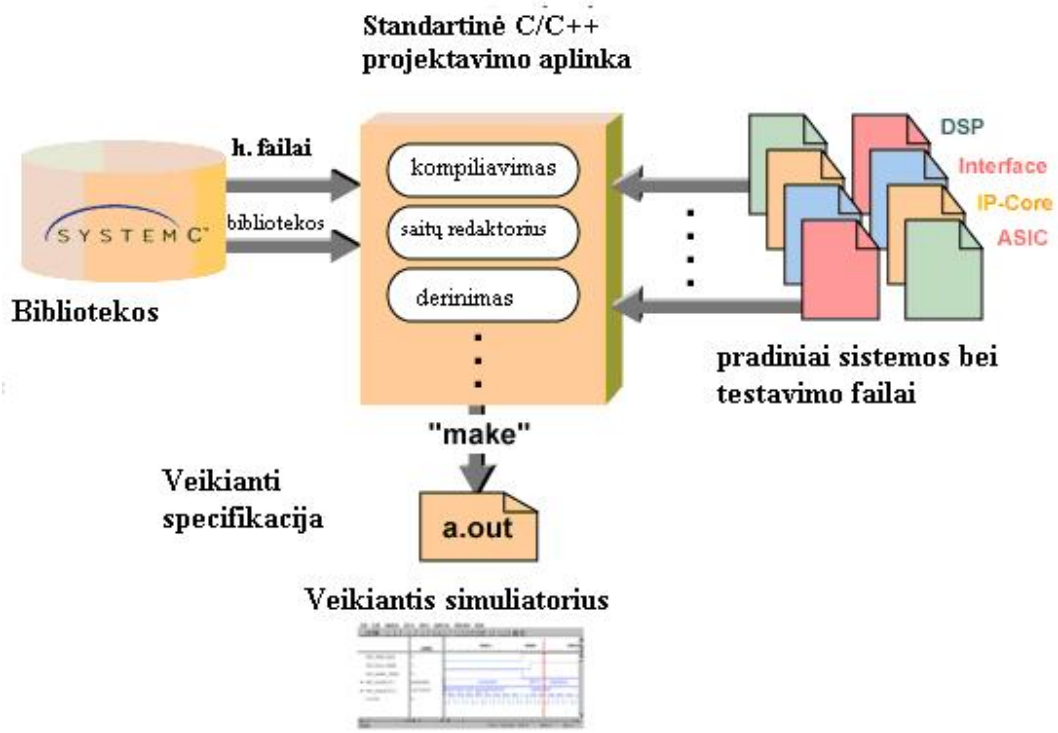
1999 m. rugsėjo mėnesį buvo išleista pirmoji *SystemC* kalbos versija. Jos vystymusi bei tobulinimu rūpinosi *OSCI(Open SystemC Initiative)* organizacija, taip pat prie prisidėjo ir *Synopsys, Frontier Design, CoWare*.

Kai *C++* kalba yra tinkama sistemos programinei įrangai modeliuoti, yra mažiausiai trys priežastys, kodėl *C++* netinka aparatūrai modeliuoti.

1. Nėra laiko sąvokos nuosekliems įvykiams, vėlinimams.
2. Nėra „lygiagretumo“. *C++* nepalaiko konkurencinių procesų.
3. Aparatūriniai duomenų tipai. *C++* nepalaiko nei trijų, nei keturių būsenų loginių reikšmių. Pvz.: „z“ ar „x“.

SystemC sudaryta iš *C++* klasių bibliotekos ir šerdies, kuri adresuoja šiuos nesutapimus. Taip pat ši kalba išplečiama panaudojant klases, bet nekeičiant sintaksės. Moduliai palaikomi įterpiant elgseną bei aprašant hierarchiją.

Kadangi *SystemC* yra sukurta *C++* kalbos pagrindu, tai ir projektavimo aplinka tinka ta pati, tik reikia prisijungti *SystemC.h* biblioteką.



9 pav. *SystemC* projektavimo aplinka

Yra skiriami šie modeliai:

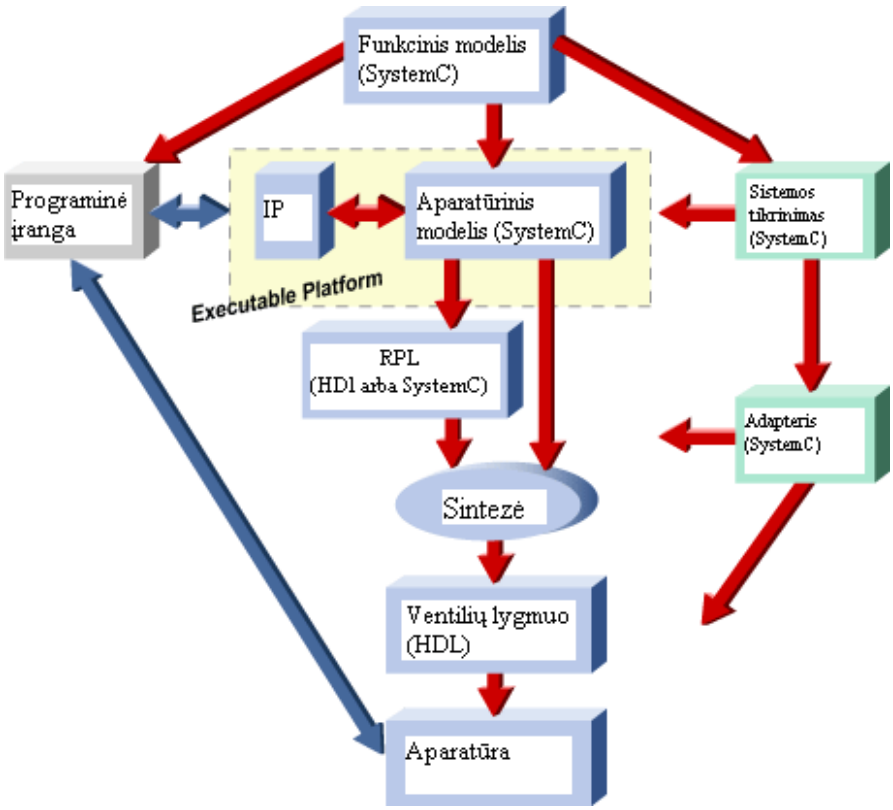
- *nelaikinis funkcinis (NF)*. Susijęs tiek su modelio sąsaja, tiek su funkcionalumu. Atlikimui laikas nenaudojamas. Procesai vykdomi iš karto bei duomenų perdavimas neužima laiko. Tipiniai *C++* duomenų tipai;
- *laikinis funkcinis (LF)*. Susijęs tiek su modelio sąsaja, tiek su funkcionalumu. Atlikimui laikas naudojamas. Tiek procesų, tiek duomenų perdavimo laikas yra baigtinis. Modeliuojami vėlinimai. Tipiniai *C++* duomenų tipai. NF ir LF gali būti maišyti;
- *magistralinis ciklinis (MC)*. Šis modelis susijęs su sąsaja, bet ne su funkcionalumu. Laikas ciklišk ai tikslus bei įprastai susietas su TDG. Nesusijęs su išvadų lygio detalėmis. Informacija paprastai perduodama kaip transakcija. Galimi kai kurie *SystemC* duomenų tipai;
- *išvadų ciklinis (IC)*. Šis modelis susijęs su sąsaja, bet ne su funkcionalumu. Laikas ciklišk ai tikslus bei įprastai su TDG. Sąsaja aprašoma išvadų lygyje. Galimi kai kurie *SystemC* duomenų tipai;

- *registrų perdavimo (RP).* Susijęs su modelio funkcionalumu. Viskas pilnai sinchronizuojama. TDG naudojami taip pat sinchronizavimui. Būdingas pilnas funkcinis aprašymas. Kiekvienas registras, kiekviena magistralė, kiekvienas bitas aprašomas kiekvienam TDG ciklui. Galimi kai kurie *SystemC* duomenų tipai. Sinonimas registrų perdavimo lygmens (TPL) modeliui.

Modeliams priskiriami šie tipai:

- *sisteminis architektūrinis.* Šie modeliai sudaro vykdomąją sistemos specifikaciją. Tipiškai aprašomi tiek aparatūriniai, tiek programiniai komponentai. Modelio sąsaja – FN be išvadų detalių. Tipiškai modeliuojami komunikacijos protokolai. Modelio funkcionalumas – FN. Elgsena modeliuojama algoritmiškai ir tipiškai aprašoma nuosekliai. Įmanoma, bet labai sudėtinga modeliuoti konkurencinius sakinius, kadangi nėra sinchronizacijos. Šie modeliai naudojami architektūrai apibrėžti bei algoritmų nustatymui ir patikrinimui;
- *sisteminis vykdomasis.* Šie modeliai sudaro laikinę vykdomąją sistemos specifikaciją. Tipiškai aprašomi tiek aparatūriniai, tiek programiniai komponentai. Modelio sąsaja – FN ar FL be išvadų detalių. Tipiškai modeliuojami komunikacijos protokolai. Modelio funkcionalumas – FN ar FL. Elgsena modeliuojama algoritmiškai. Veikia procesai bei konkurenciniai sakiniai su laikinėmis charakteristikomis, nesinchronizuojama. Šie modeliai naudojami aukšto lygio modeliavimui bei laiko paskirstymui;
- *perdavimų lygmens (PL).* Šie modeliai naudojami kurti vykdomąją platformą ir tipiškai aprašo tik aparatūrinę dalį. Sąsaja – FL be išvadų detalių, sinchronizuojamas arba ne. Duomenų perdavimas modeliuojamas kaip transakcijos. Modelio funkcionalumas yra FL ir nesinchronizuotas. Elgsena aprašoma naudojant transakcijas;
- *funkcinis modelis.* Apima architektūrinį bei vykdantįjį laikinį modelius;
- *sisteminis modelis* . Apima modelius aukštesnius už TPL;
- *elgsenos sintezės.* Modelis, kuris seka elgsenos sintezės taisyklėmis. Naudingas architektūrinei analizei ir pritaikymui. Sąsaja sinchronizuojama iki išvadų detalių. Funkcionalumas FL. Elgsena modeliuojama algoritmiškai;
- *magistralinis funkcinis (MF).* Tipiškai naudojamas simuliacijos poreikiams. Įprastai modeliuojami procesoriai, bet nenumatomas sintezei. Modelio sąsaja išvadų sinchronizuojama. Funkcionalumas – transakcijos.

- *Registrų perdavimo lygmens(RPL).* Šie modeliai naudojami detaliai aprašyti aparatūrai. Sąsaja – išvadų detalaus bei registrų lygmens. Tokie modeliai tipiškai aprašomi aparatūrinėmis kalbomis, pavyzdžiui *Verilog* ar *VHDL*.
- *Ventilių lygmens. SystemC* kalboje šį modelį realizuoti yra sudėtinga bei nerekomenduojama.



10 pav. Tipinis *SystemC* projektavimo procesas.

Vienuoliktame paveiksle pavaizduota *SyctemC* kalbos architektūra. Pilki blokai vaizduoja *SystemC* kalbos standartą. *SystemC* sudaryta iš standartinės *C++*. Balti blokai vaizduoja konstrukcines bibliotekas, kurios neįeina į *SystemC* standartą. Kalbos šerdis sudaryta iš įvykius valdančio stimuliatoriaus kaip bazės. Jis dirba su įvykiais ir procesais. Kiti kalbos elementai sudaryti iš modulių ir prievadų, reprezentuojančių struktūras, kai sąsajos ir kanalai aprašo komunikaciją.

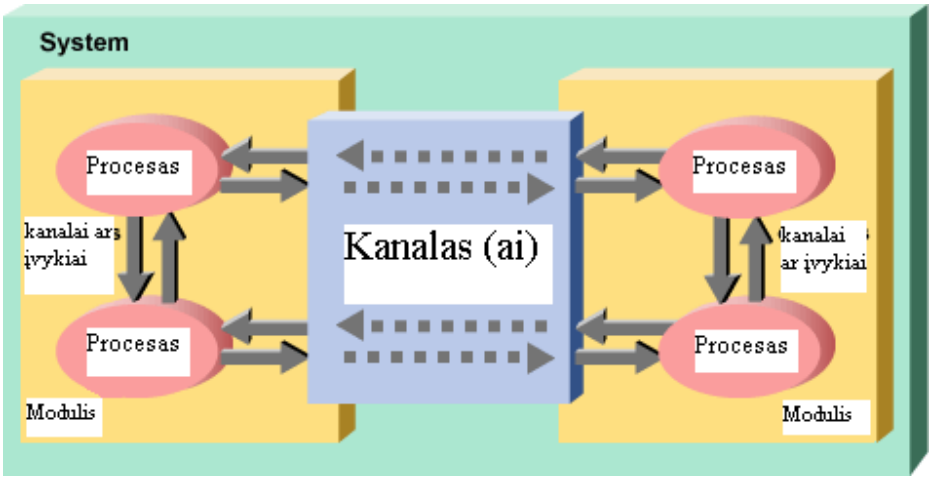
Duomenų tipai yra naudingi aparatūrai modeliuoti. Primityvūs kanalai yra integruoti į kalbą ir plačiai naudojami, kaip kad signalai ir *FIFO* metodai.

<table>
<tr><td colspan="2">Specifinės metodologijos
bibliotekos (Master/slave biblioteka)</td><td colspan="2">Sluoksninės bibliotekos
(statiniai duomenų srautai, verifikavimo biblioteka)</td></tr>
<tr><td colspan="4">Primityvūs kanalai
Signalai, multiplekseriai, FIFO</td></tr>
<tr><td colspan="2">Šerdies kalba
Moduliai
Portai
Procesai
Sąsajos
Kanalai
Įvykiai
Įvykių valdoma simuliacija</td><td colspan="2">Duomenų tipai
Loginis
Loginiai vektoriai
Bitai ir bitų vektoriai
Sveikieji
Fiksuoto kablelio tipai
C++ vartotojo nustatyti tipai</td></tr>
<tr><td colspan="4">C++ kalbos standartas</td></tr>
</table>

11 pav. *SystemC* kalbos architektūra

SystemC sudaryta iš komplekto, iš vieno arba kelių modulių. Moduliai sudaro galimybę aprašyti struktūrą. Į modulius tipiškai įeina procesai, prievadai, vidiniai duomenys, kanalai ir galimi kitų modulių komponentai. Vidiniai duomenys ir kanalai sudaro kelią komunikuoti procesams, palaiko modulio būsenas. Komponentai skirti hierarchinėms sistemos.

Komunikacija tarp skirtinguose moduliuose esančių procesų organizuojama per prievadus, sąsajas bei kanalus. Modulio prievadas – tai objektas, per kurį procesas pasiekia kanalo sąsają. Sąsaja nustato kompleksą kanalo pasiekimo funkcijų, kai kanalas pats savyje realizuoja šių funkcijų įdiegimą. Detalizavimo (*elaboration time*) metu modulio prievadai sujungiami su sukurtais kanalais.



12 pav. *SystemC* sudaryta iš komplekso modulių

Sąsajos, prievado, kanalo struktūra suteikia didelį lankstumą modeliuojant komunikacijas tarp modulių.

Įvykiai – tai pagrindiniai sinchronizuojantys objektai. Jie naudojamo sinchronizuoti procesus ir realizuoti blokų elgseną kanaluose.

Procesai nustatomi veikimui pagal jų jautrumo sąrašą. Palaikomas tiek dinaminis, tiek statinis jautrumas.

Priėjimas prie visų *SystemC* klasių ir funkcijų suteikiamas per antraštinį failą *SystemC.h*.

C++ integruoti duomenų tipai gali būti naudojami visuose modelių lygiuose. Aukštesniuose abstrakcijos lygmenyse, kaip kad funkciniame ar MF, tipiškai tinka dauguma duomenų tipų, bet leidžiantis žemyn ir žemėjant tipo abstrakcijos lygmeniui, modeliavimui tinkamų tipų yra vis mažiau. *SystemC* kalba suteikia modeliavimui reikalingus tipus:

- fiksuoto kablelio sveikieji skaičiai;
- parenkamo tikslumo sveikieji skaičiai;
- 4-reikšmių loginiai tipai (`0`, `1`, `Z`, `X`);
- 4-reikšmių loginiai vektoriai;
- fiksuoto kablelio tipai.

Sąsaja nustato priėjimo metodų kompleksą, yra visiškai funkcinė ir nerealizuoja jokių pritaikymų. Ji tik yra susieta su prievadu. Sąsajos realizaciją atlieka kanalas.

Kanalai naudojami komunikacijai tarp moduliuose esančių procesų. Modulio viduryje esantis procesas gali tiesiogiai kreiptis į kanalą. Skiriami primityvūs ir hierarchiniai kanalai. Primityvūs kaip *sc_signal*, o hierarchiniai gali apimti modulius su procesais ir prievadais ir kt. Naudojant primityvius

kanalus galima dažnai sumažinti delta ciklų skaičių. Hierarchiniai kanalai naudojamo sudėtingesnėse struktūrose.

Modulis *SystemC* kalboje yra *C++* kalbos klasė, todėl turi kostruktorių. Konstruktorius - tai funkcija neišduodanti rezultato. Ji naudojama kurti ir paskelbti modulio atsiradimą ir sukurti vidines duomenų struktūras, kurios naudojamos modulyje priskiriant joms bendras reikšmes. Procesai užregistruojami ir taip pat paskelbiami konstruktoriaus viduryje. *SystemC* turi specialią makro funkciją konstruktoriui *SC_CTOR*. Jei yra daugiau argumentų, naudojama *SC_HAS__PROCESS* makro funkcija.

Konstruktoriaus aprašymo pavyzdys:

```
SC_MODULE(modulio_vardas) {
  // Portai, vidiniai  signalai, procesai,kiti metodai

  // Konstruktorius
  SC_CTOR(modulio_vardas) /* : paskelbiamas sąrašas */ {
    // procesų registracija
    // jautrumo sąrašo deklaravimas
    // modulių paskelbimas
    // portų jungimosi deklaracija
  }
};
```

Sc_event (Įvykiai) yra pagrindiniai *SystemC* kalboje naudojami sinchronizavimo objektai. Jie naudojami sinchronizuoti procesams.

Funkcionalumas aprašomas procesuose. Procesai yra funkcijos, kurios identifikuojamos *SystemC* šerdyje tam tikru tipu. Į procesus nesikreipiama tiesiogiai, o pagal statinį arba dinaminį jautrumo sąrašą. Procesai yra labai panašūs į *C++* metodus ir funkcijas su mažomis išlygomis. Kai registruojami procesai *SystemC* šerdyje, jie priskiriami tam tikram metodui. Procesai užregistruojami modulio klasės konstruktoriaus viduje. Yra skiriami trys procesų tipai:

- metodiniai (*SC_METHOD*);
- sinchronizuoti žingsniniai (*SC_THREAD*);
- TDG sinchronizuoti žingsniniai (*SC_CTHREAD*).

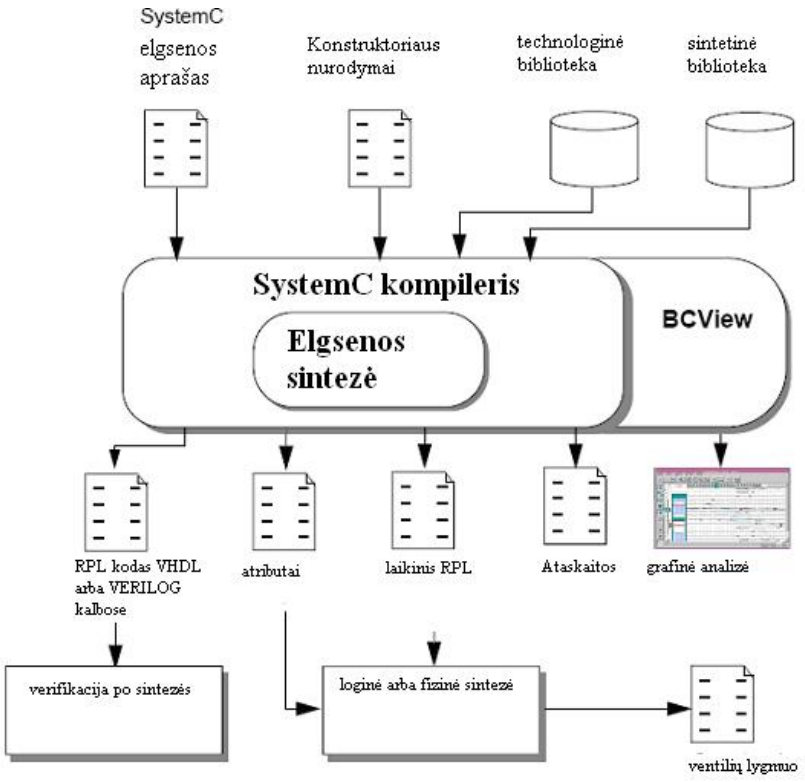
Žingsniniai procesai turi savo nustatytus žingsnius atlikimui. Tai atliekama per dinaminį arba statinį (prievadai, kanalai ir kt.) jautrumo sąrašą ir iki *wait(0)*. Žingsniniai procesai realizuoti kaip amžini ciklai.

Metodiniai procesai skiriasi tuo, kad jie negali būti sustabdyti.

Aukščiausiojo lygmens modulį *SystemC* kalboje atstoja speciali funkcija *sc_main()*.

2.3.3.1. *SystemC* sintezė

SystemC kalbos sintezę palaiko mažiau įrankių nei *Verilog*, be to čia yra tam tikrų papildomų niuansų.



13 pav. *SystemC* kalba aprašytų modulių sintezės eiga

Norint atlikti sintezės procesą reikia pirma atlikti tam tikrus nustatymus ir turėti atitinkamus failus. Pvz.: *desing_analyzer* programoje:

- *default* skyrelyje nustatome technologinę, sintetinę bei simbolinę bibliotekas, taip pat paieškos kelius sintezuojamiems failams bei bibliotekoms;
- galime visą procesą atlikti patys arba naudoti iš anksto paruoštus *.scr* skriptų failus, kuriuose nurodomas nuskaitomas projektas, konstruktoriaus nurodymai, atributai, išdėstymas laike, jo modelis, TDG bei kitos norimos funkcijos;
- po kiekvienos operacijos galima išsisaugoti *.db* failuose visą schemą su duomenimis;
- atlikus sintezę, išsisaugojus kalbos RPL kodavimo tipo failus atliekama verifikacija;
- paruošiamos ataskaitos apie elementus, bibliotekas, kiekį, plotą, laikines charakteristikas;
- *BC_View* ar kitos panašios programos leidžia nagrinėti signalus grafinėje aplinkoje.

3. *RISC 8* MIKROVALDIKLIO KŪRIMAS IR ANALIZĖ

3.1. Pradinė specifikacija

Specifikuoti norimą sukurti produktą yra bene sunkiausia, daugiausiai laiko reikalaujanti ir didžiausią atsakomybę kelianti projekto dalis. Gera specifikacija mažina projekto kainą, kadangi suprojektavus įrenginį, keisti specifikaciją yra daug brangiau.

Objektas: *RISC*, 8 bitų, Harvardo architektūros mikrovaldiklis su išplėtimo sąsaja.

Pagrindiniai parametrai surašyti 1 lentelėje.

1 lentelė. Specifikuojami projekto parametrai

Parametras	Reikšmė	Žymėjimas projekte
Taktinis dažnis, MHz	50	clk
Asinchroninis perkrovimas	Taip	Reset
Procesoriaus apdorojamo žodžio ilgis, (bitais)	8	
Duomenų atmintis, (baitais)	70	
Programų atmintis, (bitais)	2048x12	
Programų atmintis, adresavimas, (bitais)	11	paddr
Programų atmintis, duomenys, (bitais)	12	pdata
Komandų dekoderio žodžio ilgis, (bitais)	12	inst
Magistralės, (bitais)	8	Dbus, Sbus
Laikmatis žodžio ilgis, (bitais)	8	Tmr0
Laikmačio buferio žodžio ilgis, (bitais)	8	prescaler
Registrų kiekis, vnt.	8	
Aritmetinis loginis įtaisas (ALĮ), dvigubas, (bitais)	2 x 8	alua, alub
ALĮ operacijos	8	
ALĮ operacijos	ADD, SUB, AND, OR, XOR, COM, ROR, ROL, SWAP	ALUOP_...
ALĮ opeacijų žodžio ilgis, (bitais)	4	
Prievadai	A, B, C	
Prievadų žodžio ilgis, (bitais)	8	
Maksimalus elementų plotas, santykiniai vnt.	10000	
Maksimalus bendras plotas, santykiniai vnt.	30000	
Maksimalus bito perdavimo laikas, ns	500	
Maksimaliai suvartojama elementų energija, mW	1	

Keturioliktame paveikslėlyje apibrėžta mikrovaldiklio „juodos dėžės“ realizacija su išvadų žodžio ilgiais.

Mikrovaldiklį sudaro:

cpu		**c**entrinis procesorinis elementas (CPE)
	idec	komandų dekoderis
	alu	aritmetinis loginis įtaisas
	regs.	registrų failas
	dram	**d**uomenų atminties modelis, sinchroninis 72x8.
pram		**p**rogramų **a**tminties modelis, sinchroninis 2048x12.
exp		**p**avyzdinis išplėtimo modulis (*DDS*).

Signalai: *expdin, expdout, expwrite, expread* – išplėstinės sąsajos signalai.

Pdata, paddr – programų atminties signalai.

Portain – prievadas A.

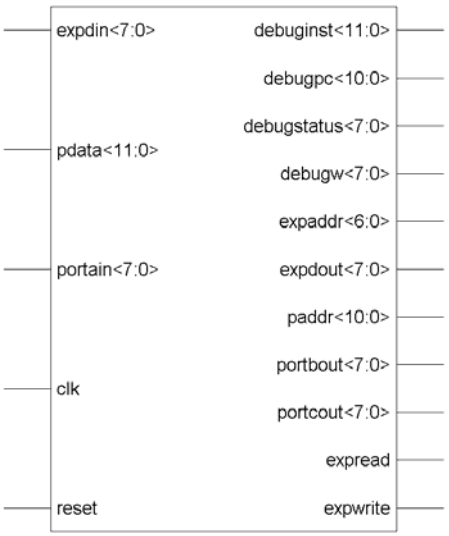
Porbout – prievadas B.

` *Portcout* – prievadas C.

Clk – taktinio dažnio generatoriaus signalas.

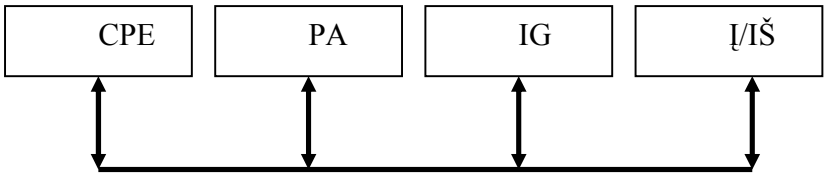
Reset – programos perkrovimo signalas.

Debuginst, debugpc, debugstatus, debugw – procesoriaus derinimo signalai.

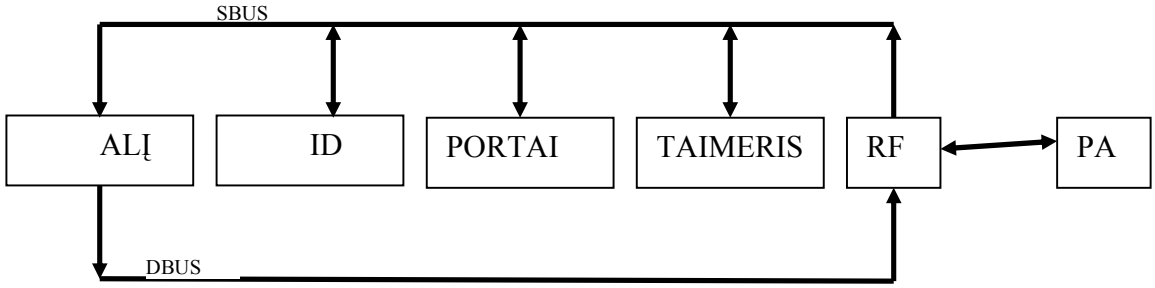


14 pav. Centrinio procesorinio elemento(CPE) modulis specifikacijai

CPE sudarytas iš instrukcijų dekoderio (ID), registrų failo (RF), duomenų atminties (DA) bei aritmetinio loginio įtaiso (ALĮ). Taip pat šis elementas valdo programų atmintį bei išplėtimo grandinę.



15 pav. Mikrovaldiklio schema



16 pav. CPE schema

Specialieji vidiniai registrai:

- komandu registras *inst [11:0];*
- programų skaitiklis *pc, pc_in [10:0];*
- dėklai *stacklevel[1:0], stack1, stack2[10:0];*
- registras W *w[7:0];*
- būklės registras *status[7:0],* naudojami tik 0 ir 2 bitai: C ir Z;
- žymeklio registras netiesioginiam adresavimui *fsr[7:0];*
- laikmatis su buferiu *tmr0, prescaler [7:0];*
- opcijų registras *option[7:0];*
- trijų būsenų kontrolės registrai, *trisa, trisb, trisc[7:0];*
- prievadų registrai *porta, portb, portc;*
- *dbus[7:0]* – magistralė, išeinanti iš ALĮ ir pasiekiama visiems registrams;
- *sbus[7:0]* – magistralė, įeinanti į ALĮ multiplekserius iš registrų;

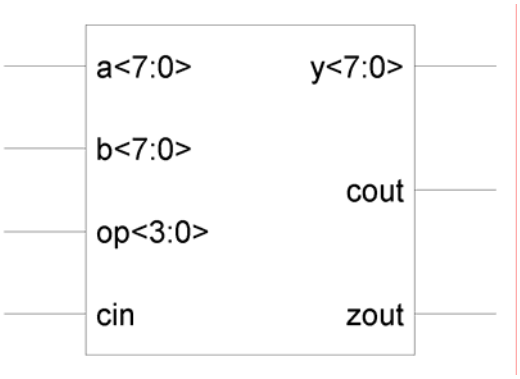
Adresavimas į registrus vykdomas per *fileaddr[6:0]*.

Aritmetinį loginį įtaisą sudaro:

- op - operacijos pasirinkimas;

```
parameter ALUOP_ADD  = 4'b0000;
parameter ALUOP_SUB  = 4'b1000;
parameter ALUOP_AND  = 4'b0001;
parameter ALUOP_OR   = 4'b0010;
parameter ALUOP_XOR  = 4'b0011;
parameter ALUOP_COM  = 4'b0100;
parameter ALUOP_ROR  = 4'b0101;
parameter ALUOP_ROL  = 4'b0110;
parameter ALUOP_SWAP = 4'b0111;
```

- a, b, ALĮ multiplekseriniai įėjimai. Pats ALĮ suskaidytas į du;
- y – išėjimas;
- cout – pernešimas;
- cin, zout – operacijoms naudojami signalai.

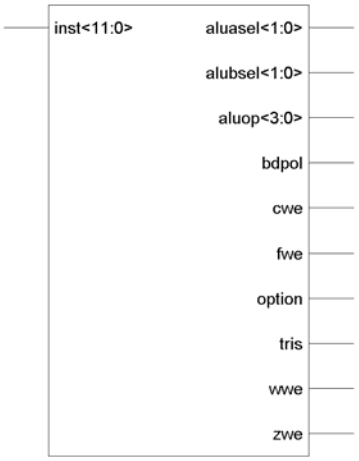


17 pav. Aritmetinio loginio įtaiso modulis specifikacijai

Komandų dekoderį, skirtą dekoduoti komandas, sudaro:

- *inst[11:0]* – komandos kodas;
- *aluasel, alubsel, aluop* – aritmetinio loginio įtaiso signalai;
- *bdpol* – dekoduojančio vektoriaus poliarumas;
- wwe – W registro leidimas įrašyti;
- fwe – failų registro leidimas įrašyti;
- zwe, cwe, būsenos registro z, c bitai;
- tris, option - derinimo komandos

Pagal 12 bitų *inst* gaunamas 15 bitų dekoduotas registras, pagal kurį nustatomi visi išėjimai.



18 pav. Komandų dekoderio modulis specifikacijai

Registrų failą sudaro:

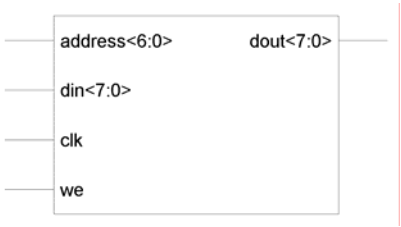
- *bank* – banko išrinkimas atmintyje;
- *din* – duomenų įėjimas atmintyje;
- *dout* – duomenų išėjimas atmintyje;
- *location* – nustato banko vietą;
- *re, we* – rašymo ir skaitymo leidimo signalai.

19 pav. Registrų failo modulis specifikacijai

Kiekvienas atminties bankas sudarytas iš 32 vietų. Vieta atitinka baitą. Šešiolika kiekvieno banko vietų priskirta tam pačiam bankui. Pirmos 8 vietos skirtos specialiems registrams. Likusios 16 vietų yra unikalios kiekvienam bankui. Koks pasirinktas tam tikru momentu bankas, nustato registro FSR du vyriausieji bitai. Tai galioja tik aukštesnėms unikalioms 16 vietų. Kitos pirminės vietos prieinamos bet kada, nesvarbu, koks būtų nustatytas bankas.

Duomenų atminties modulį sudaro:

- *address* – adresas;
- *dout, din* – duomenų išėjimas, įėjimas;
- *we* – leidimo įrašyti signalas.



20 pav. Programų atminties modulis specifikacijai

Programų atminties modulį sudaro:

- *address* – adresas;
- *dout, din* – duomenų išėjimas, įėjimas;
- *we* – įrašyti leidimo signalas.

21 pav. Programų atminties modulis specifikacijai

Išplėtimo sąsajos modulis turi savyje realizuotą skaitmeninį sintezatorių, kurio pagalba galima gauti moduliuotą sinusoidę.

Modulį sudaro:

- *ads_out* – sintezatoriaus išėjimas;
- *expdin, expdout* – duomenų išėjimai, įėjimai;
- *expaddr* – adresas;
- *expread, expwrite* – skaitymo ir rašymo leidimo signalai.

Dažniausiai sinusoidė realizuojama kaip bitų seka atmintyje. Skaitomas *sindata.hex* failas, kuriame surašyta seka, per *expread* grandinę vėl signalas nukreipiamas į procesorių ir galima jį naudoti apdorojimui.

3.2. Sintezuojama specifikacija

Tai *RISC (Reduced inctruction set cycle)* architektūros 8 bitų *Verilog* kalba aprašytas procesoriaus failų paketas analogoškas *PIC 16C57* mikrovaldikliui. Pats *RISC8* mikrovaldiklis aprašomas vėliau esančiuose skyriuose, o pagrindiniai skirtumai tarp aprašomo mikrovaldiklio ir *PIC 16C57* pateikti 2 lentelėje. Tai pilnai sintezuojamas ir turintis galimybę būti realizuotas *FPGA* struktūroje failų paketas.

2 lentelė. Paketo struktūra

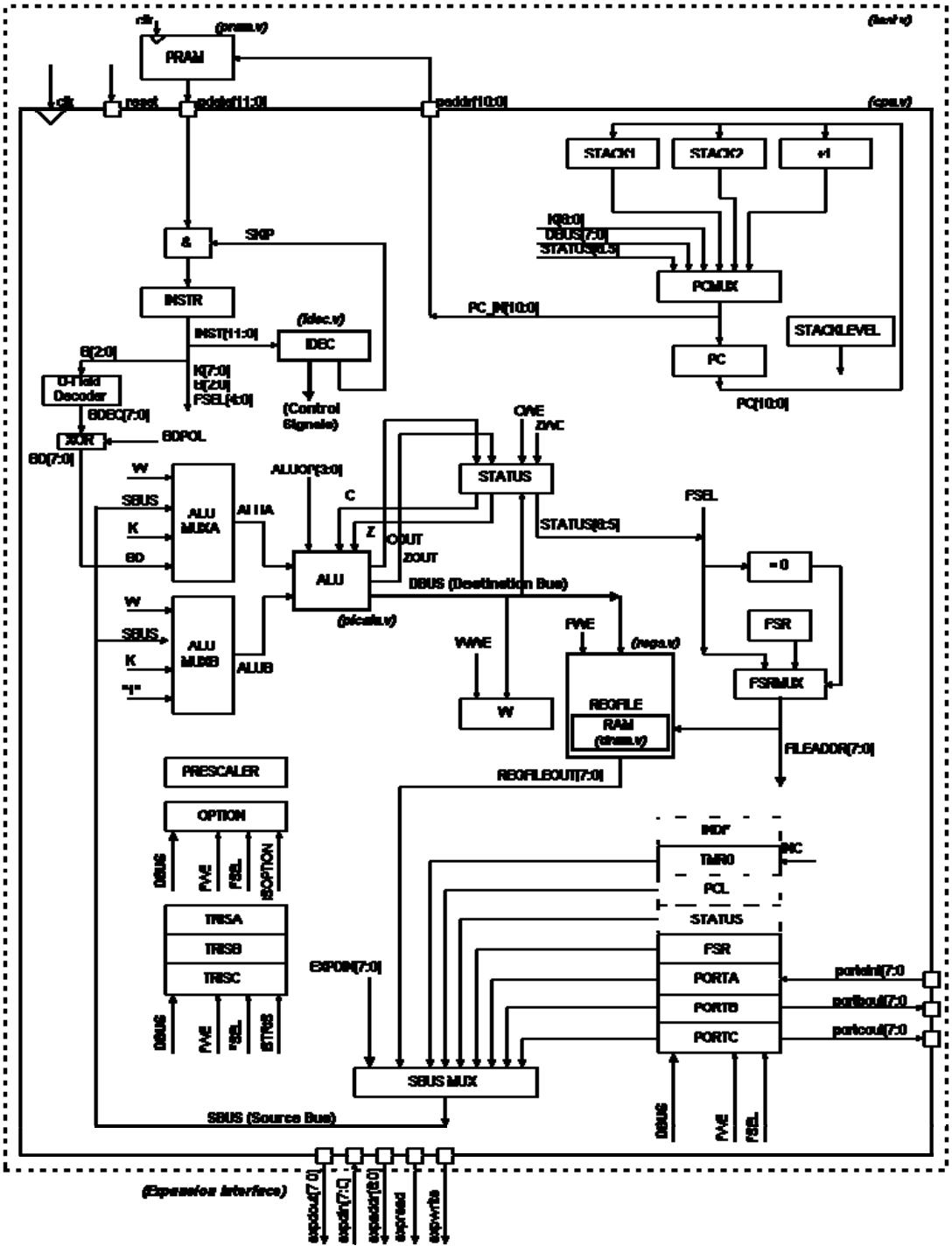
Failas	Aprašymas
test.v	Aukščiausio lygmens testavimo programa, įskaitant elgsenos tipu *Verilog* kalba programuotą programų atmintį (PA).
cpu.v	Aukščiausio lygmens sintezuojamas centrinio procesorinio elemento(CPE) modulis.
idec.	Komandų dekoderis(KD). Žemesnio lygio modulis po CPE moduliu.
alu.v	Aritmetinio loginio įtaiso(ALĮ) modulis. Taip pat po CPE moduliu.
regs.v	Registrų failas (RF). Taip pat po CPE moduliu.
exp.v	Išplėtimo modulis (priedas). Tai pavyzdinis modulis, kuris vaizduoja kaip pridedama išplėtimo grandinė prie konstrukcijos. Šis modulis realizuoja labai paprastą *DDS(Direct Digital Synthesis)* grandinę, kuri naudojama *DDS demo* vaizduoti.
dram.v	Atminties modelis RF moduliui – duomenų atmintis (sinchroninė, RAM).
pram.v	Atminties modelis programų moduliui – duomenų atmintis (sinchroninė, RAM).
basic.rom	Failas su komandomis.

Sistemos architektūra parodyta 22-ame paveikslėlyje.

Mikrovaldiklio komandų žodžio ilgis – 12 bitų, o duomenų kelias – 8 bitų pločio. Taip pat yra 72 žodžiai duomenų ir iki 2048 programų atminties. Mikrovaldiklis turi kaupikliu pagrįstą 33 komandų komplektą. W registras ir yra kaupiklis (akumuliatorius). Programų skaitiklis (PS) ir du dėklo registrai leidžia vykdyti 2 lygių paprogrames. Registrų failas naudoja bankus ir netiesioginio adresavimo būdą. Šerdies registrų failas realizuotas „flip-flop“ tipo registrų pagrindu. PA (PRAM) – tai atskira atmintis nuo registrų failo ir realizuota už šerdies ribų. PA (PRAM) – paprastas *Verilog* kalbos pagrindu programuotas atminties masyvas realizuotas test.v modulyje. Šerdis yra sinchroninė, vieno ciklo įvykdymo, turi vieną sinchroninę *reset* funkciją.

ALĮ turi minimalų 8 bitų operacijų komplektą (ADD, SUB, OR, AND, XOR, ROTATE ir kt). KD – tai visiškai kombinatorinė logikos lentelė, kuri siunčia raktinius signalus. Mikrovaldiklis turi vienpuses magistrales, bet priešingų krypčių.

Komandos vykdomos vieno ciklo metu. Argumentas pertraukčių stokai – komandų įvykdymas per vieną taktinį ciklą.



22 pav. Architektūra

Išplėtimas padarytas per išplėtimo grandinę pagrindinio modulio CPE sąsajoje. Magistralė palaiko adresus, skaitymą, rašymą, iš/į signalų komplektus. Modulis exp.v vaizduoja paprastą išplėtimo grandinę.

Šis failų paketas RISC8 gali vykdyti kodą suderinamą su skirtu mikrovaldikliui *PIC 16C57*. Bet yra keletas skirtumų kaip kad skirtingas atminties kiekis ir Į/IŠ sistema.

Šios savybės ar charakteristikos skiriasi:

3 lentelė. Suderinamumas su 16C57 mikrovaldikliu

Savybė	Mikrovaldiklis 16C57	RISC8 failų paketas
TDG	Turi keletą rėžimų.	Paprastas tiesioginis įėjimas
TDG veikimas	Naudoja 4 fazes	Vienfazis
Reset	Naudoja aktyvų žemo lygio MRST ir „power-up“ grandinę, kai kurie turi apsaugas nuo perdegimo.	Paprastas aktyvus aukšto lygio.
Parengties rėžimai	Turi komandą ir grandinę.	Nėra.
Trijų būsenų jungtys	Turi dvipusius prievadus su TRIS komanda krypčiai nurodyti.	Nėra. Prievadas A – įėjimas, Prievadai B, C – išėjimai.
Stebintis laikmatis	WDT grandinė	Nėra
Laikmatis 0	Yra.	Sinchronizuotas su vidiniu TDG, naudoja tris buferio bitus OPTION registre.
OPTION registras ir komanda	Naudoja.	Tik laikmačiui 0.

Hierarchija:

test.v testinė programa, apimanti ir PA.

- **cpu.v** aukščiausio lygio CPE modulis.
  - **idec.v** KD (kombinatoriais, po CPE moduliu).
  - **alu.v** ALĮ (po CPE moduliu).
  - **regs.v** RF (po CPE moduliu).
  - **dram.v** atminties modelis, sinchroninis 72x8.
- **pram.v** aAtminties modelis, sinchroninis 2048x12.
- **exp.v** pavyzdinis išplėtimo modulis (*DDS*).

CPE modulis (*cpu.v*).

Tai aukščiausio lygio sintezuojamas modulis. Čia realizuoti pagrindiniai registrai kaip kad *INST, W, STACK1, STACK2 ir PC*. Programų vykdymas kontroliuojamas kaip ir realizuoti vidinės magistralės sutankintojai. Įėjimai ir išėjimai, laikmatis ir kitos grandinės randasi taip pat šiame modulyje.

Komandų vykdymo būdo keitimas realizuojamas per:

- *GOTO* komandą;
- *CALL* paprogramę;
- sąlygines *SKIP* komandas.

GOTO komanda dekoduoja paskirties adresą simboliniame komandos lauke. Paprogramės realizuotos panaudojant elementinius dėklų registrus (vietoj programinio dėklo ar dėklo žymeklio registro). Tai dalinai Harvardo architektūros nuopelnas ir griežtas programų, duomenų erdvės atskyrimas. *SKIP* komandos yra sąlyginės ir dažniausiai susiję su bitų registre testu.

Atminties sąsaja (dram.v, pram.v).

Atminties sąsaja yra griežtai ir tiesiogiai susieta su šerdimi. 11- kos bitų adresas išsiunčiamas ir laukiamas 12 – bitų duomenų įėjimas. Skaitymas sinchronizuotas.

RF sąsaja – tai sinchroninė sąsaja su *clk* ir *reset* įėjimais. Įėjimai adresavimui apima 2 bitų bankus bei 5 bitų vietos įėjimus. Skaitymo ir rašymo leidžiamieji signalai yra įėjimai ir realizuoti per dvi atskiras duomenų magistrales įėjimams ir išėjimams. Registrų modulis atlieka adresavimo logiką, kur kai kurių žodžių kopijos tiesiog patalpinamos bendroje adresų erdvėje. Po registrų moduliu yra duomenų atminties modulis t.y. žemesnio lygio nei RF sinchroninis atminties modelis kaip ir programų modulis.

Aritmetinis loginis įtaisas (alu.v).

ALĮ realizuotas *alu.v* faile. Tai visiškai kombinatorinę logiką atspindintis modulis. Jis turi du 8 bitų įėjimus, A ir B taip pat ir *CON Carry* įėjimą kaip vienetinį bitą. Keturių bitų operandas parenka ALĮ operacijas. Jis turi 8 bitų duomenų išėjimą, bitų *carry* bei nulinio išėjimus. ALĮ neparenka tinkamo šaltinio savo įėjimams ir nenusprendžia kada reikia būsenos bitą keisti. Tai atlieka KD arba aukštesnio lygio CPE modulis.

ALĮ palaiko sekančius operandus:

4 lentelė. ALĮ operandai

ALĮ operandų pasirinkimo kodas	Operacija	Aprašymas
0000	ADD	A + B (be pernešimo)
1000	SUB	A – B (be pasiskolinimo)
0001	AND	A and B
0010	OR	A arba B
0011	XOR	A xor B
0100	COM	Ne A
0101	ROR	{A[0], A[7:1]}
0110	ROL	{A[6:0], A[7]}
0111	SWAP	{A[3:0], A[7:4]}

Komandų dekoderis (idec.v).

KD realizuotas *idec.v* modulyje. Tai taip pat visiškai kombinatorinis modelis. Tai specialiai realizuota kaip didelė *casex* sakinio struktūra; per vieną arba du komandos skirsnius. KD išėjimai – tai komplektai dekoduotų duomenų.

Kai pradedama vykdyti komanda, ji iš karto užregistruojama *INST* registre. Tai atliekama kiekvieną ciklą. *RISC8* turi 33 komandas. *INST* registre reziduojančios komandos būna 12 bitų pločio. Keletas laukų dažnai priskiriami komandose, įskaitant F, K, B laukus. Šie laukai sukurti šerdyje iš originalių 12 *INST* registro bitų.

Komandų santrauka pateikiama sekančioje lentelėje:

5 lentelė.KD kontroliniai signalai komandoms

Komanda	ALĮ A šaltinis	ALĮ B šaltinis	ALĮ operandas	ALĮ išėjimas	WWE	FWE	ZWE	CWE	BDPOL
Baitinės registrų Komandos									
NOP	X	X	X	X	0	0	0	0	0
MOVWF	W	W	OR	W	0	1	0	0	0
CLRW	W	W	XOR	0	1	0	1	0	0
CLRF	W	W	XOR	0	0	1	1	0	0
SUBWF (d=0)	F	W	SUB	F-W	1	0	1	1	0
SUBWF (d=1)	F	W	SUB	F-W	0	1	1	1	0
DECF (d=0)	F	1	SUB	F-1	1	0	1	0	0
DECF (d=1)	F	1	SUB	F-1	0	1	1	0	0
IORF (d=0)	W	F	OR	W|F	1	0	1	0	0
IORWF(d=1)	W	F	OR	W|F	0	1	1	0	0
ANDWF(d=0)	W	F	AND	W&F	1	0	1	0	0
ANDWF(d=1)	W	F	AND	W&F	0	1	1	0	0
XORWF(d=0)	W	F	XOR	W^F	1	0	1	0	0
XORWF(d=1)	W	F	XOR	W^F	0	1	1	0	0
ADDWF(d=0)	W	F	ADD	W+F	1	0	1	1	0
ADDWF(	W	F	ADD	W+F	0	1	1	1	0
MOVF(d=0)	F	F	OR	F	1	0	1	0	0
MOVF(d=1)	F	F	OR	F	0	1	1	0	0
COMF(d=0)	F	X	NOT	~F	1	0	1	0	0
COMF(d=1)	F	X	NOT	~F	0	1	1	0	0
INCF(d=0)	F	I	ADD	F+1	1	0	1	0	0
INCF(d=1)	F	I	ADD	F+1	0	1	1	0	0
DECFS	F	I	SUB	F-1	1	0	0	0	0

Z(d=0)									
DECFS Z(d=1)	F	I	SUB	F-1	0	1	0	0	0
RRF(d=0)	F	X	ROR	{C,F[7:1]}	1	0	0	1	0
RRF(d=1)	F	X	ROR	{C,F[7:1]}	0	1	0	1	0
RLF(d=0)	F	X	ROL	{F[6:0],C}	1	0	0	1	0
RLF(d=1)	F	X	ROL	{F[6:0],C}	0	1	0	1	0
SWAPF (d=0)	F	X	SWAP	{F[3:0],F[7:4]}	1	0	0	0	0
SWAPF (d=1)	F	X	SWAP	{F[3:0],F[7:4]}	0	1	0	0	0
INCFSZ (d=0)	F	I	ADD	F+1	1	0	0	0	0
INCFSZ (d=1)	F	I	ADD	F+1	0	1	0	0	0
Bitinės failų registrų komandos									
BCF	K	R	BCL	F&(~(1<<K))		1	0	0	1
BSF	K		BSET	F|~(1<<K)		1	0	0	0
BTFSC	K		BTST	F&(1<<K)		0	0	0	0
BTFSS	K		BTST	F&(1<<K)		0	0	0	0
Simbolinėss ir kontrolinės Komandos									
OPTIKON	W		OR	W		1	0	0	
SLEEP	X		X	X		0	0	0	
CLRWDT	X		X	X		0	0	0	
TRIS	W		OR	W		1	0	0	
RETLW	K		OR	K		0	0	0	
CALL	X		X	X		0	0	0	
GOTO	X		X	X		0	0	0	
MOVLW	K		OR	K		0	0	0	
IORLW	K		OR	K|W		0	1	0	
ANDLW	K		AND	K&W		0	1	0	
XORLW	K		XOR	K^W		0	1	0	

Kiekviena komanda apima dalinius kontrolinių signalų komplektus, ALĮ šaltinių įėjimus, programų skaitiklio atnaujinimą, būsenos registro rašymo leidimą, registrų failų adresus ir kt. Kontroliniai signalai dekoduojami vienoje vietoje, modulyje *idec.v*. Šis modulis generuoja 15 kontrolinių signalų.

KD kontroliuoja siunčiamus į ALĮ signalus ir ALĮ atliekamas operacijas. ALĮ turi du įėjimus: A ir B. A ir B įėjimai valdomi sutankintojo, kuris išrenka arba *W, SBUS, K, BD* vektorius ALĮA įėjimui, arba *W,*

SBUS, K ar simbolinį 00000001. Beveik visi duomenys, kurie bus įrašyti atgal į registrų failą, siunčiami per ALĮ. Dažnai ALĮ vaidina duomenų persiuntėjo vaidmenį. Taip panaudojant ALĮ konstrukcijoje sumažinamas magistralių kiekis. Pvz.: norint išvalyti, W registrui taikoma operacija *XOR* su juo pačiu tam, kad gautume 00000000l, arba *OR* operacija paprasčiausiam duomenų perkopijavimui per ALĮ.

Būsenos bitai, kaip kad z ir c (nulis ir pernešimas) atnaujinami priklausomai nuo komandos. Kiekvienai komandai turi būti generuojamas leidimo signalas. Lentelėje 5 vaizduojami kontroliniai KD signalai.

Registrų failas (*regs.v*)

Registrų failas realizuotas modulyje *regs.v*. RF – tai sudėtingesnė struktūra nei PA. PA yra šerdies išorėje ir realizuota kaip paprastas atminties modelis. RF reikalingas įėjimas įrašymui, išėjimas skaitymui. Taip pat jis suskirstytas į keletą „bankų“. Šie bankai kartais būna priskiriami į vieną bendrą atminties sritį. Šis modulis apima visą logiką, kuri priskiria registrų adresams (bankų ir kitiems) fizinius duomenų atminties adresus. Po šiuo moduliu (žemesnis lygis) realizuotas duomenų atminties modelis dram.v.

Šeštoje lentelėje vaizduojami šie sąryšiai:

6 lentelė.RF loginiai adresų ryšiai

Failų registras		Galutinis duomenų atminties adresas	Aprašymas
FSR [6:5]	f[4:0]		
00	0x00 – 0x07	N/D	Specialios paskirties registrai (8)
00	0x08 – 0x0F	0x00 – 0x07	Bendri registrai (8)
00	0x10 – 0x1F	0x08 – 0x17	Bankas #0 registrai (16)
01	0x00 – 0x07	N/D	Specialios paskirties registrai (8 su kopijomis)
01	0x08 – 0x0F	0x00 – 0x07	Bendri registrai (8 su kopijomis)
01	0x10 – 0x1F	0x18 – 0x2F	Bankas #1 registrai (16)
10	0x00 – 0x07	N/D	Specialios paskirties registrai (8 su kopijomis)
10	0x08 – 0x0F	0x00 – 0x07	Bendri registrai (8 su kopijomis)
10	0x10 – 0x1F	0x30 – 0x47	Bankas #2 registrai (16 su kopijomis)
11	0x00 – 0x07	N/A	Specialios paskirties registrai (8 su kopijomis)
11	0x08 – 0x0F	0x00 – 0x07	Bendri registrai (8 su kopijomis)
11	0x10 – 0x1F	0x48 – 0x5F	Bankas #3 registrai (16 su kopijomis)

RF apima 70 8 bitų duomenų žodžių. Mirovaldiklis turi 72 registrus, bet čia 2 registrai panaudojami specialiems išorinių įrenginių poreikiams. Jei norima pridėti papildomų išorinių įrenginių, tai turi būti atlikta būtent šiuo keliu vietas imant iš atminties erdvės.

RISC8 naudoja vienfazį TDG įėjimą. Šerdis naudoja fazes tam, kad atliktų skaitymą ir rašymą per vieną ciklą (periodą).

RF gali skaityti ir rašyti per vieną ciklą. Naudojant *Q1- Q4* fazes, paprasta sinchroninė atmintis taip pat gali būti naudojama. Duomenys perskaitomi iš RF per Q2 (Q1 duoda leidimą). RF atnaujinamas per Q4. Duomenis, gauti iš RF, siunčiami į *SBUS* sutankintoją (*SBUS mux*). Vykstant komandos ciklui ALĮ išėjimas valdo *DBUS* magistralę, kuri sujungta su RF duomenų įėjimu. Jei KD įterpia *FWE* leidimą, tada šie duomenys turi būti įrašyti atgal į RF per Q4 fazę.

Išplėtimo grandinė (exp.v).

Išplėtimo grandinė (IG) leidžia integruoti į sistemą naujus pageidaujamus modulius. Tai atliekama per specialius signalų komplektus CPE modulio sąsajoje. Registrų adresų erdvėje gali būti rezervuotas bet koks skaičius adresų, reikalingų IG. Pats modulis suteikia 2 tokias vietas. IG realizuoja paprastą *DDS* grandinę naudojamą *DDS demo* programos. IG grandinei rezervuotos grandinės turi būti dekoduotos CPE modulyje (*expsel* signalas). *Case* struktūrą, kai norima, galima modifikuoti. Pagal dabartinę konfigūraciją rezervuojamos 4 vietos ir šios vietos negali būti naudojamos RF.

Išplėtimo grandinės signalai aprašyti 7 lentelėje.

7 lentelė. Išplėtimo grandinės signalai

Signalas	Aprašymas
expdin[7:0]	Tai 8 bitų duomenys iš IG modulio į šerdį. Galioja, kai įterpiamas *expread*.
expdout[7:0]	Išėjimas iš šerdies. Tai 8 bitų duomenys į IG modulį iš šerdies. Galioja, kai įterpiamas *expwrite*. IG atsakinga už *expaddr* dekodavimą tam, kad žinotų įkuri išplėtimo adresą įrašoma.
expaddr[6:0]	Galutinis duomenų erdvės adresas skaitymui ir rašymui. Įskaitant bet kokį ne tiesioginį adresavimą. Turi būti nustatytas *expsel* signalas, kai adresuojama IG vieta vietoje to, kad RF adresuojamas.
expread	Nustatomas aukštu lygiu, kai skaitoma iš IG adreso.
expwrite	Nustatomas aukštu lygiu, kai rašoma į IG adresą.

IG paprastai naudoja *clk* ar *reset* signalą per vieną ciklą.

3.2.1. RISC8 mikrovaldiklio modeliavimas

Modeliavimui panaudojami visų modelių failai, naudojama programa *Cadence LDV-5.1.*

Pirminis etapas sukompiliuoti failus. Kompiliuojant modulių failus patikrinama sintaksė ir semantika. Kai kodas parašytas pagal taisykles, tada atliekamas detalizavimo procesas (*elaborate*). Jo metu susiejami komponentai pagal prievadus, nustatomas komponentų funkcionalumas ir kiti aparatūriniai parametrai. Jei šis procesas sėkmingai įvykdomas, galima parašius aukščiausio lygio modulio - testavimo programos kodą, testuoti norimus modulius. Testinėje programoje paskelbiami moduliai, kuriuos norime testuoti. Išėjimo prievadai paskelbiami įėjimo prievadais, kad būtų galima į testuojamų modulių prievadus siųsti signalus. Modulių išėjimai turi būti paskelbti signalais, kad juos galėtume stebėti. Paleidę, kad ir interaktyvios aplinkos *SIMVIR* grafinę signalų atvaizdavimo aplinką, galime pasirinkti programos atpažintus signalus ir pasirinkus testuojamą laiką paleidžiame testavimą.

Gautus rezultatus matome laikinių diagramų pavidalu.

Testavimo programa ir testuojami moduliai:

test.v		testinė programa, apimanti ir PA.
	cpu.v	aukščiausio lygio CPE modulis.
		idec.v KD (kombinatoriais, po CPE moduliu).
		alu.v ALĮ (po CPE moduliu).
		regs.v RF (po CPE moduliu).
		dram.v aminties modelis, sinchroninis 72x8.
	pram.v	atminties modelis, sinchroninis 2048x12.
	exp.v	pavyzdinis išplėtimo modulis (DDS).

Testavimo metu atliekami 9 tipų testai. Jų metu testuojami įėjimai, išėjimai, komandos ir kt.

Test.v programa atlikdama testus skaito iš kelių pagalbinių failų: *basic.rom, dds.rom*, kuriuose yra patalpinti asemblerio kodai.

Taktinis dažnis nustatomas 50 MHz, pagal 10ns pusperiodžius.Paskelbiami registrai ir signalai. Paskelbiami ir inicijuojami komponentai.

Programa, įvykdžius kiekvieną testą, parodo rezultatus. Tai atliekama per išėjimų bei sėkmingų ir nesėkmingų testų skaičiavimą per *num_outputs, num_matches, num_mismatches*.

Testavimas vykdomas šiais etapais:

1. Atliekamas programos nulinis perkrovimas (*reset*).
2. Skaitomas failas *basic.rom.*

Basic.rom failas. Verifikacinė šerdies programa. Kiekvieno testo metu, sėkmingai atlikus testą išmetamas rezultatas *success* arba *fail*.

3. Testas Nr.1. Didinimas ir mažinimas.

Didinamos bei mažinamos kintamųjų reikšmės ir gale testo patikrinima ar galutinis rezultatas teisingas su įrašyta reikšme. Į prievadą B įrašoma reikšmė H'01 sėkmingo testo atveju, o H'F1, jei testas nesėkmingas. Pagal šias reikšmes išmetamos į ekraną *success* arba *fail*.

4. Testas Nr. 2. Sudėtis ir atimtis.

   Testo metu tikrinama sudėtis ir atimtis, taip tikrinamas pernešimas ar pasiskolinimas. Ir analogiškai išmetamas pranešimas.

5. Testas Nr.3. Perstūmimo testas.

   Testo metu atliekamas postūmis į dešinę, postūmis į kairę, tai pat testuojamas *carry* fiksavimas.

6. Testas Nr.4. Testuojamas laikmatis.

   Užkraunamas buferis, po to testuojamas skaitliukas aukštyn žemyn.

7. Testas Nr.5. Įvairios loginės operacijos.

   Testuojamas *or, com, and, xow* ir patikrinami x kintamojo visi bitai.

8. Testas Nr.6. Testuojamas paprogramių veikimas.

   Testo metu paprasčiausiai tikrinamas nuorodų veikimas assemblerio kalboje, ar tam tikroje vietoje esančios aprašytos paprogramės įvykdomos.

9. Testas Nr.7. RF testas.

   Testo metu tikrinamas atminties bankų įrašymas, atminties adresavimas, taip pat netiesioginis adresavimas. Visiems keturiems bankams įrašomos tam tikros reikšmės, nuskaitomos, patikrinamas adresavimas.

10. Testas Nr.8. Prievadų testas.

    Testo metu tikrinamas prievadas A ir prievadas C. Tikrinama kiek laiko išsilaiko reikšmės.

    Prievadas B testuojamas automatiškai, kadangi kiekvieno testo metu parodo rezultatus.

11. Testas Nr. 9. Atliekamas amžinas ciklas.

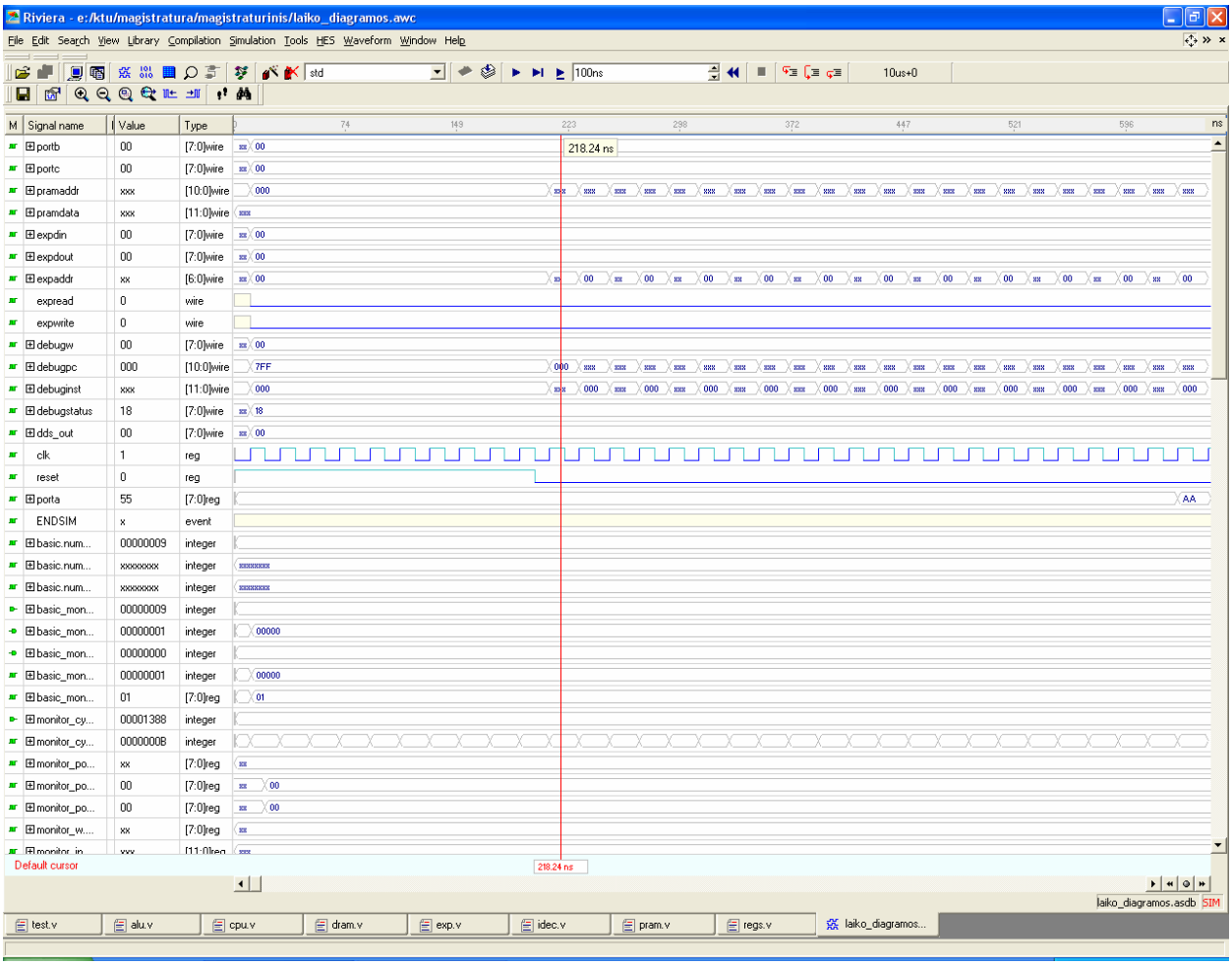
Pastariesiems testams praėjus, testuojama išplėtimo grandinė, kurios pagalba sintezuojamas sinusinis signalas.

Testo seka:

1. Atliekamas programos nulinis perkrovimas (reset).
2. Skaitomas failas *dds.rom*.
3. Į prievadą C paduodam 3 baitų seka, kuri indikuoja moduliuotos sinusoidės veikimą.

Sekančio testo metu patikrinami procesoriaus derinimo registrai, bei keičiant prievadus, jie testuojami.

Toliau yra pradedamas testas komandoms. Paprasčiausiai įvedus 12 bitų komandų dekoderiui skirtą kodą, tikrinama ar teisingai dekoduojamos komandos.



23 pav. Modeliavimo rezultatų atvaizdavimas banginėmis diagramomis

3.2.2. Mikrovaldiklio modulių sintezė

Sintezei naudojamas *Synopsys* kompanijos įrankis ***design_analyzer*.**

Aprašius mikrovaldiklio modulius, juos pratestavus ir patikrinus, atliekama sintezė.

Sintezę atliekame keliais etapais.

- nustatomos technologinės bibliotekos, pvz.: *class.db;*
- nuskaitomi aparatūrine kalba aprašyti moduliai sukuriant schematinį vaizdą;
- nustatomas grandinių modelis, pvz.:05x05;
- nustatomas temperatūrinis diapazonas, maitinimo įtampa ir kt.;
- sukuriamas taktinio dažnio generatorius;
- patikrinamas projektas;
- patikrinamos laikinės charakteristikos;
- detalizuojama schema elementinėje bazėje bei atvaizduojama duotoje technologijoje;
- optimizuojama schema pagal nustatytus kriterijus, pvz.:laikas, plotas, energija;
- užsaugome schemą bei kitus parametrus duomenų bazės failų formatu *.db.*
- užsaugomi realizuoto ir optimizuoti dizaino TPL tipo failai;
- sukuriamos norimos ataskaitos.

Naudojant simbolinę, technologinę biblioteką ***class*,** gauti rezultatai atvaizduoti lentelėje.

8 lentelė.Class bibliotekoje sintezuotų modulių parametrai

Parametras	CPE	ALĮ	KD	RF	DA	IG
Plotas (kombinacinė logika, santykiniais vienetais	8530(3798)	(276)	(183)	6399(2497)	6380(2460)	405(160)
Laikas (Nuo vieno iš įėjimų prievado į vieną iš išėjimų prievadą), ns	426,51	25,02	11,64	228,46	18,83	12,48
Energija, uW	813,19	280	115	2700	2570	58,14
Įtampa, V	4,75					

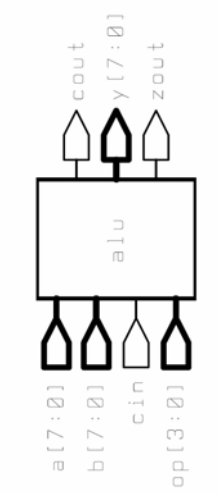
Naudojant simbolinę, technologinę biblioteką ***tc6a_cbacore*****,** gauti rezultatai atvaizduoti lentelėje.

9 lentelė.*Tc6a_cbacore* bibliotekoje sintezuotų modulių parametrai

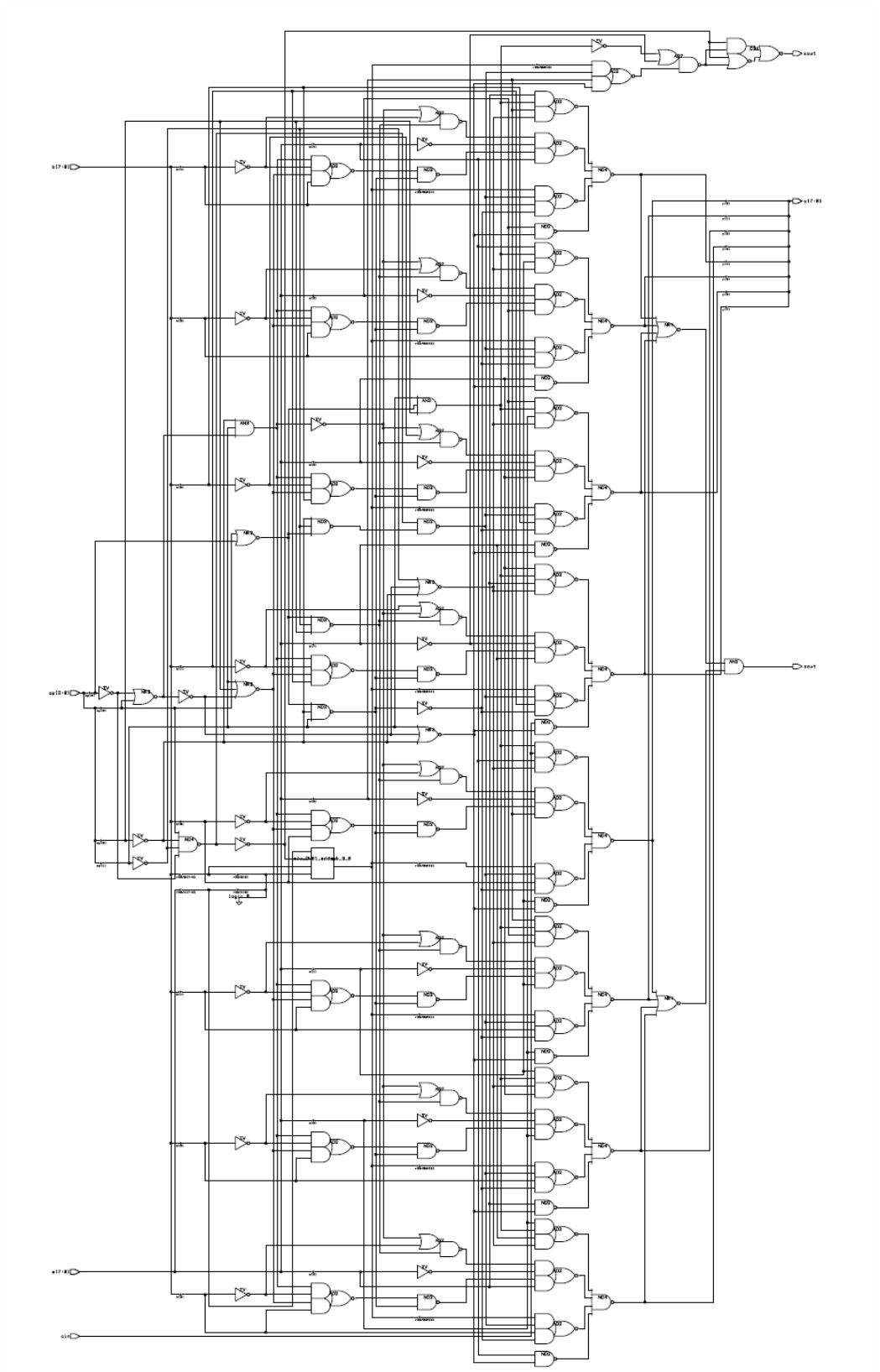
Parametras	CPE	ALĮ	KD	RF	DA	IG
Plotas (kombinacinė logika, santykiniais vienetais	5492(2145)	(198)	(164)	3957(1140)	3937(1120)	246(132)
Laikas (Nuo vieno iš įėjimų prievado į vieną iš išėjimų prievadą), ns	32,84	9,08	4,72	20	6,34	3,47
Energija, mW	250	116	67, 51	1050	956	19.7
Įtampa, V	4,75					

Palyginę lenteles matome, kad naudojant ***tc6a_cbacore*** biblioteką:

- svarbu sintezuojant eksperimentuoti ir parinkti tinkamą technologinę biblioteką, atitinkančią užduotus parametrus;
- sumažėja ne tik bendras elementų plotas, bet ir procentinė kombinacinės logikos dalis;
- kur kas greičiau įvykdomos operacijos, bei apie 3x sutrumpėja maksimalus kelias;
- šios bibliotekos elementai suvartoja kur kas daugiau energijos, ir ji jau matuojama mW;
- daugiausiai ploto ir energijos sunaudoja atmintis, todėl itin racionaliai reikia pasirinkti atminties technologiją;
- daugiausiai laiko užtrunkantis CPE modulis yra RF - -registrų failas, todėl pravartu itin racionaliai išnaudoti registrų operacijas bei jų kiekį.



24 pav. ALĮ modulio atvaizdavimas programinės įrangos *desing_analyzer* pagalba aukščiausiame lygmenyje



25 pav. ALĮ atvaizdavimas ventilių lygmenyje *class* bibliotekoje, realizuojant operacijų resursų dalinimąsi.

4. *VERILOG* KALBOS SINTEZUOJAMOS KONSTRUKCIJOS *SYSTEMC* KALBOJE

Loginė sintezė – tai aukšto lygio aprašymo konvertavimo procesas į optimizuotą ventilių lygio logiką. Anksčiau projektavimo procesas trukdavo ilgai. Viskas buvo daroma rankiniu būdų: piešiamos principinės schemos, optimizuojama logiką ir kt. Tai veikia normaliai, tik kai projektuojame keletą šimtų ventilių. Bet kai šiuolaikinėje elektronikoje šis kiekis matuojamas tūkstančiais bei milijonais be automatizuotų projektavimo priemonių neišsiverčiama. Tačiau čia vėl gi susiduriame su problema, kadangi visose aparatūros aprašymo kalbose apstu nesintezuojamų konstrukcijų, reikia laikytis taisyklingo kodavimo. Ir visa tai taikoma tam, kad optimizuoti dizainą, sumažinti resursus ir pagreitinti prekės pateikimo laiką į rinką.

4.1. Skirtumai tarp modeliavimo ir sintezės.

Modeliuojant naudojamos visos konstrukcijos tiek *Verilog*, tiek *SystemC* kalbose. Visos jos naudojamos testuojant, tačiau sintezuojamų konstrukcijų yra pakankamai mažai ir neatsižvelgus į tai nuo pat projektavimo pradžios, vėliau tenka naudoti pakankamai daug resursų taisant kodą į sintezuojamą.

SystemC kalboje aukščiausiojo lygmens modulį atstoja *sc_main* funkcija, kuri dažniausiai aprašoma *main.cpp* faile, kuriame tik paskelbiami moduliai ir nurodomi tam tikri argumentai. Todėl mikrovaldiklio bendrąjį funkcionalumą tenka aprašinėti papildomame modulyje, kai tuo tarpu *Verilog* kalboje tai galima atlikti viename modulyje.

SystemC kompileris sintezuoja procesus modulyje ir jų sąsajas su kitais moduliais priklausomai nuo aprašyto modulio. Norint sintezuoti hierarchinius modulius reikia naudoti TPL sintezę.

4.2. Sintezuojamų konstrukcijų ypatumai.

Yra keletas kelių kaip sintezės metu suprantama konstrukcija:

- kodas pilnai sintezuojamas, modeliavimo ir sintezės rezultatai sutampa;
- kodas nepilnai sintezuojamas, kadangi yra tam tikrų klaidingų ar ignoruojamų konstrukcijų, todėl modeliavimo ir sintezės rezultatai gali nesutapti;
- kodas nesintezuojamas, kadangi yra nesintezuojamos konstrukcijos.

Tiek *Verilog,* tiek *SystemC* kalbose sintezuojamų konstrukcijų nėra daug. Jas galima suskirstyti į:

- pilnai sintezuojamas;
- ignoruojamas;
- nesintezuojamas.

Prievadai.

Verilog kalboje palaikomi *input, output, inout*. SystemC kalboje palaikomi *sc_in, sc_out* prievadai.

Signalai ir kintamieji. Duomenų tipai.

Verilog kalboje palaikomi *wire, reg tri* taip pat ir vektorinio tipo. *Verilog* kalbos signalo aprašą *wire SystemC* kalboje atitinka *sc_signal*. Šis tipas abiejose kalbose suprantamas kaip tam tikras signalas jungiantis procesus, nes modulius jungia prievadai.

Verilog kalbos konstrukciją *reg*, turinčią atmintį *SystemC* kalboje atitinka taip pat bet kokio sintezuojamo duomenų tipo kintamasis su atminčia.

Pavyzdys *SystemC* kalbos kintamojo su atmintimi:

```
//Duomenų kintamojo paskelbimas, kuris turės atmintį
int count_val; // vidinis skaitiklis
sc_int<8> mem[1024]; // sc_int masyvas
```

Priskyrimus šiems kintamiesiems galima atlikti ir konstruktoriuje *SC_CTOR*.

Konstrukcijos *tri* atitikmens *SystemC* kalboje nėra, todėl trijų būsenų buferį reikia aprašinėti atskirai.

SystemC sintezuojamų duomenų tipai:

- *sc_bv, sc_int, sc_uint, sc_bigint, sc_biguint, bool, int, unsigned int, long, unsigned long, char, unsigned char, short, unsigned short, struct, enum.*

Verilog kalbos sintezuojamų duomenų tipai:

- visi *integer* tipo, vektoriniai tipai, loginiai.

Procesai.

Verilog kalboje yra vienintelis sintezuojamas procesas, kuris turi būti būtinai sinchronizuotas *@, posedge, negedge* operatoriais. Niekada negalima sumaišyti lygio, fronto bei skirtingų frontų sinchronizavimų viename procese.

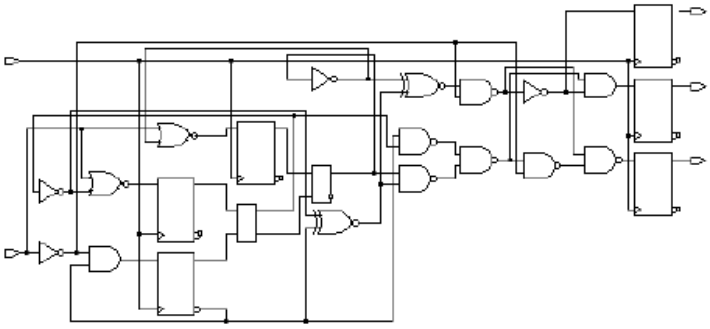
Always @ (“jautrumo sąrašas)

Naudojamas modeliuoti tiek kombinacinei, tiek nuosekliai logikai. Šiame bloke, kuris yra sinchronizuotas su TDG, kiekvienas kintamasis yra saugomas *flip-flop* tipo trigeryje.

Norint išvengti papildomų registrų, nesinchronizuotus priskyrimus reikia iškelti į kitą bloką.

Pavyzdys modulio, kuriame realizuoti 6 registrai. Visi jie aprašyti *always* bloko viduje:

```
module count (clock, reset, and_bits, or_bits, xor_bits);
input clock, reset;
output and_bits, or_bits, xor_bits;
reg and_bits, or_bits, xor_bits;
reg [2:0] count;
always @(posedge clock) begin
if (reset)
count = 0;
else
count = count + 1;
and_bits = & count;
or_bits = | count;
xor_bits = ^ count;
end
endmodule
```



26 pav. Realizuoto kodo sintezė šešiais registrais

Taip pat nuo papildomų registrų saugo papildomas kintamojo priskyrimas iškeliant už vykdomojo sakinio ribų:

```
always @ (in)
            begin
                  out = 0;
                  case (in)
                        3'b001 : out = 8'b0000_0001;
                        3'b010 : out = 8'b0000_0010;
                        3'b011 : out = 8'b0000_0100;
                        3'b100 : out = 8'b0000_1000;
                        3'b101 : out = 8'b0001_0000;
                        3'b110 : out = 8'b0100_0000;
                        3'b111 : out = 8'b1000_0000;
                  endcase
            end
```

SC_CTHREAD.

Verilog kalbos *always* konstrukcija realizuotą sintezuojamą procesą SystemC kalboje atitinka *SC_CTHREAD* procesas. Procesų paskelbimas, jautrumo sąrašai bei sinchronizavimas atliekamas konstruktoriuje. Šis procesas, skirtingai nuo kitų S*ystemC* kalbos procesų, jautrus yra TDG signalui, o ne jautrumo sąrašui. Praktiškai rezultatas gaunamas toks pat tiek *SystemC* kalboje, tiek *Verilog*, tik visa tai skirtingai realizuojama.

Skirtingai nuo *Verilog* kalbos, kur *wait* operatorius nesintezuojamas, čia jis turi būti nurodomas kiekviename procese, kad sustabdyti jį iki kito aktyvaus sinchronizuoto TDG signalo.

Sinchronizuoto proceso *SystemC* kalboje paskelbimas nurodant aktyvų įėjimo frontą:

```
SC_CTOR(bios) {
      SC_CTHREAD(entry, CLK.pos());
```

***SystemC* konstruktorius *SC_CTOR*.**

SystemC konstruktorius – tai vieta, kurioje vykdoma procesų registracija, deklaruojami procesų jautrumo sąrašai, paskelbiami moduliai ar prievadų jungimosi deklaracijos. Svarbu yra konstruktoriuje neatlikinėti jokių papildomų priskyrimų ar veiksmų su kitomis konstrukcijomis, išskyrus kintamųjų su atmintimi priskyrimus.

Konstruktoriaus aprašo pavyzdys:

```
  SC_CTOR(modulio_vardas) /* : paskelbiamas sąrašas */ {
    // procesų registracija
    // jautrumo sąrašo deklaravimas
    // modulių paskelbimas
    // portų jungimosi deklaracija
  }
};
```

Ciklai.

For, while, forever ciklai sintezuojami *Verilog* kalboje. *While bei forever* turi būti būtinai sinchronizuoti *@(posedge clk) or @(negedge clk)* operatorių pagalba.

Sakiniai.

*Case***.**

Nepabaigti specifikuoti sakiniai kompiliuojant prideda papildomų trigerių, tačiau *Verilog* kalbos atveju galima nurodyti *full_case* direktyvą kompiliuojant ir kompiliatorius sakinį supranta kaip pilną. Pavyzdys *casex* sakinio *Verilog* kalboje:

```
always @(inst) begin
      casex (inst) // synopsys parallel_case
            12'b0000_0000_0000: decodes = 15'b00_00_0000_0_0_0_0_0_0_0; // NOP
            12'b0000_001X_XXXX: decodes = 15'b00_00_0010_0_1_0_0_0_0_0; // MOVWF
            12'b0000_0100_0000: decodes = 15'b00_00_0011_1_0_1_0_0_0_0; // CLRW
            12'b0000_011X_XXXX: decodes = 15'b00_00_0011_0_1_1_0_0_0_0; // CLRF
            12'b0000_100X_XXXX: decodes = 15'b01_00_1000_1_0_1_1_0_0_0; // SUBWF (
            ...
            default:
                  decodes = 15'b00_00_0000_0_0_0_0_0_0_0;
      endcase
     end
```

Verilog kalboje *casex* bei *casez* - tai *case* sakinių modifikacijos panaudojant „z“ bei „x“ reikšmes. Rašant tokį kodą kaip pateikta pavyzdyje, taupomi resursai bei nereikia aprašinėti visų variantų. Nors sintezuojant aparatūra gali priimti, kad „x“ reikšmės tai *false* šaka, tačiau naudojant *full_case* ar *parallel_*case direktyvą galima išvengti šių trūkumų. *SystemC* kalboje naudojant *case* sakinį tenka aprašinėti visas galimas sąlygas.

If.

Nebaigti specifikuoti sakiniai kompiliuojant prideda papildomų trigerių. Tiek *SystemC* kalboje, tiek *Verilog* rašant sintezuojamą kodą būtina aprašyti visas sąlygas tame tarpe ir *else* šaką. *Verilog* kalboje čia taip pat galimas „x“ bei „z“ reikšmių naudojimas.

Assign.

Labai patogus sakinys modeliuojant kombinacinę logiką. Tiek trijų būsenų buferį, tiek multiplekserį yra pakankamai nesudėtinga aprašyti. Taip pat juo galima aprašyti suvedimą į vieną tašką.

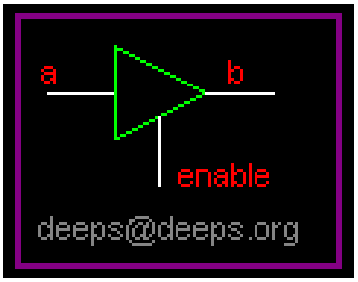
Bet šis sakinys sintezuojamas tik su *wire* tipo reikšmėmis.

Trijų būsenų buferio realizacijos pavyzdys:

```
module tri_buf (a,b,enable);
 input a;
 output b;
 input enable;
 wire b;

assign b = (enable) ? a : 1'bz;

endmodule
```



27 pav. Trijų būsenų buferis

Šiuo sakiniu galima sukurti multiplekserius, signalų sujungimus.

4.3. Sintezuojamų konstrukcijų dviprasmiškumai.

„X“, „Y“ reikšmių naudojimas modeliavime, sintezėje.

Panaudojant šias reikšmes galima smarkiai sumažinti plotą. Geriausiai panaudojama sakiniuose „pagal nutylėjimą“. Pvz.: *Verilog casex, casez* sakiniai, *if* sakinio *else* šakoje.

Kai kuriais atvejais šios reikšmės gali sukelti tam tikrų problemų.

- šios reikšmės sukelia tikimybę nesutapimams tarp modeliavimo ir sintezės rezultatų;
- taip pat gali paslėpti tam tikras klaidas kintamiesiems.

Kadangi simuliatorius supranta „x“ kaip „1“ arba „0“, o sintezuojant priskiriama arba „0“ arba „1“, išeitis yra priskirti šias reikšmes tik išėjimams ir įsitikinti, kad išėjimo prievadai neskaitomi vidinių resursų.

Grįžtamieji ryšiai ir trigeriai. Kartais sintezuojamas *Verilog* kalbos kodas gali realizuoti grįžtamuosius ryšius ar papildomus trigerius. Tai atsitinka, kai kombinatorinės logikos bloke signalas ar kintamasis nėra pilnai apibrėžtas. Pvz.: *always* blokas be *posedge ar negedge*. Kintamasis ar signalas yra tik tada pilnai apibrėžiami, kai jiems priskiriamos reikšmės prie visų įmanomų būsenų.

Sinchronizavimas. Sinchronizuotame dizaine, visi registrai naudoja tą patį TDG. Šis signalas turi būti pirminis įėjimas. Tokiame projektavimo stiliuje nėra nei grįžtamųjų ryšių, nei vėlinimo linijų. Sinchronizuojant atliekama ta pati funkcija priklausomai nuo TDG tol, kol per visą kelią t.y. visus registrus perduodami signalai.

Synopsys sintezės priemonėse yra ribotas asinchroninio dizaino palaikymas. Bendriausias kelias tam padaryti yra susieti registrus ventilių lygmenyje prie TDG. Bet šią techniką naudojant galimas modeliavimo ir sintezės rezultatų nesutapimas, kadangi *Synopsys design compiler* kompiliuojant asinchroninį dizainą nepateikia perspėjimų, todėl tokiu atveju reikalinga verifikuoti pačiam.Tokiems atvejams dažniausiai naudojama arba „paleidimo“ (enable), arba asinchroninio „reset“ linija.

Taip pat nepalaikomos kai kurios tokios technikos formos kaip kad trigubas neigimas ir kt.

Sinchronizuotuose procesuose reikia naudoti neblokuojančius priskyrimus. Pvz.:

```
always @ (posedge clock)
q <= d;
```

Nepilni jautrumo sąrašai gali paveikti sintezės rezultatus, kadangi neįtraukus į sąrašą atitinkamo signalo, sintezuojant, aparatūra bus jau nereguojanti į šį signalą.

Nereikalingi skaičiavimai cikluose. *For* cikluose naudojant priskyrimus, kurie nesikeičia sukantis ciklui papildomai apkraunamas kompileris optimizuojant esamus resursus, todėl geriau juos iškelti už ciklo ribų.

Pavyzdys, kai for ciklas naudojamas teisingai:

```
for(i =0; i<32; i++){
      printf("  R%2d(%08x)  ",i, cpu_reg[i]);
            if ((i==3) || (i== 11) || (i==19) || (i== 27) ||(i==7) || (i==15) ||
            (i==23) || (i==31)){
      printf("\n");
```

Pavyzdys, kai for ciklas naudojamas neteisingai:

```
for(i =0; i<32; i++){
      sig1 = sig2;
      printf("  R%2d(%08x)  ",i, cpu_reg[i]);
            if ((i==3) || (i== 11) || (i==19) || (i== 27) ||(i==7) || (i==15) ||
            (i==23) || (i==31)){
      printf("\n");
                              }
                  }
```

Resursų dalinimasis. Resursai sintezės metu dalinami tik tada, kai jie panaudojami *if* ar *case* sakinių šakose. Tokiame sakinyje dalinimosi nebus:

```
z = (a>b)?(a+b):(c+d);
```

Tokiame sakinyje dalinimasis bus:

```
if (cond)
z = a+b;
else
z = c+d;
```

Kombinacinėje logikoje kaip parodyta pavyzdyje reikia naudoti blokuojančius priskyrimus.

4.4. Nesintezuojamos *Verilog* bei *SystemC* kalbų konstrukcijos

Reikia pastebėti, kad nesintezuojamos konstrukcijos yra šios:

- **vėlinimai.**

  Norint realizuoti vėlinimus, realiausia tai padaryti per buferius. Paprasčiausiai tokia konstrukcija: *a=#10 b* bus suprantama kaip a=b;

- **„z“ ir „x“ reikšmių lyginimas.**

  ```
  if ((b == 1'bz) || (b == 1'bx)) begin
   a = 1;
  ```

  Šio tipo sakinys bus paprasčiausiai ignoruojamas;

- priskyrimas išrinktų bitų kairėje priskyrimo pusėje;
- globaliniai kintamieji.

Šioje lentelėje aprašytos visiškai nesintezuojamos Verilog kalbos konstrukcijos:

10 lentelė.Nesitezuojamos *Verilog* kalbos kosntrukcijos

Konstrukcija	**Pastebėjimai**
initial	Naudojama tik testinėse programose.
events	Naudojama sinchronizavimui testinėse programose.
real	Nepalaikomas duomenų tipas.
time	Nepalikomas tipas.
force, release	Nepalaikoma.
fork join	Naudojant neblokuojančius priskyrimus galima gauti tą patį efektą.
primitives	Tik ventilių lygio *primitives* yra palaikomi
table	Nepalaikoma.
cnmos, nmos, rcmos, pmos, rpmos	Nepalaikoma.
defparam	Nepalaikoma.
force	Nepalaikoma.
pullup, pulldown	Nepalaikoma.
release	Nepalaikoma.
repeat	Nepalaikoma.
rtran, tran, tranif	Nepalaikoma.

Ignoruojamos *Verilog* kalbos konstrukcijos:

- laikinės specifikacijos;
- *specify;*
- *weak1, weak0 ...*
- *$keyword* (kai kurių sintezės įrankių suprantam kaip konstruktoriaus nustatymai);
- wait (kai kurie sintezės įrankiai supranta šią konstrukciją su apribojimais).

Šioje lentelėje pateiktos nesintezuojamos *SystemC* konstrukcijos:

11 lentelė.Nesitezuojamos *SystemC* kalbos konstrukcijos

Tipas	**Konstrukcija**	**Komentaras**	**Rekomendacijos**
Procesai	*SC_THREAD* *SC_METHOD*	Naudojamas modeliavimui, testavimui elgsenos lygmenyje. Naudojamas RT lygmenyje.	Keisti i *SC_CTHREAD*
Kanalai	*sc_channel*	Naudojamas tik modeliavime	Naudoti sc_signal
TDG	*sc_start()*	simuliacija	Nesintezuojama
Komunikacija	*sc_interface, sc_port, sc_mutex, sc_fifo*	Modeliuoti komunikacijai	Nesintezuojama
Sinchronizacija	Master-slave biblioteka	Naudojama įvykių sinchronizacijai	Nesintezuojama
Sekimas	*Sc_trace, sc_create*	Kuria bangines diagramas	Nesintezuojama
Dinaminė atmintis			Pakeisti į statinę
Išimtinų situacijų valdymas	*Try, catch, throw*		Nesintezuojama
Funkcijų perdengimas		Tik *SystemC* klasėms	Pakeisti į unikalius vardus
Paveldimumas			Nesintezuojama
C++ integruotos funkcijos	Matematinės bibliotekos, iėjimų/išėjimų bibliotekos		Nesintezuojama
Kreipimasis į *Struct* narius su „->“ operatoriumi	-> operatorius	Neleidžiama	Pakeisti į (.)
Statiniai nariai		Neleidžiama	Pakeisti nestatiniais
Perdavimo operatoriai	* ir &	Neleidžiama	Pakeisti į tiesioginį kreipimąsi
Ciklo operatorius (,)		Neleidžiama	Išimti iš ciklo.
Žymeklis	*	Tik hierarchiniuose moduliuose. **char* suprantama kaip eilutė, ne kaip žymeklis. Taip pat neleidžiama konversija	Pakeisti priėjimu prie elementų
Žymeklis	this	Neleidžiama	
Nuoroda	&	Parametrų perdavimas į funkciją	Kitais atvejais pakeisti
Unions		Neleidžiama	Pakeisti į *struct*.

Bendros rekomendacijos sintezei:

- rekomenduojama kruopščiai nustatyti struktūrą, duomenų tipus bei kontrolę, nes kaip žinome sintezuojamų konstrukcijų nėra daug, todėl tai iškeltų papildomų resursų poreikį;
- tiek S*ystemC*, tiek *Verilog* kalbos turi pakankamai daug nesintezuojamų konstrukcijų, todėl išskirti vieną iš jų pakankamai sunku. Tačiau imant atskiras pranašumus galima pasirinkti poreikius atitinkančią kalbą;
- *Verilog* kalba iš *C++* kalbos perėmusi tik sintaksę, bet turimomis papildomomis modeliavimo funkcijomis operatoriais bei konstrukcijomis labiau panaši į struktūrinę-aparatūrinę kalbą, paprastesnė programuoti ir turinti pakankamai panašias galimybes kaip ir *SystemC*;
- *SystemC* kalba iš *C++* kalbos perėmusį ir sintaksę, ir funkcijas, ir struktūrą: *.h* bei *.cc*, turi daug daugiau galimybių modeliavime, tačiau perėjus į sintezės fazę ji tampa labai panaši į *Verilog* kalbą ir tenka susidurti su nesintezuojamų konstrukcijų problemomis, ko nepasakysi apie duomenų tipus. Tačiau ateitis priklauso *SystemC*;
- kiek žinome slankaus kablelio tipo nepalaiko jokie sintezės įrankiai ar kalbos, todėl rekomenduojama būtų naudotis *Synopsys* kompanijos optimizavimo rekomendacijomis kaip aprašyti fiksuoto kablelio modulių funkcionalumą.

5. IŠVADOS

1. Norint efektyviai išnaudoti resursus svarbu būti įsisavinus modeliavimo ir sintezės programas, kadangi skirtingos programos gali skirtingai vertinti jūsų parašytą kodą ir tokias savybes, kaip kad programavimo kalbos ir modeliavimo programos paskutinės versijos. Geriausius rezultatus bei produktyvumą galima gauti dirbant su *Cadence* ir *Synopsys* kompanijų produktais *LDV-5.1, design analyzer*.
2. Kuriant programą aukštesniame abstrakcijos lygmenyje pvz.: elgsenos lygyje sutaupomas laikas, programa tampa profesionalesnė, lengviau pakeičiama ir pritaikoma kitiems tikslams, tuo pačiu užkertamas kelias elementarioms konstruktoriaus klaidoms. Galima pasiekti geresnes laiko, ploto ir kitas charakteristikas. Reikia pastebėti, kad elgsenos lygmenyje sunkiau atvaizduoti funkcinius lygmenis, bet tai gali būti atliekama TPL lygmenyje, kadangi yra galimas šių lygių sumaišymas, bet aukščiausio lygio išlieka TPL modulis.
3. Svarbu iš karto žinoti, kokių konstrukcijų, operatorių ar funkcijų negalima naudoti, kad po to nebūtų problemų su sinteze, nes kuo sudėtingesnė konstrukcija, kuo sudėtingesnis aprašas, tuo didesnė tikimybė, kad aprašas nebus sintezuojamas ar kurios tai jo dalys.
4. Prieš kuriant pravartu žinoti, kokias konstrukcijas naudosime, kad galėtumėme pritaikyti bibliotekas, pvz.: duomenų atmintį projektuojant ir vykdant sintezę standartinėje „class“ bibliotekoje atvaizduojama paprastai trigeriais, o panaudojant įrankį memory_wrappers galima atmintį atvaizduoti kitais elementais sutaupant vietą ir gerinant funkcionalumą.
5. Sintezuojamų konstrukcijų V*erilog* kalba yra labai panaši į *SystemC* kalbą lyginant sintezuojamas konstrukcijas, nors ir daug seniau sukurta nei pastaroji. *SystemC* kalboje yra daugiau sintezuojamų duomenų tipų, nors dauguma *C++* paveldėtų privalumų neįmanoma sintezuoti.
6. Modeliuojant *SystemC* kalba yra kur kas pranašesnė nei *Verilog.* Automatinis TDG sukurimas, paveldėti *C++* kalbos privalumai, duomenų srautų valdymas suteikia *SystemC* pranašūmų, tačiau *Verilog* kalbos konstrukcijos tranzistoriniam lygmeniui atvaziduoti, *wire, reg tipų* kombinacijos, kalbos paprastumas ir lankstumas verčia susimąstyti renkantis kalbą. *Verilog* kalba paveldėjusi iš C tik sintaksę, o *SystemC* paveldėjo ir struktūrą, ir sintaksę, ir semantiką.
7. Abiejose kalbose vis dar labai daug skirtumų tarp modeliavimo ir sintezės, taip pat labai daug dar problemų ir su tomis pačiomis sintezuojamomis konstrukcijomis, kadangi kiekviena, kad ir labai maža nežinoma sintezės kosntrukcijų ypatybė lemia resursų padidėjimą.

LITERATŪRA

[1] Aboulhadim El Mustapha, Mike Baird ir kt. SystemC 2.0.1 language reference manual. Revision 1.0. Open SystemC, 2003. Prieiga per internetą: www.cs.tus.de/eis/klingauf/systemc/SystemC_LRM.pdf.

[2] Cadence LDV 5.0 Nclaunch online documentation. Cadence, 2003.

John G. Elias. Verilog XL. 1993. Prieiga per internetą: http://www.eecis.udel.edu/~elias/verilog/verilog_manuals/chap_1.pdf.

[3] Jusas V., Bareiša E., Šeinauskas R.. Skaitmeninių sistemų projektavimas VHDL kalba. – K: Technologija, 1997 – 206p.

[4] Keating M., Bricaud P. Reuse methology. Manual. II edition. 2001. – 300 p.

[5] Leila Mahmoudi Ayough. Verilog2SC: A Methology for converting Verilog HDL to SystemC. 2002. Prieiga per internetą: https://www.systemc.org/docman2/ViewCategory.php?group_id=4&category_id=2

[6] Levitan S.P. SoC design. 2003. Prieiga per internetą:http://www.engr.pitt.edu/eecourses/2140/CourseNotes/Overview/SlideSet1.pdf.

[7] Open cores. Prieiga per internetą: www.opencores.org.

[8] Regan O Regan. Verilog synthesis methology, 2001. Prieiga per internetą: http://www.et5.tu-harburg.de/Lehre/Praktikum/Cadence_digital/Synthesizable%20code.pdf.

[9] Synopsys online documentation. SOLD U-2003.06, Volume 1. Synopsys, 2003.

[10] Sylvia Liu, Savitha Gandikota, Mohammad Shallal ir kt. Synopsys synthesis tutorial. 2002. Prieiga per inernetą: http://www.scudc.scu.edu/mentortu/synopsys_tutorial.html.

[11] Synopsys. Galaxy design platform. Desing compiler. 2005. Prieiga per internetą: http://www.synopsys.com/products/logic/design_compiler.html.

1 PRIEDAS. Kompaktinis diskas.

Šiame diske pateikti *Verilog* bei S*ystemC* kalbomis aprašytų modulių failai, testinės programos, sintezuotos konstrukcijos, *.db* failai, sintezuoti TPL failai.

2 PRIEDAS. Synopsys programos skriptai

Kaip jau anksčiau minėta yra trys aukščiausio lygio moduliai – *cpu.v, pram.v ir exp.v.* Todėl yra trys pagrindiniai skriptai programos vykdymui.

Skriptų turinys:

CPE modulio:

Failas ***main_cpu.script:***

```
/* Projekto aprasymu transliavimas i sistemos formata (*.db) */

read -format verilog {"./alu.v"}
read -format verilog {"./idec.v"}
read -format verilog {"./regs.v"}
read -format verilog {"./dram.v"}
read -format verilog {"./cpu.v"}
write -format db -hierarchy -output "./db/CPU.db"

create_schematic -size infinite -symbol_view

/* Signalu grandiniu modelis*/
set_wire_load "05x05" -library "class"

/* temperatr diapozonas, maitinimo itampa ir kt..*/
set_operating_conditions -library "class" "WCCOM"

/* Uzsaugome projekta*/
write -format db -hierarchy -output "./db/CPU_attributes.db" {"CPU.db:cpu"}

/* ismetame visas schemas is atminties*/
remove_design -all

/* atidarome is DB schema */
read -format db {"./db/CPU_attributes.db"}
```

```
    create_schematic -size infinite -gen_database
    /* Create a clock object */
    create_clock -name CLK -period 15 -waveform { 0 20 } { CLK }

    /* Run check design */
    check_design

    /* Run check timing */
    check_timing

    /* Use uniquify to resolve multiple design instances*/
    uniquify
    create_schematic -size infinite -gen_database

    /* Uzsaugome projekta*/
    write -format db -hierarchy -output "./db/CPU_constraints.db"
{"CPU_attributes.db:cpu"}


    /* ismetame visas schemas is atminties*/
    remove_design -all

    /* atidarome is DB schema */
    read -format db {"./db/CPU_constraints.db"}
    compile -map_effort medium -verify -verify_effort low -boundary_optimization
    create_schematic -size infinite -schematic_view -symbol_view -hier_view

    /* Save the optimized design*/
    write -format db -hierarchy -output "./db/CPU_compiled.db"
{"CPU_constraints.db:cpu"}
    write -format verilog -hierarchy -output "./verilog/CPU_compiled.v"
    create_schematic -size infinite -schematic_view -symbol_view -hier_view

    report_bus
    report_cell
    report_net
    report_compile_options
    report_hierarchy
    report_timing
```

PA modulio:

Failas ***main_pram.script:***

```
/* Projekto aprasymu transliavimas i sistemos formata (*.db) */

read -format verilog {"./pram.v"}
write -format db -hierarchy -output "./db/pram.db"

/* atidarome is DB schema */

read -format db {"./db/pram.db"}
create_schematic -size infinite -symbol_view

/* Signalu grandiniu modelis*/

set_wire_load "05x05" -library "class"

/* temperatr diapozonas, maitinimo itampa ir kt..*/

set_operating_conditions -library "class" "WCCOM"

 /*Uzsaugome projekta*/

write -format db -hierarchy -output "./db/pram_attributes.db"

 /*ismetame visas schemas is atminties*/

remove_design -all

/* atidarome is DB schema*/

read -format db {"./db/pram_attributes.db"}
```

```
current_design "pram"

create_schematic -size infinite -gen_database

 /*Use uniquify to resolve multiple design instances*/

uniquify

create_schematic -size infinite -gen_database
current_design "pram"

/*Uzsaugome projekta*/

write -format db -hierarchy -output "./db/pram_constraints.db"


/* ismetame visas schemas is atminties*/

remove_design -all

/*atidarome is DB schema*/

read -format db {"./db/pram_constraints.db"}

compile -map_effort medium -verify -verify_effort low -boundary_optimization

create_schematic -size infinite -schematic_view -symbol_view -hier_view
current_design "pram"

/* Save the optimized design*/

write -format db -hierarchy -output "./db/pram_compiled.db"
write -format verilog -hierarchy -output "./verilog/pram_compiled.v"

create_schematic -size infinite -schematic_view -symbol_view -hier_view
current_design "pram"

report_bus
report_cell
report_net
```

```
report_compile_options
report_hierarchy
report_timing
```

IG modulio:

Failas ***main_exp.script:***

```
/* Projekto aprasymu transliavimas i sistemos formata (*.db) */

read -format verilog {"./exp.v"}
write -format db -hierarchy -output "./db/exp.db"

/* atidarome is DB schema */

read -format db {"./db/exp.db"}
create_schematic -size infinite -symbol_view

/* Signalu grandiniu modelis*/

set_wire_load "05x05" -library "class"

/* temperatr diapozonas, maitinimo itampa ir kt..*/

set_operating_conditions -library "class" "WCCOM"

 /*Uzsaugome projekta*/

write -format db -hierarchy -output "./db/exp_attributes.db"

 /*ismetame visas schemas is atminties*/

remove_design -all

/* atidarome is DB schema*/

read -format db {"./db/exp_attributes.db"}

current_design "exp"
```

```
create_schematic -size infinite -gen_database

 /*Use uniquify to resolve multiple design instances*/

uniquify

create_schematic -size infinite -gen_database
current_design "exp"

/*Uzsaugome projekta*/

write -format db -hierarchy -output "./db/exp_constraints.db"


/* ismetame visas schemas is atminties*/

remove_design -all

/*atidarome is DB schema*/

read -format db {"./db/exp_constraints.db"}

compile -map_effort medium -verify -verify_effort low -boundary_optimization

create_schematic -size infinite -schematic_view -symbol_view -hier_view
current_design "exp"

/* Save the optimized design*/

write -format db -hierarchy -output "./db/exp_compiled.db"
write -format verilog -hierarchy -output "./verilog/exp_compiled.v"
create_schematic -size infinite -schematic_view -symbol_view -hier_view
current_design "exp"
report_bus
report_cell
report_net
report_compile_options
report_hierarchy
report_timing
```