KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS
PROGRAMŲ INŽINERIJOS KATEDRA

Mindaugas Mikulis

Procesorinio komponento bendrinimo tyrimas: analizės aspektai

Magistro darbas

Vadovas
doc. dr. Giedrius Ziberkas

Kaunas, 2007

KAUNO TECHNOLOGIJOS UNIVERSITETAS

INFORMATIKOS FAKULTETAS

PROGRAMŲ INŽINERIJOS KATEDRA

Mindaugas Mikulis

Procesorinio komponento bendrinimo tyrimas: analizės aspektai

Magistro darbas

Recenzentas

prof. dr. Vacys Jusas

2007-05-

Vadovas

doc. dr. Giedrius Ziberkas

2007-05-

Atliko

IFM-1/5 gr.stud.
Mindaugas Mikulis
2007-05-

Kaunas, 2007

TURINYS

LENTELIŲ SĄRAŠAS 5
PAVEIKSLŲ SĄRAŠAS 6
ĮVADAS 10
1. PROBLEMOS ANALIZĖ.. 12
1.1. Projektavimo etapo problematika 12
1.2. Atkartojimo technologija 14
1.2.1. Pagrindinių terminų apibrėžimai 14
1.2.2. Kas yra komponentas 15
1.2.3. Kas yra generatorius 17
1.2.4. Kas yra ruošinys 18
1.3. Motyvacija, kodėl naudoti pakartotinio naudojimo metodiką 18
1.3.1. Atkartojimas gali būti pasiektas naudojant metaprogramavimą 20
1.4. Mikroprocesoriai 21
1.4.1. Mikroarchitektūrinė koncepcija 23
1.4.2. Instrukcijų rinkinio architektūros pasirinkimas 23
1.4.3. Instrukcijų lygiagretinimas 24
1.5. Procesorinio komponento architektūra 25
1.5.1. Procesoriaus komponentinės dalys 25
1.5.2. CISC architektūra 27
1.5.3. RISC architektūra 27
1.5.4. MISC architektūra 28
1.5.5. VLIW architektūra 28
1.5.6. EPIC architektūra 29
1.5.7. Flynn taksonomija 30
1.5.8. CISC prieš RISC 36
1.6. Apžvalgos išvados 37
2. PROJEKTINĖ DALIS 39
2.1. Atskirų procesoriaus komponentų bendrinimo kryptis 39
2.1.1. ACC komponentas 41
2.1.2. ALU komponentas 42
2.1.3. CTRL komponentas 43
2.1.4. IAR komponentas 45
2.1.5. IR komponentas 46

2.1.6. PC komponentas ... 47
2.1.7. PORT komponentas ... 48
2.1.8. RAM komponentas ... 49
2.1.9. REG komponentas ... 49
2.1.10. CPU komponentas ... 50
2.1.11. CORE komponentas ... 52
2.1.12. SYSTEM komponentas ... 52
2.2. Instrukcijų dekoderio ir valdymo įrenginio bendrinimo kryptis ... 53
3. TYRIMAS ... 56
3.1. Tyrimo metodika ... 56
3.2. Procesoriaus komponentų bendrinimo krypties rezultatai ... 56
3.3. Instrukcijų registro ir valdymo įrenginio bendrinimo krypties rezultatai ... 61
3.3.1. IR ir CTRL komponentų bendrinimo rezultatai (VHDL) ... 61
3.3.2. IR ir CTRL komponentų bendrinimo rezultatai (SystemC) ... 62
4. IŠVADOS ... 64
LITERATŪRA ... 66
PRIEDAI ... 68
1. Priedas. Bendrinių komponentų pavyzdinis VHDL kodas ... 68
2. Priedas. CTRL ir IR komponentų pavyzdinis SystemC kodas ... 82
3. Priedas. Bendrintų CTRL ir IR komponetų pavyzdinis SystemC kodas. ... 90
4. Priedas. PIC10F200 blokinė diagrama ... 98
5. Priedas. PIC10F200 instrukcijų rinkinys ... 98
6. Priedas. ATtiny25 blokinė diagrama ... 100
7. Priedas. ATtiny25 instrukcijų rinkinys ... 100

LENTELIŲ SĄRAŠAS

1 lentelė. Instrukcijų rinkinys 40
2 lentelė. ALU operacijos 43
3 lentelė. Peršokimų instrukcijos 54
4 lentelė. Bendrinto ir nebendrinto komponento sintezės rezultatų palyginimas 57
5 lentelė. Core komponento sintezės rezultatai 57
6 lentelė. Skirtingų parametrų bendrinto komponento sintezės rezultatai (core_8) 58
7 lentelė. Skirtingų parametrų bendrinto komponento sintezės rezultatai (core_12) 58
8 lentelė. Skirtingų parametrų bendrinto komponento sintezės rezultatai (core_16) 58
9 lentelė. CTRL ir IR komponentų sintezės rezultatai (VHDL) 62
10 lentelė. CTRL ir IR komponentų sintezės rezultatai, be reliatyvių peršokimų (VHDL) 62
11 lentelė. CTRL ir IR komponentų sintezės rezultatai (SystemC) 62
12 lentelė. CTRL ir IR komponentų sintezės rezultatai, be reliatyvių peršokimų (SystemC). 62
13 lentelė. CTRL ir IR bendrinių komponentų sintezės rezultatai (SystemC) 63
14 lentelė. CTRL ir IR bendrinių komponentų sintezės rezultatai, be reliatyvių peršokimų (SystemC) 63
15 lentelė. PIC10F200 instrukcijų rinkinys [5] 98
16 lentelė. ATtiny25 instrukcijų rinkinys [4] 101

PAVEIKSLŲ SĄRAŠAS

1 pav. Našumo plyšys 12
2 pav. Algoritminio sudėtingumo (Shannon dėsnis) palyginimas su Moore dėsniu........ 13
3 pav. Verifikavimo plyšys 13
4 pav. Instrukcijų vykdymas be lygiagretinimo [angl. Pipeline] 24
5 pav. Keturių lygių lygiagretinimas[angl. Pipeline] 24
6 pav. Keturių lygių, dviejų vykdymo įrenginių [angl. Superscalar pipeline]........ 25
7 pav. VLIW instrukcijų vykdymas, keturi vykdymo įrenginiai 29
8 pav. Kompiuterių klasifikavimas pagal architektūras pagal Flynn........ 31
9 pav. SISD 31
10 pav. MISD........ 32
11 pav. SIMD........ 33
12 pav. MIMD 34
13 pav. Supaprastinta procesoriaus struktūrinė schema........ 38
14 pav. Bendrinių komponentų įdiegimas 39
15 pav.ACC komponentas 41
16 pav. ALU komponentas 42
17 pav. CTRL komponentas 43
18 pav. IAR komponentas........ 45
19 pav. IR komponentas 46
20 pav. PC komponentas........ 48
21 pav. PORT komponentas 48
22 pav. RAM komponentas 49
23 pav. REG komponentas 50
24 pav. CPU komponento struktūra........ 51
25 pav. CPU komponentas........ 51
26 pav. CORE komponentas........ 52
27 pav. SYSTEM komponento struktūra........ 53
28 pav. SYSTEM komponentas........ 53
29 pav. Sintezės rezultatai........ 59
30 pav. Simuliacijos rezultatai........ 59
31 pav. Ventilių skaičiaus prognozė 60
32 pav. Bendro ploto prognozė........ 60
33 pav. Vėlinimo prognozė........ 61
34 pav. PIC10F200 blokinė diagrama [5]........ 98
35 pav. ATtiny25 blokinė diagrama [4]........ 100

Mikulis, M. Procesorinio komponento bendrinimo tyrimas: analizės aspektai. Informatikos inžinerijos magistro darbas / vadovas doc. dr. Giedrius Ziberkas. Kauno technologijos universitetas, Informatikos fakultetas, Programų inžinerijos katedra. Kaunas, 2007. – 67 p.

SANTRAUKA

Mikroelektronikos technologinėms galimybėms stipriai lenkiant projektavimo galimybes, projektavimo etapas reikalauja naujų metodų. Vienas iš problemos sprendimų būdų yra atkartojimo technologija.

Pirmoje dalyje yra analizuojama literatūra. Apžvelgiamas atkartojimo technologijos objektas. Remiantis literatūra, pateikiamas platus ir siauras atkartojimo technologijos apibrėžimas. Pateikiami komponento apibrėžimai, komponento pakartotinio panaudojimo sąvokos ir metodai. Taip pat apžvelgiami mikroprocesoriai, mikroprocesorių architektūros.

Antroje dalyje išanalizuojamas pateiktas mikroprocesorius, jo komponentai. Pasirenkama procesorinių komponentų bendrinimo kryptis. Taip pat analizuojamas procesoriaus instrukcijų rinkinys, bei galimybė bendrinti instrukcijų dekodavimo ir valdymo įrenginius.

Trečioje dalyje, suformuluotiems uždaviniams pateikiami tyrimo rezultatai. Pateikiami procesoriaus komponentų bendrinimo bei sintezės rezultatai. Taip pat įvertinami instrukcijų dekodavimo ir valdymo įrenginių bendrinimo bei sintezės rezultatai.

Ketvirtoje dalyje pateikiamos išvados bei rekomendacijos.

Mikulis, M. Research of processor component generalisation: analysis aspects. Master's Work in Informatics Engineering / supervisor doc. dr. Giedrius Ziberkas. Kaunas University of Technology, Faculty of Informatics, Department of Software Engineering. Kaunas, 2007. – 67 p.

SUMMARY

The design process requires new methods, because technological abilities of microelectronics overtake design possibilities. One way of the solution is a reuse technology.

In the first chapter the analysis of literature has been made. Also the reuse technology object has been reviewed. According to literature the wide and narrow definitions of reuse technology are presented. Definitions of component, methods and concepts of generic components have been delivered. Overlook through the microprocessors and their architectures have been made.

In the second chapter a microprocessor and its components are analysed. The directions of generalisation for microprocessor components are proposed. Also analysis of instruction set, instruction decoder and control units generalisation possibility is discussed.

The third chapter provides generalisation results for formulated tasks. Results of generalisation and synthesis of processor components are presented. Also results of instruction decoder, control units generalisation and synthesis are delivered.

Conclusions and recommendations are formulated in the fourth part.

SANTRUMPŲ IR TERMINŲ ŽODYNAS

ACC – akumuliatoriaus registras [angl. *Accumulator register*]

ALU – aritmetinis loginis įrenginys [angl. *Arithmetic logic unit*]

ASIP – specializuoto instrukcijų rinkinio procesorius [angl. *Application specific instruction set processors*]

CISC – sudėtingo instrukcijų rinkinio kompiuteris [angl. *Complex Instruction Set Computer*]

CTRL – valdymo įrenginys [angl. *Control unit*]

DSP – skaitmeninio signalo mikroprocesoriams [angl. *Digital Signal Processor*]

EPIC – išskirtinai lygiagrečių instrukcijų [angl. *Explicitly Parallel Instruction Computing*]

IAR – netiesioginis adreso registras [angl. *Indirect address register*]

ILP – lygiagretinimas instrukcijų lygmenyje [angl. *Instruction level parallelism*]

IR – instrukcijų registras [angl. *Instruction register*]

ISA – instrukcijų rinkinio architektūra [angl. *Instruction Set Architecture*]

ISE – instrukcijų rinkinio išplėtimas [angl. *Instruction Set Extension*]

MISC – minimalaus instrukcijų rinkinio kompiuteris [angl. *Minimal Instruction Set Computer*]

PC – programinis skaitiklis [angl. *Program counter*]

REG – registras [angl. *Register*]

RISC – sumažinto (racionalaus) instrukcijų rinkinio kompiuteris [angl. *Reduced Instruction Set Computer*]

SystemC – aukšto lygio aparatūros aprašymo kalba, C++ kalbos atmaina

VHDL – aukšto lygio aparatūros aprašymo kalba [angl. *Very High Speed Integrated Circuit Hardware Description Language*]

VLIW – labai ilgas instrukcijos žodis [angl. *Very Long Instruction Word*]

ĮVADAS

Darbo aktualumas

Pastaraisiais dešimtmečiais labiausiai besiplėtojanti technologinių mokslų šaka yra mikroelektronika. Šioje srityje yra pasiekta nemažai laimėjimų projektavimo etapo automatizavime, bet tai neatitinka elektroninės aparatūros poreikių. Padidėjus technologinėms mikroelektronikos galimybės, projektavimo sistemos nebetenkina norimo projektavimo našumo ir patikimumo. Šios priežasties pasekoje yra didelis atotrūkis tarp projektavimo poreikių ir galimybių. Literatūroje tai yra vadinama „našumo plyšys“ [angl. *productivity gap*].

Šiam atotrūkiui sumažinti siūloma didinti projektavimo abstrakcijos lygį. Tam reikia tobulinti esamus metodus. Taip pat kurti naujus aukštesnės abstrakcijos projektavimo metodus. Šios srities tyrinėtojai atkartojimo [angl. *reuse*] technologiją laiko pagrindine prielaida iškilusioms problemoms spręsti.

Tai yra aktualu, nes tobulėjant mikroelektronikos technologinėms galimybėms, atsiveria vis naujos jos pritaikymo galimybės. Šiuolaikinėje mikroelektronikoje labai išaugo poreikis DSP mikroprocesoriams. Atsirado didelis poreikis mus supančius analoginius signalus perdavinėti šiuolaikinėmis komunikacijomis, t.y. vaizdas, garsas, taip pat medicinos sritis ir panašiai. Apskritai šiuolaikinė mikroelektronikoje plačiai naudojami mikroprocesoriai.

Darbo tikslai

Šio darbo esmė atlikti procesorinio komponento bendrinimo analizę. Išanalizuoti procesorinio komponento bendrinimo galimybes, tikslus, pagrįstumą. Iki kokio lygio galima bendrinti, kad neprarasti funkcionalumo. Ar bendrintas komponentas bus sintezuojamas. Tikslo siekimui suformuluosime uždavinius:

- išanalizuoti mikroprocesorių architektūras;
- išanalizuoti homogeninio parametrizavimo galimybes;
- išanalizuoti procesorinio komponento modelį;
- pasiūlyti bendrinimo kryptis;

- šiuolaikinėmis modeliavimo ir sintezės priemonėmis eksperimentiškai ištirti siūlomus metodus.

Darbo metodai

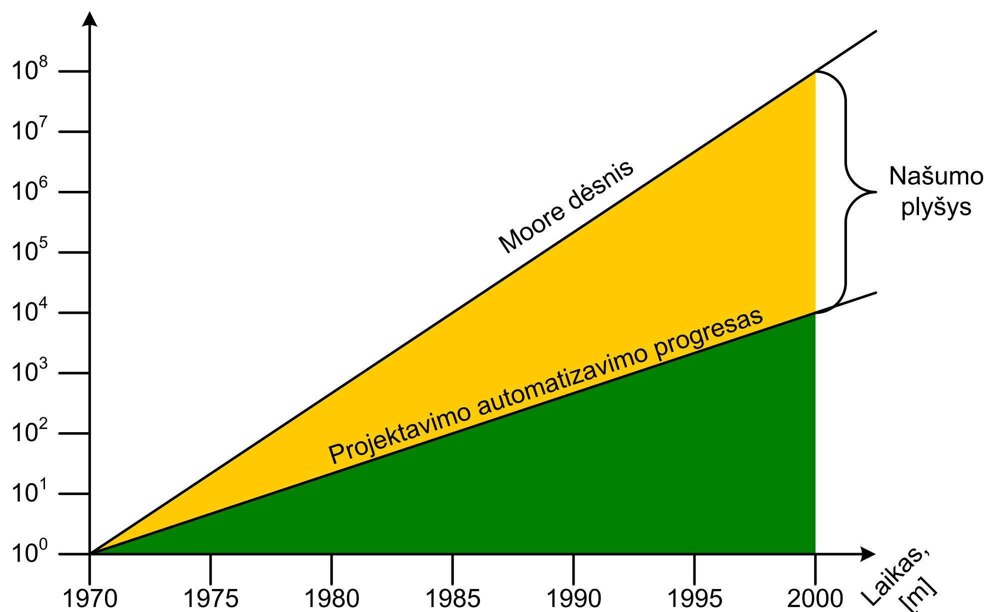
Norint paspartinti bendrinių komponentų projektavimo procesą stengiamasi ne kurti naują komponentą, o ieškoma kelių jau esamų ištestuotų, patikrintų komponentų bendrinimui. Tai svarbu, nes naujų bendrinių komponentų testavimui nebereikia skirti tiek daug laiko. Todėl šiame darbe bus išnagrinėta parametrizavimo galimybė jau esančio komponento, neprojektuojant jo iš naujo.

Darbo apimtis 67 p. (be priedų). Darbe pateikta 35 paveikslai ir 16 lentelių.

1. PROBLEMOS ANALIZĖ

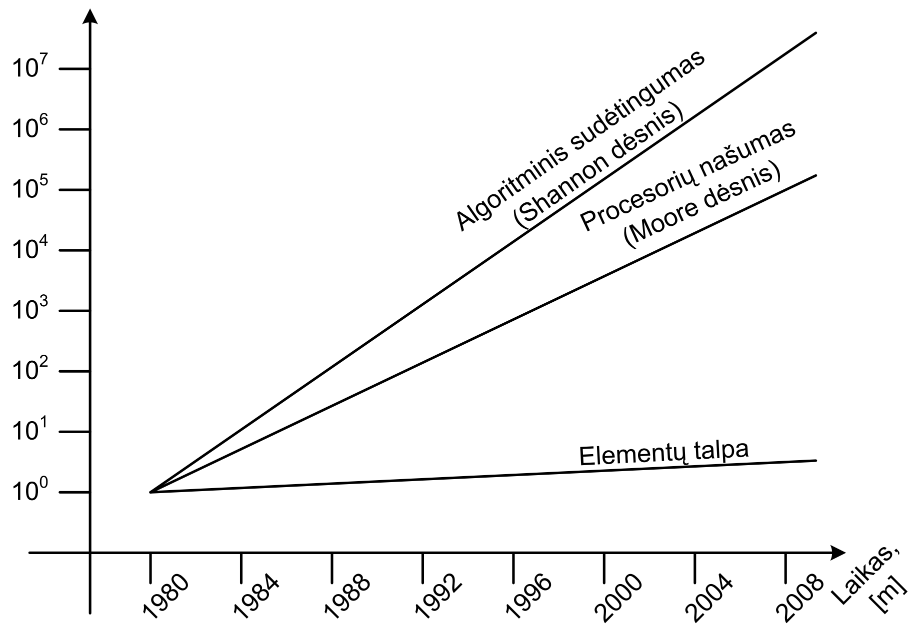
1.1. Projektavimo etapo problematika

Nuolatos didėjant atotrūkiui tarp mikroelektronikos technologinių galimybių ir projektavimo proceso našumo, kyla natūralus poreikis naujiems projektavimo metodams, ar esamų tobulinimui [2]. Nors pasiekta nemažai laimėjimų projektavimo proceso automatizavime, bet tai netenkina poreikių (1 paveikslas).



1 pav. Našumo plyšys

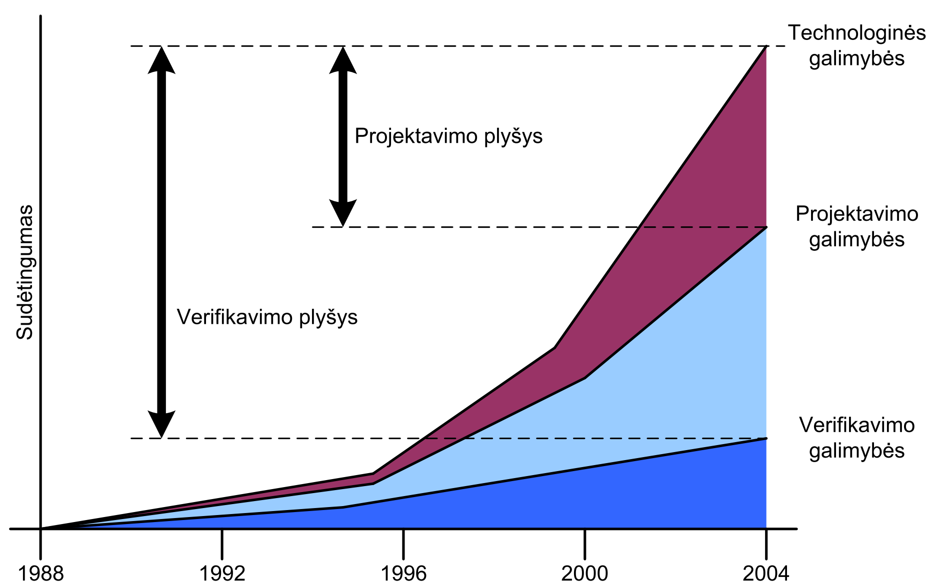
Didėjantis gaminių sudėtingumas dar daugiau didina našumo ir kokybės problemą. 2 paveiksle pateikiamas atotrūkis tarp algoritminio sudėtingumo poreikio ir Moore dėsnio procesorių našumo. Taip pat šiuolaikinė rinka verčia skubinti projektavimo procesą [12]. Noras pateikti naują sudėtingesnį gaminį per kiek įmanomą trumpesnį laiką netenkina projektavimo, verifikavimo galimybių. Gausios projektuotojų pajėgos nežymiai paspartina produkto pasirodymą rinkoje, bet ženkliai įtakoja galutinę produkto kainą.



2 pav. Algoritminio sudėtingumo (Shannon dėsnis) palyginimas su Moore dėsniu

Šiam atotrūkiui sumažinti siūloma didinti projektavimo abstrakcijos lygį. Tam reikia tobulinti esamus metodus, taip pat kurti naujus aukštesnės abstrakcijos projektavimo metodus. Šios srities tyrinėtojai atkartojimo [angl. *reuse*] technologiją laiko pagrindine prielaida iškilusioms problemoms spręsti [2].

Dar viena priežastis vedanti link atkartojimo technologijos metodų tobulinimo – verifikavimas [1][14]. Čia yra jaučiamas dar didesnis atotrūkis (3 paveikslas).



3 pav. Verifikavimo plyšys

1.2. Atkartojimo technologija

Pakartotinio naudojimo technologija [angl. *reuse technology*] iki šiol vis dar yra tyrimo objektas. Ši technologija figūruoja daugelyje skirtingų formų nuo tikslinio iki sisteminio atkartojimo, nuo juodosios dėžės [angl. *black box*] iki baltosios dėžės [angl. *white box*] atkartojimo, nuo bendrinio atkartojimo iki specifinės srities atkartojimo. Plačiai paplitusi nuomonė, kad pakartotinio naudojimo technologija yra esminė kryptis tobulinant programinės įrangos kūrimo produktyvumą ir kokybę [20].

1.2.1. Pagrindinių terminų apibrėžimai

Yra sunku apibrėžti vienokius ar kitokius terminus, nes ši technologija vis dar yra tyrimų objektas. Pabandysime apibrėžti esminius terminus, tokius kaip: programinis pakartotinis naudojimas [angl. *software reuse*], komponentas [angl. *component*], programos generatorius [angl. *program generator*] ir ruošiniai [angl. *templates*]. Tai nėra visi terminai, bet jų turėtų pakakti pirminių, atkartojimo technologijos, aspektų suvokimui.

Atkartojimo technologijos tikslas – naudoti komponentus „iš lentynos“ [angl. *off the shelf components*] kaip blokus naujos sistemos kūrime.

„Pakartotinio panaudojimo procesas yra panaudojimas esančių programinių elementų, juos pritaikant ir modifikuojant naujiems reikalavimams“ – su šiuo teiginiu sutinka 66% iš 133 įvairaus išsilavinimo respondentų, iš kurių – 32% bakalauro laipsnio, 55% magistro, 12% daktaro [20].

Prieš ką nors naudojant pakartotinai, tai turi būti naudinga [24]. Autoriai nori pasakyti, kad tarp naudojimo ir pakartotinio naudojimo yra plona riba: „Kai kurie gali manyti, kad naudojimas paketo iš bibliotekos yra geras pakartotinio naudojimo pavyzdys, kiti gali tai pavadinti tik geru inžineriniu planavimu“. Patys autoriai teigia, kad po truputį yra ir to, ir to.

Siūloma vadovautis šiuo apibrėžimu

Pakartotinio panaudojimo technologija – tai procesas apimantis [20]:

a) komponentų kūrimas pakartotiniam panaudojimui, greičiausiai, naudojant jau esančius komponentus, juos modifikuojant arba ne;

b) naujos sistemos kūrimas naudojant pakartotinio panaudojimo elementus su kontroliuojamomis modifikacijoms.

Šis požiūris pabrėžia komponentų panaudojimo, kaip sistemos kūrimo blokų, svarbą. Tai atitinka taip vadinama kompozicinį pakartotinį naudojimą. Kitas būdas, taip pat naudingas, yra ne kompozicinis atkartotinas naudojimas, o generatyvinis. Daugelis autorių sutinka, kad komponentinis pakartotinis naudojimas yra fundamentinis, nes generatyvinio pakartotinio naudojimo komponentą kurti galima naudojant komponentinę koncepciją.

1.2.2. Kas yra komponentas

Platus apibrėžimas

Komponentas – tai programinės įrangos kūrimo procesas, apimantis realius elementus (specifikacijos, architektūra, kodas, dizainas, algoritmai, sąsajos, testavimo sąlygos) ir neapčiuopiamus elementus (žinios, metodologija) [25][22].

Komponentas – aukštos kokybės tipas, klasė, ar kitas darbo produktas, suprojektuotas, dokumentuotas, sukomplektuotas pakartotiniam naudojimui. Komponentas yra rišlus ir turi stabilią sąsają. Komponentas apima sąsają, posistemes, naudojimo sąlygas, tipus ar klases ir atributus. Komponentas apima kitus darbo produktus kaip ruošinius, testavimo sąlygų specifikaciją [21].

Siauras apibrėžimas

Komponentas – savarankiškas, aiškiai apibrėžtas programos gabalas, kuris apibūdina specifinę funkciją, turi aiškią sąsają ir atitinkamą dokumentaciją [23].

Remiantis literatūra [20] *siūloma struktūrą problemos apibūdinimui*

- komponentas kartais yra kaip turtas [angl. *asset*] ar darbo produktas (artefaktas). Visi šie terminai turi šiek tiek skirtingas reikšmes. Komponentas – reiškia sąsają ir paketą. Turtas [angl. *asset*] – perduoda savininko ir valdymo aspektus. Darbo produktas – rodo faktą, kad komponentas yra darbo ciklo dalis.

Sekančios komponento savybės yra vertingos, kad komponentas būtų pakartotinai naudojama dalis, jis privalo būti prieinamas, suprantamas ir naudingas.

Kas yra bendro visuose šituose komponento sąvokos apibrėžimuose, tai, kad komponentas yra – savarankiška, gerai dokumentuota ir ištestuota struktūra, jis turi turėti funkcionalumą, griežtai apibrėžtą sąsają komunikavimui su aplinka [20].

Funkciniai požymiai rodo ką komponentas daro ir kaip jis įdiegiamas. Kiekybiniai požymiai rodo objektyvines ir subjektyvines reikšmes, kurios įvertina komponento kokybę. Objektyvinė reikšmė apima eilučių skaičių, sudėtingumą. Subjektyvinė reikšmė apima suprantamumą, dokumentaciją ir testavimus.

Ypatingai svarbu pakartotinai naudojamiems komponentams turėti griežtai apibrėžtą sąsają. Bendrai sakant, sąsaja turi parodyti kaip naudoti komponentą, nežinant jo įdiegimo smulkmenų. Visa reikalinga informacija sėkmingam komponento panaudojimui turi būti jo specifikacijoje. Pavyzdžiui ADA ir VHDL teikia tokias vartotojui suprantamas sąsajas, paketus.

Yra du pagrindiniai metodai susieti su komponentiniu pakartotiniu naudojimu

- keičiant komponentą, jo vidinę struktūrą tam, kad jis atitiktų vartotojo reikalavimus. Šis būdas priartinamas prie baltos dėžės [angl. *white-box*];
- naudojat komponentą be jokių pakeitimų jo vidinėje struktūroje. Šis būdas priartinamas prie juodos dėžės [angl. *black-box*].

Abiem paminėtiems būdams gali būti sukurti sričiai specifiniai komponentai, kaip srities specifinių komponentų pakartotinio naudojimo rezultatas.

Nėra vieningo komponento sąvokos apibrėžimo. Kuo daugiau informacijos galime surinkti ir apibendrinti, tuo tiksliau galime apibrėžti komponento sąvoką. Tai galima atlikti tik su laiku, sukaupus pakankamai patirties šioje srityje [20].

Taip pat galime suformuluoti komponentų tipų sritį.

- Komponentai susideda iš:
    - visų tipų programų, taip pat jų specifikacijų, konstrukcijų, išeities kodo;
    - visų dydžių – nuo individualių fragmentų iki galutinių posistemių;
    - abiejų tipų sričiai specializuotų ir bendros paskirties komponentų;
    - abiejų tipų komponentų iš lentynos (juodos dėžės) bei pritaikomųjų komponentų (baltos dėžės).
- Komponentų dydžiai sritis:
    - vientisa sistema;
    - segmentas;
    - mažas fragmentas.
- Modifikacijos laipsnio sritis:
    - negalima (juoda dėžė);
    - nereikšminga (mažiau nei 5%);
    - nedideli (mažiau nei 25%);
    - svarbūs (daugiau kaip 25%).
- Užbaigtumo sritis:
    - industrinis;
    - korporacinis;
    - departamentinis;
    - vienam projektui.

1.2.3. Kas yra generatorius

Šis terminas naudojamas generatyvinio pakartotinio naudojimo esmei paaiškinti. Generatorius tai įrankis skirtas palaikyti generatyvinį požiūrį.

- Generatorius – tai aukšto lygio automatinis kūrėjas, kuris išlaisvina nuo rankinio komponentų sujungimo, naudojant į problemą orientuotas kalbas, ruošinius ar filtrus, ar vizualinę programavimo aplinką. Generatorius naudoja glaustas specifikacijas reikiamos aplikacijos, ir tada sugeneruoja reikiamą kodą ar procedūrų šaukinius tam tikrose kalbose [22];
- Generatorius kuria įvairius kalbų ar ruošinių ryšius naudodamas jų komponentus. Generatorius dažnai yra kalbos ar kalbos valdomas įrankis [21].

1.2.4. Kas yra ruošinys

Šis terminas yra svarbus tuo, kad dauguma efektyvių generatorių būtent pagrįsti šiuo terminu.

Ruošinys ar griaučiai yra programinio darbo produktas su neapibrėžtais parametrais, kurie gali būti panaudoti sukurti ar sugeneruoti gatavą produktą.

Viena iš pirmųjų publikacijų, kur šis terminas buvo pristatytas su dabartine esme, buvo Sametinger [23].

1.3. Motyvacija, kodėl naudoti pakartotinio naudojimo metodiką

Visada yra svarbu išnagrinėti naujų dalykų motyvacines priežastis. Plačiai paplitusi nuomonė, kad pakartotinis naudojimas yra raktinė kryptis tobulinant programinės įrangos kūrimo produktyvumą ir kokybę.

- Programuotojų ir kompiuterių mokslininkų bendruomenė kovoja su „programine krize" jau daugiau kaip dvi dekadas (kompiuterinė įranga vis tampa spartesnė ir pigesnė; kompanijos vis daugiau pinigų išleidžia programoms, programos tampa vis sudėtingesnės; programinės įrangos atliekamos užduotys vis auga, vis mažiau studentų pasirenka programavimą);
- Poreikis tobulinti programinės įrangos produktyvumą, kokybę ir patikimumą;
- Poreikis sumažinti laiką kelio į rinką [angl. *time-to-market*].

Apžvelgus programinės įrangos projektus, matyti, kad tik 16% iš išleidžiamų projektų yra pilnai sėkmingi (pelningi, laiku, su visomis galimybėmis), net 53% projektų viršija biudžetus, neišeina laiku, nepilno funkcionalumo, ir 31% projektų nutraukiama [20].

Atkartojimas yra pagrindas tiek programiniams, tiek aparatūriniams komponentams kurti ir apskritai vienlustėms sistemoms projektuoti. Nors atkartojimas ir negali būti laikomas visų projektavimo problemų sprendimu, jis yra metodologinis (plačiąja prasme) ir technologinis (siaurąja prasme) pagrindas. Atkartojimas gali būti pasiekiamas per abstrakcijos lygmens kėlimą, apibendrinimą ir automatizavimą [19].

Aukšto lygmens abstrakcijos, tokios kaip paketai (VHDL) arba klasės (SystemC), leidžia kurti lengvai atkartojamus komponentus ir jų bibliotekas.

Apibendrinimas leidžia aprašyti atkartojamus (bendrinius) komponentus naudojant bendrines kalbos abstrakcijos (pvz., C++/SystemC šablonus) arba aukštesnės kalbos (pvz., preprocesoriaus komandų) abstrakcijas.

Bendrinis komponentas – apibendrintas komponentas, aprašantis panašių komponentų šeimyną ir turintis bendrinius parametrus, kurių pagalba vartotojas gali pasirinkti konkretų komponento egzempliorių.

Automatizavimas apima programų generatorių kūrimą ir leidžia sumažinti pastangas reikalingas atkartojamumui realizuoti.

Atkartojimo metodai gali būti klasifikuojami taip:

1. komponentinis atkartojimas yra pagrįstas sistemos dalių atkartojimu, t.y. siekiama panaudoti komponentus „iš lentynos“ kaip naujos sistemos blokelius.

Komponentinio atkartojimo tipai:

a) baltos dėžės atkartojimas [angl. *white-box reuse*] – vidinė komponento struktūra matoma programuotojui; atkartojimo metu ją galima modifikuoti pritaikant prie sistemos konteksto.

b) juodos dėžės atkartojimas [angl. *black-box reuse*] – vidinė komponento struktūra nematoma programuotojui; atkartojimo metu jos modifikuoti negalima.

c) pilkos dėžės atkartojimas [angl. *gray-box reuse*] – vidinė komponento struktūra matoma programuotojui, tačiau atkartojimo metu jos modifikuoti negalima.

2. generatyvinis atkartojimas yra pagrįstas komponentų kūrimo ir modifikavimo procesų atkartojimu naudojant specialius įrankius (pvz., procesorius, generatorius).

Plačiąja prasme atkartojimas apima visus resursus, kurie yra naudojami ir sukuriami sistemų kūrimo metu. Galutinis atkartojimo tikslas yra sistemų projektuotojo darbo našumo padidinimas.

Sėkmingo atkartojimo paslaptis yra nuodugni srities analizė ir daugeliui srities taikomųjų programų būdingos elgsenos išskyrimas į nedidelės apimties komponentų aibę. Geresnio atkartojamumo pasiekiama dirbant su aukštesnio lygmens, abstraktesnėmis programų konstrukcijomis. Šiuo metu geriausiai atkartojami komponentai yra siauros probleminės srities, tokios kaip elektroninių schemų projektavimo paketai, abstrakcijos.

1.3.1. Atkartojimas gali būti pasiektas naudojant metaprogramavimą

Metaprogramavimas – programavimo metodas, kai tikslo kalba užrašome konkretų srities funkcionalumą, o su metakalba jį apibendriname.

Paprasčiausias metaprogramavimo pavyzdys – C++/SystemC komponentų apibendrinimas naudojant CPP (C preprocesorių). Konkretus programos egzempliorius yra sugeneruojamas su metakalbos procesoriumi arba srities tikslo kalbos kompiliatoriumi. Šiuo metu jau yra sukurta žymiai galingesnių preprocesorių negu CPP.

Atskiras metaprogramavimo atvejis yra taip vadinamas homogeninis metaprogramavimas, kai bendriniai komponentai kuriami vienos kalbos aplinkoje, naudojant vien tik tos kalbos abstrakcijas, pvz., VHDL generics arba SystemC šablonus (templates). Bendrinių komponentų kūrimas vienoje aplinkoje yra paprastesnis, tačiau sukurti komponentai ne visada yra sintezuojami.

Bazinis MPG mechanizmas yra parametrizavimas. Šis mechanizmą palaiko metakalbos arba bendrinės tikslo kalbos poaibio abstrakcijos. Bendriniai parametrai turi vertes (reikšmes), kurios išreiškia tam tikrus taikomosios srities aspektus, todėl galima tvirtinti, kad parametrizavimo mechanizmas leidžia sujungti programavimo ir metaprogramavimo sritis.

Bendrinių komponentų kūrimą sudaro trys stadijos:

1. koncepcijų atskyrimas – atliekama srities analizė, kurios metu atskiriamas pastovus srities funkcionalumas, kuris nepriklauso nuo bendrinių parametrų ir yra bendras visai komponentų šeimynai; ir kintamas srities funkcionalumas, kuris priklauso nuo bendrinių parametrų reikšmių ir gali būti skirtingas kiekviename komponento egzemplioriuje. Išskiriami bendriniai parametrai, nuo kurių reikšmių priklauso konkretaus komponento egzemplioriaus realizavimas.

2. koncepcijų realizavimas – pastovus srities funkcionalumas yra aprašomas tikslo kalba be papildomų programos kodo modifikacijų. Kintamas srities funkcionalumas turi būti modifikuojamas priklausomai nuo bendrinių parametrų reikšmių ir yra aprašomas naudojant metakalbos abstrakcijas.

3. koncepcijų integravimas – metakalbos ir tikslo kalbos abstrakcijos yra suintegruojami į bendrinį komponentą.

Bendrinio komponento testavimas yra atskira ir labai sudėtinga problema, kadangi bendrinis komponentas gali turėti labai daug skirtingų egzempliorių, kuriuos visus ištestuoti gali būti fiziškai neįmanoma. Tokiu atveju reikėtų sukurti egzempliorius su ribinėmis bendrinių parametrų reikšmėmis ir remiantis jų testavimo rezultatais daryti išvadas apie viso bendrinio komponento teisingumą.

1.4. Mikroprocesoriai

Bevielio ryšio komunikacijose ir kitose įterptinėse sistemose, projektavimo įrankiai ASIP sistemoms yra svarbi disciplina sisteminio projektavimo lygyje. Kai kurios ASIP sistemos, siekiant atitikti ekstremalius efektyvumo poreikius, projektuojamos nuo nulio. Bet ryškėja tendencijos, vedančios prie dalinai apibrėžtų, konfigūruojamų RISC tipo įterptinių procesorių branduolių, kurie gali būti greitai priderinti prie keliamos užduoties naudojant ISE technologiją [11].

Sparčiai besivystant mikroelektronikos technologijoms atsiveria vis naujos pritaikymo sritys. Ypatingai didelis poreikis skaitmeninio signalo apdorojimo mikroprocesoriams. Tai labai plačiai naudojama šiuolaikinėse komunikacijose vaizdui ir garsui apdoroti, taip pat medicinoje. Tai kelia vis sudėtingesnius uždaviniu projektuotojams bei reikalauja sudėtingesnių programinių aparatūrinių sprendimų.

Šiek tiek išsamiau panagrinėsime pačius procesorius, jų dedamąsias dalis.

Procesorius struktūriškai gali būti suskaidytas į komponentus, žinoma, čia neišvardinsime visų dalių, bet paminėsime pagrindines:

- Atmintys [angl. *Memory*]:
  - ROM;
  - RAM;
  - EPROM;
  - EEPROM;
  - cache;
  - flash;
  - OTP.
- Registrai [angl. *Register*]:
  - instrukcijų rinkinio [angl. *Instruction Set*];
  - bendro naudojimo [angl. *GPR – General Purpose Register*].
- Magistralės [angl. *Bus*]:
  - duomenų [angl. *Data Bus*];
  - adresų [angl. *Address Bus*].
- Aritmetiniai-loginiai įrenginiai [ALU].
- Periferinės sąsajos elementai [angl. *Peripheral Interface*]:
  - RS-232;
  - USB;
  - ethernet;
  - CAN;
  - I^2C;
  - SPI;
  - 2-wire;
  - 3-wire;

    - parallel.
- Valdymo logika [angl. *Control Logic*].
- Skaitikliai [angl. *Counter*].

1.4.1. Mikroarchitektūrinė koncepcija

Bendruoju atveju visi mikroprocesoriai vykdo programas atliekant šiuos žingsnius:

- nuskaityti instrukciją ir ją dešifruoti;
- išrinkti visus duomenis reikalingus šiai instrukcijai įvykdyti;
- įvykdyti instrukciją;
- išvesti rezultatus.

Šią iš esmė paprastą veiksmų seką komplikuoja atminties hierarchija, kuri apima pagrindinę atmintį ir išorines atmintis, kuri paprastai lėtesnė negu vidinė procesoriaus. Antrasis žingsnis paprastai įveda pailgintą vėlinimą, kol duomenys pasiekia procesorių per magistrales. Atliekama daug tyrimų ir jų rezultatai įtraukiami į projektavimo etapą, norint kiek įmanoma daugiau išvengti šių vėlinimų. Pagrindinis tikslas buvo, lygiagretinti instrukcijų vykdymą, kas didiną programų vykdymo našumą, spartą. Šios pastangos iššaukia sudėtingas logines ir schematines struktūras. Iš pradžių šie techniniai sprendimai, dėl sudėtingumo, galėjo būti įtraukiami tik į brangias sistemas – superkompiuterius. Sparčiai besivystančių mikroelektronikos technologijų dėka, šios techninės priemonės vis labiau įtraukiamos ir į paprastesnius, pigesnius produktus.

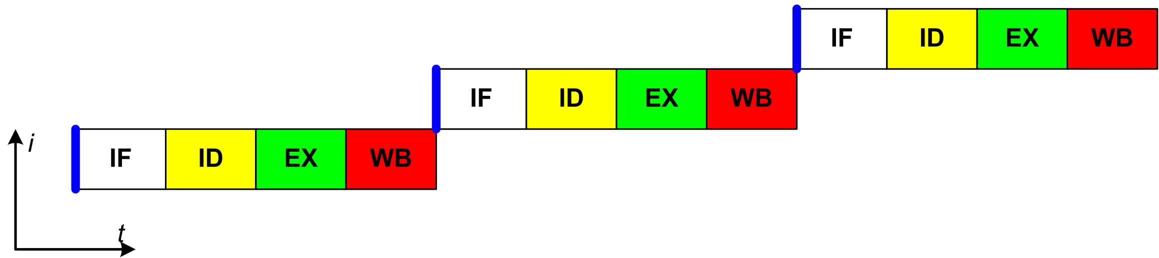
1.4.2. Instrukcijų rinkinio architektūros pasirinkimas

Pasirinkimas, kurią instrukcijų rinkinio architektūrą naudoti, ypatingai įtakoja įrenginio sudėtingumą ir našumą. Kompiuterių architektai nuolat stengėsi supaprastinti instrukcijų rinkinius, kas įgalino didesnį našumą. Projektuotojai galėjo daugiau laiko ir pastangų skirti našumo didinimui, o ne sudėtingų instrukcijų projektavimui.

Instrukcijų rinkinio projektavimo progresas nuo CISC, RISC, VLIW, EPIC tipų į architektūras, kurios dirba su duomenų lygiagretumu apimant SIMD ir vektorius.

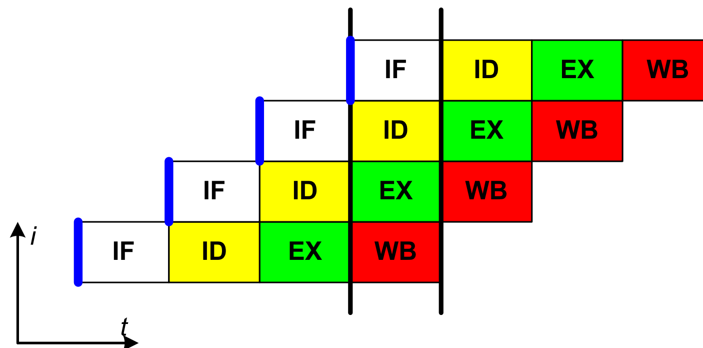
1.4.3. Instrukcijų lygiagretinimas

Tai viena iš pirmųjų ir galingiausių technologijų našumui padidinti. Pirmieji procesoriai pradėdavo vykdyti sekančią instrukciją tik įvykdę visus prieš tai buvusios instrukcijos žingsnius (4 paveikslas). Didelės schematinės dalys likdavo be darbo, pavyzdžiui, instrukcijos dekodavimo schematika neatlieka jokių veiksmų instrukcijos vykdymo metu ir panašiai.

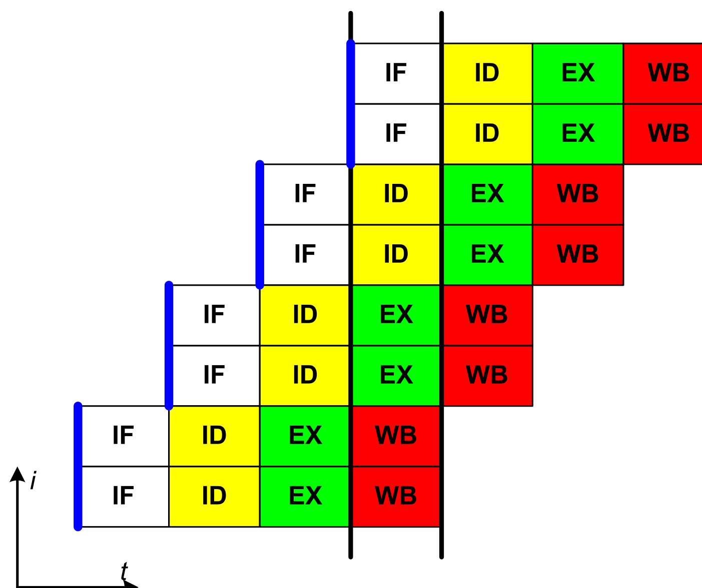


4 pav. Instrukcijų vykdymas be lygiagretinimo [angl. Pipeline]

Lygiagretinimas padidino našumą, įgalindamas keletą instrukcijų atlikti skirtingus žingsnius tuo pačiu metu, plaukti procesoriumi. Pavyzdžiui procesorius leis instrukcijos dekodavimo logikai dešifruoti instrukciją, kol paskutinioji instrukcija dar lauks rezultato (5, 6 paveikslas). Toks metodas leidžia vykdyti keturias instrukcijas vienu metu, o tai praktiškai padidina procesoriaus našumą keturis kartus. Žinoma pačios instrukcijos įvykdymo laikas lieka toks pats, ji vis tiek susideda iš keturių žingsnių.



5 pav. Keturių lygių lygiagretinimas[angl. Pipeline]



6 pav. Keturių lygių, dviejų vykdymo įrenginių [angl. Superscalar pipeline]

RISC architektūros lygiagretinimas daug paprastesnis projektuoti, nes yra aiškiai atskirti atskiri žingsniai, jie užima vienodą procesoriaus laiką – vieną ciklą. Bet lygiagretinimas nėra apribojamas, jis taip pat sėkmingai realizuojamas ir CISC architektūrose, nors tai ir sudėtingiau, bet dėka besivystančių technologijų tai tapo įmanoma.

1.5. Procesorinio komponento architektūra

Apibendrinti visą procesorių, kaip vieną komponentą, nėra realu ir tikslinga. Nes bus prarandamas lankstumas, funkcionalumas, optimalumas. Parametrizavimas savo sudėtingumu taps panašus į programavimą. Yra neįmanoma apžvelgti visų esamų ir būsimų uždavinių. Taip pat kyla sunkumų atnaujinant komponentą.

Komponavimas iš atskirų, fundamentalių procesoriaus komponentinių elementų, išlaiko optimalumo, funkcionalumo, išplečiamumo savybes. Dėka atskirų komponentų, galima kurti itin specifinius procesorius pritaikytus specializuotoms užduotims.

1.5.1. Procesoriaus komponentinės dalys

Dauguma procesoriaus komponentų yra pakankamai elementarūs, vienokio ar kitokio tipo atmintys, skaitikliai, registrai, magistralės ir panašiai. Jie iš esmės skiriasi pločiu, dydžiu. Kiek daugiau problemų kelia ALU, nors tai iš esmės apibrėžtas elementas, bet pakankamai platus. Galimas problemos sprendimo variantas skirstyti ALU.

ALU elementariems aritmetiniams, loginiams veiksmams:

- Aritmetiniai veiksmai:
  - aritmetinė sudėtis;
  - aritmetinė atimtis;
  - aritmetinė daugyba;
  - aritmetinė dalyba.
- Loginiai veiksmai:
  - loginis neigimas;
  - loginė konjunkcija;
  - loginė disjunkcija.

ALU trigonometriniams veiksmams:

- Trigonometriniai veiksmai:
  - sinuso funkcija;
  - kosinuso funkcija;
  - tangento funkcija;
  - kotangento funkcija;
  - arksinuso funkcija;
  - arkkosinuso funkcija;
  - arktangento funkcija;
  - arkkotangento funkcija;

Taip pat galimas ALU kėlimo laipsniu, šaknies traukimo, logaritmo skaičiavimui. Priimtiniau daryti ALU atskiroms veiksmų grupėms, nes tai lengvina išplečiamumą. Taip pat kuo daugiau komponentas padengia veiksmų, tuo jis mažina sistemos lankstumą, nors supaprastina naudojimą.

Sekanti dalis – instrukcijų rinkinio architektūra (ISA). Tai pakankamai specifinis elementas kiekvienai procesorinei sistemai, kuri apsprendžia procesoriaus architektūros tipą ir jo našumą.

Instrukcijų rinkinys – tai visų instrukcijų sąrašas ir jų variacijos, kurias procesorius supranta ir gali įvykdyti. Instrukcijos apima:

- aritmetines operacijas – sudėtis, atimtis ir pan.;
- logines operacijas – loginė konjunkcija, disjunkcija, neigimas;
- duomenų operacijas – skaitymas, rašymas, perkėlimas, įėjimai, išėjimai;
- vykdymo sekos operacijas – peršokimai, sąlygos peršokimai, kvietimai, grįžimai ir pan.

Šis elementas taip pat gali būti pakankamai platus. Įvairovė šioje techninėje dalyje taip pat yra gana plati. Išvardinsime keletą architektūros modelių ir jų išskirtinumus.

1.5.2. CISC architektūra

CISC – sudėtingų instrukcijų rinkinio kompiuteris – tai mikroprocesoriaus instrukcijų rinkinio architektūra (ISA), kurioje kiekviena instrukcija gali įvykdyti keletą žemo lygio operacijų, tokių kaip užkrovimas iš atminties, aritmetinė operacija ir įrašymas į atmintį, viską atliekant viena instrukcija.

CISC architektūros procesorių pavyzdžiai: System/360, VAX, PDP-11, Motorola 68000 šeima, bei Intel x86.

1.5.3. RISC architektūra

RISC – tai mikroprocesorių instrukcijų rinkinio architektūra, kuri sudaryta iš elementarių instrukcijų, kurios atlieka po vieną operaciją. Šios architektūros atsiradimą įtakojo supratimas, kad daugelis ypatybių, kurios buvo įtraukiamos į tradicinius procesorių projektus siekiant lengvesnio programavimo, buvo pradėtos ignoruoti [6]. Taip pat šios sudėtingos ypatybės užimdavo keletą procesoriaus vykdymo ciklų. Dar papildomos įtakos turėjo nuolat didėjantis efektyvumo plyšys tarp procesoriaus ir pagrindinės atminties. Tai privedė prie kelių techninių sprendimų, kad išlaikyti tą pačią apdorojimų kryptį procesoriuje, bet tuo pačiu sumažinti kreipinių į atmintį skaičių. Ankstyvosiomis kompiuterių pramonės dienomis kompiliavimo technologijos nebuvo apskritai. Programavimas vyko arba mašininiu kodu arba asemblerio kalba. Norint padaryti programavimą lengvesnį, kompiuterių projektuotojai kurdavo vis sudėtingesnes ir sudėtingesnes instrukcijas, kurios tiesiogiai

atitikdavo aukšto lygio programavimo kalbos funkcijas. Tai lėmė požiūris, kad aparatūros projektavimas buvo lengvesnis negu kompiliatoriaus projektavimas. Taigi visas sudėtingumas buvo realizuojamas aparatūriškai.

Tai leidžia šias instrukcijas įvykdyti kaip galima sparčiau. Šios architektūros sukūrimą lėmė tai, kad nuolat sudėtingėjanti CISC architektūra buvo sunkiai realizuojama schematiškai, dėl riboto tranzistorių skaičiaus luste, didelės vieno tranzistoriaus kainos. Taip pat programuotojau vis mažiau išnaudodavo CISC architektūros instrukcijų rinkinio teikiamus privalumus. Tai iš dalies ir lėmė RISC architektūros atsiradimą. Dabar tai labiausiai išplitusi architektūra.

Labiausiai paplitę RISC mikroprocesoriai yra: Alpha, ARC, ARM, AVR, MIPS, PA-RISC, PIC, Power Architecture ir SPARC.

1.5.4. MISC architektūra

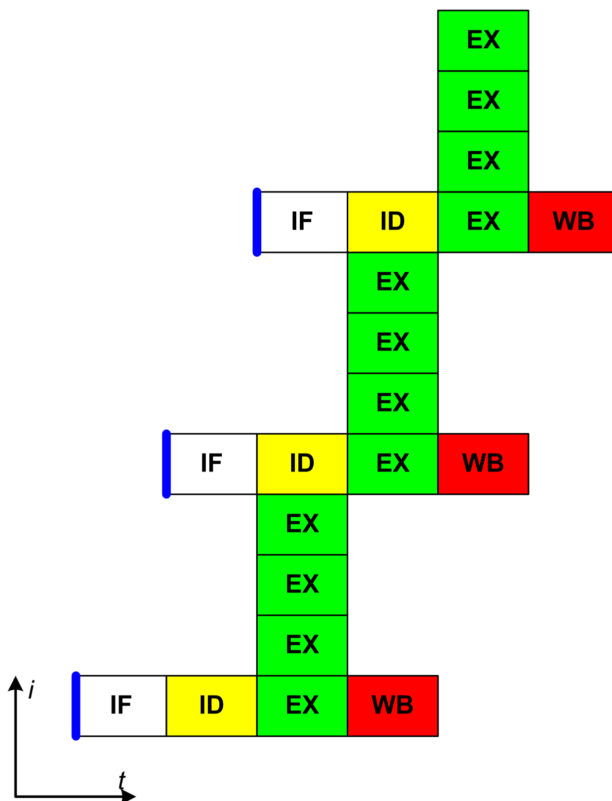
MISC – tai procesoriaus architektūra su labai mažu skaičiumi elementarių operacijų ir atitinkamais operacijų kodais. Paprastai šie instrukcijų rinkiniai realizuojami „steke“ [angl. *stack*], o ne instrukcijų registre, kad sumažinti specifinių operandų dydį. Tokios „steko“ mašinų architektūros iš esmės yra paprastesnės, kadangi visos instrukcijos vykdomos aukščiausiuose „steko“ įrašuose. Šio mažo instrukcijų rinkinio rezultatas yra paprastesnė, greitesnė instrukcijų dekodavimo logika ir bendrai greičiau vykdomos instrukcijų operacijos. Minusas tokios architektūros yra tai, kad instrukcijos linkusios turėti nuoseklią priklausomybę, o tai mažina instrukcinio lygio lygiagretinimą. MISC architektūros turi daug bendro su „Forth programming language“ programavimo kalba ir „Java Virtual Machine“ virtualiąja mašina.

Ko gero, pats komerciškai sėkmingiausias MISC architektūros produktas buvo INMOS transpiuteris.

1.5.5. VLIW architektūra

VLIW – tai procesorių architektūra, siekianti apimti visus instrukcijų lygio lygiagretinimo privalumus (ILP). Šio tipo procesoriai turi keletą vykdymo įrenginių, o šio tipo instrukcijos visada turi bent vieną užduotį kiekvienam vykdymo įrenginiui (7 paveikslas).

Skirtumas tarp kitų sistemų naudojančių keletą vykdymo įrenginių yra tai, kad instrukcijos turi užduotį tik vienam vykdymo įrenginiui. Instrukcijos žodis paprastai nebūna trumpesnis negu 64 bitų [3][8][9].



7 pav. VLIW instrukcijų vykdymas, keturi vykdymo įrenginiai

Minusas tas, kad šio tipo instrukcijų rinkinio architektūra nepalaiko suderinamumo tarp naujų ir senų versijų, nes pasikeitus vykdymo įrenginių skaičiui keičiasi ir instrukcijos kodas.

Programų vykdymo lygiagretumo planavimą atlieka kompiliatorius, procesoriui nebereikia tikrinti ar vienos instrukcijos rezultatai nebus naudojami sekančioje instrukcijoje.

1.5.6. EPIC architektūra

EPIC – Išskirtinai lygiagrečių instrukcijų skaičiavimas - tai skaičiavimo paradigma, kuri buvo pradėta tyrinėti 1990 m. Ši paradigma taip pat vadinama nepriklausomomis architektūromis [angl. *Independence Architectures*] [7]. Intel ir HP ją naudojo Intel firmos IA-64 architektūros vystyme. Ji buvo pritaikyta Intel firmos Itanium ir Itanium2 linijos serveriniuose procesoriuose. EPIC tikslas yra padidinti mikroprocesoriaus

galimybes vykdyti instrukcijas lygiagrečiai atsisakant sudėtingos schematikos. O tam naudojant kompiliatorių, kuris identifikuoja ir pasveria lygiagretaus vykdymo galimybę. Tai turėtų ateityje leisti žymiai padidinti procesorių našumą nedidinant darbinio dažnio, kas dabar tampa aktuali problema dėl energijos ir aušinimo problemų.

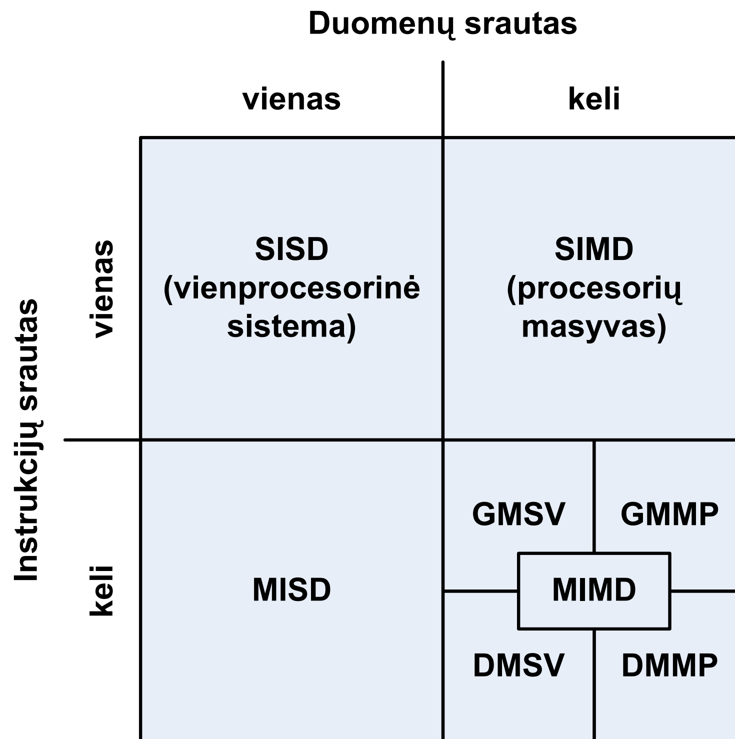
EPIC šaknys VLIW architektūroje

Vykdymas ne iš eilės ir spekuliacinis vykdymas sėkmingai buvo naudojama daugelį metų, siekiant padidinti lygiagretaus programinio kodo vykdymą mikroprocesoriuose. Dėl augančio sudėtingumo siekiant šių tikslų, 1990-tais metais procesorių pramonė pradėjo iš naujo tyrinėti instrukcijų rinkinius, kurie aiškiai koduoja keletą operacijų per instrukciją. Tokių tyrinėjimų pagrindas yra VLIW, kur keletas operacijų yra užkoduotos kiekvienoje instrukcijoje ir vykdomos kelių vykdymo įrenginių.

Vienas iš šios strategijos tikslų iškelti sudėtingą instrukcijų prognozavimą iš procesoriaus aparatūrinio lygmens į programinį kompiliatorių, kuris gali statistiškai atlikti prognozavimą su grįžtamojo ryšio pagalba. Tai eliminuoja sudėtingos prognozavimo schematikos poreikį procesoriuje, ko pasekoje atlaisvinama vieta ir energija kitoms funkcijoms, įtraukiant ir papildomus vykdymo resursus. Ekvivalentiškai svarbus tikslas papildomai naudoti instrukcijų lygio lygiagretinimą, naudojant kompiliatorių surasti ir naudoti papildomas galimybes lygiagrečiam vykdymui.

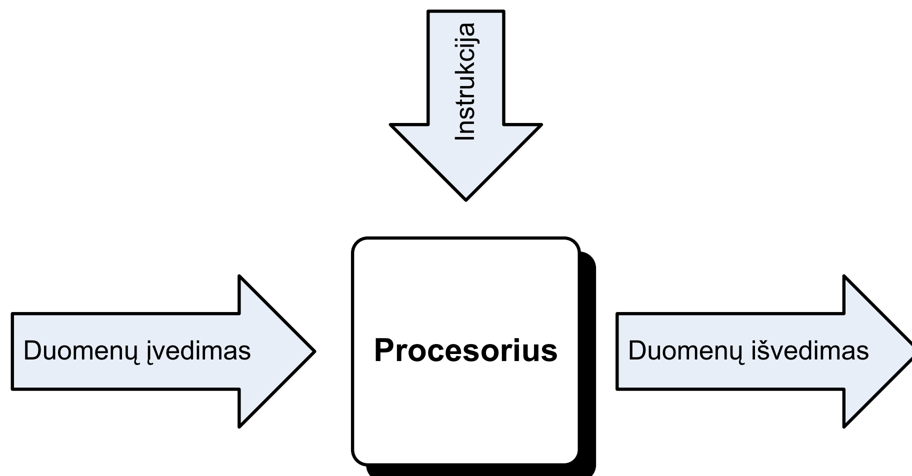
1.5.7. Flynn taksonomija

Flynn taksonomija apima kompiuterių klasifikavimą pagal architektūras, pasiūlyta Michael J.Flynn 1966 m. (8 paveikslas). Flynn apibūdino keturias klasifikacijas paremtas skaičiumi galimų konkuruojančių instrukcijų ir duomenų srautų [16].



8 pav. Kompiuterių klasifikavimas pagal architektūras pagal Flynn

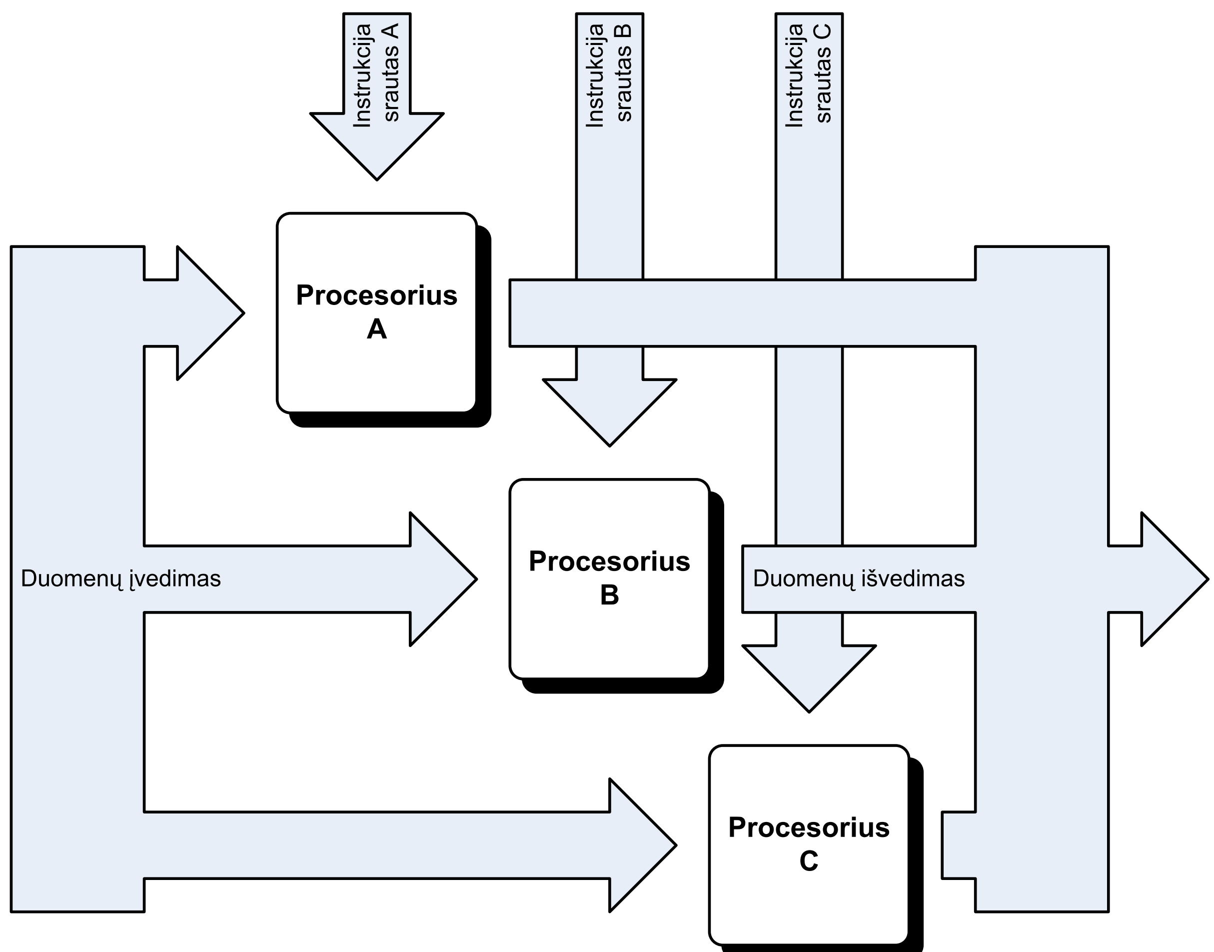
SISD – viena instrukcija su vienu duomenų srautu [angl. *Single Instruction, Single Data stream*]. Tai nuoseklus procesorius, nepalaikantis lygiagretumo nei instrukcijų, nei duomenų rinkinių srautų atžvilgiu (9 paveikslas). Tai tradicinės vieno procesoriaus mašinos.



9 pav. SISD

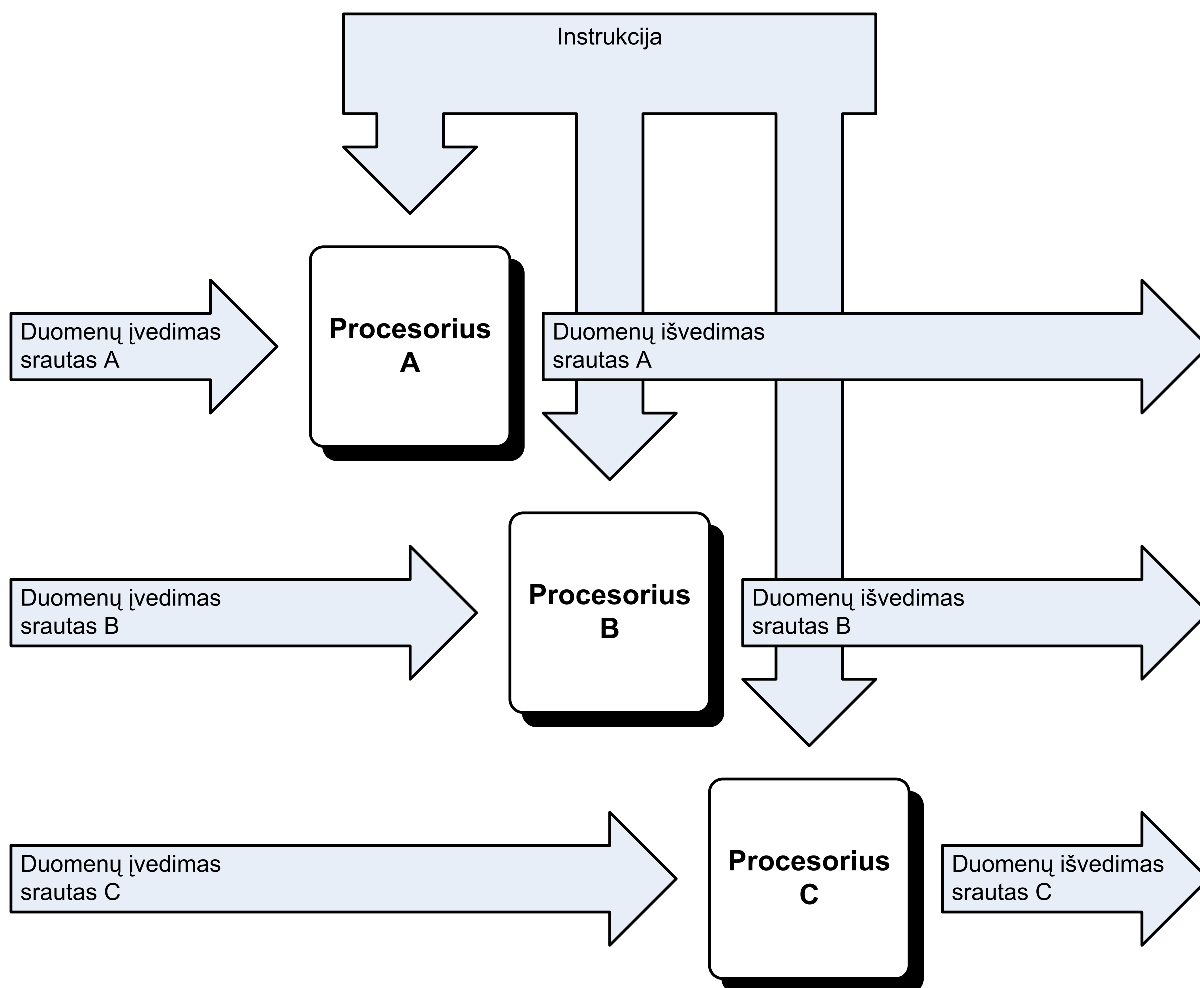
MISD – kelių instrukcijų su tuo pačiu duomenų srautu [angl. *Multiple Instruction, Single Data*] (10 paveikslas). Kiek neprasta architektūra, nes keletas instrukcijų bendrai reikalauja kelių duomenų srautų, kad dirbtų efektyviai. Bet kaip bebūtų, ši architektūra yra

naudojama tada, kai reikalingas rezervuotas lygiagretumas, kaip pavyzdys moderniuose lėktuvuose, kuriems reikalinga keletas rezervuojančių sistemų, vienos gedimo atveju. Buvo pasiūlyta keletas procesorių su MISD architektūra, bet nė viena neišsivystė iki masinės produkcijos lygio.



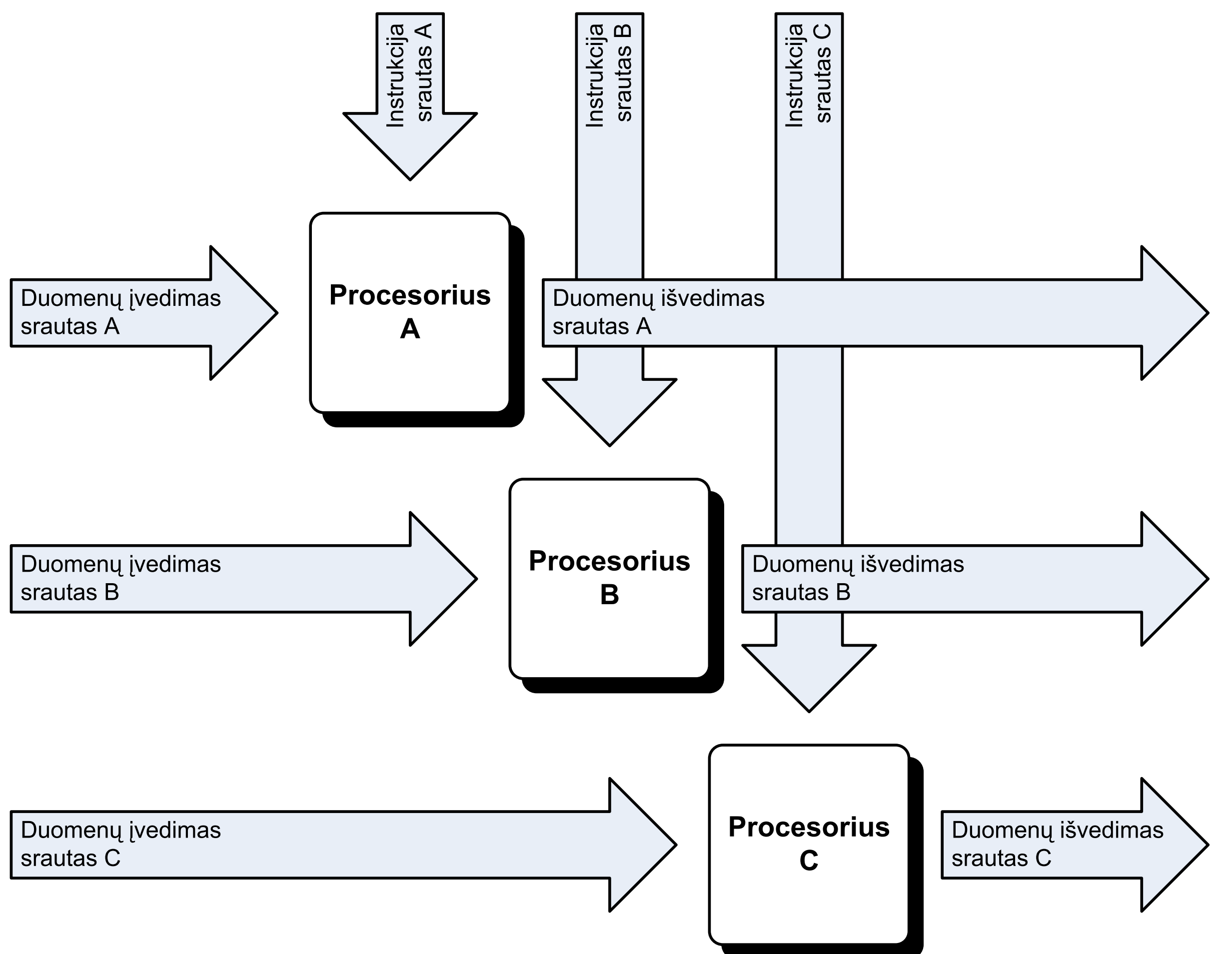
10 pav. MISD

SIMD – viena instrukcija su keliais duomenų srautais [angl. *Single Instruction, Multiple Data*]. Tai architektūra, kai keli duomenų srautai pastatomi prieš vieną instrukcijų srautą, siekiant atlikti operacijas kurios natūraliai gali būti vykdomos lygiagrečiai (11 paveikslas). Naudojama grafiniuose procesoriuose [angl. *GPU – grphical processing unit*] arba masyvų [angl. *array*] procesoriuose.



11 pav. SIMD

MIMD – kelių instrukcijų su keliais duomenų srautais [angl. *Multiple Instruction, Multiple Data*]. Keli autonominiai procesoriai tuo pačiu metu vykdo skirtingas instrukcijas su skirtingais duomenų srautais (12 paveikslas). Paskirstytos sistemos yra priskiriamos prie MIMD architektūros sistemų, nepriklausomai ar dalinasi viena atmintimi, ar naudoja paskirstytą atmintį.



12 pav. MIMD

Toliau aptarsime labiausiai išplitusias architektūras. Yra dvi pagrindinės procesorių instrukcijų rinkinio architektūros:

- CISC;
- RISC.

Pastaruoju metu vis mažėja skirtumai tarp šių architektūrų. Paminėsime pagrindinius bruožus būdingus šioms architektūroms:

- RISC architektūrų tipiški bruožai:
  - Visos instrukcijos yra vienodo ilgio, taigi, nereikia papildomų operacijų jų ilgiui nustatyti.
  - Visos instrukcijos yra neišskaidomos ir minimalios. Dėl to nereikia naudoti papildomų mikroprograminių įrenginių. Programa gali būti geriau

optimizuota. Kai kuriuose RISC procesoriuose atsisakyta netgi daugybos ir dalybos veiksmų.

  - Visos operacijos atliekamos tik su registruose esančiomis reikšmėmis. Taip išvengiama bet kokių galimų prastovų, atsirandančių kreipiantis į operacinę atmintį.
  - Procesoriai turi kelis registrų rinkinius bei instrukcijas, leidžiančias procesoriaus darbo metu pakrauti registrų rinkinį duomenimis tam, kad paskui programa galėtų pereiti prie naujo registrų rinkinio nesikreipdama į atmintį. Tai itin tipiškas RISC procesorių bruožas, kuris nors ir nėra būtinas, tačiau egzistuoja praktiškai visuose RISC procesoriuose. Toks registrų rinkinys vadinamas registrų langu.
  - Tipiški RISC procesoriai yra 32 ar daugiau bitų procesoriai (kai kada jie turi net 256 bitų registrus), nors yra sukurta ir RISC bruožus turinčių 16 bitų procesorių.
  - Visi RISC procesorių registrai yra bendros paskirties.
  - Vektorinės instrukcijos – galimybė vienu metu vykdyti tokį patį kelių operandų sekų apdirbimą, pvz., skaičių A sudėti su skaičiumi B ir paraleliai skaičių C sudėti su skaičiumi D.
  - „Matematiška“ procesoriaus optimizacija – kuriant procesorių, neatsižvelgiama į programavimo kalbų ar kompiliatorių ypatybes, kaip ir į tai, kad kartais tenka rašyti programas asembleriu. Visas komandų rinkinys parenkamas taip, kad procesorius dirbtų maksimaliai efektyviai, nesvarbu kiek sunku bebūtų jam programuoti. Optimizacija užsiima kompiliatoriai.

- CISC architektūrų tipiški bruožai:
  - Instrukcijos gali būti skirtingų ilgių. Tai leidžia efektyviau panaudoti operacinę atmintį.
  - Dažnai pasitaikančios instrukcijų sekos apjungiamos įvedant naujas instrukcijas.
  - Dėl tokio komandų apjungimo programuoti asembleriu darosi daug paprasčiau.
  - Daugelis aritmetinių operacijų vykdoma su duomenimis, esančiais atmintyje. Netgi gali būti operacijų, kurios vienos atminties ląstelės turinį sudeda su kitos ląstelės turiniu ir rezultatą įrašo į trečios ląstelės turinį kartu naudodamos adresus esančius atmintyje.

- Procesoriai turi vieną registrų rinkinį, o darbas su atmintimi pagreitinamas naudojant sudėtingus „kešavimo“ [angl. *cache*] mechanizmus bei kreipimųsi į atmintį prognozavimą.
- Tipiški CISC procesoriai yra ne daugiau nei 32 bitų ir turi daugelį architektūros bruožų būdingų 8 ar 16 bitų procesoriams.
- Dažnai naudojami specializuoti registrai, kurie gali būti net nenurodomi kai kuriose komandose.
- CISC procesoriuose nebūna vektorinių instrukcijų, nebent jos būtų realizuotos atskirame procesoriaus galimybes padidinančiame priede, turinčiame atskirą instrukcijų ir registrų rinkinį.
- „Programinė“ procesoriaus optimizacija. Procesorius kuriamas taip, kad programuoti asembleriu būtų patogu, stengiantis sukurti kuo daugiau aukšto lygio programavimo kalbose naudojamų konstrukcijų ekvivalentų.

Tai daugiausiai problemų kelianti dalis. Tai pagrindinis elementas apibrėžiantis sistemos veiksmų seką, atliekamus veiksmus. Pagrindinė problema yra ta, kad šis elementas priklauso nuo kitų sistemos dedamųjų, kurios šiuo atveju yra dinaminės, nėra iš karto apibrėžtos. Toliau, gilinantis į procesorius, keliamas procesų lygiagretinimo [angl. *pipeline*] klausimas, kas dar labiau apsunkina komponento bendrinimą.

1.5.8. CISC prieš RISC

Pirminė procesorių projektuotojų užduotis - pagerinti procesorių našumą. Našumas apibūdinamas atliktu darbu arba galimu atlikti darbu per duotą laiko tarpą. Skirtingos instrukcijos atlieką skirtingą darbo kiekį.

Padidinti procesoriaus darbo našumą galima dviems būdais: įvykdyti instrukciją per kiek įmanoma trumpesnį laiko tarpą, arba instrukcija atlieka daugiau operacijų. Našumo didinimas, instrukcijos vykdymo laiko mažinimas paprastai reiškia didinti procesoriaus darbinį dažnį. Našumo didinimas, instrukcijos darbo kiekio didinimas reiškia instrukcijos sudėtingumo didinimą. Idealiu atveju siekiama abiejų tikslų, bet tai suprantama yra projektavimo kompromisas. Sunku sudėtingas instrukcijas įvykdyti per kuo trumpesnį laiką.

Pateikimas realaus gyvenimo analogiškas pavyzdys. Įsivaizduokite, kad jūs minate dviratį. Norint pasiekti tikslą kuo greičiau jūs galite naudoti žemesnę pavarą ir minti

labai greitai, arba naudoti aukštesnę pavarą ir minti iš visų jėgų. Jūs galite minti ir stipriai ir greitai, bet jūs niekada neminsite taip greitai su aukšta pavara, kaip galite su žema.

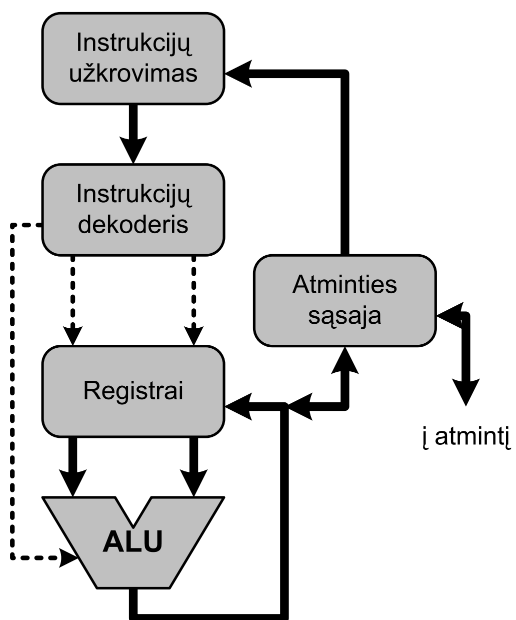
Kompromisą bazinėse instrukcijų rinkinio projektavimo filosofijose atspindi du pavadinimai suteikti instrukcijų rinkinių architektūroms. CISC atstovauja sudėtingo instrukcijų rinkinio architektūros kompiuterius. Šio tipo procesoriai naudoja daug sudėtingų instrukcijų ir su kiekviena stengiasi atlikti kuo daugiau užduočių. RISC atstovauja racionalaus instrukcijų rinkinio architektūros kompiuterius. Šio tipo procesoriai naudoja mažai paprastų instrukcijų ir su kiekviena stengiasi atlikti kuo mažesnį kiekį užduočių, bet jas stengiasi įvykdyti kaip galima greičiau.

Tai yra svarbus argumentas pasirenkant architektūros tipą šiuolaikinių procesorių projektavime, nes šis pasirinkimas suriša su ateities projektais. Jis įpareigoja užtikrinti, kad naujieji produktai palaikys esamą programinę įrangą. Be to riba tarp RISC ir CISC architektūrų pastaraisiais metais tapo ne be tokia akivaizdi. Vis labiau populiarėja šių architektūrų maišymas sistemoje. Dėl to pradėti naudoti vertimo procesoriai.

1.6. Apžvalgos išvados

Atkartojimas yra pagrindas tiek programiniams, tiek aparatūriniams komponentams kurti ir apskritai vienlustėms sistemoms projektuoti. Nors atkartojimas ir negali būti laikomas visų projektavimo problemų sprendimu, jis yra metodologinis (plačiąja prasme) ir technologinis (siaurąja prasme) pagrindas. Atkartojimas gali būti pasiekiamas per abstrakcijos lygmens kėlimą, apibendrinimą ir automatizavimą [2][17].

Taigi apibendrinant galima apibūdinti supaprastintą procesoriaus struktūrą (13 paveikslas). Instrukcijų užkrovimas [angl. *Fetch*], instrukcijos dekodavimas, jos vykdymas, rezultatų išsaugojimas. Duomenys saugojami registruose ir išorinėje atmintyje. Aritmetinei veiksmai atliekami ALU.



13 pav. Supaprastinta procesoriaus struktūrinė schema

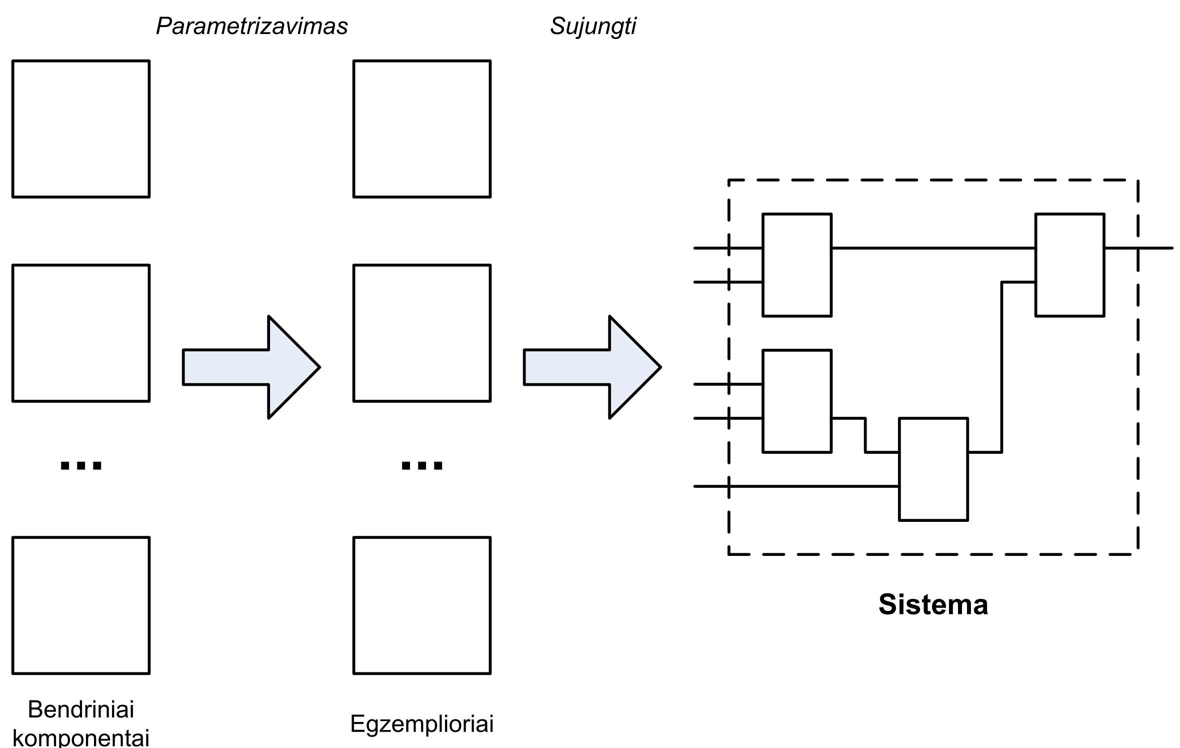
2. PROJEKTINĖ DALIS

Šioje dalyje bus pasiūlytos ir aptartos procesorinio komponento bendrinimo kryptis. Panaudotas mikroprocesorius, kuris yra realizuotas VHDL kalboje. Šis mikroprocesorius realizuotas funkciniame lygmenyje, bei sintezuojamas.

2.1. Atskirų procesoriaus komponentų bendrinimo kryptis

Atskirų mikroprocesoriaus komponentų bendrinimo sieksime naudodamiesi VHDL metaprogramavimo – parametrizavimo galimybėmis (14 paveikslas). Yra trys pagrindinės VHDL kalbos parametrizavimo konstrukcijos, tai:

- Constant struktūra;
- Generic struktūra;
- Generate struktūra.



14 pav. Bendrinių komponentų įdiegimas

Bendrinimo kryptis duomenų magistralės, adreso magistralės ir registrų adreso magistralės parametrizavimas naudojant „generic“ struktūrą. Esamo procesoriaus

komponentai modifikuojami taip, kad būtų galima parametrizuoti pasirinktus parametrus. Patikrinsime sintezavimo galimybė. Jei komponentas bus sintezuojamas, palyginsime sintezės rezultatus.

Procesorius susideda iš sekančių komponentų:

- ACC – akumuliatoriaus registras
- ALU – aritmetinis loginis įrenginys
- CTRL – valdymo logika
- IAR – netiesioginio kreipimosi adreso registras
- IR – instrukcijų rinkinys
- PC – programinis skaitiklis
- PORT – įėjimo/išėjimo registras
- RAM – atmintis
- REG – bendros paskirties registrai

- CPU – apjungia ACC, ALU, CTRL, IAR, IR, PC, PORT, REG ir RAM komponentus
- CORE – apjungia ACC, ALU, CTRL, IAR, IR, PC, PORT ir REG komponentus
- SYSTEM – apjungia CORE ir RAM komponentus

Procesoriaus instrukcijų rinkinys pateiktas 1 lentelėje.

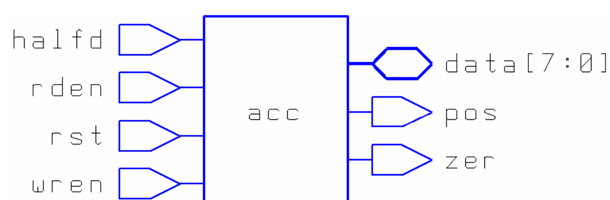
1 lentelė. Instrukcijų rinkinys

Nr.	**Operacija**	**Operacijos kodas**	**Operacijos veiksmas**	**Operacijos aprašymas**
1	LDA A, rrr	0001xrrr	A<-R[rrr]	akumuliatoriaus užkrovimas iš registro
2	STA rrr,A	0010xrrr	R[rrr]<-A	akumuliatoriaus išsaugojimas registre
3	LDM A, aaaaaa	0011xxxx xxaaaaaa	A<-M[aaaaaa]	akumuliatoriaus užkrovimas iš atminties
4	STM aaaaaa, A	0100xxxx xxaaaaaa	M[aaaaaa]<-A	akumuliatoriaus išsaugojimas atmintyje
5	LDI A, iiii	0101iiii	A<-0000iiii	netiesioginis akumuliatoriaus užkrovimas

Nr.	Operacija	Operacijos kodas	Operacijos veiksmas	Operacijos aprašymas
6	JMPR rel	0110jjjj	if(jjjj!=0000) then PC+=jjjj	reliatyvus peršokimas
7	JMP addr	01100000 xxaaaaaa	if(jjjj==0000) then PC=aaaaaa	peršokimas adresu
8	JZR rel	0111jjjj	if((A==0)&(jjjj!=0000)) then PC+=jjjj	reliatyvus peršokimas, kai akumuliatorius lygus nuliui
9	JZ addr	01110000 xxaaaaaa	if((A==0)&(jjjj==0000)) then PC=aaaaaa	peršokimas adresu, kai akumuliatorius lygus nuliui
10	JPR rel	1000jjjj	if((A>0)&(jjjj!=0000)) then PC+=jjjj	reliatyvus peršokimas, kai akumuliatorius teigiamas
11	JP addr	10000000 xxaaaaaa	if((A>0)&(jjjj==0000)) then PC=aaaaaa	peršokimas adresu, kai akumuliatorius teigiamas
12	AND A, rrr	1001xrrr	A<-A&&R[rrr]	loginė konjunkcijos (ir) operacija
13	OR A, rrr	1010xrrr	A<-A||R[rrr]	loginė disjunkcijos (arba) operacija
14	ADD A, rrr	1011xrrr	A<-A+R[rrr]	aritmetinė sudėtis
15	SUB A, rrr	1100xrrr	A<-A-R[rrr]	aritmetinė atimtis
16	NOT A	1101x000	A<-!A	loginė inversijos operacija
17	INC A	1101x001	A<-A+1	didinimas vienetu
18	DEC A	1101x010	A<-A-1	mažinimas vienetu
19	SHFL A	1101x011	A<-A<<1	loginis postūmis į kairę
20	SHFR A	1101x100	A<-A>>1	loginis postūmis į dešinę
21	INa A	1110xxx0	A<-input	akumuliatoriaus užkrovimas iš išorės
22	OUTa A	1110xxx1	output<-A	akumuliatoriaus įrašymas į išorę
23	HALT	1111xxxx		procesoriau stabdymas
24	NOP	00000000		jokios operacijos

2.1.1. ACC komponentas

Tai akumuliatorius arba darbinis registras (15 paveikslas). Jis naudojamas vienam iš operandų saugoti, reikalingų aritmetiniams, loginiams veiksmams atlikti, bei išsaugojami šių operacijų rezultatai.



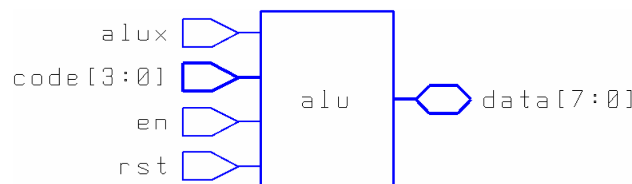
15 pav.ACC komponentas

ACC komponento valdymo signalai:

- Įėjimai:
  - rst – perkrovimo valdymo signalas;
  - halfd – nurodo, kad naudojama tik pusė duomenų magistralės;
  - rden – registro reikšmės nuskaitymo valdymo signalas;
  - wren – registro reikšmės įrašymo valdymo signalas.
- Išėjimai:
  - pos – vėliavėlė, kuri nurodo, kad registre esantys duomenys yra teigiami;
  - zer – vėliavėlė, kuri nurodo, kad registre esantys duomenys lygūs nuliui.
- Įėjimai/išėjimai:
  - data[7:0] – sąsaja su duomenų magistrale.

2.1.2. ALU komponentas

Aritmetinis loginis įrenginys (16 paveikslas). Jis skirtas atlikti aritmetiniams, loginiams veiksmams.



16 pav. ALU komponentas

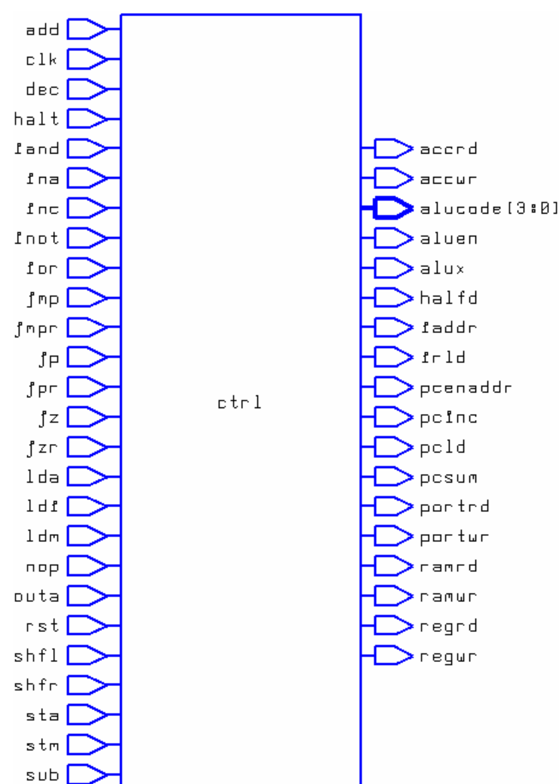
ALU komponento valdymo signalai:

- Įėjimai:
  - rst – perkrovimo valdymo signalas;
  - alux – operando užkrovimo iš duomenų magistralės valdymo signalas;
  - en – operacijos aktyvavimo valdymo signalas;
  - code[3:0] – operacijos kodas (2 lentelė).
- Įėjimas/išėjimas:
  - data[7:0] – sąsaja su duomenų magistrale.

operacija	operacijos kodas	operacijos aprašymas
AND	0000	loginė konjunkcijos (ir) operacija
OR	0001	loginė disjunkcijos (arba) operacija
ADD	0010	aritmetinė sudėtis
SUB	0011	aritmetinė atimtis
NOT	0100	loginė inversijos operacija
INC	0101	didinimas vienetu
DEC	0110	mažinimas vienetu
SHFL	0111	loginis postūmis į kairę
SHFR	1000	loginis postūmis į dešinę

2.1.3. CTRL komponentas

Valdymo įrenginys (17 paveikslas). Tai pagrindinis mikroprocesoriaus komponentas. Jis atsakingas už visą mikroprocesoriaus valdymo veiksmų seką. Iš esmės tai yra baigtinės būsenos automatas. Tai yra kiekvieno mikroprocesoriaus specifinis įrenginys. Kadangi šiuo metu atmintys yra pakankamai greitos, tai šis komponentas įtakoja pačio procesoriaus našumą. Pagrindinis projektuotojų uždavinys, šiuolaikinių procesorių projektavimo etape, yra suprojektuoti gerą valdymo įrenginį.



17 pav. CTRL komponentas

CTRL komponento valdymo signalai

- Įėjimai
  - Bendri
    - rst – perkrovimo valdymo signalas
    - clk – taktavimo valdymo signalas
  - Instrukcijos
    - lda – „lda“ instrukcijos valdymo signalas
    - sta – „sta“ instrukcijos valdymo signalas
    - ldm – „ldm“ instrukcijos valdymo signalas
    - stm – „stm“ instrukcijos valdymo signalas
    - ldi – „ldi“ instrukcijos valdymo signalas
    - jmpr – „jmpr“ instrukcijos valdymo signalas
    - jmp – „jmp“ instrukcijos valdymo signalas
    - jzr – „jzr“ instrukcijos valdymo signalas
    - jz – „jz“ instrukcijos valdymo signalas
    - jpr – „jpr“ instrukcijos valdymo signalas
    - jp – „jp“ instrukcijos valdymo signalas
    - iand – „iand“ instrukcijos valdymo signalas
    - ior – „ior“ instrukcijos valdymo signalas
    - add – „add“ instrukcijos valdymo signalas
    - sub – „sub“ instrukcijos valdymo signalas
    - inot – „inot“ instrukcijos valdymo signalas
    - inc – „inc“ instrukcijos valdymo signalas
    - dec – „dec“ instrukcijos valdymo signalas
    - shfl – „shfl“ instrukcijos valdymo signalas
    - shfr – „shfr“ instrukcijos valdymo signalas
    - ina – „ina“ instrukcijos valdymo signalas
    - outa – „outa“ instrukcijos valdymo signalas
    - halt – „halt“ instrukcijos valdymo signalas
    - nop – „nop“ instrukcijos valdymo signalas
- Išėjimai
  - ACC valdymo signalai
    - accrd – akumuliatoriaus skaitymo valdymo signalas

- accwr – akumuliatoriaus rašymo valdymo signalas
- halfd – pusės duomenų magistralės valdymo signalas

o ALU valdymo signalai

- aluen – operacijos vykdymo valdymo signalas
- alux – operando užkrovimo valdymo signalas
- alucode[3:0] – operacijos kodas

o IAR valdymo signalas

- iaddr – netiesioginio adreso registro valdymo signalas

o IR valdymo signalas

- irld – instrukcijų registro-dekoderio valdymo signalas

o PC valdymo signalai

- pcinc – pc didinimo vienetu valdymo signalas
- pcld – pc užkrovimo valdymo signalas
- pcsum – pc sumavimo valdymo signalas
- pcenaddr – pc aktyvavimo valdymo signalas

o RAM valdymo signalai

- ramrd – atminties skaitymo valdymo signalas
- ramwr – atminties įrašymo valdymo signalas

o REG valdymo signalai

- regrd – registrų skaitymo valdymo signalas
- regwr – registrų įrašymo valdymo signalas

o PORT valdymo signalai

- portrd – įvedimo skaitymo valdymo signalas
- portwr – išvedimo įrašymo valdymo signalas

2.1.4. IAR komponentas

Netiesioginio adreso registras (18 paveikslėlis). Naudojamas užkrauti adresui iš duomenų magistralės apeinant „pc“.

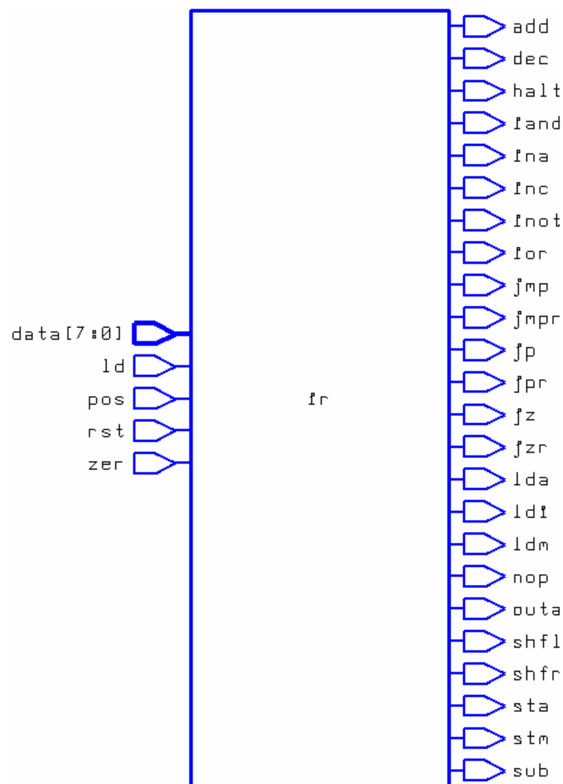
18 pav. IAR komponentas

IAR komponento valdymo signalai:

- Įėjimai:
  - iaddr – aktyvavimo valdymo signalas;
  - data[5:0] – sąsaja su duomenų magistrale.
- Išėjimai:
  - addr[5:0] – sąsaja su adresų magistrale.

2.1.5. IR komponentas

Instrukcijų registras ir dekoderis (19 paveikslas). Šis komponentas atpažįsta instrukciją ir išduoda valdymo signalą mikroprocesoriaus valdymo įrenginiui. Tai taip pat yra kritinis sistemos komponentas, nes jis glaudžiai susijęs su valdymo įrenginiu, ir gavus neapibrėžtą komandą sustabdo sistemos darbą.



19 pav. IR komponentas

IR komponento valdymo signalai

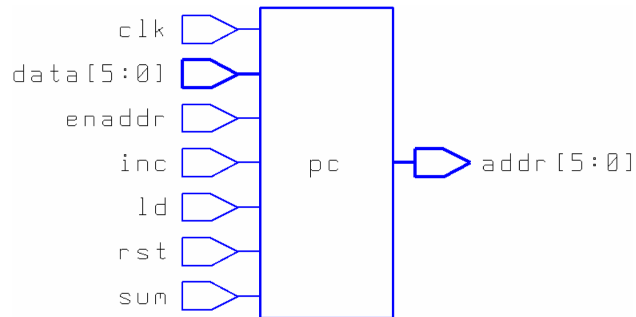
- Įėjimai
  - rst – perkrovimo signalas

- ld – instrukcijos užkrovimo signalas
- pos – teigiamo ženklo vėliavėlė
- zer – nulio vėliavėlė
- data[7:0] – sąsaja su duomenų magistrale

- Išėjimai
  - lda – „lda“ instrukcijos valdymo signalas
  - sta – „sta“ instrukcijos valdymo signalas
  - ldm – „ldm“ instrukcijos valdymo signalas
  - stm – „stm“ instrukcijos valdymo signalas
  - ldi – „ldi“ instrukcijos valdymo signalas
  - jmpr – „jmpr“ instrukcijos valdymo signalas
  - jmp – „jmp“ instrukcijos valdymo signalas
  - jzr – „jzr“ instrukcijos valdymo signalas
  - jz – „jz“ instrukcijos valdymo signalas
  - jpr – „jpr“ instrukcijos valdymo signalas
  - jp – „jp“ instrukcijos valdymo signalas
  - iand – „iand“ instrukcijos valdymo signalas
  - ior – „ior“ instrukcijos valdymo signalas
  - add – „add“ instrukcijos valdymo signalas
  - sub – „sub“ instrukcijos valdymo signalas
  - inot – „inot“ instrukcijos valdymo signalas
  - inc – „inc“ instrukcijos valdymo signalas
  - dec – „dec“ instrukcijos valdymo signalas
  - shfl – „shfl“ instrukcijos valdymo signalas
  - shfr – „shfr“ instrukcijos valdymo signalas
  - ina – „ina“ instrukcijos valdymo signalas
  - outa – „outa“ instrukcijos valdymo signalas
  - halt – „halt“ instrukcijos valdymo signalas
  - nop – „nop“ instrukcijos valdymo signalas

2.1.6. PC komponentas

Programinis skaitiklis (20 paveikslas). Šis komponentas yra valdomas valdymo įrenginio ir užtikrina teisingą adresą adresų magistralėje. Taip pat turi papildomų funkcijų, specifinių šiam mikroprocesoriniam komponentui.



20 pav. PC komponentas

PC komponento valdymo signalai

- Įėjimai
  - rst – perkrovimo valdymo signalas
  - clk – taktavimo valdymo signalas
  - inc – leidimas skaičiuoti valdymo signalas
  - ld – duomenų užkrovimas valdymo signalas
  - sum – sumavimo funkcijos aktyvavimo valdymo signalas
  - enaddr – adreso išdavimo valdymo signalas
  - data[5:0] – sąsaja su duomenų magistrale
- Išėjimai
  - addr[5:0] – sąsaja su adresų magistrale

2.1.7. PORT komponentas

Išorinių duomenų apsikeitimo komponentas (21 paveikslas). Šis komponentas jungia vidinę duomenų magistralę su išorine. Tai įgalina prie procesoriaus prijungti papildomas atmintis ar kito tipo signalus, taip pat valdyti kitus komponentus.

21 pav. PORT komponentas

PORT komponento valdymo signalai

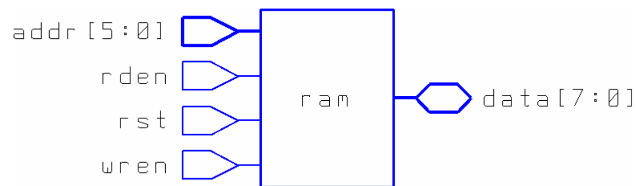
- Įėjimai

- portrd – skaitymo valdymo signalas
- portwr – rašymo valdymo signalas

- Įėjimai/išėjimai
  - data[7:0] – sąsaja su vidine duomenų magistrale
  - datai[7:0] – sąsaja su išorine duomenų magistrale

2.1.8. RAM komponentas

Atminties komponentas (22 paveikslas). Tai elementas; kuriame saugojama programos kodas, kintamųjų reikšmės, taip pat rezultatai. Šis komponentas ventilių skaičiumi ir plotu užima didžiąją sistemos dalį. Šios sistemos atveju naudojama 64 įrašai po 8 bitus. Iš viso susidaro 64 baitai.



22 pav. RAM komponentas

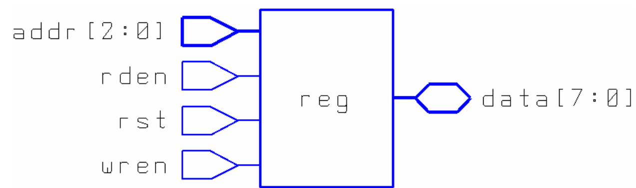
RAM komponento valdymo signalai

- Įėjimai
  - rst – perkrovimo valdymo signalas
  - rden – skaitymo valdymo signalas
  - wren – rašymo valdymo signalas
  - addr[5:0] – sąsaja su adresų magistrale
- Įėjimai/išėjimai
  - data[7:0] – sąsaja su duomenų magistrale

2.1.9. REG komponentas

Registrų failo komponentas (23 paveikslas). Šiame komponentą laikomi duomenys reikalingi atlikti aritmetiniams, loginiams veiksmams. Taip pat išsaugojami aritmetinių, loginių veiksmų rezultatai. Šios sistemos atveju vienas operandas laikomas akumuliatoriaus registre, kitas registrų faile. Duomenys iš atminties įkraunami per

akumuliatoriaus registrą, taip pat ir išsaugojami per akumuliatoriaus registrą. Šios sistemos atveju yra 8 registrai po 8 bitus.



23 pav. REG komponentas

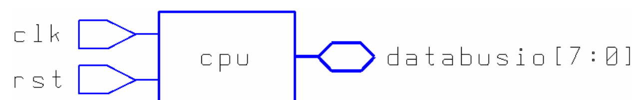
REG komponento valdymo signalai

- Įėjimai
  - rst – perkrovimo valdymo signalas
  - rden – skaitymo valdymo signalas
  - wren – rašymo valdymo signalas
  - addr[2:0] – sąsaja su adresų magistrale
- Įėjimai/išėjimai
  - data[7:0] – sąsaja su duomenų magistrale

2.1.10. CPU komponentas

Tai komponentas apjungiantis ACC, ALU, CTRL, IAR, IR, PC, PORT, REG ir RAM komponentus (24 paveikslas). Tai yra baigtinis mikroprocesoriaus komponentas (25 paveikslas).

24 pav. CPU komponento struktūra



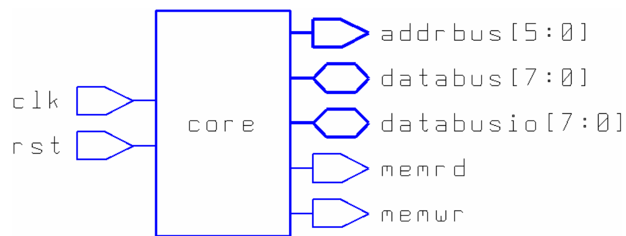
25 pav. CPU komponentas

CPU komponento valdymo signalai

- Įėjimai
  - rst – perkrovimo valdymo signalas
  - clk – taktavimo valdymo signalas
- Įėjimai/išėjimai
  - databusio[7:0] – sąsaja su išorine duomenų magistrale

2.1.11. CORE komponentas

Tai komponentas apjungiantis ACC, ALU, CTRL, IAR, IR, PC, PORT, ir REG komponentus (26 paveikslas). Šis komponentas yra sistemos branduolys.



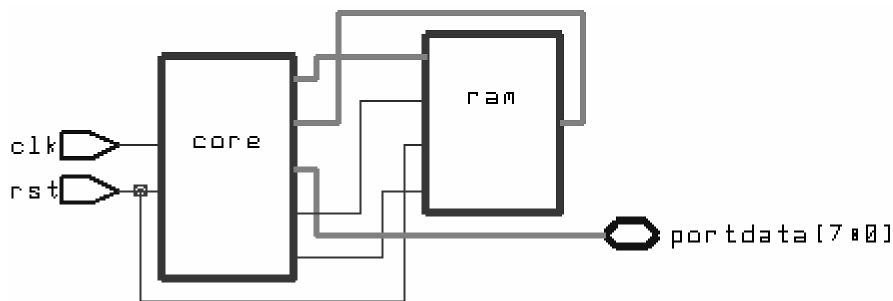
26 pav. CORE komponentas

CORE komponento valdymo signalai

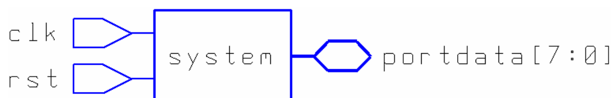
- Įėjimai
  - rst – perkrovimo valdymo signalas
  - clk – taktavimo valdymo signalas
- Išėjimai
  - memrd – atminties skaitymo valdymo signalas
  - memwr – atminties įrašymo valdymo signalas
  - addrbus[5:0] – sąsaja su duomenų magistrale
- Įėjimai/išėjimai
  - databus[7:0] – sąsaja su vidine duomenų magistrale
  - databusio[7:0] – sąsaja su išorine duomenų magistrale

2.1.12. SYSTEM komponentas

Tai komponentas apjungiantis CORE ir RAM komponentus (27 paveikslas). Šis komponentas kaip ir CPU yra baigtinis mikroprocesoriaus komponentas (28 paveikslas).



27 pav. SYSTEM komponento struktūra



28 pav. SYSTEM komponentas

SYSTEM komponento valdymo signalai.

- Įėjimai:
  - rst – perkrovimo valdymo signalas;
  - clk – taktavimo valdymo signalas.
- Įėjimai/išėjimai:
  - portdata[7:0] – sąsaja su išorine duomenų magistrale.

2.2. Instrukcijų dekoderio ir valdymo įrenginio bendrinimo kryptis

Instrukcijų dekoderis ir valdymo komponentai yra labai svarbūs, bendrame viso mikroprocesorinio elemento plane. Jie apsprendžia mikroprocesoriaus architektūrą, našumą, komandų sistemą. Tai daugiausiai pastangų, iš projektuotojų, reikalaujančios dalys. Kaip pavyzdys, Intel ir AMD firmų procesoriai, naudojami šiuolaikiniuose kompiuteriuose, naudoja tą patį instrukcijų rinkinį ir priskiriami CISC architektūrai. Instrukcijų vykdymo seka ir pati procesoriaus vidinė architektūra yra skirtinga. Šių kompanijų projektuotojai skirtingai realizuoja valdymo įrenginius. Tai labai akivaizdžiai įrodo procesorių našumo skirtumas.

Šiuo metu patys galingiausi procesoriai yra CISC architektūros, bet rinkoje didžiąją dalį užima RISC. CISC architektūros procesoriai dažniausiai naudojami kompiuteriuose, o RISC visoje buitinėje elektronikoje, mobilūs telefonai, smulkioji elektronika ir panašiai.

Naudosime pasiūlyto mikroprocesoriaus IR ir CTRL komponentus. Ieškosime būdų juos apibendrinti. Nors tai atstovauja RISC architektūros, kas rodo, kad šio mikroprocesoriaus komandų sistema yra racionali, bet nuolat mažėjant skirtumams tarp RISC ir CISC architektūrų, galima ieškoti bendrinimo krypčių. Kaip pavyzdys yra Atmel ir Microchip firmų mikroprocesoriai. Jie abu priklauso RISC architektūrai, bet instrukcijų rinkiniai skiriasi Atmel – 120 instrukcijų (7 priedas), o Microchip – 33 instrukcijos (5 priedas).

Pasirinktasis mikroprocesorius turi 24 instrukcijas. Žinoma nėra tikslinga siekti kiekvienos instrukcijos parametrizavimo, kadangi prarasime mikroprocesoriaus, kaip sistemos, funkcionalumą. Prieš bendrinant būtina išanalizuoti, suskirstyti instrukcijas į grupes. Be atminties valdymo instrukcijos mikroprocesorius netektų funkcionalumo, nors jis ir būtų sintezuojamas. Nors ir kaip kiltų noras skirstyti aritmetines ir logines operacija pagal jų tipą, bet tai nebūtų teisinga. Tiksliau būtų skirstyti jas pagal operandų skaičių operacijai atlikti. Atrodyto iš pirmo žvilgsnio galima atsisakyti aritmetinių operacijų, nes jas galima realizuoti loginėmis operacijomis, bet yra loginių operacijų kurios atliekamos su vienu operandu, ir dviem kaip ir aritmetinės. Pažvelgus iš kitos pusės, tai supaprastintų ALU. Bet mes orientuojamės į IR ir CTRL komponentus.

Išanalizavus, pasirinkto mikroprocesoriaus instrukcijų rinkinį yra pasirenkama bendrinimo kryptis. Atsisakoma reliatyvių peršokimų, kadangi juos pilnai dengia peršokimai adresu (3 lentelė).

3 lentelė. Peršokimų instrukcijos

Nr.	**Operacija**	**Operacijos kodas**	**Operacijos veiksmas**	**Operacijos aprašymas**
6	JMPR rel	`0110jjjj`	`if(jjjj!=0000)` `then PC+=jjjj`	reliatyvus peršokimas
7	JMP addr	`01100000` `xxaaaaaa`	`if(jjjj==0000)` `then PC=aaaaaa`	peršokimas adresu
8	JZR rel	`0111jjjj`	`if((A==0)&(jjjj!=0000))` `then PC+=jjjj`	reliatyvus peršokimas, kai akumuliatorius lygus nuliui
9	JZ addr	`01110000` `xxaaaaaa`	`if((A==0)&(jjjj==0000))` `then PC=aaaaaa`	peršokimas adresu, kai akumuliatorius lygus nuliui
10	JPR rel	`1000jjjj`	`if((A>0)&(jjjj!=0000))` `then PC+=jjjj`	reliatyvus peršokimas, kai akumuliatorius teigiamas
11	JP addr	`10000000` `xxaaaaaa`	`if((A>0)&(jjjj==0000))` `then PC=aaaaaa`	peršokimas adresu, kai akumuliatorius teigiamas

Komponentai realizuoti VHDL ir SystemC kalbomis. Jie yra sintezuojami. Komponentai buvo modifikuoti taip, kad būtų galima parametrizuoti reliatyvius peršokimus. Bus palyginti sintezės rezultatai parametrizuotų ir neparametrizuotų komponentų.

3. TYRIMAS

3.1. Tyrimo metodika

Pasirenkamos dvi bendrinimo kryptys. Procesoriaus komponentų bendrinimas siekiant parametrizuoti duomenų, adresų ir registrų adresų magistralių pločius. Bendrinimas atliekamas naudojantis tik VHDL metaprogramavimo priemones. Bendrinių komponentų funkcionalumas tikrinamas naudojant *Cadence SimVision 5.70-s005* programinį paketą. Sintezės rezultatams gauti naudojamas *Synopsys Design Analyzer* programinis paketas. Komponentai sintezuojami naudojant *tc6a_cbacore* biblioteką.

Antrojo tyrimo metu siekiama instrukcijų rinkinio parametrizavimo, bendrinant CTRL ir IR komponentus. Eksperimentui naudojami komponentai realizuoti VHDL ir SystemC kalba. Komponentų realizuotų VHDL kalba funkcionalumas tikrinamas naudojama *Cadence SimVision 5.70-s005* programinį paketą. Komponentų realizuotų SystemC modeliavimui naudojamas SystemC kompiliatorius. Po modeliavimo gaunama „*trace*“ byla, kuri analizuojama naudojant *gtkwave* programą. Šių elementų sintezė atliekama naudojant *Synopsys Design Analyzer* programinį paketą. Komponentai sintezuojami naudojant *tc6a_cbacore* biblioteką.

3.2. Procesoriaus komponentų bendrinimo krypties rezultatai

Komponento bendrinimo kryptis yra duomenų magistralės parametrizavimas ir adresų magistralės parametrizavimas. Pateiktasis mikroprocesorius naudoja 8 bitų duomenų magistralę ir 6 bitų adresų magistralę.

Bendrinimas atliktas VHDL kalbos metaprogramavimo priemonėmis naudojant generic konstrukciją. Įvesti parametrizavimo kriterijai, tai duomenų magistralės, adresų magistralės ir registrų adresų magistralės pločiai, kas tiesiogiai atspindi registrų skaičių.

Apibendrinto procesorinio komponento CPU_8 ir SYSTEM_8 palyginimas su nebendrintu pradiniu variantu CPU ir SYSTEM (4 lentelė). Pradinis variantas yra su 8 bitų duomenų ir 6 bitų adresų magistralėmis. Taigi bendrinį procesorinį komponentą parametrizuojame taip, kad duomenų ir adresų magistralių pločiai sutaptų

4 lentelė. Bendrinto ir nebendrinto komponento sintezės rezultatų palyginimas

komponentas	**ventilių skaičius, vnt.**	**ventilių plotas**	**bendras plotas**	**vėlinimas, ns**
cpu	2405	5093,279785	29674,525391	1,04
cpu_8	2405	5093,279785	29674,525391	1,04
cpu_ram	1681	3583,879883	21297,458984	6,45
cpu_8_ram	1681	3583,879883	21297,458984	6,45
system	2420	5116,250000	29837,669922	1,04
system_8	2420	5116,250000	29837,669922	1,04
system_ram	1681	3550,510010	21264,089844	6,16
system_8_ram	1681	3550,510010	21264,089844	6,16

Kadangi matome, kad didžiąją ventilių skaičiaus dalį užima atmintis, tad į tolimesnius palyginimus jos neįtraukiame. Naudojame komponentą „Core“ kaip atraminį pavyzdį, o Core_gen atitinkamai sintezuojame Core_8 – 8 bitų duomenų magistralė ir 6 bitų adresų magistralė, Core_12 – 12 bitų duomenų magistralė ir 10 bitų adresų magistralė ir Core_16 – 16 bitų duomenų magistralė ir 14 bitų adresų magistralė (5, 6, 7, 8 lentelės).

5 lentelė. Core komponento sintezės rezultatai

komp.	**Core**			
	ventilių skaičius, vnt.	**ventilių plotas**	**bendras plotas**	**vėlinimas, ns**
core	24	1564,300049	8629,611328	36,66
acc	30	92,099998	365,225006	2,24
alu	107	248,460007	1272,678833	8,41
ctrl	212	324,619995	2362,276123	6,82
iar	6	9,480000	74,167503	0,37
ir	64	83,669998	644,295044	3,65
pc	56	153,339996	674,433777	1,46
port	17	67,099998	246,787506	1,68
reg	231	526,890015	2949,077637	1,95
core_all	739	1564,300049	8629,611328	36,66

6 lentelė. Skirtingų parametrų bendrinto komponento sintezės rezultatai (core_8)

komp.	Core_8			
	ventilių skaičius, vnt.	**ventilių plotas**	**bendras plotas**	**vėlinimas, ns**
core	24	1564,300049	8629,611328	36,66
acc	30	92,099998	365,225006	2,24
alu	107	248,460007	1272,678833	8,41
ctrl	212	324,619995	2362,276123	6,82
iar	6	9,480000	74,167503	0,37
ir	64	83,669998	644,295044	3,65
pc	56	153,339996	674,433777	1,46
port	17	67,099998	246,787506	1,68
reg	231	526,890015	2949,077637	1,95
core_all	739	1564,300049	8629,611328	36,66

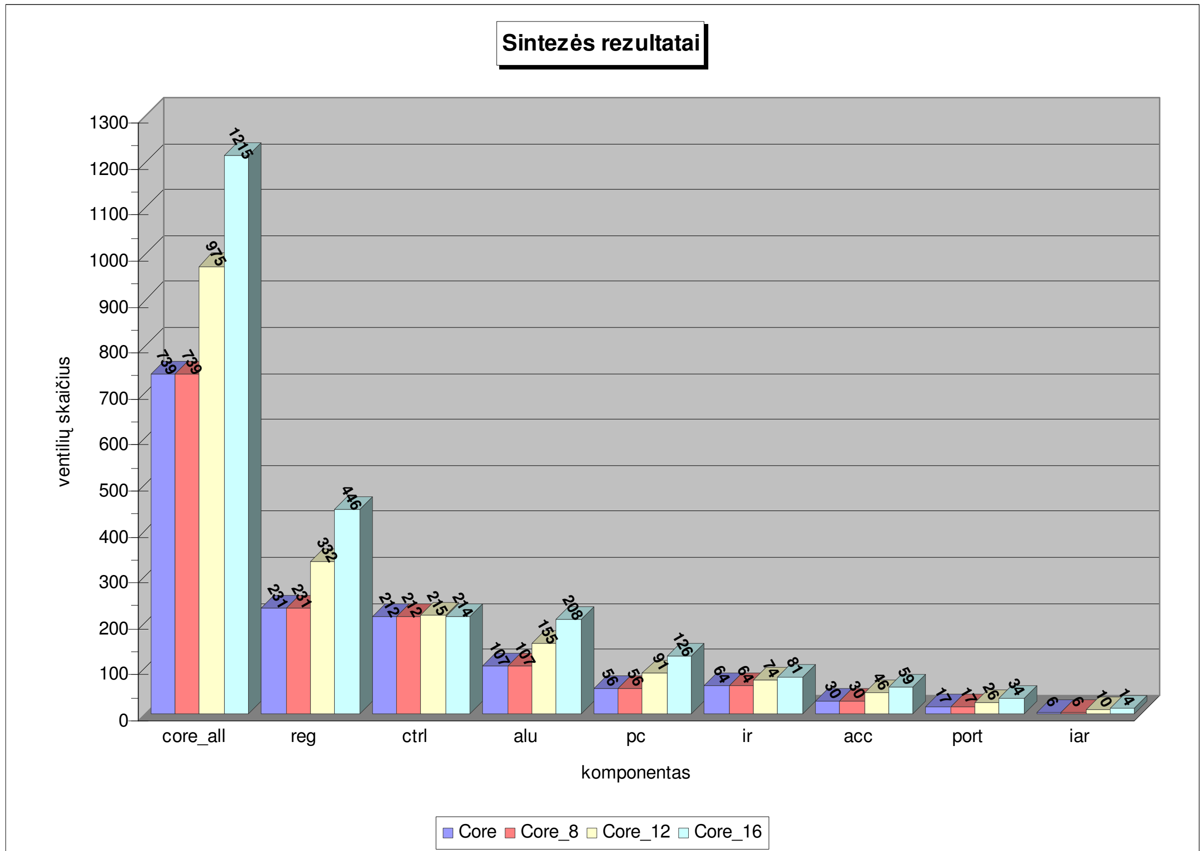
7 lentelė. Skirtingų parametrų bendrinto komponento sintezės rezultatai (core_12)

komp.	Core_12			
	ventilių skaičius, vnt.	**ventilių plotas**	**bendras plotas**	**vėlinimas, ns**
core	34	2184,169922	11574,633789	51,80
acc	46	137,350006	539,850037	4,60
alu	155	364,750000	1859,750244	10,78
ctrl	215	327,200012	2375,637451	6,79
iar	10	16,520000	124,332504	0,39
ir	74	101,770004	741,457520	4,01
pc	91	257,959991	1106,084961	1,58
port	26	102,160004	371,691254	1,63
reg	332	784,619995	4302,901367	1,95
core_all	975	2184,169922	11574,631836	51,80

8 lentelė. Skirtingų parametrų bendrinto komponento sintezės rezultatai (core_16)

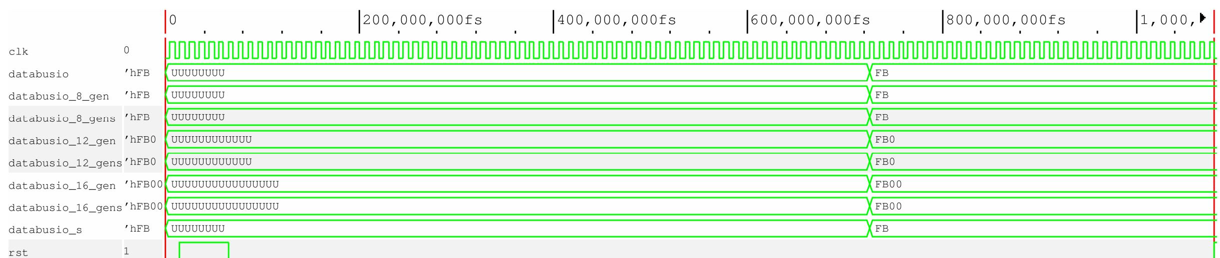
komp.	Core_16			
	ventilių skaičius, vnt.	**ventilių plotas**	**bendras plotas**	**vėlinimas, ns**
core	41	2801,149902	14538,327148	65,69
acc	59	181,610001	702,703796	3,10
alu	208	487,880005	2475,223877	13,23
ctrl	214	328,200012	2373,043457	6,78
iar	14	22,120001	173,057495	0,37
ir	81	120,870003	828,838806	3,81
pc	126	361,309998	1536,466309	1,46
port	34	135,639999	491,421265	1,68
reg	446	1043,939941	5708,625977	1,95
core_all	1215	2801,149902	14538,326172	65,69

Kaip matyti, bendrinis komponentas sintezuojamas, iš to seka išvada, kad esamą procesorinį komponentą bendrinti galime naudojant ir paprastas VHDL kalbos bendrinimo savybes (29 paveikslas).



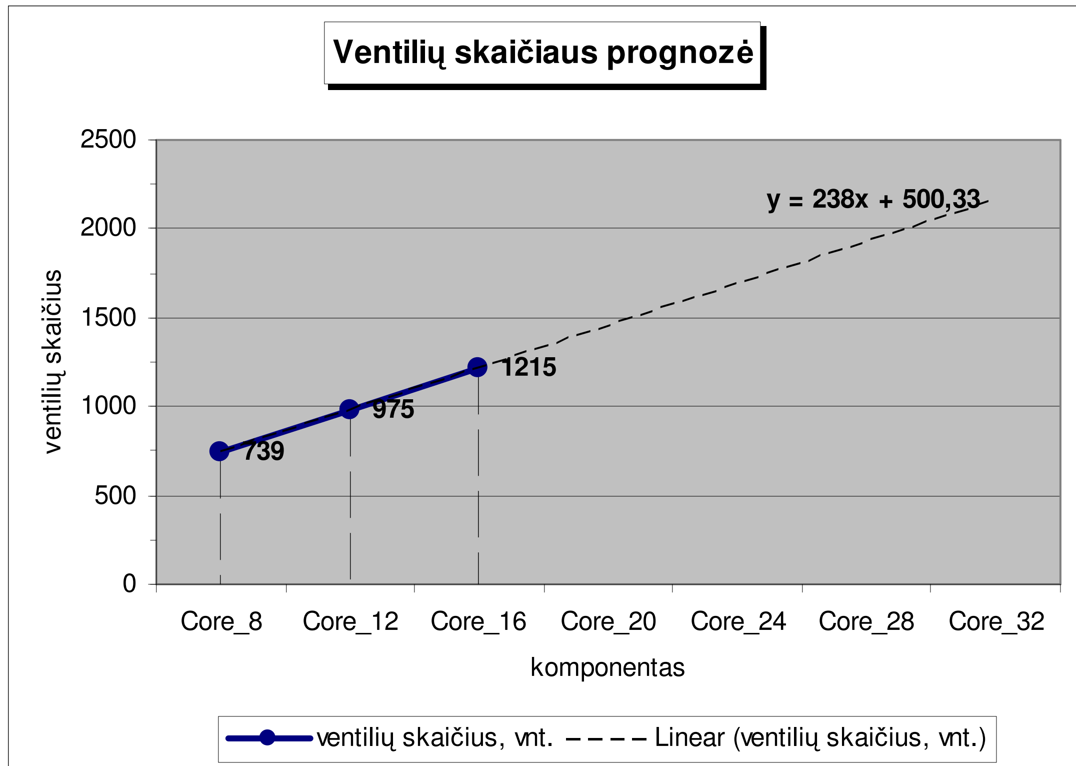
29 pav. Sintezės rezultatai

Simuliacijos rezultatai pateikiami 30 paveiksle.

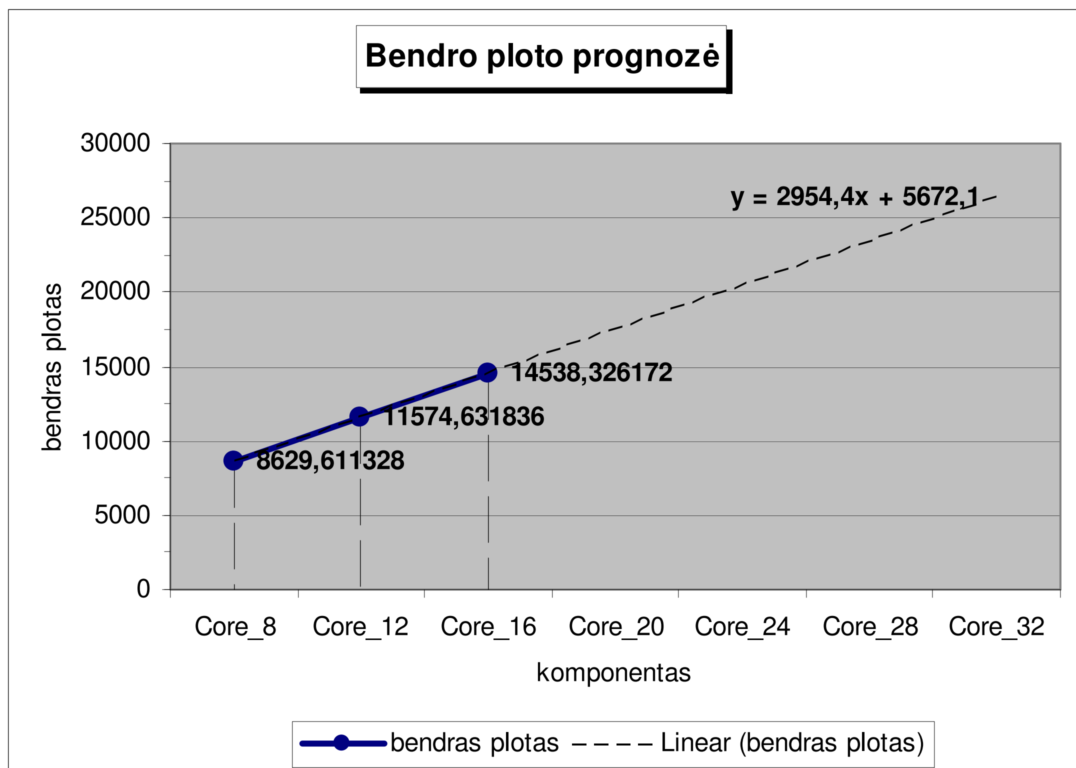


30 pav. Simuliacijos rezultatai

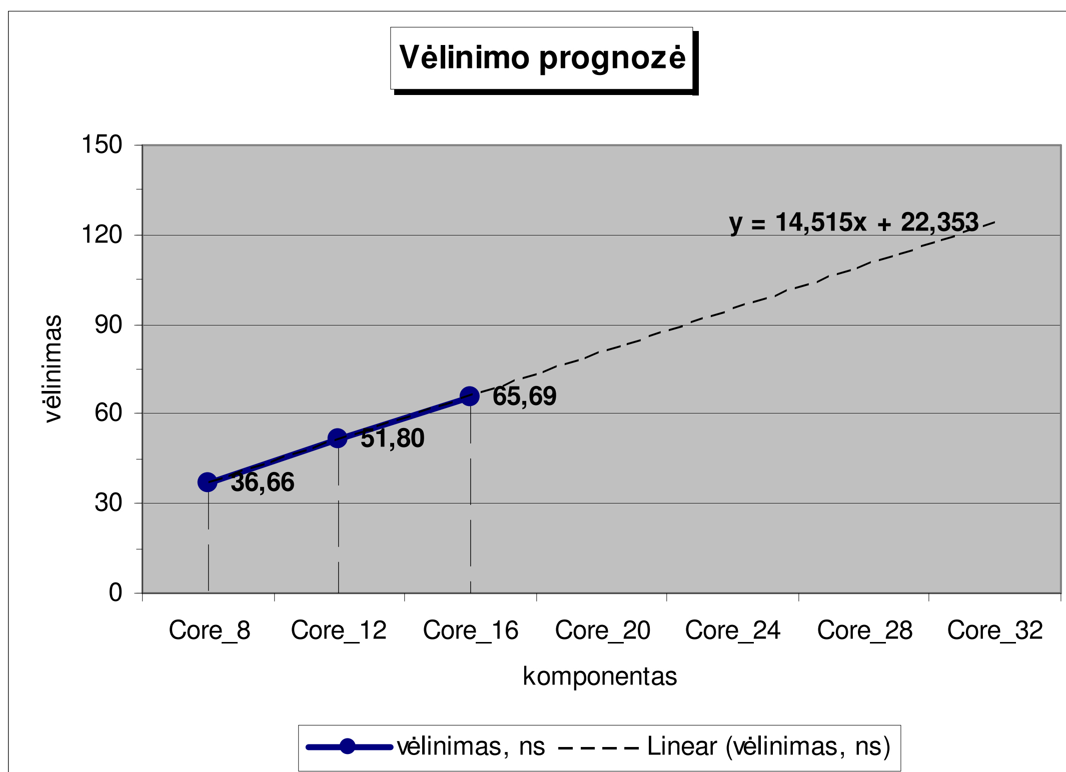
Bendrinio komponento parametrizavimo be atminties prognozė, kai išlaikoma sąlyga, kad n – duomenų magistralė plotis bitais, n – 4 adresų magistralė plotis bitais, registrų adresų magistralės plotis 3 bitai. Ventilių skaičiaus prognozė pateikiama 31 paveiksle. Bendro ploto prognozė 32 paveiksle. Vėlinimo prognozė 33 paveiksle.



31 pav. Ventilių skaičiaus prognozė



32 pav. Bendro ploto prognozė



33 pav. Vėlinimo prognozė

3.3. Instrukcijų registro ir valdymo įrenginio bendrinimo krypties rezultatai

Toliau bus nagrinėjami CTRL ir IR komponentai. Bus siekiam jų bendrinimo. Kadangi šie du komponentai ir apsprendžia sistemos instrukcijų rinkinį, konkrečiai IR komponentas ir jų vykdymo seką – CTRL komponentas. Dėl šių elementų sudėtingumo, vien tik VHDL kalbos esamų metaprogramavimo galimybių nepakanka, todėl šiuos elementus realizuojame ir SystemC kalboje.

3.3.1. IR ir CTRL komponentų bendrinimo rezultatai (VHDL)

Kadangi VHDL kalbos priemonių nepakanka komponentui bendrinti, pateikiami sintezės rezultatai, kurie gaunami pašalinus atitinkamus kodo blokus (9, 10 lentelės). Tai bus atraminiai rezultatai naudojami palyginimui su SystemC kalba realizuotų komponentų bendrinimo rezultatais.

9 lentelė. CTRL ir IR komponentų sintezės rezultatai (VHDL)

komp.	VHDL			
	ventilių skaičius, vnt.	**ventilių plotas**	**bendras plotas**	**vėlinimas, ns**
ctrl	204	313,000000	2321,906250	6,19
ir	64	83,669998	644,295044	3,65

10 lentelė. CTRL ir IR komponentų sintezės rezultatai, be reliatyvių peršokimų (VHDL)

komp.	VHDL			
	ventilių skaičius, vnt.	**ventilių plotas**	**bendras plotas**	**vėlinimas, ns**
ctrl	190	312,739990	2307,271240	6,24
ir	56	79,950005	583,075012	3,67

3.3.2. IR ir CTRL komponentų bendrinimo rezultatai (SystemC)

Kadangi esami komponentai yra realizuoti VHDL kalba, funkciniu aprašu, stengiantis išlaikyti proporcijas, SystemC kalboje taip pat kiek įmanoma išlaikoma ta pati programavimo struktūra. Komponentų realizuotų SystemC kalboje sintezės rezultatai pateikiami 11 lentelėje. Kad būtų galima atlikti tikslesnę analizę, komponentai taip pat sintezuojami ir pašalinus atitinkamas kodo vietas ne parametrizuojant (12 lentelė).

11 lentelė. CTRL ir IR komponentų sintezės rezultatai (SystemC)

komp.	SystemC			
	ventilių skaičius, vnt.	**ventilių plotas**	**bendras plotas**	**vėlinimas, ns**
ctrl	218	339,799988	2553,549316	3,14
ir	65	86,120003	661,120056	4,14

12 lentelė. CTRL ir IR komponentų sintezės rezultatai, be reliatyvių peršokimų (SystemC)

komp.	SystemC			
	ventilių skaičius, vnt.	**ventilių plotas**	**bendras plotas**	**vėlinimas, ns**
ctrl	212	322,529999	2457,217041	3,41
ir	58	81,250000	595,156250	3,64

Komponentų parametrizavimui SystemC kalboje naudojamas C kalbos preprocesorius. Naudojama `#define`, kur nurodoma ar naudojami reliatyvūs peršokimai, ar

ne. Bei `#ifdef … #endif` preprocesoriaus komandos, atitinkamam kodo blokui įtraukti į kompiliavimo, sintezavimo procesą ar ne.

```
#define nrj

#ifdef nrj
…
//Kodo blokas
…
#endif
```

Bendrinių komponentų IR ir CTRL sintezės rezultatai su reliatyviais peršokimais pateikti 13 lentelėje. Bendrinių komponentų IR ir CTRL sintezės rezultatai, parametrizavus komponentus, kad nebūtų naudojami reliatyvūs peršokimai pateikiami 14 lentelėje.

13 lentelė. CTRL ir IR bendrinių komponentų sintezės rezultatai (SystemC)

komp.	**SystemC**			
	ventilių skaičius, vnt.	**ventilių plotas**	**bendras plotas**	**vėlinimas, ns**
ctrl	218	335,769989	2535,144287	3,09
ir	65	86,120003	661,120056	4,14

14 lentelė. CTRL ir IR bendrinių komponentų sintezės rezultatai, be reliatyvių peršokimų (SystemC)

komp.	**SystemC**			
	ventilių skaičius, vnt.	**ventilių plotas**	**bendras plotas**	**vėlinimas, ns**
ctrl	212	322.529999	2457,217041	3,41
ir	58	81,250000	595,156250	3,64

Sintezės rezultatai rodo, kad ženklaus ventilių skaičiaus ar ploto sumažėjimo nėra, bet galime daryti prielaidą, kad instrukcijų rinkinio parametrizavimas, taip pat įtakoja ir kitus sistemos elementus. Šios architektūros atveju, supaprastėja programinis skaitiklis PC. Jame nebelieka sumavimo funkcijos, kuri būtent buvo skirta atliekant reliatyvius peršokimus.

4. IŠVADOS

1. Išnagrinėjus paplitusias procesorių architektūras, pasirinktos tokios bendrinimo kryptys:
    - atskirų procesoriaus komponentų bendrinimas, siekiant ištirti bendrinimo įtaką galutinės sistemos atžvilgiu, nekeičiant procesoriaus pradinės architektūros ir instrukcijų rinkinio;
    - ištirti instrukcijų registro ir valdymo įrenginių bendrinimo galimybes, parametrizuojant instrukcijų rinkinį.
2. Atlikus procesoriaus komponentų bendrinimo tyrimą, galima teigti:
    - Kiekvienas komponentas parametrizuojamas individualiai, o sistemos (sudarytos iš bendrinių komponentų) parametrizavimas susiveda į duomenų, adresų ir registrų adresų magistralių pločio parametrų keitimą.
    - Procesoriau komponentų bendrinimas, parametrizuojant duomenų magistralės, adresų magistralės ir registrų adresų magistralės plotį, galimas naudojant VHDL kalbos metaprogramavimo priemones. Gaunami sintezuojami komponentai, kurių parametrų kitimą, galutinėje sistemoje, galime prognozuoti:
        - skirtumo tarp ventilių skaičiaus, ploto ir vėlinimo nėra, kai bendrinta sistema parametrizuojama taip, kad atitiktų pradinę;
        - padidinus duomenų magistralės plotį 4 bitais, adresų magistralės plotį 4 bitais, kai registrų adresų magistralės plotis nekeičiamas, sistemos ventilių skaičiius, plotas ir vėlinimas atitinkamai padidėja ~32%, ~34%, ~41%;
        - padidinus duomenų magistralės plotį 8 bitais, adresų magistralės plotį 8 bitais, kai registrų adresų magistralės plotis nekeičiamas, sistemos ventilių skaičius, plotas ir vėlinimas atitinkamai padidėja: ~64%, ~68% ir ~79%.
3. Išanalizavus pasirinkto mikroprocesoriaus instrukcijų registrą ir valdymo įrenginį, galima teigti:
    - komponentų parametrizavimui VHDL metaprogramavimo priemonių nepakanka;
    - pašalinant kodo blokus, ventilių skaičius ir bendrasis plotas nežymiai mažėja, vėlinimas atvirkščiai – nežymiai išauga.
4. Komponentų perrašymas į SystemC kalbą, leido juos parametrizuoti instrukcijų rinkinio atžvilgiu. SystemC kalboje komponentus parametrizuojame naudojant C kalbos preprocesoriaus komandas.

- Parametrizuojant, palyginimui pašalinant kodo blokus, taip pat lyginant su VHDL realizacija, gaunami labai panašūs sintezės rezultatai tiek ventilių skaičiaus, tiek bendro ploto ir vėlinimo kitime. Ventilių skaičius, bendras plotas mažėja, o vėlinimas didėja.

5. Norint pasiekti viso procesoriaus detalaus parametrizavimo, ne tik magistralių pločių parametrizavimo, bet ir komandų rinkinio ar net architektūros, VHDL ar SystemC priemonių nepakanka. Reikia kitų priemonių, tokių kaip generatoriai. Kokios kalbos kodą generatorius turėtų generuoti vienareikšmiškai pasakyti negalima. Komponentai parašyti VHDL kalba labiau prognozuojami, bet SystemC kalba yra naujesnė ir turi daugiau galimybių, be to yra nuolat tobulinama.

LITERATŪRA

[1] Bailey, B. Dealing with SoC hardware/software design complexity with scalable verification methods. Embedded [interaktyvus]. 2007, balandis [žiūrėta 2007-04-17]. Prieiga per internetą: <http://www.embedded.com>

[2] Ziberkas, G. Komponentinio taikomosios srities programų generavimo metodų tyrimas: daktaro disertacijos darbas. KTU, Informatikos fakultetas. [Kaunas], 2001. 122 p.

[3] Lodi, Andrea, at el.. A VLIW Processor With Reconfigurable Instruction Set for Embedded Applications. Ieeexplore [interaktyvus]. 2003, lapkritis [žiūrėta 2006-05-08]. Prieiga per internetą <http://ieeexplore.ieee.org>

[4] Atmel Corporation. ATtiny25/45/85: 8-bit AVR Microcontroller with 2/4/8KBytes In-System Programmable Flash. [žiūrėta 2005-10-18]. Prieiga per internetą <http://www.atmel.com>

[5] Microchip Technology Inc. PIC10F200/202/204/206: 6-Pin, 8-bit Flash Microcontrollers. [žiūrėta 2005-10-18]. Prieiga per internetą <http://www.microchip.com>

[6] Stallings, W. Reduced instruction set computer architecture. Ieeexplore [interaktyvus]. 1988, sausis [žiūrėta 2006-05-10]. Prieiga per internetą <http://ieeexplore.ieee.org>

[7] Smotherman, M. Understanding EPIC Architectures and Implementations. Department of Computer Science, Clemson University [interaktyvus]. 2005, rugsėjis [žiūrėta 2006-05-15]. Prieiga per internetą <http://www.cs.clemson.edu>

[8] Joseph, A., Fisher. Very Long Instruction Word Architectures and the ELI- 512. Hewlett-Packard Laboratories, Cambridge, Massachusetts [interaktyvus]. 2005, liepa. [žiūrėta 2006-05-08]. Prieiga per internetą <http://www.hpl.hp.com>

[9] Philips Semiconductors. Very-Long Instruction Word (VLIW) Computer Architecture. [žiūrėta 2006-09-12]. Prieiga per internetą <http://www.nxp.com>

[10] Ziberkas, G. The Development of Generic Components in VHDL. ITC [interaktyvus]. 2000, vasaris [žiūrėta 2005-10-12]. Prieiga per internetą <http://itc.ktu.lt/>

[11] Leupers, R. at el. A Design Flow for Configurable Embedded Processors based on Optimized Instruction Set Extension Synthesis. Inst. for Integrated Signal Process. Syst., KWTH Aachen Univ., Germany. Ieeexplore [interaktyvus]. 2006, kovas [žiūrėta 2007-04-12]. Prieiga per internetą <http://ieeexplore.ieee.org>

[12] Charest, L.; Aboulhamid, E.M. A VHDL/SystemC Comparison in Handling Design Reuse, Université de Montréal. 2002. [žiūrėta 2006-10-12]. Prieiga per internetą <www.iro.umontreal.ca>

[13] Charest , L.; Marquet, P. Comparisons of Different Approaches of Realizing IP Block Configuration in SystemC. Universite des Sciences et Technologies de Lille, France. Ieeexplore [interaktyvus]. 2005, birželis [žiūrėta 2006-02-02]. Prieiga per internetą <http://ieeexplore.ieee.org>

[14] Bailey, B. Address verification issues with scalable methods. Mentor Graphics Corp., Design Verification and Test Division, Chief Technologist. 2007, balandis [žiūrėta 2007-05-03]. Prieiga per internetą <http://www.eetasia.com/>

[15] Advanced Micro Devices, Inc. AMD Eighth-Generation Processor Architecture. 2001, spalis [žiūrėta 2006-10-22]. Prieiga per internetą <http://www.amd.com>

[16] Duncan R. A Survey of Parallel Computer Architectures. IEEE Computer. Ieeexplore [interaktyvus]. 1990, vasaris [žiūrėta 2007-01-05]. Prieiga per internetą <http://ieeexplore.ieee.org>

[17] Barna, C.; Rosenstiel, W. Object-Oriented Reuse Methodology for VHDL. Ieeexplore [interaktyvus]. 1999, kovas [žiūrėta 2007-04-28]. Prieiga per internetą <http://ieeexplore.ieee.org>

[18] Rincon, F. at el. Model Reuse through Hardware Design Patterns. Ieeexplore [interaktyvus]. 2005, kovas [žiūrėta 2007-01-09]. Prieiga per internetą <http://ieeexplore.ieee.org>

[19] Štuikys, V.; Damaševičius, R. Soft IP Customisation Model Based on Metaprogramming Techniques. INFORMATICA, 2004 Vol. 15, No. 1, 1–16

[20] Štuikys, V.; Ziberkas, G. Domain specific reuse with VHDL. Kaunas University of Technology. 1999. [žiūrėta 2007-01-20] Prieiga per internetą <http://soften.ktu.lt/~stuik/knyga/>

[21] Jacobson I.; Griss, M.; Jansson, P. Software Reuse Architecture Process and Organization for Business success. ACM Press Addison-Wesley 1997.

[22] Lim W. C. Managing Software Reuse. Prentice Hall PTR 1998

[23] Sametinger J. Software Engineering with Reusable Components. Springer-Verlag 1997.

[24] Tracz, W. Developing Reusable Java Components, Symposium on Software Reusability, ACM 1997 100-103 p.

[25] Jones, G. Methodology/environment Support for Reusability. In: „Software Reuse: Emerging Technology“. IEEE Computer Society, 1990 p.190-193 p.

PRIEDAI

1. Priedas. Bendrinių komponentų pavyzdinis VHDL kodas

acc_gen.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity acc_gen is
generic(width : natural);
port   (
       data     : inout std_logic_vector(width-1 downto 0);
       rden     : in    std_logic;
       wren     : in    std_logic;
       rst      : in    std_logic;
       halfd    : in    std_logic;
       pos, zer : out   std_logic
       );
end acc_gen;

architecture behv of acc_gen is

signal accreg : std_logic_vector(width-1 downto 0);

begin
   process (rden, wren, rst)
   begin
      if (rst = '1')
         then accreg <= (others => '0'); data <= (others => 'Z');
         elsif (wren = '1' and rden = '0')
            then
               if (halfd = '1')
                  then
                     accreg <= "0000" & data(width-5 downto 0);
                     data <= (others => 'Z');
                  else accreg <= data; data <= (others => 'Z');
               end if;
            elsif (wren = '0' and rden = '1')
               then data <= accreg;
               else data <= (others => 'Z');
      end if;
   end process;

pos <= '1'   when accreg(width-1) = '0' else '0';
zer <= '1'   when accreg = 0 else '0';

end behv;
```

alu_gen.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;

entity alu_gen is
generic(width : natural);
port   (
       data : inout std_logic_vector(width-1 downto 0);
       alux : in    std_logic;
       code : in    std_logic_vector(3 downto 0);
       en   : in    std_logic;
       rst  : in    std_logic
       );
end alu_gen;

architecture behv of alu_gen is

signal tmp : std_logic_vector(width-1 downto 0);

begin
   process (en, alux, rst)
```

alu_gen.vhd

```
   begin
      if (rst = '1')
         then tmp <= (others => '0'); data <= (others => 'Z');
         elsif (alux = '1')
            then tmp <= data; data <= (others => 'Z');
            elsif (en = '1')
               then
                  case code is
                     when "0000" => data <= data and tmp;
                     when "0001" => data <= data or tmp;
                     when "0010" => data <= data + tmp;
                     when "0011" => data <= data - tmp;
                     when "0100" => data <= not data;
                     when "0101" => data <= data + 1;
                     when "0110" => data <= data - 1;
                     when "0111" => data <= data(width-2 downto 0) & "0";
                     when "1000" => data <= "0" & data(width-1 downto 1);
                     when others => data <= (others => 'Z');
                  end case;
               else data <= (others => 'Z');
      end if;
   end process;
end behv;
```

ctrl_gen.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity ctrl_gen is
port(
    clk, rst                    : in  std_logic;
    accrd, accwr                : out std_logic;
    halfd                       : out std_logic;
    aluen                       : out std_logic;
    alux                        : out std_logic;
    alucode                     : out std_logic_vector(3 downto 0);
    iaddr                       : out std_logic;
    irld                        : out std_logic;
    lda, sta, ldm, stm, ldi,
    jmpr, jmp, jzr, jz, jpr, jp,
    iand, ior, add, sub, inot,
    inc, dec, shfl, shfr, ina,
    outa, halt, nop             : in  std_logic;
   pcinc, pcld, pcsum, pcenaddr : out std_logic;
   ramrd, ramwr                 : out std_logic;
   regrd, regwr                 : out std_logic;
   portrd, portwr               : out std_logic
   );
end ctrl_gen;

architecture behv of ctrl_gen is

type busenos is (reset, bhalt, bnop,
                fetch, ftch0, ftch1, ftch2,
                lda0, lda1, lda2, lda3,
                sta0, sta1, sta2, sta3,
                ldm0, ldm1, ldm2, ldm3, ldm4, ldm5, ldm6,
                stm0, stm1, stm2, stm3, stm4, stm5, stm6,
                ldi0, ldi1,
                jumpr,
                jump0, jump1, jump2, jump3,
                arit0, arit1, arit2, arit3, arit4, arit5, arit6, arit7,
                log0, log1, log2, log3, log4,
                ina0, ina1, ina2,
                outa0, outa1, outa2);
signal busena : busenos;

begin
   process (rst, clk)
   begin
      if (rst = '1')
         then busena <= reset;
         elsif (clk = '1' and clk'event)
            then
               case busena is
```

ctrl_gen.vhd

```
                    when reset => busena <= fetch;
                    when fetch => busena <= ftch0;
                    when ftch0 => busena <= ftch1;
                    when ftch1 => busena <= ftch2;
                    when ftch2 =>
                       if    lda = '1'  then busena <= lda0;
                       elsif sta = '1'  then busena <= sta0;
                       elsif ldm = '1'  then busena <= ldm0;
                       elsif stm = '1'  then busena <= stm0;
                       elsif ldi = '1'  then busena <= ldi0;
                       elsif jmpr = '1' then busena <= jumpr;
                       elsif jmp = '1'  then busena <= jump0;
                       elsif jzr = '1'  then busena <= jumpr;
                       elsif jz = '1'   then busena <= jump0;
                       elsif jpr = '1'  then busena <= jumpr;
                       elsif jp = '1'   then busena <= jump0;
                       elsif iand = '1' then busena <= arit0;
                       elsif ior = '1'  then busena <= arit0;
                       elsif add = '1'  then busena <= arit0;
                       elsif sub = '1'  then busena <= arit0;
                       elsif inot = '1' then busena <= log0;
                       elsif inc = '1'  then busena <= log0;
                       elsif dec = '1'  then busena <= log0;
                       elsif shfl = '1' then busena <= log0;
                       elsif shfr = '1' then busena <= log0;
                       elsif ina = '1'  then busena <= ina0;
                       elsif outa = '1' then busena <= outa0;
                       elsif halt = '1' then busena <= bhalt;
                       elsif nop = '1'  then busena <= bnop;
                       end if;
                    when lda0 => busena <= lda1;
                    when lda1 => busena <= lda2;
                    when lda2 => busena <= lda3;
                    when lda3 => busena <= fetch;

                    when sta0 => busena <= sta1;
                    when sta1 => busena <= sta2;
                    when sta2 => busena <= sta3;
                    when sta3 => busena <= fetch;

                    when ldm0 => busena <= ldm1;
                    when ldm1 => busena <= ldm2;
                    when ldm2 => busena <= ldm3;
                    when ldm3 => busena <= ldm4;
                    when ldm4 => busena <= ldm5;
                    when ldm5 => busena <= ldm6;
                    when ldm6 => busena <= fetch;

                    when stm0 => busena <= stm1;
                    when stm1 => busena <= stm2;
                    when stm2 => busena <= stm3;
                    when stm3 => busena <= stm4;
                    when stm4 => busena <= stm5;
                    when stm5 => busena <= stm6;
                    when stm6 => busena <= fetch;

                    when ldi0 => busena <= ldi1;
                    when ldi1 => busena <= fetch;

                    when jumpr => busena <= fetch;

                    when jump0 => busena <= jump1;
                    when jump1 => busena <= jump2;
                    when jump2 => busena <= jump3;
                    when jump3 => busena <= fetch;

                    when arit0 => busena <= arit1;
                    when arit1 => busena <= arit2;
                    when arit2 => busena <= arit3;
                    when arit3 => busena <= arit4;
                    when arit4 => busena <= arit5;
                    when arit5 => busena <= arit6;
                    when arit6 => busena <= arit7;
                    when arit7 => busena <= fetch;

                    when log0 => busena <= log1;
                    when log1 => busena <= log2;
                    when log2 => busena <= log3;
```

ctrl_gen.vhd

```
                  when log3 => busena <= log4;
                  when log4 => busena <= fetch;

                  when ina0 => busena <= ina1;
                  when ina1 => busena <= ina2;
                  when ina2 => busena <= ftch0;

                  when outa0 => busena <= outa1;
                  when outa1 => busena <= outa2;
                  when outa2 => busena <= ftch0;

                  when bhalt => busena <= bhalt;

                  when bnop => busena <= fetch;

                  when others => busena <= bhalt;
               end case;
            else null;
      end if;
   end process;

accrd <= '1'      when busena = sta1 or
                       busena = sta2 or
                       busena = stm4 or
                       busena = stm5 or
                       busena = arit4 or
                       busena = log1 or
                       busena = outa0 or
                       busena = outa1
                       else '0';

accwr <= '1'      when busena = lda2 or
                       busena = ldm5 or
                       busena = ldi0 or
                       busena = arit6 or
                       busena = log3 or
                       busena = ina1
                       else '0';

aluen <= '1'      when busena = arit5 or
                       busena = arit6 or
                       busena = log2 or
                       busena = log3
                       else '0';

alux <= '1'       when busena = arit3
                       else '0';

iaddr <= '1'      when busena = ldm3 or
                       busena = ldm4 or
                       busena = ldm5 or
                       busena = stm3 or
                       busena = stm4 or
                       busena = stm5 or
                       busena = lda0 or
                       busena = lda1 or
                       busena = lda2 or
                       busena = sta0 or
                       busena = sta1 or
                       busena = sta2 or
                       busena = arit1 or
                       busena = arit2 or
                       busena = arit3
                       else '0';

irld <= '1'       when busena = ftch2
                       else '0';

pcinc <= '1'      when busena = lda3 or
                       busena = sta3 or
                       busena = ldm0 or
                       busena = ldm6 or
                       busena = stm0 or
                       busena = stm6 or
                       busena = ldi1 or
                       busena = jump0 or
                       busena = arit7 or
                       busena = log4 or
```

ctrl_gen.vhd

```
                        busena = ina2 or
                        busena = outa2 or
                        busena = bnop
                        else '0';

pcld <= '1'       when busena = jump3
                        else '0';

pcsum <= '1'      when busena = jumpr
                        else '0';

pcenaddr <= '1'   when busena = ftch0 or
                        busena = ftch1 or
                        busena = ftch2 or
                        busena = ldm1 or
                        busena = ldm2 or
                        busena = stm1 or
                        busena = stm2 or
                        busena = ldi0 or
                        busena = jump1 or
                        busena = jump2 or
                        busena = jump3
                        else '0';

ramrd <= '1'      when busena = ftch1 or
                        busena = ftch2 or
                        busena = ldm2 or
                        busena = ldm4 or
                        busena = ldm5 or
                        busena = stm2 or
                        busena = ldi0 or
                        busena = jumpr or
                        busena = jump2 or
                        busena = jump3 or
                        busena = arit0 or
                        busena = log0
                        else '0';

ramwr <= '1'      when busena = stm5
                        else '0';

regrd <= '1'      when busena = lda1 or
                        busena = lda2 or
                        busena = arit2 or
                        busena = arit3
                        else '0';

regwr <= '1'      when busena = sta2
                        else '0';

portrd <= '1'     when busena = ina0 or
                        busena = ina1
                        else '0';

portwr <= '1'     when busena = outa1
                        else '0';

alucode <= "0000" when iand = '1' else
           "0001" when ior = '1' else
           "0010" when add = '1' else
           "0011" when sub = '1' else
           "0100" when inot = '1' else
           "0101" when inc = '1' else
           "0110" when dec = '1' else
           "0111" when shfl = '1' else
           "1000" when shfr = '1' else
           "ZZZZ";

halfd <= '1'      when ldi = '1' else '0';

end behv;
```

iar_gen.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
```

iar_gen.vhd

```
entity iar_gen is
generic(width : natural);
port   (
       data  : in  std_logic_vector(width-1 downto 0);
       addr  : out std_logic_vector(width-1 downto 0);
       iaddr : in  std_logic
   );
end iar_gen;

architecture behv of iar_gen is
begin
   process(iaddr)
   begin
   case iaddr is
      when '1' => addr <= data;
      when '0' => addr <= (others => 'Z');
      when others => addr <= (others => 'Z');
   end case;
   end process;
end behv;
```

ir_gen.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity ir_gen is
generic(width : natural);
port   (
       data            : in  std_logic_vector(width-1 downto 0);
       ld              : in  std_logic;
       rst             : in  std_logic;
       pos, zer        : in  std_logic;
       lda, sta, ldm,
       stm, ldi, jmpr,
       jmp, jzr, jz,
       jpr, jp, iand,
       ior, add, sub,
       inot, inc, dec,
       shfl, shfr, ina,
       outa, halt, nop : out std_logic
       );
end ir_gen;

architecture behv of ir_gen is

signal irdata : std_logic_vector(width-1 downto 0);

begin
   process (ld, rst)
   begin
      if (rst = '1')
         then irdata <= (others => '0');
         elsif (ld = '1')
            then irdata <= data;
            else null;
      end if;
   end process;

lda <= '1'  when irdata(width-1 downto width-4) = "0001" else '0';
sta <= '1'  when irdata(width-1 downto width-4) = "0010" else '0';
ldm <= '1'  when irdata(width-1 downto width-4) = "0011" else '0';
stm <= '1'  when irdata(width-1 downto width-4) = "0100" else '0';
ldi <= '1'  when irdata(width-1 downto width-4) = "0101" else '0';
jmpr <= '1' when ((irdata(width-1 downto width-4) = "0110") and (not (irdata(width-5 downto 0)
= "0000"))) else '0';
jmp <= '1'  when ((irdata(width-1 downto width-4) = "0110") and (irdata(width-5 downto 0) =
0)) else '0';
jzr <= '1' when ((irdata(width-1 downto width-4) = "0111") and (not (irdata(width-5 downto 0)
= 0)) and (zer = '1')) else '0';
jz <= '1'   when ((irdata(width-1 downto width-4) = "0111") and (irdata(width-5 downto 0) = 0)
and (zer = '1')) else '0';
jpr <= '1' when ((irdata(width-1 downto width-4) = "1000") and (not (irdata(width-5 downto 0)
= 0)) and (pos = '1')) else '0';
jp <= '1'   when ((irdata(width-1 downto width-4) = "1000") and (irdata(width-5 downto 0) = 0)
```

ir_gen.vhd

```
and (pos = '1')) else '0';
iand <= '1' when irdata(width-1 downto width-4) = "1001" else '0';
ior <= '1'  when irdata(width-1 downto width-4) = "1010" else '0';
add <= '1'  when irdata(width-1 downto width-4) = "1011" else '0';
sub <= '1'  when irdata(width-1 downto width-4) = "1100" else '0';
inot <= '1' when ((irdata(width-1 downto width-4) = "1101") and (irdata(width-6 downto width-
8) = "000")) else '0';
inc <= '1'  when ((irdata(width-1 downto width-4) = "1101") and (irdata(width-6 downto width-
8) = "001")) else '0';
dec <= '1'  when ((irdata(width-1 downto width-4) = "1101") and (irdata(width-6 downto width-
8) = "010")) else '0';
shfl <= '1' when ((irdata(width-1 downto width-4) = "1101") and (irdata(width-6 downto width-
8) = "011")) else '0';
shfr <= '1' when ((irdata(width-1 downto width-4) = "1101") and (irdata(width-6 downto width-
8) = "100")) else '0';
ina <= '1'  when ((irdata(width-1 downto width-4) = "1110") and (irdata(width-8) = '0')) else
'0';
outa <= '1' when ((irdata(width-1 downto width-4) = "1110") and (irdata(width-8) = '1')) else
'0';
halt <= '1' when irdata(width-1 downto width-4) = "1111" else '0';
nop <= '1'  when irdata = 0 else '0';

end behv;
```

pc_gen.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity pc_gen is
generic(d_width : natural; a_width : natural);
port   (
       data   : in  std_logic_vector(a_width-1 downto 0);
       addr   : out std_logic_vector(a_width-1 downto 0);
       clk    : in  std_logic;
       rst    : in  std_logic;
       inc    : in  std_logic;
       ld     : in  std_logic;
       sum    : in  std_logic;
       enaddr : in  std_logic
       );
end pc_gen;

architecture behv of pc_gen is

signal temp_addr : std_logic_vector(a_width-1 downto 0);

begin
   process (clk, rst)
   begin
      if (rst = '1')
         then temp_addr <= (others => '0'); addr <= (others => 'Z');
         elsif (clk = '1' and clk'event)
            then
               if (inc = '1' and ld = '0' and sum = '0')
                  then temp_addr <= temp_addr + 1; addr <= (others => 'Z');
                  elsif (inc = '0' and ld = '1' and sum = '0')
                     then temp_addr <= data; addr <= (others => 'Z');
                     elsif (inc = '0' and ld = '0' and sum = '1')
                        then temp_addr <= temp_addr + data(d_width - 5 downto 0); addr <=
(others => 'Z');
                        elsif (enaddr = '1')
                           then addr <= temp_addr;
                           else addr <= (others => 'Z');
               end if;
      end if;
   end process;
end behv;
```

port_gen.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
use ieee.std_logic_unsigned.all;
```

port_gen.vhd

```
entity portio_gen is
generic(width : natural);
port   (
       dataio : inout std_logic_vector(width-1 downto 0);
       data   : inout std_logic_vector(width-1 downto 0);
       portrd : in    std_logic;
       portwr : in    std_logic
       );
end portio_gen;

architecture behv of portio_gen is
begin
   process (portrd, portwr)
   begin
      if (portrd = '1')
         then data <= dataio;
         elsif (portwr = '1')
            then dataio <= data; data <= (others => 'Z');
            else data <= (others => 'Z');
      end if;
   end process;
end behv;
```

ram_gen.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity ram_gen is
generic(d_width : natural; a_width : natural);
port   (
       data : inout std_logic_vector(d_width-1 downto 0);
       addr : in    std_logic_vector(a_width-1 downto 0);
       rden : in    std_logic;
       wren : in    std_logic;
       rst  : in    std_logic
       );
end ram_gen;

architecture behv of ram_gen is

subtype byte is std_logic_vector(d_width-1 downto 0);
type rammodul is array(2**a_width-1 downto 0) of byte;

signal rambyte : rammodul;

begin
   process (rden, wren, rst)
   begin
      if (rst = '1')
         then
            rambyte <= (others => (others => '0'));
--            rambyte(0) <= "0011000000000000"; -- LDM A <-
--            rambyte(1) <= "0000000000001000"; --          M[001000]
--            rambyte(2) <= "0010000000000000"; -- STA R[000] <- A
--            rambyte(3) <= "0011000000000000"; -- LDM A <-
--            rambyte(4) <= "0000000000001001"; --          M[001001]
--            rambyte(5) <= "1100000000000000"; -- A <- A sub R[000]
--            rambyte(6) <= "0110000000000000"; -- JMP
--            rambyte(7) <= "0000000000001010"; --     PC = 001010
--            rambyte(8) <= "0000101000000000"; -- 2560
--            rambyte(9) <= "0000010100000000"; --  1280
--            rambyte(10) <= "0100000000000000"; -- STA
--            rambyte(11) <= "0000000000001110"; --     M[001110] <- A
--            rambyte(12) <= "1110000100000000"; -- OutA PORT <- A
--            rambyte(13) <= "0110000000000010"; -- JMPR PC = PC + 0010
            rambyte(0) <= "001100000000"; -- LDM A <-
            rambyte(1) <= "000000001000"; --          M[001000]
            rambyte(2) <= "001000000000"; -- STA R[000] <- A
            rambyte(3) <= "001100000000"; -- LDM A <-
            rambyte(4) <= "000000001001"; --          M[001001]
            rambyte(5) <= "110000000000"; -- A <- A sub R[000]
            rambyte(6) <= "011000000000"; -- JMP
            rambyte(7) <= "000000001010"; --     PC = 001010
            rambyte(8) <= "000010100000"; -- 2560
```

ram_gen.vhd

```
            rambyte(9) <= "000001010000"; --  1280
            rambyte(10) <= "010000000000"; -- STA
            rambyte(11) <= "000000001110"; --     M[001110] <- A
            rambyte(12) <= "111000010000"; -- OutA PORT <- A
            rambyte(13) <= "011000000010"; -- JMPR PC = PC + 0010
--            rambyte(0) <= "00110000"; -- LDM A <-
--            rambyte(1) <= "00001000"; --          M[001000]
--            rambyte(2) <= "00100000"; -- STA R[000] <- A
--            rambyte(3) <= "00110000"; -- LDM A <-
--            rambyte(4) <= "00001001"; --          M[001001]
--            rambyte(5) <= "11000000"; -- A <- A sub R[000]
--            rambyte(6) <= "01100000"; -- JMP
--            rambyte(7) <= "00001010"; --     PC = 001010
--            rambyte(8) <= "00001010"; -- 10
--            rambyte(9) <= "00000101"; --  5
--            rambyte(10) <= "01000000"; -- STA
--            rambyte(11) <= "00001110"; --     M[001110] <- A
--            rambyte(12) <= "11100001"; -- OutA PORT <- A
--            rambyte(13) <= "01100010"; -- JMPR PC = PC + 0010
            data <= (others => 'Z');
         elsif (wren = '1' and rden = '0')
            then rambyte(conv_integer(addr)) <= data; data <= (others => 'Z');
            elsif (wren = '0' and rden = '1')
               then data <= rambyte(conv_integer(addr));
               else data <= (others => 'Z');
      end if;
   end process;
end behv;
```

reg_gen.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity reg_gen is
generic(d_width : natural; a_width : natural);
port   (
       data : inout std_logic_vector(d_width-1 downto 0);
       addr : in    std_logic_vector(a_width-1 downto 0);
       rden : in    std_logic;
       wren : in    std_logic;
       rst  : in    std_logic
       );
end reg_gen;

architecture behv of reg_gen is

subtype byte is std_logic_vector(d_width-1 downto 0);
type regmodul is array(2**a_width-1 downto 0) of byte;
signal regbyte : regmodul;

begin
   process (rden, wren, rst)
   begin
      if (rst = '1')
         then
            regbyte <= (others => (others => '0'));
         elsif (wren = '1' and rden = '0')
            then regbyte(conv_integer(addr)) <= data; data <= (others => 'Z');
            elsif (wren = '0' and rden = '1')
               then data <= regbyte(conv_integer(addr));
               else data <= (others => 'Z');
      end if;
   end process;
end behv;
```

core_gen.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity core_gen is
generic(d_width : natural; a_width : natural; ra_width : natural);
port   (
```

core_gen.vhd

```
       databus      : inout std_logic_vector(d_width-1 downto 0);
       databusio    : inout std_logic_vector(d_width-1 downto 0);
       addrbus      : out   std_logic_vector(a_width-1 downto 0);
       memrd, memwr : out   std_logic;
       clk, rst     : in    std_logic
       );
end core_gen;

architecture behv of core_gen is

component acc_gen
generic(width : natural := d_width);
port   (
       data     : inout std_logic_vector(width-1 downto 0);
       rden     : in    std_logic;
       wren     : in    std_logic;
       rst      : in    std_logic;
       halfd    : in    std_logic;
       pos, zer : out   std_logic
       );
end component;

component alu_gen
generic(width : natural := d_width);
port   (
       data : inout std_logic_vector(width-1 downto 0);
       alux : in    std_logic;
       code : in    std_logic_vector(3 downto 0);
       en   : in    std_logic;
       rst  : in std_logic
       );
end component;

component ctrl_gen
port(
    clk, rst                     : in  std_logic;
    accrd, accwr                 : out std_logic;
    halfd                        : out std_logic;
    aluen                        : out std_logic;
    alux                         : out std_logic;
    alucode                      : out std_logic_vector(3 downto 0);
    iaddr                        : out std_logic;
    irld                         : out std_logic;
    lda, sta, ldm, stm, ldi,
    jmpr, jmp, jzr, jz, jpr, jp,
    iand, ior, add, sub, inot,
    inc, dec, shfl, shfr, ina,
    outa, halt, nop              : in  std_logic;
    pcinc, pcld, pcsum, pcenaddr : out std_logic;
    ramrd, ramwr                 : out std_logic;
    regrd, regwr                 : out std_logic;
    portrd, portwr               : out std_logic
    );
end component;

component iar_gen
generic(width : natural := a_width);
port   (
       data  : in  std_logic_vector(width-1 downto 0);
       addr  : out std_logic_vector(width-1 downto 0);
       iaddr : in  std_logic
       );
end component;

component ir_gen
generic(width : natural := d_width);
port   (
       data            : in  std_logic_vector(width-1 downto 0);
       ld              : in  std_logic;
       rst             : in  std_logic;
       pos, zer        : in  std_logic;
       lda, sta, ldm,
       stm, ldi, jmpr,
       jmp, jzr, jz,
       jpr, jp, iand,
       ior, add, sub,
       inot, inc, dec,
       shfl, shfr, ina,
```

core_gen.vhd

```
       outa, halt, nop : out std_logic
       );
end component;

component pc_gen
generic(d_width : natural := d_width; a_width : natural := a_width);
port   (
       data   : in  std_logic_vector(a_width-1 downto 0);
       addr   : out std_logic_vector(a_width-1 downto 0);
       clk    : in  std_logic;
       rst    : in  std_logic;
       inc    : in  std_logic;
       ld     : in  std_logic;
       sum    : in  std_logic;
       enaddr : in  std_logic
       );
end component;

component reg_gen
generic(d_width : natural := d_width; a_width : natural := ra_width);
port   (
       data : inout std_logic_vector(d_width-1 downto 0);
       addr : in    std_logic_vector(a_width-1 downto 0);
       rden : in    std_logic;
       wren : in    std_logic;
       rst  : in    std_logic
       );
end component;

component portio_gen
generic(width : natural := d_width);
port   (
       dataio : inout std_logic_vector(width-1 downto 0);
       data   : inout std_logic_vector(width-1 downto 0);
       portrd : in    std_logic;
       portwr : in    std_logic
       );
end component;

signal data_bus : std_logic_vector(d_width-1 downto 0);
signal addr_bus : std_logic_vector(a_width-1 downto 0);
signal alucode  : std_logic_vector(3 downto 0);
signal lda, sta, ldm, stm, ldi, jmpr, jmp, jzr, jz, jpr, jp, iand, ior, add, sub, inot, inc,
dec, shfl, shfr, ina, outa, halt, nop : std_logic;
signal accrd, accwr, half, pos, zer, aluen, alux, iaddr, irld, pcinc, pcld, pcsum, pcaddr,
ramrd, ramwr, regrd, regwr, portrd, portwr : std_logic;

begin

   dutacc  : acc_gen    generic map(d_width)
                        port map (data_bus, accrd, accwr, rst, half, pos, zer);
   dutalu  : alu_gen    generic map(d_width)
                        port map (data_bus, alux, alucode, aluen, rst);
   dutctrl : ctrl_gen   port map (clk, rst, accrd, accwr, half, aluen, alux, alucode, iaddr,
irld, lda, sta, ldm, stm, ldi, jmpr, jmp, jzr, jz, jpr, jp, iand, ior, add, sub, inot, inc,
dec, shfl, shfr, ina, outa, halt, nop, pcinc, pcld, pcsum, pcaddr, ramrd, ramwr, regrd, regwr,
portrd, portwr);
   dutiar  : iar_gen    generic map(a_width)
                        port map (data_bus(a_width-1 downto 0), addr_bus, iaddr);
   dutir   : ir_gen     generic map(d_width)
                        port map (data_bus, irld, rst, pos, zer, lda, sta, ldm, stm, ldi,
jmpr, jmp, jzr, jz, jpr, jp, iand, ior, add, sub, inot, inc, dec, shfl, shfr, ina, outa, halt,
nop);
   dutpc   : pc_gen     generic map(d_width, a_width)
                        port map (data_bus(a_width-1 downto 0), addr_bus, clk, rst, pcinc,
pcld, pcsum, pcaddr);
   dutreg  : reg_gen    generic map(d_width, ra_width)
                        port map (data_bus, addr_bus(2 downto 0), regrd, regwr, rst);
   dutpio  : portio_gen generic map(d_width)
                        port map (databusio, data_bus, portrd, portwr);

   databus  <= data_bus when ramwr = '1' else (others => 'Z');
   data_bus <= databus when ramrd = '1' else (others => 'Z');
   memrd    <= ramrd; memwr <= ramwr;
   addrbus  <= addr_bus;

end behv;
```

system_gen.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity system_gen is
generic(d_width : natural; a_width : natural; ra_width : natural);
port	(
	portdata : inout std_logic_vector(d_width-1 downto 0);
	clk, rst : in    std_logic
	);
end system_gen;

architecture behv of system_gen is

component ram_gen
generic(d_width : natural := d_width; a_width : natural := a_width);
port	(
	data : inout std_logic_vector(d_width-1 downto 0);
	addr : in    std_logic_vector(a_width-1 downto 0);
	rden : in    std_logic;
	wren : in    std_logic;
	rst  : in    std_logic
	);
end component;

component core_gen
generic(d_width : natural := d_width; a_width : natural := a_width);
port	(
	databus      : inout std_logic_vector(d_width-1 downto 0);
	databusio    : inout std_logic_vector(d_width-1 downto 0);
	addrbus      : out   std_logic_vector(a_width-1 downto 0);
	memrd, memwr : out   std_logic;
	clk, rst     : in    std_logic
	);
end component;

signal databus : std_logic_vector(d_width-1 downto 0);
signal addrbus : std_logic_vector(a_width-1 downto 0);
signal memrd, memwr : std_logic;

begin

   dram : ram_gen  generic map(d_width, a_width)
                   port map (databus, addrbus, memrd, memwr, rst);
   dcpu : core_gen generic map(d_width, a_width)
                   port map (databus, portdata, addrbus, memrd, memwr, clk, rst);

end behv;
```

cpu_gen.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity cpu_gen is
generic(d_width : natural; a_width : natural; ra_width : natural);
port	(
	databusio : inout std_logic_vector(d_width-1 downto 0);
	clk, rst  : in    std_logic
	);
end cpu_gen;

architecture behv of cpu_gen is

component acc_gen
generic(width : natural := d_width);
port	(
	data     : inout std_logic_vector(width-1 downto 0);
	rden     : in    std_logic;
	wren     : in    std_logic;
	rst      : in    std_logic;
	halfd    : in    std_logic;
	pos, zer : out   std_logic
	);
```

cpu_gen.vhd

```
end component;

component alu_gen
generic(width : natural := d_width);
port    (
        data : inout std_logic_vector(width-1 downto 0);
        alux : in    std_logic;
        code : in    std_logic_vector(3 downto 0);
        en   : in    std_logic;
        rst  : in    std_logic
        );
end component;

component ctrl_gen
port(
    clk, rst                     : in  std_logic;
    accrd, accwr                 : out std_logic;
    halfd                        : out std_logic;
    aluen                        : out std_logic;
    alux                         : out std_logic;
    alucode                      : out std_logic_vector(3 downto 0);
    iaddr                        : out std_logic;
    irld                         : out std_logic;
    lda, sta, ldm, stm, ldi,
    jmpr, jmp, jzr, jz, jpr, jp,
    iand, ior, add, sub, inot,
    inc, dec, shfl, shfr, ina,
    outa, halt, nop              : in  std_logic;
    pcinc, pcld, pcsum, pcenaddr : out std_logic;
    ramrd, ramwr                 : out std_logic;
    regrd, regwr                 : out std_logic;
    portrd, portwr               : out std_logic
    );
end component;

component iar_gen
generic(width : natural := a_width);
port    (
        data  : in  std_logic_vector(width-1 downto 0);
        addr  : out std_logic_vector(width-1 downto 0);
        iaddr : in  std_logic
        );
end component;

component ir_gen
generic(width : natural := d_width);
port    (
        data             : in  std_logic_vector(width-1 downto 0);
        ld               : in  std_logic;
        rst              : in  std_logic;
        pos, zer         : in  std_logic;
        lda, sta, ldm,
        stm, ldi, jmpr,
        jmp, jzr, jz,
        jpr, jp, iand,
        ior, add, sub,
        inot, inc, dec,
        shfl, shfr, ina,
        outa, halt, nop : out std_logic
   );
end component;

component pc_gen is
generic(d_width : natural := d_width; a_width : natural := a_width);
port    (
        data   : in  std_logic_vector(a_width-1 downto 0);
        addr   : out std_logic_vector(a_width-1 downto 0);
        clk    : in  std_logic;
        rst    : in  std_logic;
        inc    : in  std_logic;
        ld     : in  std_logic;
        sum    : in  std_logic;
        enaddr : in  std_logic
        );
end component;

component reg_gen
generic(d_width : natural := d_width; a_width : natural := ra_width);
```

cpu_gen.vhd

```
port   (
       data : inout std_logic_vector(d_width-1 downto 0);
       addr : in    std_logic_vector(a_width-1 downto 0);
       rden : in    std_logic;
       wren : in    std_logic;
       rst  : in    std_logic
       );
end component;

component ram_gen is
generic(d_width : natural := d_width; a_width : natural := a_width);
port   (
       data : inout std_logic_vector(d_width-1 downto 0);
       addr : in    std_logic_vector(a_width-1 downto 0);
       rden : in    std_logic;
       wren : in    std_logic;
       rst  : in    std_logic
       );
end component;

component portio_gen is
generic(width : natural := d_width);
port   (
       dataio : inout std_logic_vector(width-1 downto 0);
       data   : inout std_logic_vector(width-1 downto 0);
       portrd : in    std_logic;
       portwr : in    std_logic
       );
end component;

signal data_bus : std_logic_vector(d_width-1 downto 0);
signal addr_bus : std_logic_vector(a_width-1 downto 0);
signal alucode  : std_logic_vector(3 downto 0);
signal lda, sta, ldm, stm, ldi, jmpr, jmp, jzr, jz, jpr, jp, iand, ior, add, sub, inot, inc,
dec, shfl, shfr, ina, outa, halt, nop : std_logic;
signal accrd, accwr, half, pos, zer, aluen, alux, iaddr, irld, pcinc, pcld, pcsum, pcaddr,
ramrd, ramwr, regrd, regwr, portrd, portwr : std_logic;

begin

   dutacc  : acc_gen    generic map(d_width)
                        port map (data_bus, accrd, accwr, rst, half, pos, zer);
   dutalu  : alu_gen    generic map(d_width)
                        port map (data_bus, alux, alucode, aluen, rst);
   dutctrl : ctrl_gen   port map (clk, rst, accrd, accwr, half, aluen, alux, alucode, iaddr,
irld, lda, sta, ldm, stm, ldi, jmpr, jmp, jzr, jz, jpr, jp, iand, ior, add, sub, inot, inc,
dec, shfl, shfr, ina, outa, halt, nop, pcinc, pcld, pcsum, pcaddr, ramrd, ramwr, regrd, regwr,
portrd, portwr);
   dutiar  : iar_gen    generic map(a_width)
                        port map (data_bus(a_width-1 downto 0), addr_bus, iaddr);
   dutir   : ir_gen     generic map(d_width)
                        port map (data_bus, irld, rst, pos, zer, lda, sta, ldm, stm, ldi,
jmpr, jmp, jzr, jz, jpr, jp, iand, ior, add, sub, inot, inc, dec, shfl, shfr, ina, outa, halt,
nop);
   dutpc   : pc_gen     generic map(d_width, a_width)
                        port map (data_bus(a_width-1 downto 0), addr_bus, clk, rst, pcinc,
pcld, pcsum, pcaddr);
   dutreg  : reg_gen    generic map(d_width, ra_width)
                        port map (data_bus, addr_bus(2 downto 0), regrd, regwr, rst);
   dutram  : ram_gen    generic map(d_width, a_width)
                        port map (data_bus, addr_bus, ramrd, ramwr, rst);
   dutpio  : portio_gen generic map(d_width)
                        port map (databusio, data_bus, portrd, portwr);

end behv;
```

2. Priedas. CTRL ir IR komponentų pavyzdinis SystemC kodas

ctrl.h

```
#include "systemc.h"

SC_MODULE(ctrl)
{

   sc_in<bool> clk, rst;
   sc_in<bool> lda, sta, ldm, stm, ldi, jmpr, jmp, jzr, jz, jpr, jp, iand, ior, add, sub,
inot, inc, dec, shfl, shfr, ina, outa, halt, nop;
   sc_out<bool> accrd, accwr, halfd, aluen, alux, iaddr, irld, pcinc, pcld, pcsum, pcenaddr,
ramrd, ramwr, regrd, regwr, portrd, portwr;
   sc_out< sc_bv<4> > alucode;

   sc_uint<6> busena;

   enum busenos{
               reset, bhalt, bnop,
               fetch, ftch0, ftch1, ftch2,
               lda0, lda1, lda2, lda3,
               sta0, sta1, sta2, sta3,
               ldm0, ldm1, ldm2, ldm3, ldm4, ldm5, ldm6,
               stm0, stm1, stm2, stm3, stm4, stm5, stm6,
               ldi0, ldi1,
               jumpr,
               jump0, jump1, jump2, jump3,
               arit0, arit1, arit2, arit3, arit4, arit5, arit6, arit7,
               log0, log1, log2, log3, log4,
               ina0, ina1, ina2,
               outa0, outa1, outa2
               };

   SC_CTOR(ctrl)
   {

      SC_METHOD(do_ctrl);
         sensitive << clk << rst;

   }

   void do_ctrl();

};
```

ctrl.cc

```
#include "systemc.h"
#include "ctrl.h"

void ctrl::do_ctrl()
{

   if (rst.read() == true)
      {busena = reset;}
   else if (clk.read() == true)
      {switch (busena)
         {case reset:
            {busena = fetch;}
         break;
         case fetch:
            {busena = ftch0;}
         break;
         case ftch0:
            {busena = ftch1;}
         break;
         case ftch1:
            {busena = ftch2;}
         break;
         case ftch2:
            {if (lda.read() == true)
               {busena = lda0;}
            else if (sta.read() == true)
               {busena = sta0;}
            else if (ldm.read() == true)
```

ctrl.cc

```
            {busena = ldm0;}
         else if (stm.read() == true)
            {busena = stm0;}
         else if (ldi.read() == true)
            {busena = ldi0;}
         else if (jmpr.read() == true)
            {busena = jumpr; }
         else if (jmp.read() == true)
            {busena = jump0;}
         else if (jzr.read() == true)
            {busena = jumpr;}
         else if (jz.read() == true)
            {busena = jump0;}
         else if (jpr.read() == true)
            {busena = jumpr;}
         else if (jp.read() == true)
            {busena = jump0;}
         else if (iand.read() == true)
            {busena = arit0;}
         else if (ior.read() == true)
            {busena = arit0;}
         else if (add.read() == true)
            {busena = arit0;}
         else if (sub.read() == true)
            {busena = arit0;}
         else if (inot.read() == true)
            {busena = log0;}
         else if (inc.read() == true)
            {busena = log0;}
         else if (dec.read() == true)
            {busena = log0;}
         else if (shfl.read() == true)
            {busena = log0;}
         else if (shfr.read() == true)
            {busena = log0;}
         else if (ina.read() == true)
            {busena = ina0;}
         else if (outa.read() == true)
            {busena = outa0;}
         else if (halt.read() == true)
            {busena = bhalt;}
         else if (nop.read() == true)
            {busena = bnop;}
      }
      break;
      case lda0:
         {busena = lda1;}
      break;
      case lda1:
         {busena = lda2;}
      break;
      case lda2:
         {busena = lda3;}
      break;
      case lda3:
         {busena = fetch;}
      break;
      case sta0:
         {busena = sta1;}
      break;
      case sta1:
         {busena = sta2;}
      break;
      case sta2:
         {busena = sta3;}
      break;
      case sta3:
         {busena = fetch;}
      break;
      case ldm0:
         {busena = ldm1;}
      break;
      case ldm1:
         {busena = ldm2;}
      break;
      case ldm2:
         {busena = ldm3;}
      break;
```

ctrl.cc

```
        case ldm3:
           {busena = ldm4;}
        break;
        case ldm4:
           {busena = ldm5;}
        break;
        case ldm5:
           {busena = ldm6;}
        break;
        case ldm6:
           {busena = fetch;}
        break;
        case stm0:
           {busena = stm1;}
        break;
        case stm1:
           {busena = stm2;}
        break;
        case stm2:
           {busena = stm3;}
        break;
        case stm3:
           {busena = stm4;}
        break;
        case stm4:
           {busena = stm5;}
        break;
        case stm5:
           {busena = stm6;}
        break;
        case stm6:
           {busena = fetch;}
        break;
        case ldi0:
           {busena = ldi1;}
        break;
        case ldi1:
           {busena = fetch;}
        break;
        case jumpr:
           {busena = fetch;}
        break;
        case jump0:
           {busena = jump1;}
        break;
        case jump1:
           {busena = jump2;}
        break;
        case jump2:
           {busena = jump3;}
        break;
        case jump3:
           {busena = fetch;}
        break;
        case arit0:
           {busena = arit1;}
        break;
        case arit1:
           {busena = arit2;}
        break;
        case arit2:
           {busena = arit3;}
        break;
        case arit3:
           {busena = arit4;}
        break;
        case arit4:
           {busena = arit5;}
        break;
        case arit5:
           {busena = arit6;}
        break;
        case arit6:
           {busena = arit7;}
        break;
        case arit7:
           {busena = fetch;}
        break;
```

ctrl.cc

```
        case log0:
           {busena = log1;}
        break;
        case log1:
           {busena = log2;}
        break;
        case log2:
           {busena = log3;}
        break;
        case log3:
           {busena = log4;}
        break;
        case log4:
           {busena = fetch;}
        break;
        case ina0:
           {busena = ina1;}
        break;
        case ina1:
           {busena = ina2;}
        break;
        case ina2:
           {busena = ftch0;}
        break;
        case outa0:
           {busena = outa1;}
        break;
        case outa1:
           {busena = outa2;}
        break;
        case outa2:
           {busena = ftch0;}
        break;
        case bhalt:
           {busena = bhalt;}
        break;
        case bnop:
           {busena = fetch;}
        break;
        default:
           {busena = bhalt;}
        break;
     }
  }

  if (busena == sta1 | busena == sta2 | busena == stm4 | busena == stm5 | busena == arit4 |
busena == log1 | busena == outa0 | busena == outa1)
     {accrd.write(true);}
  else
     {accrd.write(false);}

  if (busena == lda2 | busena == ldm5 | busena == ldi0 | busena == arit6 | busena == log3 |
busena == ina1)
     {accwr.write(true);}
  else
     {accwr.write(false);}

  if (busena == arit5 | busena == arit6 | busena == log2 | busena == log3)
     {aluen.write(true);}
  else
     {aluen.write(false);}

  if (busena == arit3)
     {alux.write(true);}
  else
     {alux.write(false);}

  if (busena == ldm3 | busena == ldm4 | busena == ldm5 | busena == stm3 | busena == stm4 |
busena == stm5 | busena == lda0 | busena == lda1 | busena == lda2 | busena == sta0 or   busena
== sta1 | busena == sta2 | busena == arit1 | busena == arit2 | busena == arit3)
     {iaddr.write(true);}
  else
     {iaddr.write(false);}

  if (busena == ftch2)
     {irld.write(true);}
  else
     {irld.write(false);}
```

ctrl.cc

```
   if (busena == lda3 | busena == sta3 | busena == ldm0 | busena == ldm6 | busena == stm0 |
busena == stm6 | busena == ldi1 | busena == jump0 | busena == arit7 | busena == log4 | busena
== ina2 | busena == outa2 | busena == bnop)
      {pcinc.write(true);}
   else
      {pcinc.write(false);}

   if (busena == jump3)
      {pcld.write(true);}
   else
      {pcld.write(false);}

   if (busena == jumpr)
      {pcsum.write(true);}
   else
      {pcsum.write(false);}

   if (busena == ftch0 | busena == ftch1 | busena == ftch2 | busena == ldm1 | busena == ldm2 |
busena == stm1 | busena == stm2 | busena == ldi0 | busena == jump1 | busena == jump2 | busena
== jump3)
      {pcenaddr.write(true);}
   else
      {pcenaddr.write(false);}

   if (busena == ftch1 | busena == ftch2 | busena == ldm2 | busena == ldm4 | busena == ldm5 |
busena == stm2 | busena == ldi0 | busena == jumpr | busena == jump2 | busena == jump3 | busena
== arit0 | busena == log0)
      {ramrd.write(true);}
   else
      {ramrd.write(false);}

   if (busena == stm5)
      {ramwr.write(true);}
   else
      {ramwr.write(false);}

   if (busena == lda1 | busena == lda2 | busena == arit2 | busena == arit3)
      {regrd.write(true);}
   else
      {regrd.write(false);}

   if (busena == sta2)
      {regwr.write(true);}
   else
      {regwr.write(false);}

   if (busena == ina0 | busena == ina1)
      {portrd.write(true);}
   else
      {portrd.write(false);}

   if (busena == outa1)
      {portwr.write(true);}
   else
      {portwr.write(false);}

   if (iand.read() == true)
      {alucode.write("0000");}
   else if (ior.read() == true)
      {alucode.write("0001");}
   else if (add.read() == true)
      {alucode.write("0010");}
   else if (sub.read() == true)
      {alucode.write("0011");}
   else if (inot.read() == true)
      {alucode.write("0100");}
   else if (inc.read() == true)
      {alucode.write("0101");}
   else if (dec.read() == true)
      {alucode.write("0110");}
   else if   (shfl.read() == true)
      {alucode.write("0111");}
   else if (shfr.read() == true)
      {alucode.write("1000");}
   else
      {alucode.write(NULL);}
```

ctrl.cc

```
   if (ldi.read() == true)
      {halfd.write(true);}
   else
      {halfd.write(false);}

}
```

ir.h

```
#include "systemc.h"

SC_MODULE(ir)
{

   sc_in<bool> ld;
   sc_in<bool> rst;
   sc_in<bool> pos, zer;
   sc_in< sc_bv<8> > data;
   sc_out<bool> lda, sta, ldm, stm, ldi, jmpr, jmp, jzr, jz, jpr, jp, iand, ior, add, sub,
inot, inc, dec, shfl, shfr, ina, outa, halt, nop;

   sc_bv<8> irdata;

   SC_CTOR(ir)
   {

      SC_METHOD(do_ir);
         sensitive << ld << rst;

   }

   void do_ir();

};
```

ir.cc

```
#include "systemc.h"
#include "ir.h"

void ir::do_ir()
{

   if (rst.read() == true)
      {irdata = "00000000";}
   else if (ld.read() == true)
      {irdata = data.read();}

   if (irdata.range(7,4) == "0001")
      {lda.write(true);}
   else
      {lda.write(false);}

   if (irdata.range(7,4) == "0010")
      {sta.write(true);}
   else
      {sta.write(false);}

   if (irdata.range(7,4) == "0011")
      {ldm.write(true);}
   else
      {ldm.write(false);}

   if (irdata.range(7,4) == "0100")
      {stm.write(true);}
   else
      {stm.write(false);}

   if (irdata.range(7,4) == "0101")
      {ldi.write(true);}
   else
      {ldi.write(false);}

   if ((irdata.range(7,4) == "0110") & (~(irdata.range(3,0) == "0000")))
      {jmpr.write(true);}
   else
```

ir.cc

```
    {jmpr.write(false);}

if (irdata == "01100000")
    {jmp.write(true);}
else
    {jmp.write(false);}

if (((irdata.range(7,4) == "0111") & (irdata.range(3,0) != "0000") & (zer.read() == true))
    {jzr.write(true);}
else
    {jzr.write(false);}

if ((irdata == "01110000") & (zer.read() == true))
    {jz.write(true);}
else
    {jz.write(false);}

if ((irdata.range(7,4) == "1000") & (irdata.range(3,0) != "0000") & (pos.read() == true))
    {jpr.write(true);}
else
    {jpr.write(false);}

if ((irdata == "10000000") & (pos.read() == true))
    {jp.write(true);}
else
    {jp.write(false);}

if (irdata.range(7,4) == "1001")
    {iand.write(true);}
else
    {iand.write(false);}

if (irdata.range(7,4) == "1010")
    {ior.write(true);}
else
    {ior.write(false);}

if (irdata.range(7,4) == "1011")
    {add.write(true);}
else
    {add.write(false);}

if (irdata.range(7,4) == "1100")
    {sub.write(true);}
else
    {sub.write(false);}

if ((irdata.range(7,4) == "1101") & (irdata.range(2,0) == "000"))
    {inot.write(true);}
else
    {inot.write(false);}

if ((irdata.range(7,4) == "1101") & (irdata.range(2,0) == "001"))
    {inc.write(true);}
else
    {inc.write(false);}

if ((irdata.range(7,4) == "1101") & (irdata.range(2,0) == "010"))
    {dec.write(true);}
else
    {dec.write(false);}

if ((irdata.range(7,4) == "1101") & (irdata.range(2,0) == "011"))
    {shfl.write(true);}
else
    {shfl.write(false);}

if ((irdata.range(7,4) == "1101") & (irdata.range(2,0) == "100"))
    {shfr.write(true);}
else
    {shfr.write(false);}

if ((irdata.range(7,4) == "1110") & (irdata[0] == false))
    {ina.write(true);}
else
    {ina.write(false);}

if ((irdata.range(7,4) == "1110") & (irdata[0] == true))
```

ir.cc

```
      {outa.write(true);}
   else
      {outa.write(false);}

   if (irdata.range(7,4) == "1111")
      {halt.write(true);}
   else
      {halt.write(false);}

   if (irdata == "00000000")
      {nop.write(true);}
   else
      {nop.write(false);}

}
```

3. Priedas. Bendrintų CTRL ir IR komponetų pavyzdinis SystemC kodas.

ctrl_gen_rj.h

```
#include "systemc.h"

#define jxr

SC_MODULE(ctrl_gen_rj)
{

   sc_in<bool> clk, rst;
   sc_in<bool> lda, sta, ldm, stm, ldi,
   #ifdef jxr
   jmpr,
   jzr,
   jpr,
   #endif
   jmp,
   jz,
   jp,
   iand, ior, add, sub, inot, inc, dec, shfl, shfr, ina, outa, halt, nop;
   sc_out<bool> accrd, accwr, halfd, aluen, alux, iaddr, irld, pcinc, pcld,
   #ifdef jxr
   pcsum,
   #endif
   pcenaddr, ramrd, ramwr, regrd, regwr, portrd, portwr;
   sc_out< sc_bv<4> > alucode;

   sc_uint<6> busena;

   enum busenos{
            reset, bhalt, bnop,
            fetch, ftch0, ftch1, ftch2,
            lda0, lda1, lda2, lda3,
            sta0, sta1, sta2, sta3,
            ldm0, ldm1, ldm2, ldm3, ldm4, ldm5, ldm6,
            stm0, stm1, stm2, stm3, stm4, stm5, stm6,
            ldi0, ldi1,
            #ifdef jxr
            jumpr,
            #endif
            jump0, jump1, jump2, jump3,
            arit0, arit1, arit2, arit3, arit4, arit5, arit6, arit7,
            log0, log1, log2, log3, log4,
            ina0, ina1, ina2,
            outa0, outa1, outa2
            };

   SC_CTOR(ctrl_gen_rj)
   {

      SC_METHOD(do_ctrl_gen_rj);
         sensitive << clk << rst;

   }

   void do_ctrl_gen_rj();

};
```

ctrl_gen_rj.cc

```
#include "systemc.h"
#include "ctrl_gen_rj.h"

void ctrl_gen_rj::do_ctrl_gen_rj()
{

   if (rst.read() == true)
      {busena = reset;}
   else if (clk.read() == true)
      {switch (busena)
         {case reset:
            {busena = fetch;}
         break;
```

ctrl_gen_rj.cc

```
        case fetch:
           {busena = ftch0;}
        break;
        case ftch0:
           {busena = ftch1;}
        break;
        case ftch1:
           {busena = ftch2;}
        break;
        case ftch2:
           {if (lda.read() == true)
              {busena = lda0;}
           else if (sta.read() == true)
              {busena = sta0;}
           else if (ldm.read() == true)
              {busena = ldm0;}
           else if (stm.read() == true)
              {busena = stm0;}
           else if (ldi.read() == true)
              {busena = ldi0;}
           #ifdef jxr
           else if (jmpr.read() == true)
              {busena = jumpr;}
           else if (jzr.read() == true)
              {busena = jumpr;}
           else if (jpr.read() == true)
              {busena = jumpr;}
           #endif
           else if (jmp.read() == true)
              {busena = jump0;}
           else if (jz.read() == true)
              {busena = jump0;}
           else if (jp.read() == true)
              {busena = jump0;}
           else if (iand.read() == true)
              {busena = arit0;}
           else if (ior.read() == true)
              {busena = arit0;}
           else if (add.read() == true)
              {busena = arit0;}
           else if (sub.read() == true)
              {busena = arit0;}
           else if (inot.read() == true)
              {busena = log0;}
           else if (inc.read() == true)
              {busena = log0;}
           else if (dec.read() == true)
              {busena = log0;}
           else if (shfl.read() == true)
              {busena = log0;}
           else if (shfr.read() == true)
              {busena = log0;}
           else if (ina.read() == true)
              {busena = ina0;}
           else if (outa.read() == true)
              {busena = outa0;}
           else if (halt.read() == true)
              {busena = bhalt;}
           else if (nop.read() == true)
              {busena = bnop;}
           }
        break;
        case lda0:
           {busena = lda1;}
        break;
        case lda1:
           {busena = lda2;}
        break;
        case lda2:
           {busena = lda3;}
        break;
        case lda3:
           {busena = fetch;}
        break;
        case sta0:
           {busena = sta1;}
        break;
        case sta1:
```

ctrl_gen_rj.cc

```
            {busena = sta2;}
         break;
         case sta2:
            {busena = sta3;}
         break;
         case sta3:
            {busena = fetch;}
         break;
         case ldm0:
            {busena = ldm1;}
         break;
         case ldm1:
            {busena = ldm2;}
         break;
         case ldm2:
            {busena = ldm3;}
         break;
         case ldm3:
            {busena = ldm4;}
         break;
         case ldm4:
            {busena = ldm5;}
         break;
         case ldm5:
            {busena = ldm6;}
         break;
         case ldm6:
            {busena = fetch;}
         break;
         case stm0:
            {busena = stm1;}
         break;
         case stm1:
            {busena = stm2;}
         break;
         case stm2:
            {busena = stm3;}
         break;
         case stm3:
            {busena = stm4;}
         break;
         case stm4:
            {busena = stm5;}
         break;
         case stm5:
            {busena = stm6;}
         break;
         case stm6:
            {busena = fetch;}
         break;
         case ldi0:
            {busena = ldi1;}
         break;
         case ldi1:
            {busena = fetch;}
         break;
         #ifdef jxr
         case jumpr:
            {busena = fetch;}
         break;
         #endif
         case jump0:
            {busena = jump1;}
         break;
         case jump1:
            {busena = jump2;}
         break;
         case jump2:
            {busena = jump3;}
         break;
         case jump3:
            {busena = fetch;}
         break;
         case arit0:
            {busena = arit1;}
         break;
         case arit1:
            {busena = arit2;}
```

ctrl_gen_rj.cc

```
         break;
         case arit2:
            {busena = arit3;}
         break;
         case arit3:
            {busena = arit4;}
         break;
         case arit4:
            {busena = arit5;}
         break;
         case arit5:
            {busena = arit6;}
         break;
         case arit6:
            {busena = arit7;}
         break;
         case arit7:
            {busena = fetch;}
         break;
         case log0:
            {busena = log1;}
         break;
         case log1:
            {busena = log2;}
         break;
         case log2:
            {busena = log3;}
         break;
         case log3:
            {busena = log4;}
         break;
         case log4:
            {busena = fetch;}
         break;
         case ina0:
            {busena = ina1;}
         break;
         case ina1:
            {busena = ina2;}
         break;
         case ina2:
            {busena = ftch0;}
         break;
         case outa0:
            {busena = outa1;}
         break;
         case outa1:
            {busena = outa2;}
         break;
         case outa2:
            {busena = ftch0;}
         break;
         case bhalt:
            {busena = bhalt;}
         break;
         case bnop:
            {busena = fetch;}
         break;
         default:
            {busena = bhalt;}
         break;
      }
   }

   if (busena == sta1 | busena == sta2 | busena == stm4 | busena == stm5 | busena == arit4 |
busena == log1 | busena == outa0 | busena == outa1)
      {accrd.write(true);}
   else
      {accrd.write(false);}

   if (busena == lda2 | busena == ldm5 | busena == ldi0 | busena == arit6 | busena == log3 |
busena == ina1)
      {accwr.write(true);}
   else
      {accwr.write(false);}

   if (busena == arit5 | busena == arit6 | busena == log2 | busena == log3)
      {aluen.write(true);}
```

ctrl_gen_rj.cc

```
   else
      {aluen.write(false);}

   if (busena == arit3)
      {alux.write(true);}
   else
      {alux.write(false);}

   if (busena == ldm3 | busena == ldm4 | busena == ldm5 | busena == stm3 | busena == stm4 |
busena == stm5 | busena == lda0 | busena == lda1 | busena == lda2 | busena == sta0 or   busena
== sta1 | busena == sta2 | busena == arit1 | busena == arit2 | busena == arit3)
      {iaddr.write(true);}
   else
      {iaddr.write(false);}

   if (busena == ftch2)
      {irld.write(true);}
   else
      {irld.write(false);}

   if (busena == lda3 | busena == sta3 | busena == ldm0 | busena == ldm6 | busena == stm0 |
busena == stm6 | busena == ldi1 | busena == jump0 | busena == arit7 | busena == log4 | busena
== ina2 | busena == outa2 | busena == bnop)
      {pcinc.write(true);}
   else
      {pcinc.write(false);}

   if (busena == jump3)
      {pcld.write(true);}
   else
      {pcld.write(false);}

   #ifdef jxr
   if (busena == jumpr)
      {pcsum.write(true);}
   else
      {pcsum.write(false);}
   #endif

   if (busena == ftch0 | busena == ftch1 | busena == ftch2 | busena == ldm1 | busena == ldm2 |
busena == stm1 | busena == stm2 | busena == ldi0 | busena == jump1 | busena == jump2 | busena
== jump3)
      {pcenaddr.write(true);}
   else
      {pcenaddr.write(false);}

   if (busena == ftch1 | busena == ftch2 | busena == ldm2 | busena == ldm4 | busena == ldm5 |
busena == stm2 | busena == ldi0 |
   #ifdef jxr
   busena == jumpr |
   #endif
   busena == jump2 | busena == jump3 | busena == arit0 | busena == log0)
      {ramrd.write(true);}
   else
      {ramrd.write(false);}

   if (busena == stm5)
      {ramwr.write(true);}
   else
      {ramwr.write(false);}

   if (busena == lda1 | busena == lda2 | busena == arit2 | busena == arit3)
      {regrd.write(true);}
   else
      {regrd.write(false);}

   if (busena == sta2)
      {regwr.write(true);}
   else
      {regwr.write(false);}

   if (busena == ina0 | busena == ina1)
      {portrd.write(true);}
   else
      {portrd.write(false);}

   if (busena == outa1)
      {portwr.write(true);}
```

ctrl_gen_rj.cc

```
   else
      {portwr.write(false);}

   if (iand.read() == true)
      {alucode.write("0000");}
   else if (ior.read() == true)
      {alucode.write("0001");}
   else if (add.read() == true)
      {alucode.write("0010");}
   else if (sub.read() == true)
      {alucode.write("0011");}
   else if (inot.read() == true)
      {alucode.write("0100");}
   else if (inc.read() == true)
      {alucode.write("0101");}
   else if (dec.read() == true)
      {alucode.write("0110");}
   else if   (shfl.read() == true)
      {alucode.write("0111");}
   else if (shfr.read() == true)
      {alucode.write("1000");}
   else
      {alucode.write(NULL);}

   if (ldi.read() == true)
      {halfd.write(true);}
   else
      {halfd.write(false);}

}
```

ir_gen_rj.h

```
#include "systemc.h"

#define jxr

SC_MODULE(ir_gen_rj)
{

   sc_in<bool> ld;
   sc_in<bool> rst;
   sc_in<bool> pos;
   sc_in<bool> zer;
   sc_in< sc_bv<8> > data;
   sc_out<bool> lda, sta, ldm, stm, ldi,
   #ifdef jxr
   jmpr,
   jzr,
   jpr,
   #endif
   jmp,
   jz,
   jp,
   iand, ior, add, sub, inot, inc, dec, shfl, shfr, ina, outa, halt, nop;

   sc_bv<8> irdata;

   SC_CTOR(ir_gen_rj)
   {

      SC_METHOD(do_ir_gen_rj);
         sensitive << ld << rst;

   }

   void do_ir_gen_rj();

};
```

ir_gen_rj.cc

```
#include "systemc.h"
#include "ir_gen_rj.h"

void ir_gen_rj::do_ir_gen_rj()
```

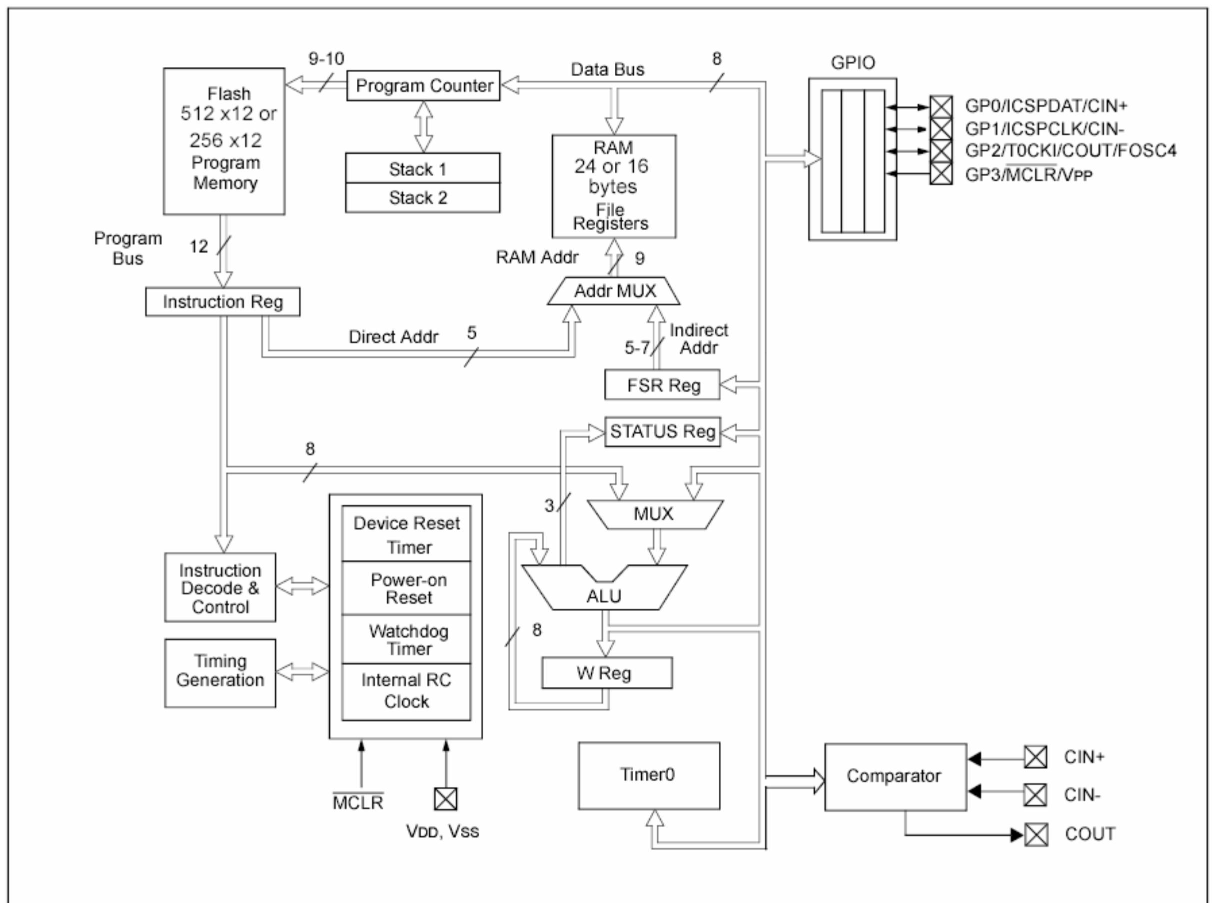
ir_gen_rj.cc

```
{

   if (rst.read() == true)
      {irdata = "00000000";}
   else if (ld.read() == true)
      {irdata = data.read();}

   if (irdata.range(7,4) == "0001")
      {lda.write(true);}
   else
      {lda.write(false);}

   if (irdata.range(7,4) == "0010")
      {sta.write(true);}
   else
      {sta.write(false);}

   if (irdata.range(7,4) == "0011")
      {ldm.write(true);}
   else
      {ldm.write(false);}

   if (irdata.range(7,4) == "0100")
      {stm.write(true);}
   else
      {stm.write(false);}

   if (irdata.range(7,4) == "0101")
      {ldi.write(true);}
   else
      {ldi.write(false);}

   #ifdef jxr
   if ((irdata.range(7,4) == "0110") & (irdata.range(3,0) != "0000"))
      {jmpr.write(true);}
   else
      {jmpr.write(false);}

   if ((irdata.range(7,4) == "0111") & (irdata.range(3,0) != "0000") & (zer.read() == true))
      {jzr.write(true);}
   else
      {jzr.write(false);}

   if ((irdata.range(7,4) == "1000") & (irdata.range(3,0) != "0000") & (pos.read() == true))
      {jpr.write(true);}
   else
      {jpr.write(false);}
   #endif

   if (irdata == "01100000")
      {jmp.write(true);}
   else
      {jmp.write(false);}

   if ((irdata == "01110000") & (zer.read() == true))
      {jz.write(true);}
   else
      {jz.write(false);}

   if ((irdata == "10000000") & (pos.read() == true))
      {jp.write(true);}
   else
      {jp.write(false);}

   if (irdata.range(7,4) == "1001")
      {iand.write(true);}
   else
      {iand.write(false);}

   if (irdata.range(7,4) == "1010")
      {ior.write(true);}
   else
      {ior.write(false);}

   if (irdata.range(7,4) == "1011")
      {add.write(true);}
   else
      {add.write(false);}
```

ir_gen_rj.cc

```
   if (irdata.range(7,4) == "1100")
      {sub.write(true);}
   else
      {sub.write(false);}

   if ((irdata.range(7,4) == "1101") & (irdata.range(2,0) == "000"))
      {inot.write(true);}
   else
      {inot.write(false);}

   if ((irdata.range(7,4) == "1101") & (irdata.range(2,0) == "001"))
      {inc.write(true);}
   else
      {inc.write(false);}

   if ((irdata.range(7,4) == "1101") & (irdata.range(2,0) == "010"))
      {dec.write(true);}
   else
      {dec.write(false);}

   if ((irdata.range(7,4) == "1101") & (irdata.range(2,0) == "011"))
      {shfl.write(true);}
   else
      {shfl.write(false);}

   if ((irdata.range(7,4) == "1101") & (irdata.range(2,0) == "100"))
      {shfr.write(true);}
   else
      {shfr.write(false);}

   if ((irdata.range(7,4) == "1110") & (irdata[0] == false))
      {ina.write(true);}
   else
      {ina.write(false);}

   if ((irdata.range(7,4) == "1110") & (irdata[0] == true))
      {outa.write(true);}
   else
      {outa.write(false);}

   if (irdata.range(7,4) == "1111")
      {halt.write(true);}
   else
      {halt.write(false);}

   if (irdata == "00000000")
      {nop.write(true);}
   else
      {nop.write(false);}

}
```

4. Priedas. PIC10F200 blokinė diagrama



34 pav. PIC10F200 blokinė diagrama [5]

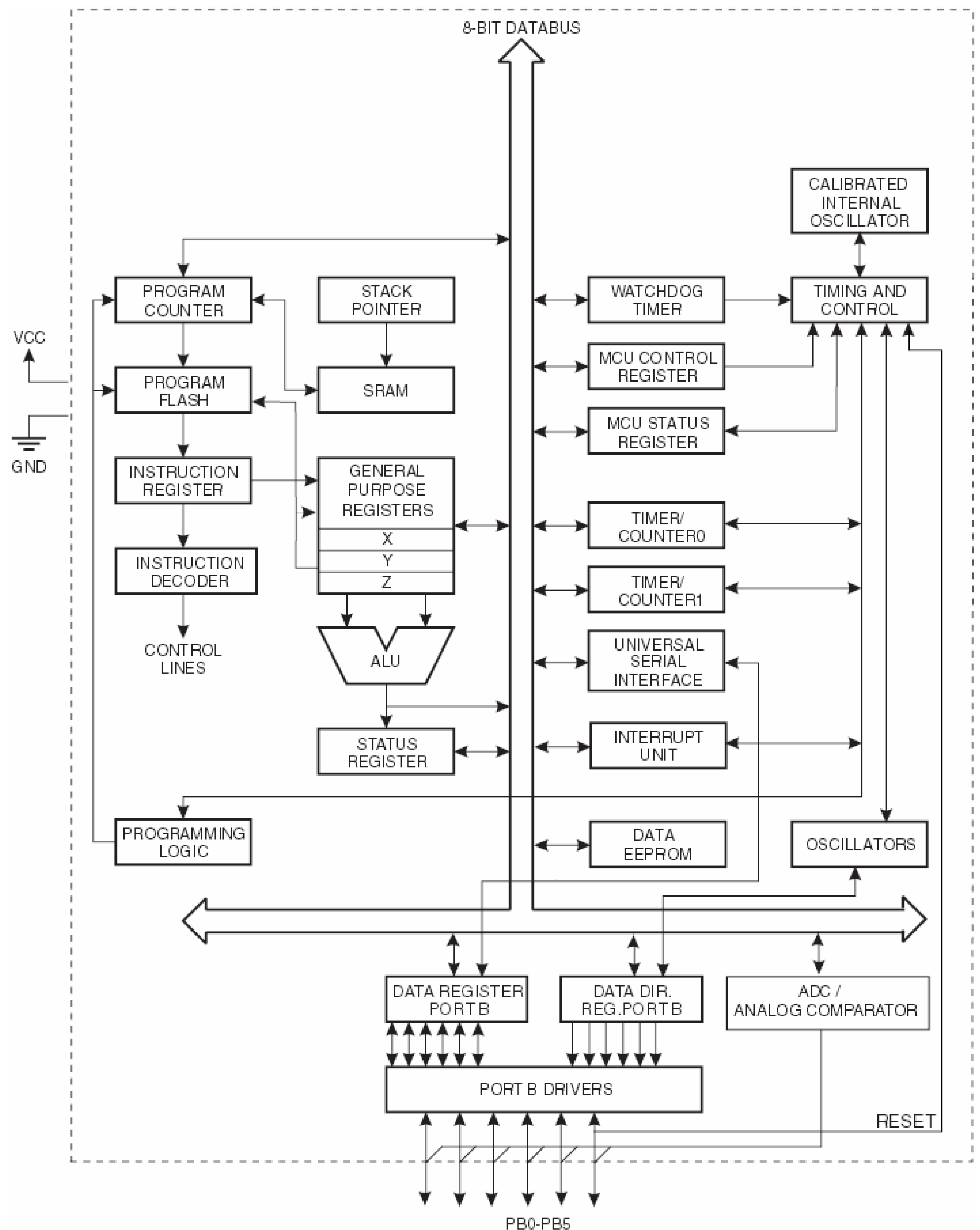
5. Priedas. PIC10F200 instrukcijų rinkinys

15 lentelė. PIC10F200 instrukcijų rinkinys [5]

Mnemonic	Operands	Description	Cycles	12-Bit Opcode			Status Affected
				MSb		LSb	
ADDWF	f, d	Add W and f	1	0001	0001	0000	C, DC, Z
ANDWF	f, d	AND W with f	1	0000	0010	0000	Z
CLRF	f	Clear f	1	0010	0010	0011	Z
CLRW	—	Clear W	1	0001	0010	0000	Z
COMF	f, d	Complement f	1	0000	0011	0011	Z
DECF	f, d	Decrement f	1	0000	0011	0001	Z
DECFSZ	f, d	Decrement f, Skip if 0	1	11df	01df	011f	None
INCF	f, d	Increment f	1	0100	01df	11df	Z
INCFSZ	f, d	Increment f, Skip if 0	1	11df	10df	11df	None
IORWF	f, d	Inclusive OR W with f	1	00df	00df	001f	Z
MOVF	f, d	Move f	1	0000	01df	00df	Z
MOVWF	f	Move W to f	1	10df	10df	10df	None

				12-Bit Opcode	
NOP	—	No Operation	1	ffff ffff ffff	None
RLF	f, d	Rotate left f through Carry	1	0000 ffff ffff	C
RRF	f, d	Rotate right f through Carry	1	ffff ffff ffff	C
SUBWF	f, d	Subtract W from f	1	ffff ffff ffff	C, DC, Z
SWAPF	f, d	Swap f	1	0000 ffff ffff	None
XORWF	f, d	Exclusive OR W with f	1	ffff ffff ffff	Z
BIT-ORIENTED FILE REGISTER OPERATIONS					
BCF	f, b	Bit Clear f	1	0100 bbbf ffff	None
BSF	f, b	Bit Set f	1	0101 bbbf ffff	None
BTFSC	f, b	Bit Test f, Skip if Clear	1	0110 bbbf ffff	None
BTFSS	f, b	Bit Test f, Skip if Set	1	0111 bbbf ffff	None
LITERAL AND CONTROL OPERATIONS					
ANDLW	k	AND literal with W	1	1110 kkkk kkkk	Z
CALL	k	Call Subroutine	2	1001 kkkk kkkk	None
CLRWDT		Clear Watchdog Timer	1	0000 0000 0100	TO, PD
GOTO	k	Unconditional branch	2	101k kkkk kkkk	None
IORLW	k	Inclusive OR literal with W	1	1101 kkkk kkkk	Z
MOVLW	k	Move literal to W	1	1100 kkkk kkkk	None
OPTION	—	Load OPTION register	1	0000 0000 0010	None
RETLW	k	Return, place Literal in W	2	1000 kkkk kkkk	None
SLEEP	—	Go into Standby mode	1	0000 0000 0011	TO, PD
TRIS	f	Load TRIS register	1	0000 0000 0fff	None
XORLW	k	Exclusive OR literal to W	1	1111 kkkk kkkk	Z

6. Priedas. ATtiny25 blokinė diagrama



35 pav. ATtiny25 blokinė diagrama [4]

7. Priedas. ATtiny25 instrukcijų rinkinys

Mnemonics	Operands	Description	Operation	Flags	#Clocks
ARITHMETIC AND LOGIC INSTRUCTIONS					
ADD	Rd, Rr	Add two Registers	Rd ← Rd + Rr	Z,C,N,V,H	1
ADC	Rd, Rr	Add with Carry two Registers	Rd ← Rd + Rr + C	Z,C,N,V,H	1
ADIW	Rdl,K	Add Immediate to Word	Rdh:Rdl ← Rdh:Rdl + K	Z,C,N,V,S	2
SUB	Rd, Rr	Subtract two Registers	Rd ← Rd – Rr	Z,C,N,V,H	1
SUBI	Rd, K	Subtract Constant from Register	Rd ← Rd – K	Z,C,N,V,H	1
SBC	Rd, Rr	Subtract with Carry two Registers	Rd ← Rd - Rr – C	Z,C,N,V,H	1
SBCI	Rd, K	Subtract with Carry Constant from Reg.	Rd ← Rd - K – C	Z,C,N,V,H	1
SBIW	Rdl,K	Subtract Immediate from Word	Rdh:Rdl ← Rdh:Rdl – K	Z,C,N,V,S	2
AND	Rd, Rr	Logical AND Registers	Rd ← Rd • Rr	Z,N,V	1
ANDI	Rd, K	Logical AND Register and Constant	Rd ← Rd • K	Z,N,V	1
OR	Rd, Rr	Logical OR Registers	Rd ← Rd v Rr	Z,N,V	1
ORI	Rd, K	Logical OR Register and Constant	Rd ← Rd v K	Z,N,V	1
EOR	Rd, Rr	Exclusive OR Registers	Rd ← Rd ⊕ Rr	Z,N,V	1
COM	Rd	One's Complement	Rd ← 0xFF – Rd	Z,C,N,V	1
NEG	Rd	Two's Complement	Rd ← 0x00 – Rd	Z,C,N,V,H	1
SBR	Rd,K	Set Bit(s) in Register	Rd ← Rd v K	Z,N,V	1
CBR	Rd,K	Clear Bit(s) in Register	Rd ← Rd • (0xFF - K)	Z,N,V	1
INC	Rd	Increment	Rd ← Rd + 1	Z,N,V	1
DEC	Rd	Decrement	Rd ← Rd – 1	Z,N,V	1
TST	Rd	Test for Zero or Minus	Rd ← Rd • Rd	Z,N,V	1
CLR	Rd	Clear Register	Rd ← Rd ⊕ Rd	Z,N,V	1
SER	Rd	Set Register	Rd ← 0xFF	None	1
BRANCH INSTRUCTIONS					
RJMP	k	Relative Jump	PC ← PC + k + 1	None	2

Mnemonics	Operands	Description	Operation	Flags	#Clocks
IJMP		Indirect Jump to (Z)	PC ← Z	None	2
RCALL	k	Relative Subroutine Call	PC ← PC + k + 1	None	3
ICALL		Indirect Call to (Z)	PC ← Z	None	3
RET		Subroutine Return	PC ← STACK	None	4
RETI		Interrupt Return	PC ← STACK	I	4
CPSE	Rd,Rr	Compare, Skip if Equal if (Rd = Rr)	PC ← PC + 2 or 3	None	1/2/3
CP	Rd,Rr	Compare	Rd – Rr	Z, N,V,C,H	1
CPC	Rd,Rr	Compare with Carry	Rd – Rr – C	Z, N,V,C,H 1	
CPI	Rd,K	Compare Register with Immediate	Rd – K	Z, N,V,C,H	1
SBRC	Rr, b	Skip if Bit in Register Cleared	if (Rr(b)=0) PC ← PC + 2 or 3	None	1/2/3
SBRS	Rr, b	Skip if Bit in Register is Set	if (Rr(b)=1) PC ← PC + 2 or 3	None	1/2/3
SBIC	P, b	Skip if Bit in I/O Register Cleared	if (P(b)=0) PC ← PC + 2 or 3	None	1/2/3
SBIS	P, b	Skip if Bit in I/O Register is Set	if (P(b)=1) PC ← PC + 2 or 3	None	1/2/3
BRBS	s, k	Branch if Status Flag Set	if (SREG(s) = 1) then PC←PC+k + 1	None	1/2
BRBC	s, k	Branch if Status Flag Cleared	if (SREG(s) = 0) then PC←PC+k + 1	None	1/2
BREQ	k	Branch if Equal	if (Z = 1) then PC ← PC + k + 1	None	1/2
BRNE	k	Branch if Not Equal	if (Z = 0) then PC ← PC + k + 1	None	1/2
BRCS	k	Branch if Carry Set	if (C = 1) then PC ← PC + k + 1	None	1/2
BRCC	k	Branch if Carry Cleared	if (C = 0) then PC ← PC + k + 1	None	1/2
BRSH	k	Branch if Same or Higher	if (C = 0) then PC ← PC + k + 1	None	1/2
BRLO	k	Branch if Lower	if (C = 1) then PC ← PC + k + 1	None	1/2
BRMI	k	Branch if Minus	if (N = 1) then PC ← PC + k + 1	None	1/2
BRPL	k	Branch if Plus	if (N = 0) then PC ← PC + k + 1	None	1/2
BRGE	k	Branch if Greater or Equal, Signed	if (N ⊕ V= 0) then PC ← PC + k + 1	None	1/2
BRLT	k	Branch if Less Than Zero, Signed	if (N ⊕ V= 1) then PC ← PC + k + 1	None	1/2
BRHS	k	Branch if Half Carry Flag Set	if (H = 1) then PC ← PC + k + 1	None	1/2

Mnemonics	Operands	Description	Operation	Flags	#Clocks
BRHC	k	Branch if Half Carry Flag Cleared	if (H = 0) then PC ← PC + k + 1	None	1/2
BRTS	k	Branch if T Flag Set	if (T = 1) then PC ← PC + k + 1	None	1/2
BRTC	k	Branch if T Flag Cleared	if (T = 0) then PC ← PC + k + 1	None	1/2
BRVS	k	Branch if Overflow Flag is Set	if (V = 1) then PC ← PC + k + 1	None	1/2
BRVC	k	Branch if Overflow Flag is Cleared	if (V = 0) then PC ← PC + k + 1	None	1/2
BRIE	k	Branch if Interrupt Enabled	if (I = 1) then PC ← PC + k + 1	None	1/2
BRID	k	Branch if Interrupt Disabled	if (I = 0) then PC ← PC + k + 1	None	1/2
BIT AND BIT-TEST INSTRUCTIONS					
SBI	P,b	Set Bit in I/O Register	I/O(P,b) ← 1	None	2
CBI	P,b	Clear Bit in I/O Register	I/O(P,b) ← 0	None	2
LSL	Rd	Logical Shift Left	Rd(n+1) ← Rd(n), Rd(0) ← 0	Z,C,N,V	1
LSR	Rd	Logical Shift Right	Rd(n) ← Rd(n+1), Rd(7) ← 0	Z,C,N,V	1
ROL	Rd	Rotate Left Through Carry	Rd(0)←C,Rd(n+1)← Rd(n),C←Rd(7)	Z,C,N,V	1
ROR	Rd	Rotate Right Through Carry	Rd(7)←C,Rd(n)← Rd(n+1),C←Rd(0)	Z,C,N,V	1
ASR	Rd	Arithmetic Shift Right	Rd(n) ← Rd(n+1), n=0..6	Z,C,N,V	1
SWAP	Rd	Swap Nibbles	Rd(3..0)←Rd(7..4),Rd(7..4)←Rd(3..0)	None	1
BSET	s	Flag Set	SREG(s) ← 1	SREG(s)	1
BCLR	s	Flag Clear	SREG(s) ← 0	SREG(s)	1
BST	Rr, b	Bit Store from Register to T	T ← Rr(b)	T	1
BLD	Rd, b	Bit load from T to Register	Rd(b) ← T	None	1
SEC		Set Carry	C ← 1	C	1
CLC		Clear Carry	C ← 0	C	1
SEN		Set Negative Flag	N ← 1	N	1
CLN		Clear Negative Flag	N ← 0	N	1
SEZ		Set Zero Flag	Z ← 1	Z	1
CLZ		Clear Zero Flag	Z ← 0	Z	1
SEI		Global Interrupt Enable	I ← 1	I	1

Mnemonics	Operands	Description	Operation	Flags	#Clocks
CLI		Global Interrupt Disable	I ← 0	I	1
SES		Set Signed Test Flag	S ← 1	S	1
CLS		Clear Signed Test Flag	S ← 0	S	1
SEV		Set Twos Complement Overflow	V ← 1	V	1
CLV		Clear Twos Complement Overflow	V ← 0	V	1
SET		Set T in SREG	T ← 1	T	1
CLT		Clear T in SREG	T ← 0	T	1
SEH		Set Half Carry Flag in SREG	H ← 1	H	1
CLH		Clear Half Carry Flag in SREG	H ← 0	H	1
DATA TRANSFER INSTRUCTIONS					
MOV	Rd, Rr	Move Between Registers	Rd ← Rr	None	1
MOVW	Rd, Rr	Copy Register Word	Rd+1:Rd ← Rr+1:Rr	None	1
LDI	Rd, K	Load Immediate	Rd ← K	None	1
LD	Rd, X	Load Indirect	Rd ← (X)	None	2
LD	Rd, X+	Load Indirect and Post-Inc.	Rd ← (X), X ← X + 1	None	2
LD	Rd, - X	Load Indirect and Pre-Dec.	X ← X - 1, Rd ← (X)	None	2
LD	Rd, Y	Load Indirect	Rd ← (Y)	None	2
LD	Rd, Y+	Load Indirect and Post-Inc.	Rd ← (Y), Y ← Y + 1	None	2
LD	Rd, - Y	Load Indirect and Pre-Dec.	Y ← Y - 1, Rd ← (Y)	None	2
LDD	Rd,Y+q	Load Indirect with Displacement	Rd ← (Y + q)	None	2
LD	Rd, Z	Load Indirect	Rd ← (Z)	None	2
LD	Rd, Z+	Load Indirect and Post-Inc.	Rd ← (Z), Z ← Z+1	None	2
LD	Rd, -Z	Load Indirect and Pre-Dec.	Z ← Z – 1, Rd ← (Z)	None	2
LDD	Rd, Z+q	Load Indirect with Displacement	Rd ← (Z + q)	None	2
LDS	Rd, k	Load Direct from SRAM	Rd ← (k)	None	2
ST	X, Rr	Store Indirect	(X) ← Rr	None	2

Mnemonics	Operands	Description	Operation	Flags	#Clocks
ST	X+, Rr	Store Indirect and Post-Inc.	(X) ← Rr, X ← X + 1	None	2
ST	- X, Rr	Store Indirect and Pre-Dec.	X ← X - 1, (X) ← Rr	None	2
ST	Y, Rr	Store Indirect	(Y) ← Rr	None	2
ST	Y+, Rr	Store Indirect and Post-Inc.	(Y) ← Rr, Y ← Y + 1	None	2
ST	- Y, Rr	Store Indirect and Pre-Dec.	Y ← Y - 1, (Y) ← Rr	None	2
STD	Y+q,Rr	Store Indirect with Displacement	(Y + q) ← Rr	None	2
ST	Z, Rr	Store Indirect	(Z) ← Rr	None	2
ST	Z+, Rr	Store Indirect and Post-Inc.	(Z) ← Rr, Z ← Z + 1	None	2
ST	-Z, Rr	Store Indirect and Pre-Dec.	Z ← Z – 1, (Z) ← Rr	None	2
STD	Z+q,Rr	Store Indirect with Displacement	(Z + q) ← Rr	None	2
STS	k, Rr	Store Direct to SRAM	(k) ← Rr	None	2
LPM		Load Program Memory	R0 ← (Z)	None	3
LPM	Rd, Z	Load Program Memory	Rd ← (Z)	None	3
LPM	Rd, Z+	Load Program Memory and Post-Inc	Rd ← (Z), Z ← Z+1	None	3
SPM		Store Program Memory	(z) ← R1:R0	None	
IN	Rd, P	In Port	Rd ← P	None	1
OUT	P, Rr	Out Port	P ← Rr	None	1
PUSH	Rr	Push Register on Stack	STACK ← Rr	None	2
POP	Rd	Pop Register from Stack	Rd ← STACK	None	2
MCU CONTROL INSTRUCTIONS					
NOP		No Operation		None	1
SLEEP		Sleep (see specific descr. for Sleep function)		None	1
WDR		Watchdog Reset (see specific descr. for WDR/Timer)		None	1
BREAK		Break For On-chip Debug Only		None	N/A