KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS
PROGRAMŲ INŽINERIJOS KATEDRA

Mindaugas Chaladauskas

GSM LPC komponento realizavimas ir tyrimas

Magistro darbas

Darbo vadovas:
prof. Vacius Jusas

Kaunas, 2010

KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS
PROGRAMŲ INŽINERIJOS KATEDRA

Mindaugas Chaladauskas

GSM LPC komponento realizavimas ir tyrimas

Magistro darbas

Vadovas
prof. Vacius Jusas
2010-05-

Recenzentas
doc. Antanas Lenkevičius
2010-05-

Atliko
IFM-4/5 gr. stud.
Mindaugas Chaladauskas
2010-05-20

Kaunas, 2010

SUMMARY

GSM LPC component implementation and testing

Everyone who develops hardware wants to do this as fast as possible and for low costs. Time to the market must be shortened because the competition is very substantial. This can be done by using special development software called high level synthesis tools. HLS tools automatically generate HDL RTL code and helps developers to get all possible architectures of project. HLS tools use an algorithmic C code as input information. There is the possibility for software engineers to develop hardware too.

It seems that HLS is very good technology, but HLS tools are not widely used today. It must be found the reasons of this problem and opportunities how this problem can be solved.

An experiment is made by solving this problem. A GSM LPC algorithm is written by hand from C description to VHDL RTL. This experiment explains problems of high level synthesis.

With the purpose HLS tools to use widely, high level of abstraction (C/C++) code must be written with restrictions. There must be no pointer variables and pointer arithmetic, no recursion, no difficult operations, no dynamic memory allocation.

Engineers have to think like hardware.

TURINYS

1. ĮVADAS

Projektuojant aparatūrinę įrangą, siekiama, kad projektavimo procesas būtų perkeltas į aukštesnį abstrakcijos lygmenį, atsirastų galimybė naudoti įprastas ir daugeliui puikiai žinomas programavimo kalbas (C/C++), būtų sumažinti projektavimo kaštai. Šiame kontekste įvedama aukšto lygio sintezės sąvoka (*angl. HLS – high level synthesis*). Ši sintezės technologija yra intensyviai tyrinėjama ir yra sukurta keletas nekomercinės ir komercinės paskirties įrankių, tačiau pramonėje jie nėra plačiai naudojami. Čia galimos įvairios priežastys: technologijos neišbaigtumas, didelė kaina, nepasitikėjimas programine įranga.

Šiame magistro darbe, norint atrasti atsakymus į keliamus klausimus, atliekamas aukšto lygio sintezės tyrimas, turimą funkciją perkeliant į kitą abstrakcijos lygmenį. Bendru atveju, aukšto lygio sintezė atliekama, funkciją, realizuotą programinės įrangos aprašymo kalba (pvz. C), perrašant į aparatūros aprašymo kalbą (pvz. VHDL, Verilog). Perrašant funkciją, susiduriama su įvairiomis problemomis. Skirtingai nuo PĮ aprašymo kalbų, AĮ aprašymo kalbose naudojami lygiagretaus priskyrimo sakiniai, specifiniai atributai (signalai), konstrukcijos. Aparatūroje naudojami sinchronizacijos signalai. C/C++ kalba užrašytos operacijos vykdomos nuosekliai, o aparatūroje signalai pasiskirsto lygiagrečiai ir vienu metu galimas kelių veiksmų atlikimas. Norint aparatūroje realizuoti nuoseklų operacijų (skaičiavimų) vykdymą, reikia suformuoti atitinkamas konstrukcijas, užtikrinančias nuoseklumą (pvz. naudoti baigtinius automatus). Programinėje įrangoje naudojamas dinaminis atminties rezervavimas, rodyklės tipo kintamieji, sudėtingos aritmetinės operacijos, tuo tarpu projektuojant aparatūrą, reikia mastyti aparatūriškai t.y mastyti kaip ji veikia ir kokiais principais atliekamos įvairios operacijos. Paprasčiausią daugybos operaciją aparatūroje tenka apsirašinėti kaip atskirą daugybos komponentą. Tai tik pabrėžia, jog aparatūrinės įrangos projektavimas yra žemesnio abstrakcijos lygmens nei programinės įrangos.

AĮ aprašas, atitinkantis apibrėžtus reikalavimus, naudojamas loginei sintezė į FPGA bei ACIS projektus, nes loginė sintezė – vienas iš svarbiausių lusto loginio projektavimo etapų, be kurio neįmanomas tolimesnis fizinis realizavimas. Pagrindinis AĮ aprašo reikalavimas – jis turi būti sintezuojamas, negali būti naudojamos nesintezuojamos konstrukcijos bei operacijos kaip pvz. VHDL kalboje sra, sla, ror, srl, sll. Paminėtos postūmio operacijos modeliuojamos, tačiau nesintezuojamos, norint realizuoti šias operacijas, ieškoma kitų būdų kaip masyvų fragmentų sujungimas ir t.t. Taigi loginei sintezei atlikti naudojami sintezuojami AĮ RTL (*angl. register*

transfer level) aprašai. Šie aprašai sintezės programose naudojami kaip įvesties informacija, iš kurios generuojamas ventilių lygio (*angl. gate-level netlist*) aprašas.

Loginės sintezės atlikimui C aprašas pirmiausia perrašomas į VHDL, Verilog arba SystemC RTL. Konvertavimas atliekamas rankiniu būdu arba naudojant aukšto lygio sintezės HLS įrankius.

Šiame darbe kaip aukšto abstrakcijos lygio funkcija parinktas C kalba realizuotas GSM LPC komponentas. GSM LPC – tai žemo lygmens kalbos (informacijos) suspaudimo algoritmo (GSM 06.10 RPE-LTP aut. *Jutta Degener, Carsten Bormann, Berlyno Technikos Universitetas, 1994*) komponentas. Tyrimo objektas – tiesinis nuspėjimo šifratorius LPC (*angl. linear predictive coder*), kurio aprašas pritaikytas CHStone (testavimo programos praktinei C aukšto lygio sintezei) testų programoms.

Šio darbo tikslas atlikti aukšto lygio sintezės tyrimą, išanalizuoti galimus HLS įrankius bei problemas, kylančias sintezės proceso metu. Atlikti eksperimentą – aukšto abstrakcijos lygio funkciją (LPC komponentas), aprašytą C kalba, kuris paimtas iš HLS sintezės eksperimentinių programų, perrašyti rankiniu būdu į VHDL. Gautą aprašą sintezuoti ir atlikti verifikavimą.

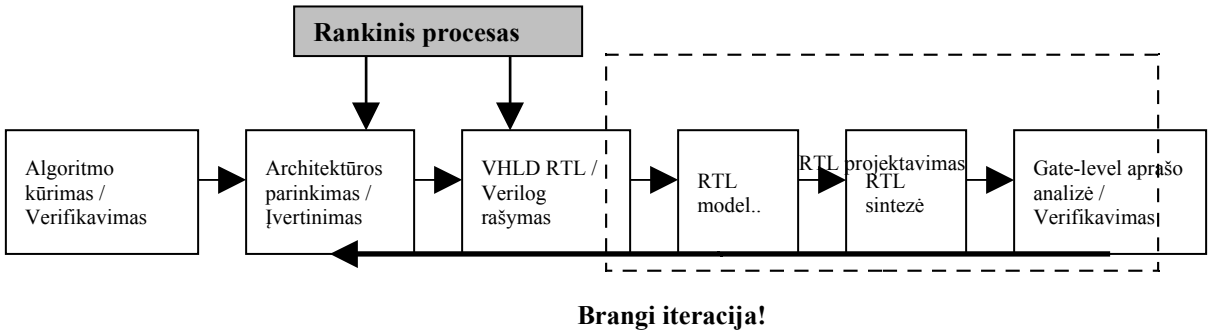
2. AUKŠTO LYGIO SINTEZĖ (HLS)

Šio darbo tikslas – aukšto lygio sintezės tyrimas, susidedantis iš įrankių analizės, problemų įvertinimo, bei praktinio eksperimento atlikimo. Šiame skyriuje paaiškinamos aukšto lygio sintezės sąvokos, aptariami nekomerciniai ir komerciniai įrankiai bei sintezės proceso problemos.

2.1 Aukšto lygio sintezė

Aukšto lygio sintezė (*angl. High-level synthesis HLS*) dar vadinama C sintezė, ESL sintezė, algoritminė sintezė arba elgsenos sintezė. Tai automatinis projektavimo procesas, kurio metu interpretuojamas duotos elgsenos algoritminis aprašas ir sukuriama aparatūrinė įranga atitinkanti duotąją elgseną. HLS sintezė prasideda nuo ANSI C/C++ bei SystemC aprašo, kuris analizuojamas, sudėliojama architektūra, atliekamas planavimas sukuriant RTL lygio aparatūrinės įrangos aprašą, kuris vėliau, atliekant loginę sintezę, sintezuojamas į elementinį (ventilių) lygį (gate level). HLS tikslas - suteikti galimybę aparatūrinės įrangos inžinieriams efektyviai kurti projektus bei atlikti jų verifikavimą, projektavimo procesą pakeliant į aukštesnį abstrakcijos lygį.

Žemiau pateikiama įprasta RTL projektavimo schema:



2.1.1 pav. Tradicinis RTL projektavimo procesas [16]

Pirmasis proceso žingsnis – algoritmo, kuris turi būti įgyvendintas, kūrimas. Kai kuriuose pramonės srityse – tai labai formalus procesas, kuriame dirba skirtingos projektavimo ir realizavimo komandos. Tai dažniausiai pasitaikantis atvejis, kai įmonė atlieka naujus projektavimo tyrimus, o standartai dažnai keičiasi. Kitose pramonės šakose, kur standartai

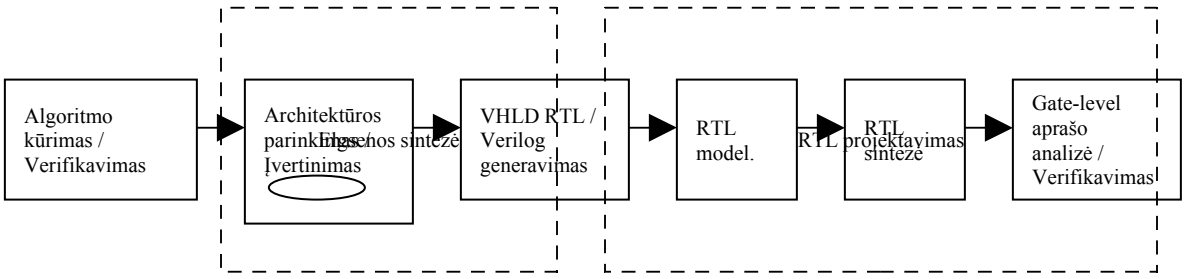
patvirtinti, projektavimo komanda gali realizuoti aiškiai apibrėžtus algoritmus ir šiuo atveju nereikalinga dar viena tyrimų komanda.

Sukurtam bei patikrintam algoritmui turi būti parinkta architektūra bei aprašyta RTL lygiu. RTL lygio aprašas – tai tipinis originalaus elgsenos aprašo sudalinimas ir tai yra ilgas rankinis procesas, reikalaujantis daug laiko bei pastangų visame projektavimo cikle. Architektūros parinkimas – kritinis žingsnis projektavimo procese, tačiau dažnai jam neskiriama pakankamai dėmesio. Tai nėra projektavimo komandos kaltė, tai didelių rezultatų siekimo per trumpą laiką, konkurencijos, pasekmė. Atsiranda projektavimo įrankių, leidžiančių projektuotojui greitai įvertinti įvairias architektūrų alternatyvas, poreikis. Nenaudojant automatinių projektavimo įrankių, atsiranda prielaida, kad kol viena architektūra bus pilnai realizuota, atsiras poreikis ieškoti kitos alternatyvios architektūros.

Algoritmo architektūrų alternatyva - tai užimamo ploto ir laiko atlikti skaičiavimams ryšys. Jei sritis didelė, skaičiavimai greiti, jei sritis maža – lėti. Galioja atvirkščias proporcingumas.

Po RTL suformavimo bei patikrinimo atliekama RTL sintezė bei tikrinamas ventilių lygio rezultatas. Jeigu šiame etape projektas neatitinka specifikacijos bei projektuotojas negali atlikti sintezės, architektūros fragmentai (porcijos) turi būti grąžinti perdarymui, kad būtų išspręsta problema ar problemos. Tokia iteracija gali būti labai brangi. Tokios problemos gali atitolinti planuojamą projekto užbaigimo laiką, kas reikštų didelius nuostolius projektams, kurie yra jautrūs konkurencijai.

Žemiau (žr. 2.1.2 pav.) pateikiamas modifikuotas tradicinis RTL projektavimo procesas, kuriame naudojama elgsenos sintezė.



2.1.2 pav. Projektavimo procesas naudojant elgsenos sintezę [16]

Elgsenos sintezė leidžia projektuotojams anksti ir greitai įvertinti daugelį skirtingų architektūrų, nelaukiant, kol bus atliktas ventilių lygio aprašo verifikavimas.

Užbaigus algoritmą, pati elgsenos specifikacija panaudojama kaip įvesties informacija elgsenos sintezės įrankiuose. Varijuojama projektavimo konstrukcijomis kaip signalo periodas, įvesties / išvesties laikas, informacijos perdavimo elementų tipas bei skaičius, bei apibrėžtas taktų skaičius. Projektuotojas gali greitai įvertinti kelias architektūras. Elgsenos sintezės įrankis, sukurdamas architektūrą, tenkinančią specifikacijas, automatiškai sugeneruoja ir RTL aprašą. Elgsenos sintezė suteikia produktyvumo.

Tradiciniame RTL projektavimo procese projektuotojas turi sukurti elgsenos modelį, kad galėtų patikrinti algoritmo funkciją. Norint įvertinti visas architektūras, reikia parašyti RTL aprašą ir atlikti to aprašo patikrinimą. RTL sintezės etapai turi būti stebimi, kad būtų galima įsitikinti, jog architektūra sėkminga. Jei architektūra nesėkminga, turi būti sukurtas naujas RTL modelis ir jis patikrinamas. Tai gali įtakoti ilgą ir nenuspėjamą procesą. Priešingai nei RTL projektavimo procese, elgsenos projektavimo procese yra tiesioginis ryšys tarp elgsenos aprašo ir RTL aprašo, kuris yra sintezuojamas. Elgsenos sintezės procese daugelis architektūrų yra greitai įvertinamos. Kai tik įrankis aptinka teisingą architektūra, iškart automatiškai sugeneruoja RTL aprašą. Toks glaudus elgsenos specifikacijos ir RTL rezultato ryšys sumažina arba visai panaikina tikimybę, jog ventilių lygio rezultatą gali tekti generuoti pakartotinai.

Elgsenos sintezės atveju parinkti kitą architektūrą dėl neatitikimų, esančių gate level apraše, yra paprasta. Alternatyvi architektūra gali būti sukurta be pradinio elgsenos aprašo koregavimo. Pakanka pakeisti elgsenos aprašo apribojimus. RTL projektavimo procese architektūros pakeitimams atlikti reikia kurti naują RTL aprašą arba koreguoti esamą.

Toliau pateikiamas paprasto algoritmo paaiškinimas.

Paprastas algoritmas. Dėl istorinių bei projektavimo priežasčių daugelis algoritmų aprašyti C arba panašiomis kalbomis. Žemiau pateikiamas C aprašo fragmentas.

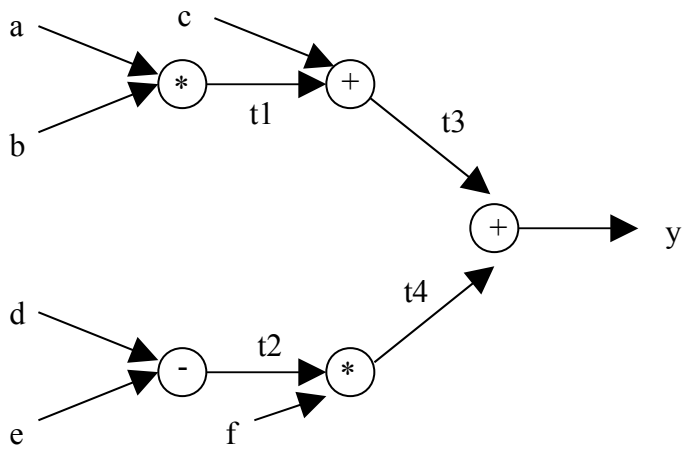
```
int design( int a, int b, int c, int d, int e, int f)
{
        int y;
        y = a * b + c + (d – e ) * f;
        return y;
}
```

Pateiktas algoritmas gali būti užrašomas VHDL kalba (tada bus galima loginė sintezė). Funkcijos sąsaja gali būti užrašoma kaip ENTITY, o pati funkcija užrašoma kaip ARCHITECTURE.

```
ENTITY design IS
        PORT(a, b, c, d, e, f : IN integer;
             y                : OUT integer );
END design;

ARCHITECTURE behavioral OF design IS
BEGIN
        PROCESS(a, b, c, d, e, f)
        BEGIN
             y <= a * b + c + (d - e) * f;
        END PROCESS;
END behavioral;
```

Duomenų priklausomybės. Norint projekte suvokti priklausomybę tarp duomenų (a, b, c, d, e, f, y) bei operatorių (*, +, -), projektą reikia pateikti duomenų srautų grafu (angl. data-flow graph DFG). Remiantis operatoriais, apibrėžtais VHDL kalboje, pavyzdinis paprasto algoritmo projektas pateikiamas DFG grafu. Tarpinės reikšmės pavadintos (t1-t4).



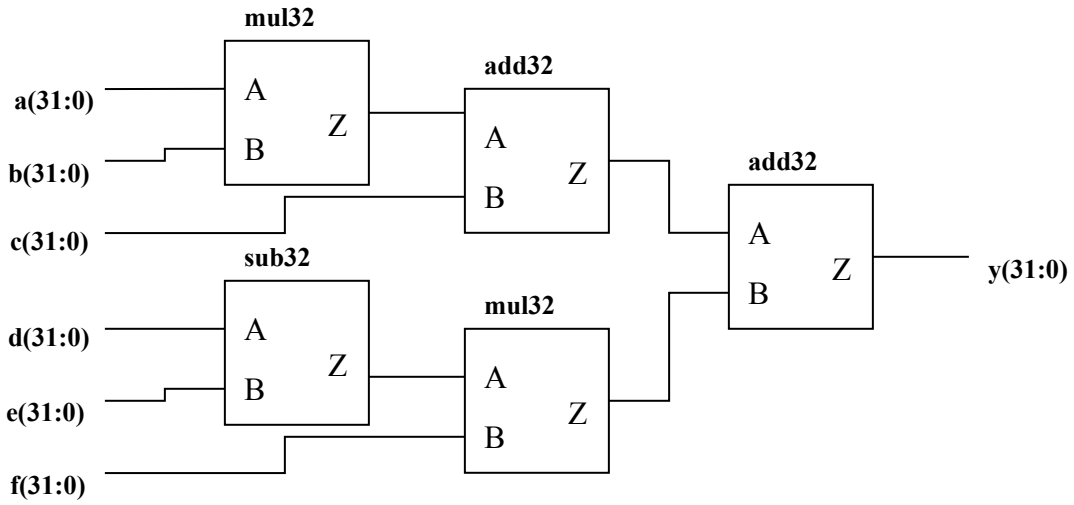
2.1.3 pav. Duomenų srautų DFG grafas

Kiekvienas grafo mazgas vaizduoja operaciją (daugyba ar sudėtį). Du mazgai sujungti tarpusavyje jei tarp jų yra duomenų priklausomybė. Duomenų priklausomybės nurodo

eiliškumą, pagal kurį, turi būti suskaičiuojamos reikšmės. Pavyzdžiui a * b sandauga turi būti apskaičiuota prieš t1 + c.

DFG grafas vaizduoja tik duomenų priklausomybę.

RTL sintezės rezultatas. RTL sintezės įrankiams PROCESS dalis gali būti kombinacinė arba sinchroninė. Procesas, aprašantis paprastą algoritmą, yra kombinacinis. Kombinacinei šio proceso realizacijai reikalingi du daugintuvai, du sumatoriai bei vienas atimties įtaisas. Kadangi sąsajos signalai pateikti kaip neapibrėžti sveikieji skaičiai (VHDL apraše), jie bus verčiami į 32 bitų magistrales. Žemiau pateikiamas sintezės rezultatas, gautas RTL sintezės įrankiu.



2.1.4 pav. Kombinacinė realizacija

Pagal pateiktą realizaciją, keliami reikalavimai užimamai sričiai:

area(total) = 2 * area(mul32) + 2 * area(add32) + area(sub32)

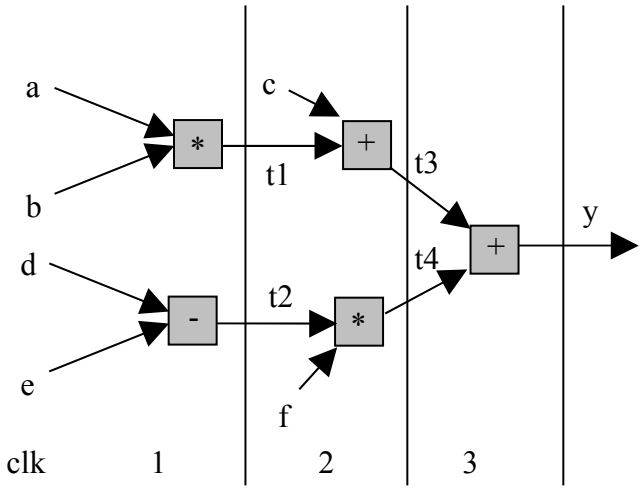
Kritinis kelias (jei visi įvesties signalai atsiranda vienu metu ir jei atimties įtaiso vėlimas didesnis nei sumatoriaus):

delay(total) = area(sub32) + area(mul32) + area(add32)

Resursų paskirstymas. Jeigu pateiktas sintezės rezultatas – pati greičiausia realizacija (veiksmai vykdomi maksimaliai lygiagrečiai), dar nereiškia, jog projektas atitinka keliamus reikalavimus. Didelio pločio daugintuvai (šiuo atveju 32 bitai) užima didelę sritį. Projektuotojams gali tekti rinktis alternatyvią architektūrą, kuri veikia lėčiau, tačiau užima mažesnę sritį. Šiame paprastame projekte naudojamos dvi daugybos ir dvi sumos operacijos.

Siekiama, kad architektūroje būtų naudojama po vieną kiekvienos operacijos įtaisą. Tokia struktūra su apribotais resursais gali ženkliai skirtis nuo kombinacinės realizacijos. Jei architektūroje naudojamas vienas daugintuvas, vadinasi daugybos operacijos atliekamos skirtingais laiko momentais. Užimamas plotas (sritis) sumažinamas prarandant lygiagretumą (greitaveiką). Daugintuvo įvesties duomenys turi būti paskirstomi, o daugybos rezultatas turi būti saugomas, kol bus apskaičiuojamas naujas rezultatas. Jeigu projektas naudoja ribotus resursus, tokių resursų naudojimas turi būti planuojamas skirtingais laiko intervalais. Jei skaičiavimai atliekami per kelis laiko taktus, toks projektas beveik visada turės baigtinį automatą. Jei skaičiavimai atliekami per vieną laiko taktą, o rezultatas turi būti išsaugotas, kad jį būtų galima naudoti kitame laiko takte, tokiame projekte turi būti naudojama atmintis (arba registrai).

Planavimas. Projekto realizavimas pradedamas DFG grafu, nustatant kada (kokiais laiko taktais) atliekamos reikiamos operacijos. Supaprastinant pavyzdį, laikoma, kad kiekviena operacija atliekama komponentų, kuriems reikalingas atskiras laiko taktas, per kurį apskaičiuojamas rezultatas. Planavimo pavyzdys gali būti modifikuotas DFG grafas.



2.1.5 pav. Modifikuotas DFG grafas su planavimo informacija

Modifikuotame DFG grafe (planavimo grafe) pateikiamas laiko paskirstymas operacijų atlikimui. Matoma jog pavyzdiniam paprastam algoritmui realizuoti reikalingi trys laiko taktai.

Valdymo logika / Baigtinis automatas. Paminėtiems 3 laiko taktams suvaldyti reikalingas baigtinis automatas. Apibrėžiamos 3 automato būsenos: s1, s2, s3, bei būsenų kitimo logika.

Atmintis. Paprasto algoritmo įtaiso architektūrai reikalingas tarpinių reikšmių saugojimas. Reikšmės gali būti saugomos RAM arba registruose. Nustatant, kurios reikšmės yra tarpinės ir kurias reikia saugoti, atliekama analizė. Analizės metu tikrinama kaip gali būti paskirstomi ir kada nuskaitomi įvesties signalai (nuskaitomi pirmame takte ir saugomi, kad būtų panaudoti antrame ar nuskaitomi tik antrame takte). Reikšmės, gaunamos kaip įvairių funkcijų rezultatai, viename takte turi būti išsaugomos, kad būtų pasiekiamos kitame takte (jeigu jos reikalingos). Žvelgiant į planavimo (modifikuotą DFG) grafą, nustatoma, kurios reikšmės turi būti saugomos. Saugomos tos reikšmės, kurios (kaip rodykles) kerta ribą tarp vieno laiko takto ir kito.

Pateiktas paprasto algoritmo pavyzdys parodo, jog galimos įvairios architektūrų realizacijos. Rankiniu būdų išanalizuoti visus galimus variantus ir juos realizuoti, kad būtų galimybė juos įvertinti, ilgas ir sudėtingas procesas. Elgsenos sintezės įrankiai tai atlieka automatiškai, pakanka nurodyti apribojimus: takto periodą, taktų skaičių, duomenų tipus, elementų skaičių).

HLS – tai intensyviai tyrinėjama sritis. Yra sukurta tiek pramonei tiek akademiniams tikslams skirtų įrankių. Pramonei skirtus įrankius kuria: Bluespec, Mentor Graphics, Forte Design Systems bei kiti. Akademinių įrankių pavyzdžiai: GAUT, Spark.

Mokslininkų S.Ahuja, S. K. Shukla, S. T. Gurumani, C. Spackman straipsnyje teigiama, jog efektyvių, gerai ištestuotų programinės įrangos algoritmų atkartojimas aparatūrinės įrangos generavime, gali ženkliai pagreitinti produkto atėjimą į rinką. Atkartojimas projektuotojams leidžia sutaupyti verifikavimui skirto laiko. Mažesniam kiekiui komponentų atliekamas RTL modeliavimas, kuris paprastai yra lėtas procesas.

Atliekant aukšto lygio sintezę, turi būti sprendžiamos svarbios problemos. Visų pirmą C bei C++ kalbose veiksmai vykdomi nuosekliai, o aparatūrinėje įrangoje veiksmai vykdomi (perduodami signalai) lygiagrečiai. Ši savybė reikalauja ieškoti nuoseklaus aprašo išlygiagretinimo galimybių, reikalingos įvairios alternatyvos. Priklausomai nuo vėlinimo, užimamo ploto, galios reikalavimų, yra įvairūs būdai realizuoti lygiagretumą.

Po sintezės turi būti patikrinama, ar gauta HDL RTL realizacija atitinka pradinę C funkciją. Generuojant Verilog ar VHDL RTL iš C aprašo, ieškomas optimalus sintezės sprendimas.

Mokslininkai: S.Ahuja, S. K. Shukla, S. T. Gurumani, C. Spackman tirdami aukšto lygio sintezę, remiasi C2R metodologija. Teigiama, jog daugeliu atveju sudėtingų C kalbos algoritmų rankinis transliavimas į RTL nėra praktinė alternatyva. Sudėtingiems algoritmams siūloma naudoti C2R transliatorių.

Problemos, kurios turi būti sprendžiamos aukšto lygio sintezės programose, vienodai aktualios rankiniam Verilog ar VHDL RTL aprašo formavimui.

2.2 Tipinis HLS procesas

Aukšto lygio sintezės procesą sudaro 6 pagrindiniai etapai:

1. Elgsenos specifikacijos:
   - Kokią kalbą naudoti? Procedūrines kalbas, funkcines kalbas, grafines notacijas.
   - Ar apibrėžtas lygiagretumas?
2. Duomenų perdavimo (dataflow) analizė:
   - Lygiagretumo formavimas
   - Aukšto lygio kalbų konstrukcijų pašalinimas
   - Ciklų išvyniojimas
   - Programos transformacijos
   - Pasikartojančių išraiškų aptikimas
3. Veiksmų (operacijų) planavimas:
   - Našumo/kainos kompromisas
   - Našumo matavimas
   - Sinchronizacijos strategija
4. Duomenų kelių paskirstymas:
   - Operatorių parinkimas
   - Registrų/atminties rezervavimas
   - Sujungimų generavimas
   - Aparatūrinis minimizavimas
5. Kontrolės pasirinkimas:
   - Valdymo stiliaus pasirinkimas (PLA, mikrokodas, atsitiktinė logika)

- Valdymo įtaisų generavimas

6. Modulių sujungimas bei valdymo įtaisų realizavimas:
   - Fizinių modulių parinkimas
   - Modulių parametrų bei konstrukcijų specifikavimas
   - Valdymo įtaisų realizavimas

Pagrindinės problemos:

- Planavimas – kiekvienai operacijai priskiriamas laiko intervalas atitinkantis sinchronizacijos signalo periodą.
- Resursų priskyrimas – aparatūrinės įrangos komponentų bei jų skaičiaus, kuris bus naudojamas galutinėje realizacijoje, parinkimas
- Modulių sujungimas – operacijų priskyrimas parinktiems aparatinės įrangos komponentams.
- Įvesties kalbos transliavimas į vidines konstrukcijas.
- Lygiagretumo formavimas - atliekant duomenų mainų analizę, ieškoma lygiagretumo galimybių.
- Operacijų dekompozicija – sudėtingų operacijų aprašymas elgsenos lygmenyje.

HLS planavimo problemos:

- Resursais apribotas planavimas (*angl. RC resource-constraints*).
- Laiko apribotas planavimas (*angl. TC time-constrained*)

RC planavimo metodika:

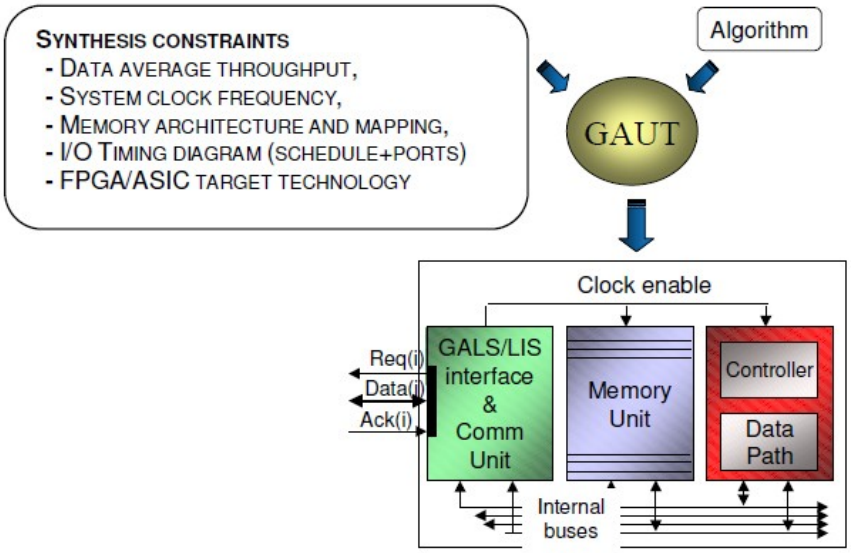
- ASAP: kaip įmanoma greičiau (*angl. ASAP as soon as possible*)
  - Topologiškai surūšiuojamos operacijos pagal duomenų mainus
  - Operacijų planavimas atliekamas pagal surūšiuotą tvarką, pradedant nuo anksčiausių žingsnių.
- ALAP: kiek įmanoma vėliau (*angl. ALAP as late as possible*)
  - Topologiškai surūšiuojamos operacijos.
  - Operacijų planavimas atliekamas pagal surūšiuotą tvarką, pradedant nuo vėliausių žingsnių.

HLS įrankiai turi efektyviai spręsti minėtas problemas visuose sintezės etapuose.

2.3 HLS įrankiai

GAUT

GAUT – vienas iš akademiniams tikslams skirtų aukšto lygio sintezės įrankių. Sintezė pirmiausia pradedama nuo C arba C++ aprašo suformuojant galimą išlygiagretinimą, vėliau formuojant apjungimą, paskirstymą srityje, planavimą. Naudojamos dvi būtinos sintezės konstrukcijos: našumas bei sinchrosignalo periodas. GAUT, generuojant VHDL arba SystemC, naudojami duomenų srautų grafai, sugeneruoti GCC, bei bibliotekinės tikslo charakteristikos sričiai, greičiui. Įrankis taip pat gali generuoti SystemC TLM modelius.



1

2.3.1 pav. GAUT principinė HLS schema [9]

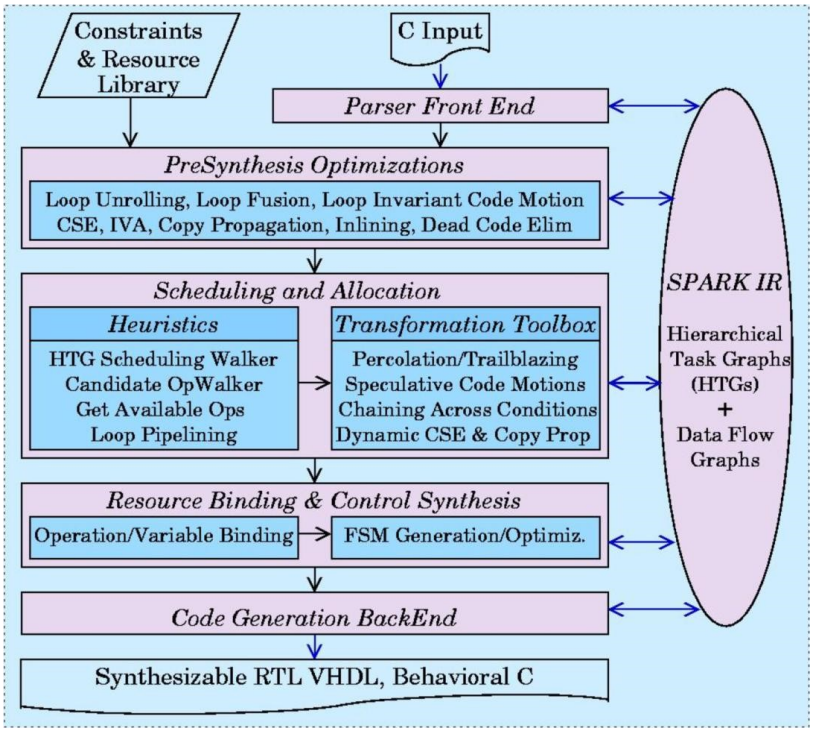
GAUT generuoja VHDL testines programas bei naudoja Modelsim grafinę sąsają. Įrankis pritaikytas Solaris, Linux, Windows operacinėms sistemoms. Sintaksę palengvinantis teksto redaktorius leidžia projektuotojams nuskaityti bei analizuoti DSP algoritmą. Analizės rezultatas – tai grafinis duomenų srautų atvaizdavimas, kuriame pateikiamas galimas aprašo lygiagretumas. Sintezės rezultatas – RTL failas.

Spark

Spark – dar vienas HLS akademinis įrankis. Sukurtas Microelectronic Embedded Systems Labaratory. Spark kaip įvesties informaciją priima ANSI C elgsenos aprašą, atlieka duomenų priklausomybės analizę, planavimą, apjungia resursus bei suformuoja RTL VHDL. Projektuotojas gali valdyti transformacijas naudodamas skriptus.

Spark nepalaiko rodyklių tipų, funkcijų rekursijos, vadinamų šuolių (*angl. control-flow jumps*).

Kaip įvesties informaciją Spark priima taip pat ir aparatūrinės įrangos bibliotekas (konstrukcijų aprašai, laikinės charakteristikos). Spark naudoja trijų lygių tarpinius atvaizdavimus: valdymo eigos grafai (*angl. control flow graphs CFGs*), duomenų kitimo grafai (*angl. data flow graphs DFGs*), hierarchiniai užduočių grafai (*ang. hierarchical task graphs HTGs*).



2.3.2 pav. Spark principinė HLS schema [10]

Spark efektyvumas patikrintas realiose projektuose kaip Intel Pentium procesoriaus instrukcijų ilgio dešifratoriuje, daugialypės terpės taikomuose uždaviniuose kaip MPEG-1, MPEG-2 bei GIMP grafikos apdorojimo įrankyje.

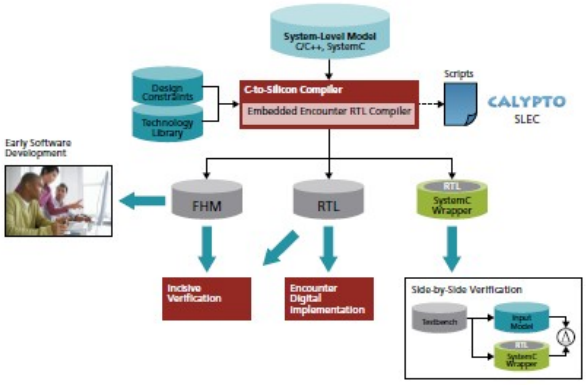
Spark projektas remiamas SRC (Semiconductor Research Corporation) bei Intel korporacijų.

C-to-Silicon Compiler

Cadence kompanijos komercinis įrankis, atliekantis C/C++/SystemC aprašo aukšto lygio sintezę. C-to-Silicon Compiler automatiškai generuoja sintezuojamą RTL iš C/C++/SystemC algoritmo aprašo. Rezultatų kokybė padidinama 90% nei rankomis rašomas RTL.

C-to-Silicon Compiler naudojant kartu su Cadence Encounter bei Cadence Incisive leidžia kurti verifikuojamus projektus. Generuodamas sparčius aparatūrinės įrangos modelius (FHM fast hardware models), leidžia kurti ankstyvą programinę įrangą.

C-to-Silicon Compiler integruotų RTL loginės sintezės įrankių pagalba iš C aprašo automatiškai generuoja kontrolės/duomenų perdavimo, laikinės analizės ataskaitas bei kitą svarbią projekto informaciją.



2.3.3 pav. C-to-Silicon Compiler sintezės proceso schema [11]

C-to-Silicon Compiler paketo savybės:

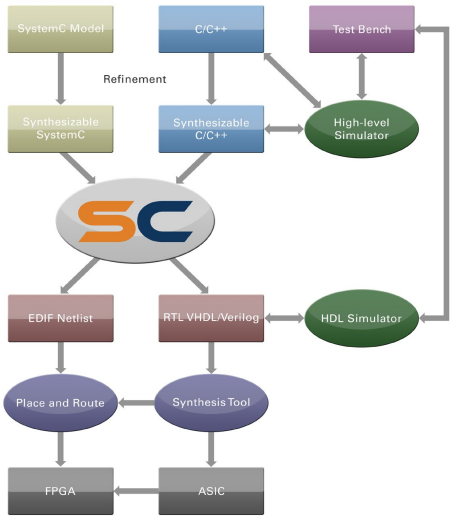
- Leidžia naudoti įvairų C/C++/SystemC programavimo stilių bei konstrukcijas, šablonus, klases, apibrėžtus tipus ir kai kuriuos rodyklių tipus.
- Automatiškai generuoja sintezuojamą IEEE-1364 Verilog bei sintezės skriptus Encounter RTL Compiler sintezei.
- Automatiškai generuoja modeliavimo skriptus.
- Patogi grafinė aplinka.
- Paruošia skriptus Calypto SLEC įrankiui.
- Palaikomos OSCI 1.0 TLM konstrukcijos.
- Integracija su XST (Xilinx Synthesis Technology)

SC Compiler Academic Edition

Tai Agility kompanijos komercinis daugiaprocesinis (multi-threaded) C, C++, SystemC aukšto lygio sintezės įrankis (akademinė versija be klaidų pataisymų). Agility SC Compiler academic edition leidžia sujungti push-button tipo projektą su RTL. SC Compiler tinka realizuoti sudėtingiems algoritmams greitam duomenų apdorojimui FPGA arba formuoti aukšto lygio įvesties informaciją ASIC projektui.

Įrankio savybės:

- Naudojami pramonėje patikrinti sprendimai:
  - Pramonėje patikrinti, efektyvūs rezultatai
- Projekto abstrakcijos lygio palaikymas
  - Pilnas klasių, šablonų, paveldėjimo, operatorių palaikymas
  - Duomenų pločio priderinimas leidžia naudoti C duomenų tipus.



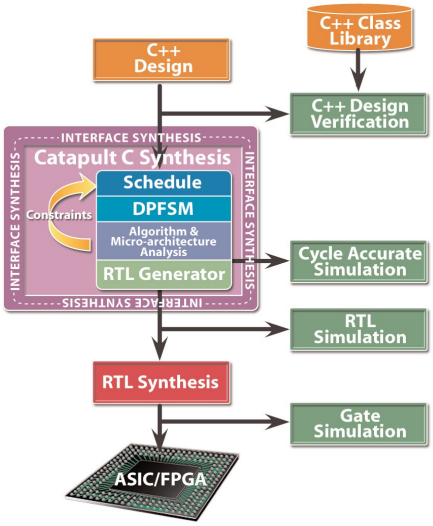
2.3.4 pav. Agility SC Compiler HLS procesas [12]

- Išlaikoma projektavimo kontrolė
  - Palaiko išorinius juodųjų dėžių komponentus (egzistuojančius IP)

- Integruota projektavimo aplinka:
  - Daugelio platformų palaikymas: Windows XP, Red Hat, Debian bei SuSE
  - Integruotos sintezės ataskaitos.
  - Integruotas valdymo bei duomenų mainų atvaizdavimo įrankis.
- Aukšto lygio sintezės rezultatai:
  - RTL VHDL (IEEE 1076.6 – 1999) ir Verilog (IEEE 1364 – 2001)
  - Optimizuotas EDIF netlist aprašas Altera bei Xilinx FPGA

Catapult C

Mentor Graphics komercinis HLS įrankis, atliekantis ANSI C/C++ aukšto lygio sintezę. C/C++ kalbos aprašuose leistini rodyklių tipo kintamieji, klasės, šablonai, operatorių perdengimas. Naudojama atkartojamo projektavimo metodologija. Įrankis turi grafinę sąsają, kurioje pateikiama projektuojamo įtaiso schema bei laiko planavimas. Palaikomas trijų tipų modeliavimas: naudojant originalius C/C++ testų rinkinius: ciklinis, RTL, ventilių (gate level).



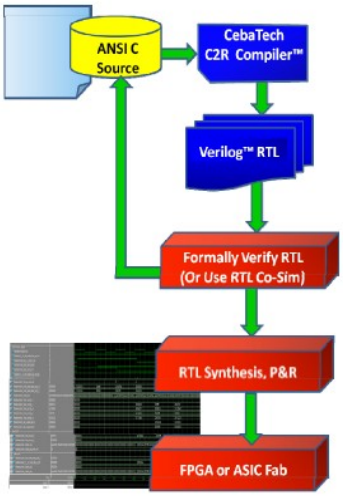
2.3.5 pav. Catapult C sintezė lusto projektavimo procese [13]

Kitos savybės:

- galios, našumo, užimamo ploto tyrimas bei optimizavimas
- viršus-apačia, apačia-viršus hierarchinio projektavimo palaikymas
- rezultatų kokybės valdymas
- integruoti analizės įrankiai kaip Ganto grafikai, kritinių kelių peržiūra, susikirtimų tikrinimas
- gamintojų sertifikuotos sintezės bibliotekos
- ASIC, FPGA technologijų palaikymas aukšto našumo aparatūrinei įrangai

C2R Compiler

CebaTech C2R Compiler – tai elgsenos bei sisteminio lygio projektavimo įrankis, kuris leidžia ANSI C naudoti kaip aparatūrinės įrangos aprašymo kalbą. Sukuriama aukšto lygio abstrakcija. C2R Compiler automatizuoja projektavimo kelia nuo C iki sintezuojamo Verilog aprašo, leidžia pagreitinti projektavimo procesą. Įrankis leidžia pagreitinti įprastus RTL metodus. C2R leidžia kurti FPGA, ASIC bei struktūrinius ASIC projektus įvairiose pramonės srityse kaip skaičiavimai, saugojimas, tinklai, plataus vartojimo elektronika.



2.3.6 pav. C2R Compiler projektavimo seka [15]

Kaip teigiama šio įrankio gamintojų aprašymuose, C kalba – tai labai galinga aparatūros aprašymo kalba, kuri yra puikiai pritaikyta vienlusčių sistemų projektavimui bet kokiu atžvilgiu. Duomenų tipai C kalboje kaip: struktūros, sąjungos, sudėtingi masyvai (su rodyklėmis lizdiniai) turi tiesioginį ryšį su aparatūrine įranga bei yra aparatūrinės įrangos abstrakcija. C2R Compiler pilnai suderintas su ANSI C kalba bei daugiagijinėmis programinės įrangos kūrimo aplinkomis.

Įrankyje palaikoma:

- Rodyklės, rodyklių aritmetika
- Fiksuoto ir slankaus kablelio aritmetika
- Funkcijų su argumentais kreipiniai taip pat rodyklėmis

- Globalūs kintamieji – būdingi trigeriams, atmintims
- Globalūs, daug kartų įrašomi bei lokalūs masyvai, struktūros, sąjungos, float, double tipai
- Ciklai, gijos

Žemiau pateikiama minėtų įrankių palyginimo lentelė:

1 lentelė

Įrankis	Paskirtis	Graf. sąsaja	Įvesties kalbos	Išvesties kalbos	Sud. aritm.
GAUT	Akademinė	+	C/C++	VHDL, SystemC	-
Spark	Akademinė	-	C	VHDL	-
C-to-Silicon Compiler	Komercinė	+	C/C++/ SystemC	Verilog	+
SC Compiler Academic Edition	Komercinė / Akademinė	+	C/C++/ SystemC	VHDL, Verilog	+
Catapult C	Komercinė	+	C/C++/ SystemC	VDL, Verilog, SystemC	+
C2R Compiler	Komercinė	+	C	Verilog	+

2.4 CHStone (aukšto lygio sintezės testinės programos)

CHStone – tai testinių programų rinkinys skirtas C kalbos aukšto lygio sintezei (autoriai Y. Hara, H. Tomiyama, S. Honda, H. Takada, K. Ishii). Programų rinkinį sudaro įvairių taikomųjų sričių programos, kai kurios iš jų priklauso kitoms testavimo programoms. CHStone sudaro 12 programų, kurios parinkto iš įvairių taikomųjų sričių, tokių kaip aritmetika, daugialypės terpės apdorojimas, saugumas bei mikroprocesoriai. CHStone programos reliatyviai didelės lyginant su plačiai seniau HLS naudotomis programomis. CHStone programos parašytos standartine C kalba, bei gali būti transliuojamos bei vykdomos kiekvieno kompiuteryje. Visos CHStone programos patvirtintos jog yra sintezuojamos HLS įrankių. CHStone lengva naudoti, nes testiniai vektoriai yra įtraukti į pačią programą ir nereikalingos jokios išorinės bibliotekos.

Testinių programų sąrašas:

2 lentelė

Programa	Projekto aprašymas	Šaltinis
DFADD	Dvigubo tiklsumo slankaus kablelio sudėtis	SoftFloat
DFMUL	Dvigubo tikslumo slankaus kablelio daugyba	SoftFloat
DFDIV	Dvigubo tikslumo slankaus kablelio dalyba	SoftFloat
DFSIN	Sin funkcijos dvigubo tikslumo slankaus kablelio skaičiams	CHStone group, SoftFloat
MIPS	Supaprastintas MIPS procesorius	CHStone group
ADPCM	ADPCM dešifratorius ir šifratorius	SNU
GSM	**GSM tiesinio nuspėjimo kodavimo analizė**	**MediaBench**
JPEG	JPEG paveiksliukų dekompresija	PVR group, CHStone group
MOTION	Judesio vektorių dešifravimas iš MPEG-2	MediaBench
AES	Pažangus šifravimo standartas	AILab
BLOWFISH	Duomenų šifravimo standartas	MiBench
SHA	Apsaugos hash algoritmas	MiBench

Tarp minėtų 12 programų yra ir šiame darbe projektuojamo komponento GSM LPC testų programa.

2.5 Išvados

- Aukšto lygio sintezė (*angl. High-level synthesis*), tai sintezė iš ANSI C aprašo dar vadinama elgsenos sinteze, algoritmine sinteze. Pagrindinė HLS savybė – tai automatizuotas sintezuojamo RTL aprašo generavimas.
- HLS įrankius kuria bei šioje tyrimų srityje dirba didelės korporacijos kaip Forte Design Systems, Mentor Graphics, Cadence.
- Atlikus HLS įrankių palyginimą, pastebėta, jog kiekvienas iš jų turi savų privalumų bei trūkumų. Jei įrankis nėra komercinis, jis negali sintezuoti sudėtingų algoritmų. Komerciniai įrankiai gana brangūs ir juos įsigyti gali tik didžiosios korporacijos.
- Aukšto lygio sintezė skirta projekto fragmentų realizavimui, tačiau ne visam projektui. Ši sintezė labiausiai tinkama patikrintiems atkartojamiems C algoritmams (vaizdo, garso suspaudimo bei kiti) sintezuoti į aparatūrinės įrangos RTL.

3. GSM KALBOS KODAVIMAS

Šiame darbe, sprendžiant HLS problemas, projektuojamas GSM LPC komponentas. Šis komponentas naudojamas GSM 06.10 šifratoriaus apraše. Šiame skyriuje bus trumpai aptartas konkretus GSM 06.10 kalbos kodavimo standartas.

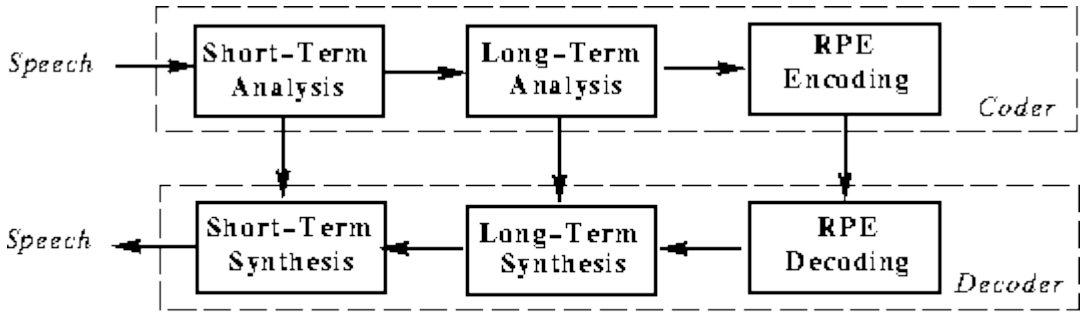
3.1 GSM 06.10 kalbos kodavimo standartas

Mobiliųjų telefonų industrijoje per jos gyvavimo metus yra sukurta įvairių standartizuotų kalbos šifratorių, naudojamų mobiliuosiuose įrengimuose. Daugiausia jų sukurta ETSI (*angl. Europian Telecommunications Standard Institute*) kaip GSM standartai ir yra plėtojami pasaulio mastu (išimtis JAV). Pirmieji šifratoriai buvo ribotų galimybių dėl mobiliųjų įrenginių trumpos baterijos veikimo trukmės bei labai riboto skaitmeninių kanalų kiekio ore. Vėliau padidėjus mobiliųjų įrenginių signalų apdorojimo galimybėms, išaugo ir kalbos kokybė.

GSM-FR arba GSM 06.10 – tai pirmasis skaitmeninio kalbos kodavimo standartas naudojamas GSM. Vidutinis šifratoriaus perduodamų bitų kiekis 13 kbit/s. Šifruotos kalbos kokybė, lyginant su moderniais standartais, yra gana prasta, tačiau kūrimo laiku (1990-tais) buvo geras skaičiavimo sudėtingumo ir kokybės derinys. Šis kodavimo standartas vis dar naudojamas pasauliniuose tinkluose.

GSM-FR specifikuotas ETSI 06.10 (ETS 300 961) remiasi PRE-LTP (*angl.* Regular Pulse Excitation – Long Term Prediction) reguliaraus energijos sužadinimo – ilgalaikės prognozės kalbos kodavimo paradigma. Kaip ir daugelis kitų kalbos šifratorių, tiesinis nuspėjimas naudojamas sintezės filtruose. Šifratorius kaip įvesties informaciją priima 13 bitų vienodus PCM (pulse-code modulation) signalus iš PSTN (viešojo telefoninio ryšio tinklo). Užkoduota kalba šifratoriaus išėjime perduodama kanalų šifravimo įrenginiui, specifikuotam GSM 05.03.

Testavimo dažnis 8000 pavyzdžių/s. Teorinis minimalus perdavimo šifravimo vėlinimas gali būti 20 ms. Reikalavimas, kad vėlinimas būtų mažesnis nei 30 ms. Perdavimo šifratoriaus vėlinimas apibrėžiamas kaip laiko intervalas tarp 160 testinių signalų priėmimo šifratoriaus įėjime ir 160 rekonstruotų testinių signalų, atsirandančių dešifratoriaus išėjime, dirbant 8 kHz testavimo dažniu.



3.1.1 pav. GSM 06.10 šifravimo blokinė diagrama

3.1.1 GSM 06.10 FR šifratorius

GSM 06.10 FR šifratoriaus (*angl. encoder*) įvesties kalbos sritis susideda iš 160 testinių signalų (*angl. samples*), kurie pirmiausiai apdorojami, generuojant nekompensuojamus (*angl. offset-free*) signalus, kurie perduodami pirmos eilės išankstinio riškinimo (pre-emphasis) filtrui. Gauti 160 signalų vėliau analizuojami, kad būtų galima apibrėžti koeficientus, naudojamus trumpos analizės filtre (LPC analizė). Šie parametrai (koeficientai) toliau naudojami tų pačių 160 testinių signalų filtravimui. Filtro parametrai, vadinami atspindžio koeficientais, verčiami į logaritmines išraiškas (*angl. log area ratios*) LAR. Kalbos signalų (pavyzdžių) aibė sudalinama į 4 poaibius po 40 trumpų testinių signalų. Kiekvienas poaibis apdorojamas nuosekliai vienas po kito einančiais funkciniais elementais (žr. 3.1.1 pav.). Prieš apdorojant poaibių trumpus signalus, LTP analizės bloke apskaičiuojami ir atnaujinami ilgos analizės filtro parametrai. [3]

3.1.2 GSM 06.10 FR dešifratorius

GSM 06.10 FR dešifratoriaus struktūra tokia pati kaip ir šifratoriaus tik visi veiksmai atliekami atvirkštine tvarka. Po sėkmingo informacijos perdavimo, išėjime turimi rekonstruoti trumpi (*angl. short term*) signalai. Šie signalai paduodami į trumpalaikį sintezės filtrą. [3]

Pagrindinės GSM 06.10 FR savybės:

- Pilnas arba pusiau dupleksinis režimas
- Suderinimas su ETSI testų programomis
- Taikymas GPRS, EDGE, VoIP

3.2 GSM 06.10 šifratoriaus bei dešifratoriaus C aprašas

GSM 06.10 aprašą C kalbą galima rasti internete ir parsisiųsti nemokamai oficialiame Jutta Degener tinklalapyje [5]. 1992 m. Berlyno Technikos Universitete dėstytojai Jutta Degener ir Dr. Carsten Bormann realizavo GSM 06.10 PRELTP šifratorių ir dešifratorių C kalba. Kodas yra laisvai pasiekiamas, nemokamas. Patys autoriai rekomenduoja jį išbandyti, analizuoti bei kurti naujus realaus laiko multimedijos protokolus bei algoritmus. Realizacija susideda iš C bibliotekos bei savarankiškos programos. Abi suprojektuotos naudoti Unix aplinkoje su mažiausiai 32 bitų sveikųjų skaičių palaikymu. Pritaikoma MSDOS 16 bitų aplinkai. GSM 06.10 algoritmas greitesnis nei CELP. [5]

Naudojant biblioteką, sukuriamas *gsm* objektas, galintis užkoduoti 160 16 bitų PCM testinius signalus į 264 bitų GSM karkasus arba dešifruoti GSM karkasus į tiesinius PCM karkasus. Bibliotekoje esanti kliento programa *toast* leidžia išbandyti suspaudimą. Įvykdžius komandą *toast myspeech,* bus suspaustas failas myspeech, ištrintas bei surinkti suspaudimo rezultatai naujame faile *myspeech.gsm*. Komanda *untoast myspeech.gsm* paminėtą procesą įvykdys atvirkštine tvarka.

GSM 06.10 C kalbos realizacija bus dar aptariama tolimesniuose šio darbo skyriuose, kalbant apie aukšto lygio sintezę bei testines sintezės programas.

3.3 GSM LPC tiesinis nuspėjimo kodavimas

Kaip jau buvo minėta 1.1 poskyryje GSM 06.10 standartas paremtas prognozavimo kodavimo paradigma. GSM šifratoriuje taikoma LPC analizė.

LPC – priemonė naudojama audio signalų apdorojime atvaizduojant skaitmeninio signalo spektrinį įvyniojimą suspaustoje formoje naudojant tiesinio nuspėjimo modelį. Tai patikima analizės technologija koduojant geros kokybės kalbos signalus naudojant mažą bitų skaičių.

LPC veikimas aiškinamas prielaida, jog kalbos signalas sukeliamas vamzdžio gale esančio švilpuko pridedant šnypštimą bei trukinėjimą. Tarpai tarp garso bangų sukelia triukšmą, kuris apibūdinamas intensyvumu bei dažniu. Garso traktas (gerklė ir burna) sukuria vamzdį, kuris apibūdinamas savo rezonansais, kurie sukelia aukštesnio dažnio bangas. Liežuvis, lupos bei gerklė sukelia šnypštimą bei trukinėjimą. LPC analizuoja garso signalą nustatydamas akustinius rezonansus (šuolius), pašalindamas jų poveikius iš garsinio signalo bei nustatydamas triukšmų intensyvumą bei dažnį. Rezonansų šalinimas vadinamas inversiniu filtravimu. Procesas, kurio metu pašalinami po filtravimo likę signalai vadinamas – nusodinimu. LPC sintezuoja balso

signalą atvirkštiniu procesu t.y naudoja triukšmo parametrus bei nusodinimą sukuriant šaltinio signalą, naudoja rezonansus sukuriant filtrą (kuris vaizduoja vamzdį) bei filtruoja šaltinio signalą. Kadangi garso signalas kinta laike, šis procesas atliekamas mažomis garso signalo porcijomis, vadinamomis karkasais (*angl. frame*). Įprastai nuo 30 iki 50 karkasų per sekundę duoda suprantamą ir gerai suspaustą garsą.

Kitaip tariant LPC – tai metodas, nuspėjantis garso signalą iš prieš tai nuskaitytų signalų.

Galima nuspėti, kad n-tasis garso signalų sekos narys (signalas) yra apibūdinamas kaip suma iš p prieš tai esančiu signalų:

$$\hat{s} = \sum_{k=1}^{p} a_k s[n-k] \qquad (1)$$

Signalų skaičius *p,* nurodo LPC eilę. Jeigu *p* būtų begalinis, būtų galimybė labai tiksliai nuspėti n-tąjį signalą, tačiau *p* dažniausiai yra dešimtos ar dvidešimtos eilės, ko pilnai pakanka gerai kokybei. Koeficientai a_k parenkami, kad sumažintų klaidas tarp realaus signalo ir jo nuspėtos reikšmės. Siekiama, kad šis koeficientas būtų kiek įmanoma mažesnis:

$$e[n] = s[n] - \hat{s}[n] = s[n] - \sum_{k=1}^{p} a_k s[n-k] \qquad (2)$$

Galima abiejų formulių z transformacija:

$$E(z) = S(z) - \sum_{k=1}^{p} a_k S(z) z^{-k} = S(z)\left[1 - \sum_{k=1}^{p} a_k z^{-k}\right] = S(z)A(z) \qquad (3)$$

Formulėje 3 matyti, jog klaidos signalas E(z) gaunamas iš originalaus signalo ir perdavimo funkcijos A(z). A(z) vaizduoja nulinį skaitmeninį filtrą, kur a_k koeficientai lygūs nuliui filtro z plokštumoje. Originalios kalbos signalas gali būti išreikštas:

$$S(z) = \frac{E(z)}{A(z)} \qquad (4)$$

Perdavimo funkcija 1/A(z) atitinka visų polių (*angl. all-pole*) skaitmeninį filtrą. [6]

Visos paminėtos formulės ir jų išvedimai paaiškina, kodėl kodavimo metodas vadinamas nuspėjimo.

Norint išsiaiškinti, kaip bet koks metodas realizuojamas programiškai, geriausia turėti tikrus pavyzdžius. LPC realizacija pateikiama Jutta Degener ir Dr. Carsten Bormann realizuoto GSM 06.10 PRELTP C apraše. Minėtas aprašas pateikiamas šio darbo prieduose.

3.4 GSM LPC C aprašas

Tarp GSM 06.10 C realizacijos failų yra ir LPC analizės komponento C aprašas. Žemiau pateikiamas pagrindinis LPC analizės metodas.

```
void Gsm_LPC_Analysis P3((S, s, LARc),
        struct gsm_state *S,
        word * s,   /* 0..159 signals IN/OUT      */
        word * LARc)      /* 0..7   LARc's  OUT   */
{
              longword    L_ACF[9];
        #if defined(USE_FLOAT_MUL) && defined(FAST)
              if (S->fast) Fast_Autocorrelation (s,  L_ACF );
              else
        #endif
              Autocorrelation(s, L_ACF);
              Reflection_coefficients(L_ACF, LARc);
              Transformation_to_Log_Area_Ratios(LARc);
              Quantization_and_coding(LARc);
}
```

Kaip matyti iš aprašo, LPC analizės metodas susideda iš keturių pagrindinių funkcijų: autokoreliacijos skaičiavimo (**`Autocorrelation(s, L_ACF)`**), atspindžio koeficientų skaičiavimo (**`Reflection_coefficients(L_ACF, LARc)`**), transformavimo į logaritminę sritį (**`Transformation_to_Log_Area_Ratios(LARc)`**), kvantavimo bei kodavimo (**`Quantization_and_coding(LARc)`**) .

Svarbiausi parametrai: rodyklė į testinių signalų masyvą (*s) bei rodyklė į logaritminės srities koeficientų masyvą (*LARc). Naudojamas papildomas dešimties elementų masyvas L_ACF, kuris reikalingas kaip tarpinis masyvas formuojant koeficientų masyvą LARc. Masyvą s sudaro 160 testinių sveikųjų skaičių, kurie traktuojami kaip garsinės informacijos vektoriai (sveikąjį skaičių pavertus dvejetainiu, gaunamas bitų vektorius). Naudojamas apibrėžtas tipas word – tai 15 bitų sveikieji skaičiai. Longword – tai 32 bitų sveikieji skaičiai.

Autokoreliacijos metodas apdoroja 160 s masyvo 15 bitų pločio elementus kitaip tariant testinius rinkinius (*angl. sample*) bei suformuoja masyvą L_ACF, kurio reikšmės 32 bitų pločio.

Atspindžio koeficientų skaičiavimo metodas iš L_ACF masyvo reikšmių suformuoja LARc masyvo reikšmes, kurios vėliau apdorojamos transformavimo į logaritminę sritį metode, bei kvantavimo ir kodavimo metode. Rezultate gaunamos galutinės LARc masyvo reikšmės.

Autokoreliacija – tai signalo koreliacija su juo pačiu, skirta apskaičiuoti pradines L_ACF reikšmes. Signalų apdorojime autokoreliacija naudojama spręsti normalinės funkcijos skaičiavimo problemą.

$$R_{xx}(m) = \frac{1}{N - |m|} \sum_{n=1}^{N-m+1} x(n)x(n+m-1) \qquad (5)$$

Atspindžio koeficientai – koeficientai, skaičiuojami naudojant Schur algoritmą. Koeficientai skaičiuojami pagal autokoreliacijos koeficientus. Atspindžio koeficientai atvaizduoja nuspėjimo filtro tinklelio parametrus pagal duotą autokoreliacijos seką.

Transformacija į logaritminę sritį (LARs) – transformacija reikalinga atvaizduoti atspindžio koeficientus, perduodant informaciją per kanalą.

$$A_k = log\frac{1 + r_k}{1 - r_k} \qquad (6)$$

3.5 LPC ir CHStone

Skyriuje 2.4 buvo pateiktas CHStone testinių programų sąrašas. Šio darbo praktiniam eksperimentui atlikti naudojama GSM LPC programa, kurios fragmentas pateikiamas žemiau.

```
/*
+-----------------------------------------+
| * Test Vectors(added for CHStone)       |
| inData : input data                     |
| outData, outLARc : expected output data |
+-----------------------------------------+
*/
#define N 160
#define M 8
```

```
const word inData[N] =
  {81, 10854, 1893, -10291, 7614, 29718, 20475, -29215, -18949, -29806,
  -32017, 1596, 15744, -3088, -17413, -22123, 6798, -13276, 3819, -16273,
  -1573, -12523, -27103, -193, -25588, 4698, -30436, 15264, -1393, 11418,
  11370, 4986, 7869, -1903, 9123, -31726, -25237, -14155, 17982, 32427,
  -12439, -15931, -21622, 7896, 1689, 28113, 3615, 22131, -5572, -20110,
  12387, 9177, -24544, 12480, 21546, -17842, -13645, 20277, 9987, 17652,
  -11464, -17326, -10552, -27100, 207, 27612, 2517, 7167, -29734, -22441,
  ...}; // iš viso 160 įrašų

const word outData[N] =
  {80, 10848, 1888, -10288, 7616, 29712, 20480, -29216, -18944, -29808,
  -32016, 1600, 15744, -3088, -17408, -22128, 6800, -13280, 3824, -16272,
  -1568, -12528, -27104, -192, -25584, 4704, -30432, 15264, -1392, 11424,
  11376, 4992, 7872, -1904, 9120, -31728, -25232, -14160, 17984, 32432, -
  12432, -15936, -21616, 7904, 1696, 28112, ...}; // iš viso 160 įrašų

const word outLARc[M] = { 32, 33, 22, 13, 7, 5, 3, 2 };
int main ()
{       int i;
        int main_result;
        word so[N];
        word LARc[M];
        main_result = 0;
        for (i = 0; i < N; i++)
             so[i] = inData[i];
        Gsm_LPC_Analysis (so, LARc);
        for (i = 0; i < N; i++)
             main_result += (so[i] != outData[i]);
       for (i = 0; i < M; i++)
             main_result += (LARc[i] != outLARc[i]);
        printf ("%d\n", main_result);
        return main_result;
}
```

CHStone GSM testinėje programoje pateikiami testiniai rinkiniai. Testiniai įvesties duomenys, atitinkantys garso signalų pavyzdžius (*angl. sample*) pateikiami masyve **inData** (160 elementų)**,** tie patys testiniai signalai, kurie turi būti gauti po LPC analizės, pateikiami **outData** masyve. LPC analizės koeficientai, kurie turi būti apskaičiuojami, pateikiami masyve **LARc**.

Masyvo **inData** duomenys nukopijuojami į masyvą **so**. Rodyklės į **so** bei **LARc** masyvus paduodamos į **Gsm_LPC_Analysis** metodą. Po GSM LPC analizės, masyvo **so** duomenys modifikuojami, o masyvas **LARc** užpildomas reikiamomis koeficientų reikšmėmis. Vėliau patikrinama ar masyvo **so** ir **dataOut** bei **LARc** ir **outLARc** duomenys sutampa. Jei duomenys minėtuose masyvuose sutampa, kintamojo **main_result** reikšmė turi būti lygi 0. Tokiu būdu patikrinamas Gsm_LPC_Analysis metodo teisingumas.

Gsm_LCP_Analysis metodas susideda iš keturių pagrindinių metodų: Autocorrelation, Reflection_coefficients, Transformation_to_Log_Area_Ratios, Quantization_and_coding. Šie metodai remiasi pagalbiniais metodais: gsm_add, gsm_mult, gsm_mult_r, gsm_abs, gsm_norm, gsm_div. Gsm_LPC_Analysis metodo pagrindinės funkcijos šiek tiek pamodifikuotos (supaprastintos) lyginant su originaliu aprašu iš GSM 06.10 C realizacijos.

3.6 Išvados

- GSM-FR arba GSM 06.10 – tai pirmasis skaitmeninio kalbos kodavimo standartas naudojamas GSM.
- 1992 m. Berlyno Technikos Universitete dėstytojai Jutta Degener ir Dr. Carsten Bormann realizavo GSM 06.10 PRELTP šifratorių ir dešifratorių C kalba. Šiame C apraše randamas ir projektuojamo LPC komponento aprašas.
- LPC aprašas – tai LPC analizės įtaisas, realizuotas C kalba. Pagrindinis LPC analizės tikslas – apskaičiuoti reikiamus koeficientus, vėliau naudojamus šifravimo, dešifravimo procese kituose GSM 06.10 komponentuose.
- Šiame skyriuje pateikti principai bei teorija, kuria pagrįstas GSM LPC veikimas ir realizacija.

4. VHDL APRAŠO FORMAVIMO ANALIZĖ

Šiame darbe, analizuojant aukšto lygio sintezę, atliekamas eksperimentas. Projektuojamas LPC komponentas. Tai daroma, turimą CHStone C aprašą rankiniu būdu perrašant į VHDL. Ši kalba pasirinkta dėl lengvesnio loginės sintezės proceso. VHDL aprašas yra lengvai sintezuojamas. Loginė sintezė - pagrindinis eksperimento tikslas, nes rankiniu būdu aprašytas įtaisas turi būti sintezuojamas, kitaip tariant „gyvas“.

4.1 Pradinio VHDL modelio sudarymas

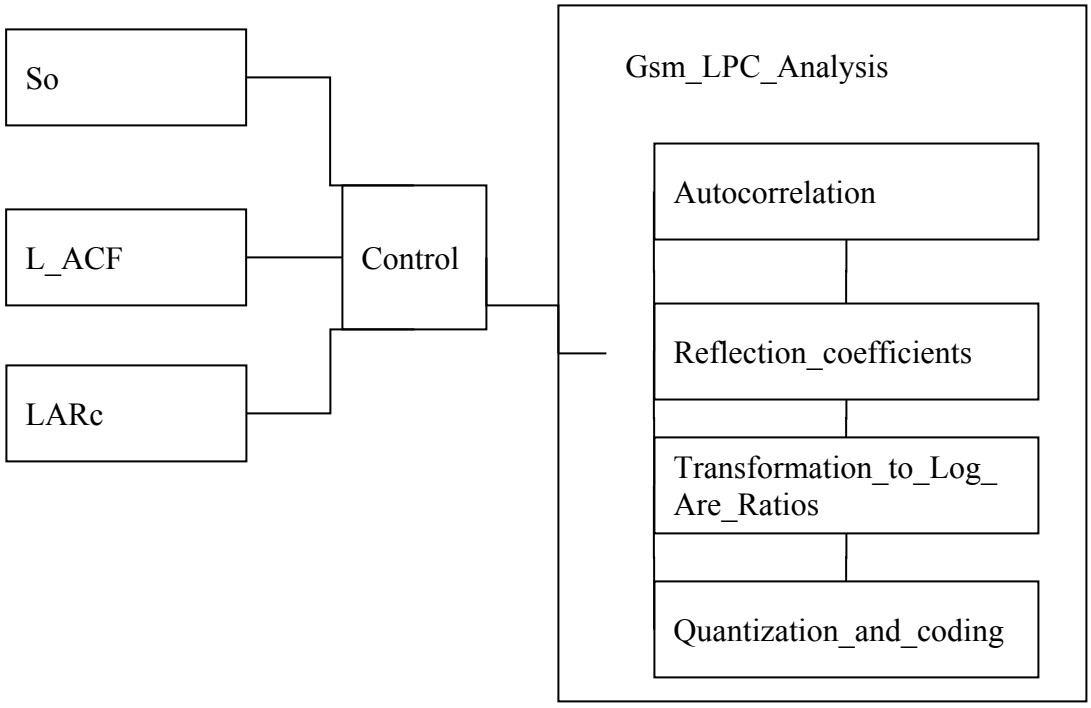
Bet kokį įtaisą realizuojant aparatūros aprašymo kalba, svarbu apsibrėžti jo struktūrą t.y iš kokių komponentų jis susideda, kaip jie tarpusavyje sąveikauja, kokie duomenų tipai, duomenų pločiai. Svarbu apsibrėžti, kas projektuojamame įtaise gali būti realizuojama elgsenos lygmenyje, kas struktūriniame. Paprastai elgsenos lygmeniu aprašomi komponentai atliekantys konkrečią apibrėžtą funkciją, o paskui tie komponentai sujungiami tarpusavyje struktūriškai, taip suformuojant bendrą įtaiso struktūrą.

LPC analizės komponentą (Gsm_LPC_Analysis funkciją) sudaro 4 pagrindinės funkcijos: Autocorrelation(s, L_ACF), Reflection_coefficients(L_ACF, LARc), Transformation_to_Log_Area_Ratios(LARc), Quantization_and_coding(LARc). Šios funkcijos tarpusavyje siejamos, per tris masyvus: s (testinių signalų), L_ACF (autokoreliacijos koeficientų), LARc (lpc analizės koeficientų). Turimą C aprašą perrašant į VHDL, paminėtas funkcijas galima realizuoti kaip atskirus komponentus (pvz. baigtinius automatus FSM). Šie komponentai sujungti tarpusavyje sudarytų vieną GSM LPC analizės komponentą. Anksčiau paminėti duomenų ir koeficientų masyvai gali būti realizuoti kaip atminties komponentai, kurie būtų prijungti prie GSM LPC analizės komponento. Visos pagalbinės funkcijos (gsm_add,), naudojamos keturiose pagrindinėse funkcijose gali būti realizuotos taip pat kaip komponentai.

Toliau pateikiama (13 pav.) LPC analizės komponento modelio supaprastinta schema, vaizduojanti kaip paminėti komponentai galėtų būti sujungti tarpusavyje. Pateiktoje schemoje vaizduojamas taip pat ir testavimo modelis, tinkamas taikyti tiek prieš loginę sintezę tiek po jos. Tolimesniuose tyrimuose reikalingas tik susintezuotas Gsm_LPC_Analysis įtaisas. Atminties komponentų sintezuoti nebūtina, nes jie reikalingi tik duomenų saugojimui paties LPC analizės komponento išorėje. Po loginės sintezės ši schema būtų tinkama naudoti atliekant verifikavimą.

Gsm_LPC_Analysis

4.1.1 pav. GSM LPC analizės komponento modelio supaprastinta schema



Pagal turimą modelio schemą bendrais bruožais galima smulkiau apibrėžti atminties bei baigtinių automatų komponentus.

4.1.1 Atminties komponentai

VHDL kalboje (taip pat ir kitose HDL kalbose) atminties komponentai gali būti realizuojami kaip vienmačiai arba dvimačiai atitinkamo (std_logic, std_logic_vector, integer ir kitų) duomenų tipo masyvai, tačiau masyvas kaip duomenų saugojimo elementas gali būti naudojamas tik konkretaus komponento viduje. Kad šiuos masyvus būtų įmanoma prijungti struktūriškai prie kitų komponentų, jie turi būti aprašomi kaip atskiri nepriklausomi komponentai, šiems masyvams turi būti sukuriamas duomenų valdymo mechanizmas t.y duomenų įrašymo į masyvą ir nuskaitymo iš jo. Kaip ir visi komponentai, taip ir šis turi turėti savo sąsają. Toks atminties komponentas savo sąsajoje turi turėti leidimo skaityti iš masyvo signalą, leidimo rašyti į masyvą signalą, adreso signalą, duomenų signalą (gali būti vienas dvikryptis arba du įvesties ir išvesties) taip pat gali turėti masyvo išvalymo signalą (reset).

Norint duomenis įrašyti į atmintį, turi būti paduodama įvesties duomenų porcija, leidimo rašyti į atmintį signalas (loginis 1), adresas, nurodantis į kurią atminties vietą turi būti rašoma informacija. Norint duomenis nuskaityti iš atminties turi būti paduodamas skaitymo iš atminties signalas (loginis 1) ir atitinkamai adresas. Atminties komponentas gavęs skaitymo leidimo signalą, duomenų signalo išėjime atiduoda duomenis saugomus nurodytu adresu.

Žemiau pateikiamas galimas tokio atminties komponento VHDL aprašo fragmentas.

```
entity LARc is
    port (data_in  : in  integer range 32767 downto -32768;
          data_out : out integer range 32767 downto -32768;
          rden     : in std_logic;
          wren     : in std_logic;
          addr     : in integer range 0 to 7);
end LARc;

architecture arch of LARc is
   type matrica is array (0 to 7) of integer range 32767 downto -32768;
   signal mem : matrica := (0,0,0,0,0,0,0,0);
begin
   process (rden, wren, addr)
   begin
      if rden = '1' and wren = '0' then
         data_out <= mem(addr);
      elsif rden = '0' and wren = '1' then
            mem(addr) <= data_in;
            data_out <= 0;
      else
            data_out <= 0;
```

```
        end if;
    end process;
end arch;
```

Aukščiau pateiktas VHDL aprašas – tai atminties komponentas pavadintas LARc, kuris gali saugoti 8 sveikuosius skaičius. Tai parodo sakinys: `type matrica is array (0 to 7) of integer range 32767 downto -32768;`. Leidimo skaityti signalas čia pavadintas rden, rašyti – wren. Skaitymas vykdomas tik esant sąlygoms: `if rden = '1' and wren = '0'`. Rašymas vykdomas esant sąlygoms: `if rden = '0' and wren = '1'`.

Pagal šį nesudėtingą pavyzdį gali būti sudaromi kitokie reikalingi atminties komponentai. Norimo dydžio, norimo duomenų tipo. Kaip ir buvo minėta, tokį atminties komponentą, remiantis tais pačiais principais, galima realizuoti ir kita aparatūros aprašymo kalba.

4.1.2 Baigtiniai automatai

Aptariant GSM LPC analizės metodą, buvo minimos keturios pagrindinės funkcijos, kurios atlieka visą duomenų apdorojimą. C realizacijoje duomenų apdorojimas vykdomas nuosekliai. Svarbiausia, jog visas keturias pagrindines funkcijas sieja tie patys duomenų masyvai, todėl kol vieną funkcija atlieka veiksmus su konkrečiu masyvu, kitos funkcijos negali dirbti su tuo masyvu. Ši logika turi būti išlaikoma ir formuojant VHDL aprašą. Aparatūroje signalų perdavimas vykdomas lygiagrečiai. Keli struktūriškai tarpusavyje sujungti komponentai kiekvienas savo funkcijas vykdo lygiagrečiai, todėl norint, kad komponentai veiksmus atliktų nuosekliai vienas po kito, reikalingas tokio nuoseklumo užtikrinimo mechanizmas. Vienas komponentas turi priversti kitą laukti, kol bus baigti veiksmai su bendrai naudojamu resursu, kuris aptariamu atveju – tai atmintis.

Veiksmų vykdymo nuoseklumą galima užtikrinti naudojant baigtinius automatus. Automatą galima priversti norimą laiką būti konkrečioje būsenoje. Kol vienas automatas skaito ir rašo į bendrai naudojamą atmintį, kitas tuo metu lieka laukimo būsenoje. Tokio nuoseklaus kelių automatų darbo užtikrinimui turi būti perduodami iš vieno automato į kitą valdymo signalai, parodantys, jog darbą baigė pirmasis ir darbą toliau gali tęsti antrasis ir t.t. Automatas baigęs darbą, lieka laukimo būsenoje, kol vėl gaus sužadinimo signalą ir galės visą veiksmų seką vykdyti iš naujo.

Šiame kontekste minimo baigtinio automato sąsają paprastai galėtų sudaryti sinchronizacijos, pradinio nustatymo, duomenų, adresų, valdymo signalai. Architektūroje būtų

pateikiamas visas elgsenos aprašas t.y. būsenų kitimo logika, atliekami veiksmai. Automato architektūroje gali būti naudojami skaitikliai, kurie leistų užtikrinti, jog automatas reikiamoje būsenoje išliks tams tikrą laiko intervalą. Toks skaitiklių automato viduje panaudojimas - vienas iš aparatūriškai vaizdingų būdų realizuoti ciklus.

Automatas VHDL kalba realizuojamas case sąlyginio sakinio pagalba.

Žemiau pateikiamas paprasto baigtinio automato aprašo fragmentas, kuriame parodoma, kaip keičiasi automato būsenos, kaip būsenų kitimas priklauso nuo konkretaus signalo ‚a' reikšmės.

```
case current_state is

            when S0 =>   x <= '0';
                         if a='0' then
                             next_state <= S0;
                         elsif a ='1' then
                             next_state <= S1;
                         end if;

            when S1 =>   x <= '0';
                         if a='0' then
                             next_state <= S1;
                         elsif a='1' then
                             next_state <= S2;
                         end if;

            when others =>
                         x <= '0';
                         next_state <= S0;

end case;
```

Aukčiau pateiktame pavyzdyje matoma kaip automato būsenos keičiasi priklausomai nuo ‚a' reikšmės. Automato valdymui pakanka valdyti signalo ‚a' reikšmę. Jei signalas ‚a' būtų skaitiklio reikšmė, kitą būseną sąlygojančiame sakinyje būtų galima priversti automatą pasilikti reikiamoje būsenoje norimą laiko intervalą.

4.1.3 Duomenų signalų paskirstymas

Kompiuterių elementų teorijoje žinoma, jog paprastai kelių komponentų išėjimo signalai negali būti sujungti viename taške. Kad toks sujungimas būtų įmanomas, komponentai turi būti trijų būsenų, jog vienu metu tik vienas iš kelių komponentų išėjime turėtų signalą ir naudotųsi

bendra duomenų perdavimo magistrale arba laidu kiti komponentai tuo metu būtų trečioje būsenoje t.y. atjungti nuo bendros magistralės.

Jeigu tarpusavyje išėjimo signalais sujungiama nedaug komponentų, tada galima sukurti suderinimo komponentą, kuris esant atitinkamiems valdymo signalams, į bendrą magistralę išduotų reikiamo komponento išėjimo signalą. Šis duomenų signalų paskirstymo komponentas dirbtų multiplekserio principu.

Remiantis pradiniu šio darbo VHDL modeliu, reikia pabrėžti, kad toks duomenų paskirstymo komponentas būtų reikalingas valdyti duomenims, kuriuos automatai siųstų į atminties komponentus. Pagal LPC algoritmą žinoma, jog kelios pagrindinės funkcijos naudojasi tuo pačiu masyvu. VHDL modelyje masyvai – tai atminties komponentai, o pagrindinės funkcijos – tai baigtiniai automatai. Keli automatai duomenų magistralėmis prijungti prie bendrų atminties komponentų. Duomenų paskirstymo komponentas galėtų pasirūpinti, jog į bendrą duomenų įrašymo į atmintį magistralę butų perduodami duomenys tik iš tuo metu dirbančio automato (automatai veiksmus atlieka vienas po kito nuosekliai). Kad duomenų paskirstymo komponentas į bendrą magistralę perduotų reikiamą informaciją, baigtiniai automatai turi šiam komponentui perduoti signalus, jog jie atlieka veiksmus arba yra laukimo būsenoje. Naudojamas busy signalas. Vienu metu vienas automatas gali būti darbinėje būsenoje busy = 1, visi kiti turi būti laukimo būsenoje busy = 0.

Svarbi automato savybė, jog kiekvienoje būsenoje jis formuoja atitinkamus išėjimo signalus. Ši savybė labai svarbi formuojant įvesties signalus pagalbinių funkcijų komponentams. Jei automatas prie savęs turi struktūriškai prijungtą kitą konkrečią funkciją atliekantį komponentą, automatas savo išėjimo signalais gali perduoti to komponento įvesties signalus. Šie įvesties signalai gali būti konkrečios funkcijos operandai. Pagalbinio komponento išėjimo signalas gali būti naudojamas kaip automato įvesties signalas (funkcijos rezultatas), reikalingas tolimesniems skaičiavimams atlikti.

4.1.4 Pagalbinių funkcijų komponentai

Šiame darbe pagalbinėmis GSM LPC funkcijomis vadinamos funkcijos: gsm_add, gsm_mult, gsm_mult_r, gsm_abs, gsm_norm, gsm_div. Pateiktos funkcijos atlieka sumavimo, specifinės daugybos, modulio skaičiavimo, normalizavimo, dalybos veiksmus. LPC C apraše šios išvardintos funkcijos (metodai) naudojamos keturiose pagrindinėse Gsm_LPC_Analysis funkcijose. Aparatūros aprašymo kalbose šias funkcijas galima realizuoti kaip komponentus.

Tokio komponento sąsajoje kaip įvesties signalai būtų nurodomi funkcijos operandai, o išėjimo signalas – funkcijos rezultatas. Komponento architektūroje būtų pateikiamas funkcijos aprašas. Komponentas gavęs įvesties reikšmes suformuotų rezultato reikšmę.

4.2 Duomenų tipų suderinimas

GSM LPC C apraše naudojami apibrėžti duomenų tipai word, bei longword:

```
typedef short word;          /* 16 bit signed int   */
typedef long longword;       /* 32 bit signed int   */
```

Tipas word atitinka short, longword – long C kalbos sveikųjų skaičių tipus. VHDL kalboje sveikiesiems skaičiams apibrėžti naudojamas integer tipas, kurio ilgis 32 bitai. VHDL kalboje nurodant signalo tipą integer šalia galima nurodyti ir ribas, kurias būtina nurodyti jei rašomas sintezuojamas aprašas. 16 bitų sveikasis skaičius su ženklu nurodomas kaip:

```
sv_sk : integer range 32767 downto -32768;
```

2^15 - 1 = 32767 šis skaičius - tai didžiausias teigiamas sveikasis skaičius naudojant 16 bitų. Skaičiams su ženklu vyriausias bitas naudojamas nurodyti ženklą, todėl skaičiaus reikšmei koduoti lieka 15 jaunesnių bitų [14..0], todėl didžiausias sveikasis skaičius apskaičiuojamas 2^15-1 = 32767. Didžiausias neigiamas skaičius -32768. Žemiau pateikiamas pavyzdys, parodantis kaip koduojamas sveikasis skaičius su ženklu, kuriam skiriami 3 bitai:

3	**0**11
2	**0**10
1	**0**01
0	**0**00
-1	**1**11
-2	**1**10
-3	**1**01
-4	**1**00

Pavyzdyje paryškintas bitas parodo skaičiaus ženklą. 0 – skaičius teigiamas, 1 – neigiamas. Kitais bitais koduojama skaičiaus reikšmė. Neigiamo skaičiaus kodas gaunamas invertuojant teigiamo skaičiaus kodą bei pridedant 1. Skaičius – A = ~A + 1, ~ 001 = 110, 110 + 1 = 111. Pateiktas pavyzdys paaiškina, kodėl 16 bitų sveikasis skaičius su ženklu patenka į intervalą [-32768..32767].

Kitas duomenų tipas longword 32 bitų. Šiam tipui galima naudoti VHDL signed tipą (numeric_std bibliotekoje). Signed (31 downto 0). Signed tipas analogiškas bitų vektoriui, tik skirtingai nuo bitų vektoriaus, čia vyriausias bitas nurodo skaičiaus ženklą. Atliekant aritmetines operacijas tarp dviejų signed kintamųjų, rezultatas gaunamas taip pat signed tipo t.y skaičius su ženklu. Signed tipą galima pavadinti specifiniu bitu vektoriumi.

VHDL kalboje atlikti postūmio (loginio, aritmetinio) operacijas galima tik su vektorių tipo kintamaisiais. Jei integer tipo kintamiesiems turi būti atliekamos postūmio operacijos, kintamąjį pirmiausia reikia konvertuoti į vektorinį tipą. Naudojant numeric_std biblioteką, konvertavimui naudojamos funkcijos: to_signed(), to_integer(). Naudojant std_logic_arith biblioteką – conv_integer(), conv_signed().

Kalbant apie postūmio operacijas, reikia paminėti, jog VHDL kalboje operacijos: sll, srl, sla, sra, rol, ror yra nesintezuojamos. Ši problema sprendžiama masyvų sukarpymu (*angl. array slice*). Jeigu turimas masyvas (vektorius) A = “1011010110”, tai loginis postūmis per 5 bitus į kairę būtų išreiškiamas per Ap = A(9 downto 5) & “00000”, į dešinę – Ap = “00000” & A(9 downto 5). Užrašas A(9 downto 5) & “00000” reiškia, jog paimami masyvo A bitai nuo devinto iki penkto (numeravimas nuo vyriausio bito iš kairės į dešinę) ir šia bitai sujungiami su bitų seka „00000“. Atliekant aritmetinį postūmį minėtu būdu, svarbu išsaugoti ženklo reikšmę (ženklą nurodančius bitus). Jeigu skaičius neigiamas tarkim A = -8 = “**1**000” (pariškintas ženklo bitas), aritmetinis postūmis į kairę per 2 bitus užrašomas Ap = “1” & A(0) & “00” = “**1**000”, į dešinę – Ap = “1” & “11” & A(2) = “**111**0” (atliekant postūmį į dešinę, į atsilaisvinusias vietas įrašomas ženklo bitas).

4.3 Išvados

- Formuojant bet kokio įrenginio aprašą, apibrėžiama jo struktūra, sudaroma įrenginio struktūros schema, apibrėžiami naudojami komponentai. Apsibrėžiami atminties bei kiti funkciniai komponentai, suderinami duomenų tipai.
- Atminties komponentai gali būti realizuojami kaip vienmačiai arba dvimačiai masyvai. Nuosekliam operacijų vykdymui galimas baigtinių automatų pasirinkimas. Atskiras algoritmo funkcijas galima realizuoti kaip nepriklausomus komponentus.

- Rašant sintezuojamą VHDL kodą, svarbu naudoti sintezuojamas konstrukcijas pvz. apibrėžtas sveikųjų skaičių ribas masyvų sudalinimą vietoj funkcijų sll, srl, sla, sra, rol, ror.

5. LPC KOMPONENTO REALIZAVIMAS

Šiame darbe atliekamas GSM LPC aprašo realizuoto C kalba perrašymas į VHDL. LCP C aprašo šaltinis – tai CHStone aukšto lygio sintezės testinių programų rinkinys. Šiame skyriuje pateikiamas komponento realizavimo planas, eiga, rezultatai, išvados.

5.1 LPC komponento realizavimo darbų planas

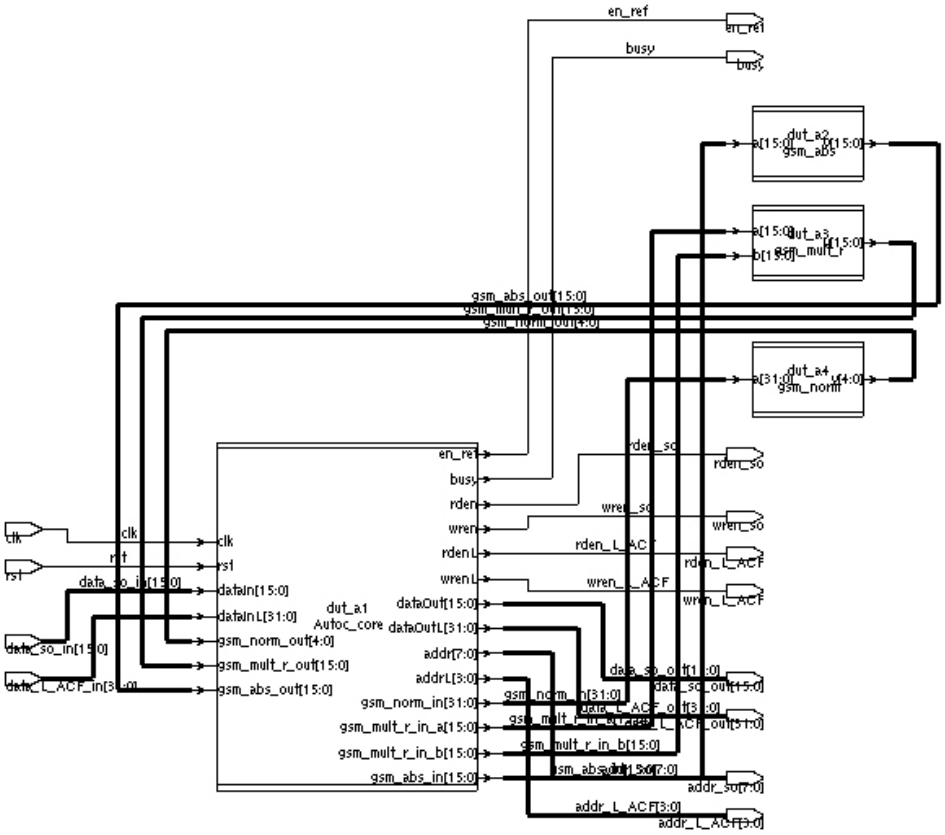
Komponento realizavimas atliekamas pagal 3 skyriuje surašytą metodiką bei logiką.

GSM LPC komponento realizavimo darbų planas:

1. C kalba realizuotos programos struktūros analizė (kokie pagrindiniai metodai, koks jų tarpusavio ryšys?)
2. Pagrindinių naudojamų duomenų tipų analizė (int, long)
3. Bendrai naudojamų resursų analizė (kokie bendri duomenų saugojimo elementai?)
4. Smulkių pagalbinių funkcijų aprašymas kaip VHDL komponentų
5. Pagrindinių masyvų aprašymas kaip VHDL atminties komponentų
6. Pagrindinių metodų realizavimas kaip VHDL komponentų (baigtinių automatų)
7. Kiekvieno komponento modeliavimas, tikrinant veikimo logiką (modeliavimo rezultatai lyginami su C programos tarpiniais rezultatais, kuriuos papildomai reikia išsivesti į ekraną, pamodifikavus analizuojamą C programą)
8. Visų komponentų apjungimas į TOP schemą
9. TOP schemos modeliavimas, tikrinant galutinį variantą (modeliuojant tikrinama, ar pagrindiniuose duomenų saugojimo masyvuose (VHDL atminties komponentuose) po visų iteracijų išsaugomos teisingos reikšmės, kurios pateikiamos C programoje)
10. Modeliuojamo aprašo loginė sintezė. Esant sintezės klaidoms, modeliuojamo aprašo koregavimas pagal sintezės programos reikalavimus Klaidos ištaisomos eksperimento būdu.

5.2 LPC komponento struktūra (vidiniai komponentai)

Autocorrelation



5.2.1 pav. Autocorrelation komponento schema

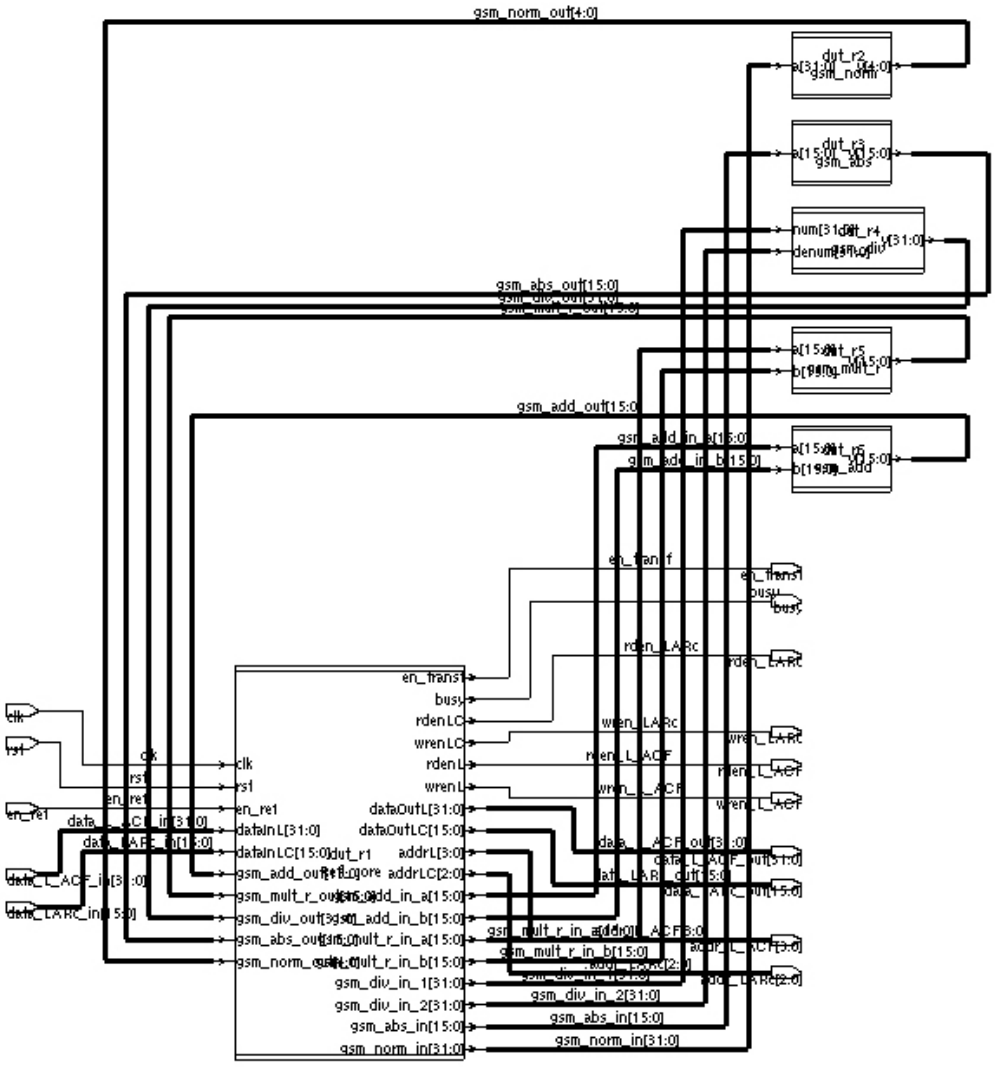
Komponento struktūrą sudaro: Autoc_core (baigtinis automatas), gsm_abs, gsm_norm, gsm_mult_r

Įvesties signalai: clk, rst, data_so_in[15:0], data_L_ACF_in[31:0]

Išvesties signalai: en_ref, busy, rden_so, wren_so, rden_L_ACF, wren_L_ACF, data_so_out[15:0], data_L_ACF_out[15:0], addr_so[7:0], addr_L_ACF[8:0]

Autoc_core – Autocorrelation komponento branduolys

Reflection_coefficients



5.2.2 pav. Reflection_coefficients komponento schema

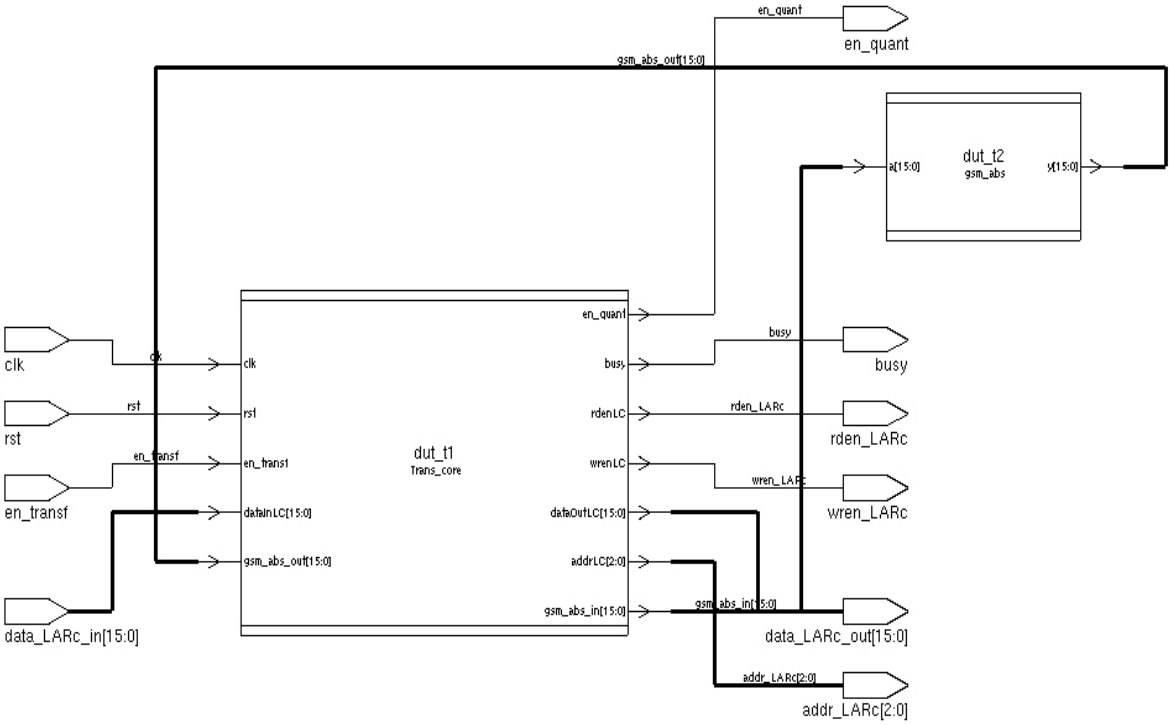
Komponento struktūrą sudaro: Refl_core (baigtinis automatas), gsm_abs, gsm_div, gsm_mult_r, gsm_add

Įvesties signalai: clk, rst, en_ref, data_L_ACF_in[31:0], data_LARc_in[15:0]

Išvesties signalai: wren_LARc, rden_L_ACF, wren_L_ACF, data_L_ACF_out[31:0], data_LARc_out[15:0], addr_L_ACF[3:0], addr_LARc[2:0], en_transf, busy, rden_LARc

Refl_core – Reflection_coefficients komponento branduolys.

Transformation_to_Log_area_ratios



5.2.3 pav. Transformation_to_Log_area_ratios komponento schema

Komponento struktūrą sudaro: Trans_core (baigtinis automatas), gsm_abs

Įvesties signalai: clk, rst, en_transf, data_LARc_in[15:0]

Išvesties signalai: en_quant, busy, rden_LARc, wren_LARc, data_LARc_out[15:0], add_LARc[2:0]

Trans_core - Transformation_to_Log_area_ratios komponento branduolys

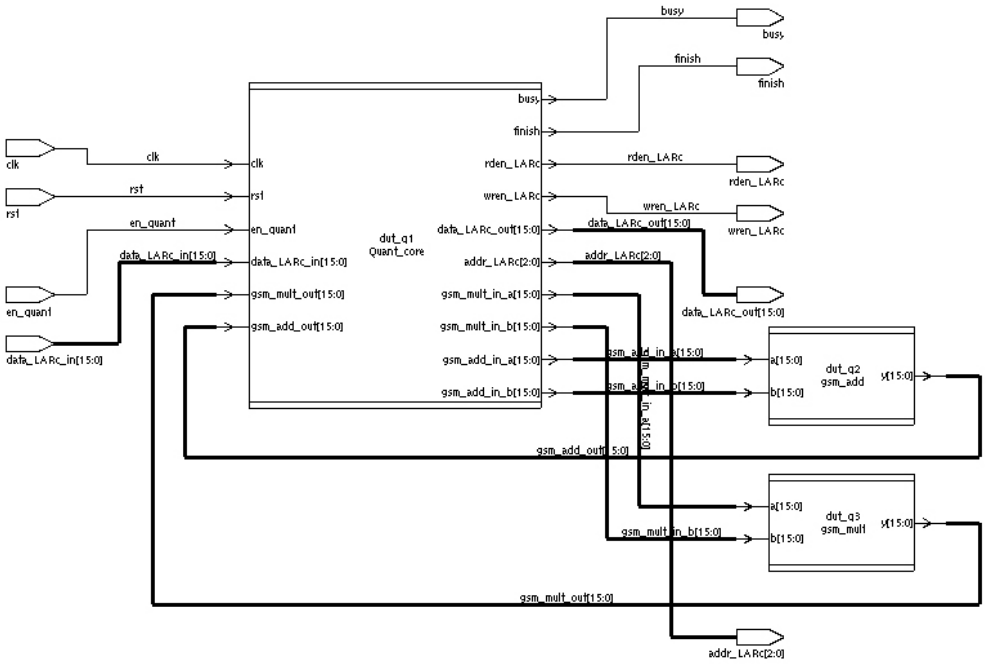
Quantization_and_coding

Komponento struktūrą sudaro: Quant_core (baigtinis automatas), gsm_add, gsm_mult

Įvesties signalai: clk, rst, en_quant, data_LARc_in[15:0]

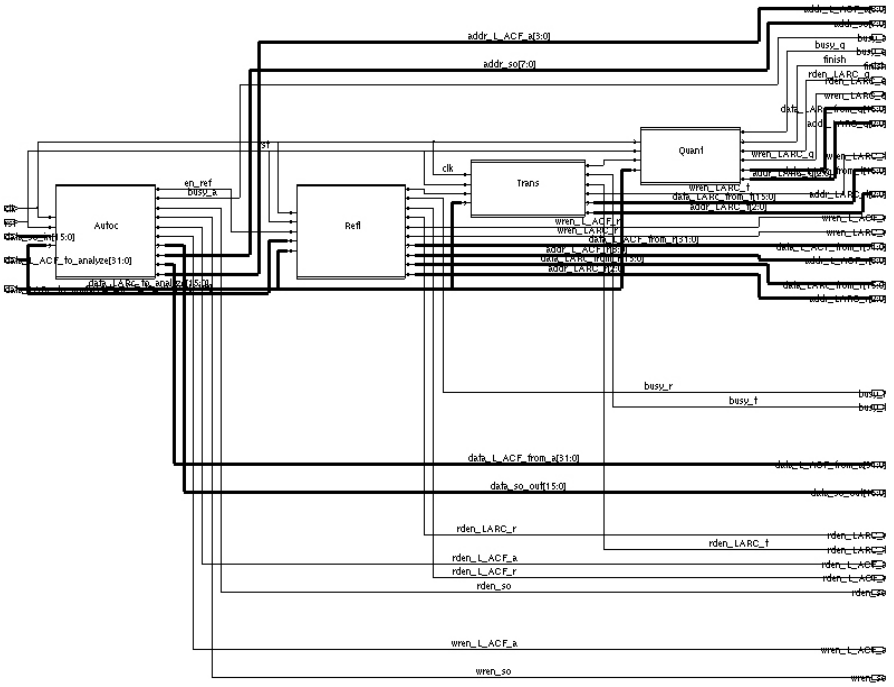
Išvesties signalai: busy, finish, rden_LARc, wren_LARc, data_LARc_out[15:0], add_LARc[2:0]

Quant_core - Quantization_and_coding komponento branduolys.



5.2.4 pav. Quantization_and_coding komponento schema

Gsm_LPC_Analysis (top schema)



5.2.5 pav. GSM_LPC_Analysis schema

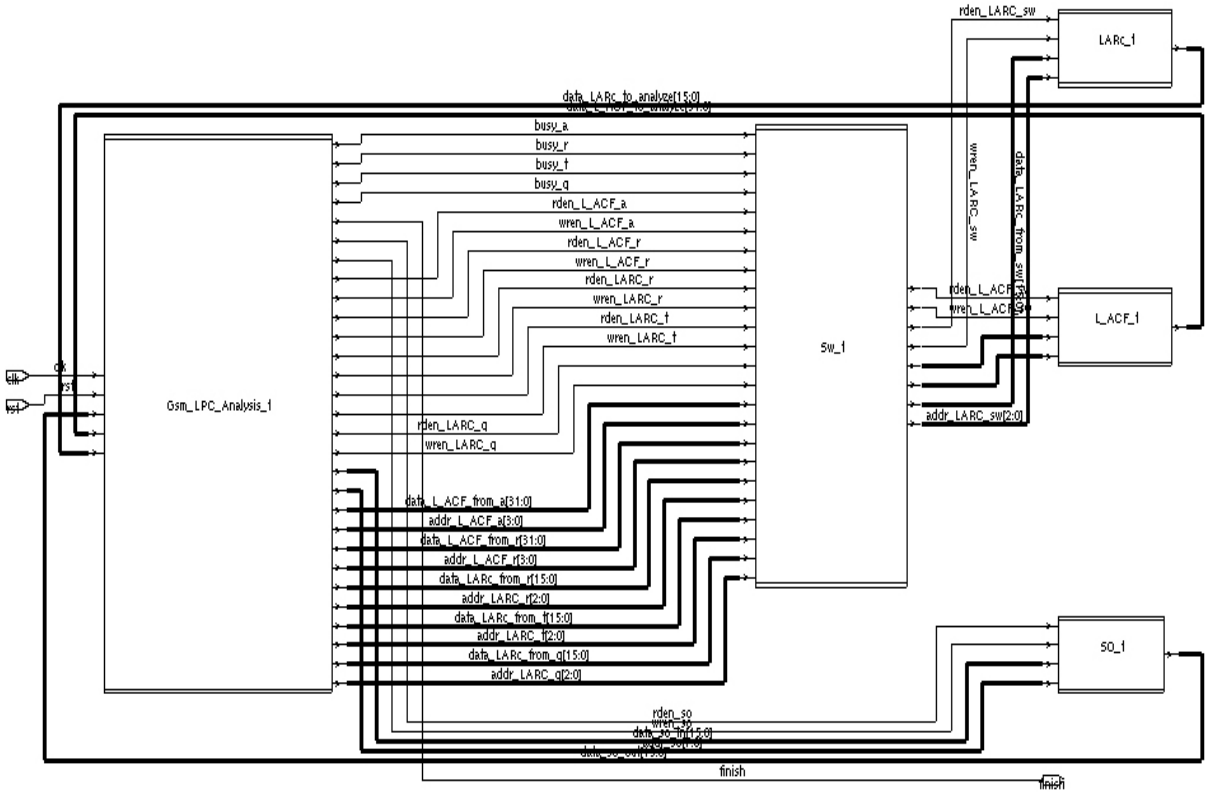
Struktūra: Autoc, Refl, Trans, Quant

Įvesties signalai: clk, rst, data_so_in[15:0], data_L_ACF_to_analyze[31:0], data_LARc_to_analyze[31:0],

Išvesties signalai: addr_L_ACF_a[3:0], addr_so[7:0], busy_a, busy_q, finish, rden_LARC_q, wren_LARC_q, data_LARc_from_q[15:0], addr_LARc_q[2:0], wren_LARC_t, data_LARc_from_t[15:0], addr_LARC_t[2:0], wren_L_ACF_r, wren_LARC_r, data_L_ACF_from_r[31:0], addr_L_ACF_r[3:0], data_LARc_from_r[15:0], addr_LARC_r[2:0], busy_r, busy_t, data_L_ACF_from_a[31:0], data_so_out[15:0], rden_LARC_r, rden_LARC_t, rden_LARC_a, rden_L_ACF_a, rden_L_ACF_r, rden_so.

Gsm_LPC_Tester

Gsm_LPC_Analysis komponento testavimui sudaroma testavimo schema, kurioje prie pagrindinio komponento prijungiami atminties komponentai: L_ACF, LARc, so. Žemiau pateikiama schema.



5.2.6 pav. Gsm_LPC_Tester schema

5.3 LPC komponento modeliavimas

Turimoms schemoms atliekamas modeliavimas, kurio metu siekiama, kad atimties komponentuose būtų įrašomos reikiamos reikšmės. Modeliavimas atliekamas remiantis CHStone programos duotosiomis reikšmėmis. CHStone programoje GSM LPC pateikiamos masyvo so reikšmės (160 sveikųjų skaičių):

```
const word inData[N] =
  { 81, 10854, 1893, -10291, 7614, 29718, 20475, -29215, -18949, -29806,
-32017, 1596, 15744, -3088, -17413, -22123, 6798, -13276, 3819, -16273,
-1573, -12523, -27103, -193, -25588, 4698, -30436, 15264, -1393, 11418,
11370, 4986, 7869, -1903, 9123, -31726, -25237, -14155, 17982, 32427,
-12439, -15931, -21622, 7896, 1689, 28113, 3615, 22131, -5572, -20110,
 12387, 9177, -24544, 12480, 21546, -17842, -13645, 20277, 9987, 17652,
 -11464, -17326, -10552, -27100, 207, 27612, 2517, 7167, -29734, -22441,
 30039, -2368, 12813, 300, -25555, 9087, 29022, -6559, -20311, -14347,
-7555, -21709, -3676, -30082, -3190, -30979, 8580, 27126, 3414, -4603,
-22303, -17143, 13788, -1096, -14617, 22071, -13552, 32646, 16689,
-8473, -12733, 10503, 20745, 6696, -26842, -31015, 3792, -19864, -20431,
 -30307, 32421, -13237, 9006, 18249, 2403, -7996, -14827, -5860, 7122,
 29817, -31894, 17955, 28836, -31297, 31821, -27502, 12276, -5587,
-22105, 9192, -22549, 15675, -12265, 7212, -23749, -12856, -5857, 7521,
 17349, 13773, -3091, -17812, -9655, 26667, 7902, 2487, 3177, 29412,
-20224, -2776, 24084, -7963, -10438, -11938, -14833, -6658, 32058, 4020,
10461, 15159 };
```

Šios reikšmės įprogramuojamos į **so** atminties komponentą. Modeliavimo sinchronizacijos signalo periodas nustatomas 20ns. Modeliavimo eksperimentai atliekami OrCad bei Cadence NCLaunch įrankiais. Pagrindiniai komponentai veiksmus atlieka nuosekliai, vienas po kito, todėl modeliavimas galimas komponentus apjungus į bendrą schemą (Gsm_LPC_Analysis). Eksperimento metu nustatomi laiko intervalai kuriuose kiekvienas pagrindinis komponentas atlieka reikiamus veiksmus.

1. Autocorrelation. – [0..135180] ns

2. Reflection_coefficients – [135180..139480] ns

3. Transformation_to_Log_area_ratios – [139480..140250] ns

4. Quantization_and_coding – [140250..141180] ns

Modeliuojant pirmąjį komponentą, tikrinama ar **so** atminties komponento reikšmės lygios duotosioms CHStone programoje:

```
const word outData[N] =
  { 80, 10848, 1888, -10288, 7616, 29712, 20480, -29216, -18944, -29808,
-32016, 1600, 15744, -3088, -17408, -22128, 6800, -13280, 3824, -16272,
 -1568, -12528, -27104, -192, -25584, 4704, -30432, 15264, -1392, 11424,
 11376, 4992, 7872, -1904, 9120, -31728, -25232, -14160, 17984, 32432,
 -12432, -15936, -21616, 7904, 1696, 28112, 3616, 22128, -5568, -20112,
12384, 9184, -24544, 12480, 21552, -17840, -13648, 20272, 9984, 17648,
 -11456, -17328, -10544, -27104, 208, 27616, 2512, 7168, -29728, -22448,
 30032, -2368, 12816, 304, -25552, 9088, 29024, -6560, -20304, -14352,
 -7552, -21712, -3680, -30080, -3184, -30976, 8576, 27120, 3408, -4608,
-22304, -17136, 13792, -1088, -14624, 22064, -13552, 32640, 16688,
 -8480, -12736, 10496, 20752,6704, -26848, -31008, 3792, -19856, -20432,
 -30304, 32416, -13232, 9008, 18256, 2400, -8000, -14832, -5856, 7120,
 29824, -31888, 17952, 28832, -31296, 31824, -27504, 12272, -5584,
 -22112, 9200, -22544, 15680, -12272, 7216, -23744, -12848, -5856, 7520,
 17344, 13776, -3088, -17808,-9648, 26672, 7904, 2480, 3184, 29408,
 -20224, -2768, 24080, -7968, -10432, -11936, -14832,-6656, 32064, 4016,
 10464, 15152 };
```

Į komponentą L_ACF įrašoma tarpinė informacija: 402349336, 1150648, -18510626, -37045272, -39221096, 35550306, 6793340, 29212618, -29045850. Šios reikšmės gaunamos pakoregavus CHStone GSM LPC programą, kad į konsolę būtų išvedami tarpiniai rezultatai.

Jeigu reikšmės sutampa, laikoma, kad duotuoju atveju pirmojo komponento veikimo logika yra teisinga.

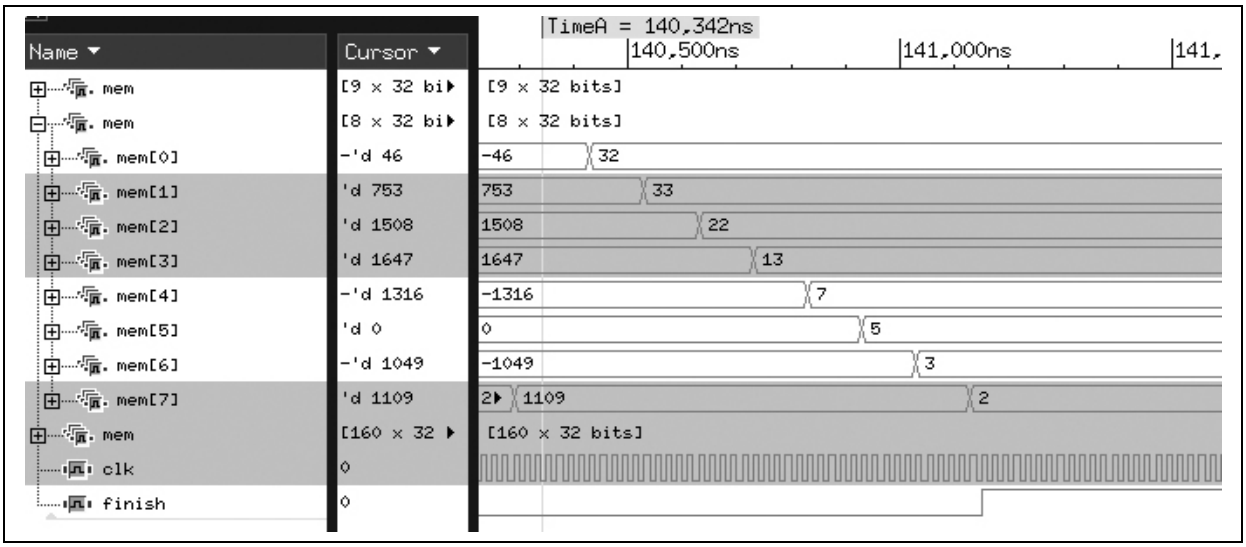
Antrasis komponentas naudoja informaciją, saugomą L_ACF ir suformuoja tarpines reikšmes bei išsaugo į LARc: -93, 1507, 3016, 3294, -2632, 0, -2099, 2219.

Trečiasis komponentas naudoja LARc informaciją, ją apdoroja ir išsaugo į tą patį LARc atminties komponentą. Gaunami tarpiniai rezultatai: -46, 753, 1508, 1647, -1316, 0, -1049, 1109.

Ketvirtasis komponentas taip pat apdoroja LARc reikšmes ir suformuoja galutinį rezultatą (gaunami LPC analizės koeficientai). Šie koeficientai pateikiami CHStone GSM LPC programoje:

const word outLARc[M] = { 32, 33, 22, 13, 7, 5, 3, 2 };

Jeigu reikšmės sutampa, laikoma, kad duotuoju atveju ketvirtojo komponento taip pat ir visos Gsm_LPC_Analysis schemos veikimo logika yra teisinga. Toliau pateikiamas laikinės diagramos fragmentas, kuriame matomos suformuotos LARc reikšmės.



5.3.1 pav. Gsm_LPC_Analysis schemos veikimo laikinės diagramos fragmentas

Pateiktame fragmente, taip pat matomos ankstesnės LARc reikšmės (-46, 753, 1508, 1647, -1316, 0, -1049, 1109), kurios įrašomos trečiojo komponento.

Kitas projektavimo etapas po modeliavimo – loginė sintezė.

5.4 LPC komponento loginė sintezė

Atlikus GSM LPC (Gsm_LPC_Analysis) schemos modeliavimo eksperimentus, atliekama modeliuojamo aprašo loginė sintezė. Sintezei naudojamas įrankis Synopsys Design Vision. Sintezuojama į class.db elementinę bazę.

Gautas sintezės rezultatas išsaugomas verilog formatu (gsm_lpc_gate_level.v). Šis rezultatas turi būti vėl modeliuojamas, taip atliekamas eksperimentinio projekto verifikavimas.

5.5 Išvados

- GSM LPC komponento algoritmas pagal pasirinktą modelį perrašytas iš C į VHDL
- Išspręstos nesintezuojamų konstrukcijų problemos

6. LPC KOMPONENTO TYRIMAS

6.1 Tyrimo planas

- Po sintezės gauto gate-level aprašo modeliavimas.

6.2 LPC komponento gate level aprašo modeliavimas

Gautas gate level aprašas suderinamas su modeliavimo įrankiu Cadence NCLaunch. Eksperimento metu nustatyta, jog po loginės sintezės, atliktos Synopsys Design Vision įrankiu, gautas verilog gate level aprašas nėra iškarto modeliuojamas su Cadence NCLaunch. Gaunamos klaidos, dėl nekorektiškų pavadinimų. VHDL apraše naudojami atminties masyvai bei matricos vėliau sintezuojami į trigerius, kurie apjungiami į registrus. Aprašant struktūrinį komponentų sujungimą, trigerių sujungimams naudojami pavadinimai pvz. mem_reg[1][1], mem_reg[1][2] ir toks pavadinimų užrašymas Verilog kalboje nėra leistinas, jei mem_reg nėra masyvas ir tai nėra reikšmės nuskaitymas iš jo. Jei užrašas mem_reg[1][1] reiškia tik pavadinimą, jis turi būti užrašomas naudojant pasvirąjį brūkšnį ‚\' ‚\mem_reg[1][1]'. Ši žymėjimo problema galioja ir signalų pavadinimams. Jei pradiniame apraše komponentai buvo jungiami magistralėmis, tai po sintezės magistralės išskleidžiamos po vieną laidą, kiekvienam laidui suteikiamas pavadinimas pvz. data_in[1], data_in[2]. Toks signalų pavadinimų užrašymas Verilog kalboje taip pat ir kitose pagrindinėse programavimo kalbose taip pat nėra leistinas.

Priklausomai nuo aprašo, gauto po loginės sintezės, dydžio rankinis tokių klaidingų pavadinimų koregavimas užima pakankamai daug laiko ir toks darbo procesas nėra efektyvus, tačiau šio eksperimento metu klaidos ištaisomos rankiniu būdu ir sprendimo būdų neieškoma.

Gate level aprašo modeliavimas atliekamas Cadence NCLaunch įrankiu. Loginės sintezė šiame eksperimente atliekama į class.db elementų biblioteką, todėl atliekant gauto Verilog gate level aprašo modeliavimą, reikia taip pat ir class.v bibliotekos, kurioje Verilog kalba aprašyti class bibliotekos komponentai. Tiek gate lavel Verilog aprašas tiek class.v biblioteka pirmiausia sutransliuojami (komanda compile), po to atliekama elaborate komanda. Modeliavimui atlikti reikalinga testinė programa, generuojanti testinius vektorius bei signalus, todėl po gate level aprašo apdorojimo, reikia sutransliuoti VHDL (arba Verilog) kalba aprašytą testavimo komponentą, kuris kreipiasi į gate level top komponentą ir jam paduoda testinius signalus. NCLaunch įrankis leidžia kartu naudoti tiek VHDL tiek Verilog kalbomis aprašytas schemas

(komponentus), tik prieš tokių komponentų panaudojimą, jie turi būti sutransliuoti (atlikta compile operacija).

Testavimo programai įvykdžius simulate komandą, bei pasirinkus reikiamus signalus, gaunama laikinė diagrama.

Po pirmojo bandymo modeliuoti gate level aprašą, gaunami rezultatai netenkina pradinės funkcijos, gaunama daug neatpažintų signalų (x), kurie parodo, jog galutinė schema neveikia arba veikia nekorektiškai. Gsm_LPC_Analysis schemos pirmasis baigtinis automatas Autocorrelation nustoja veikti jau 6500 ns laiko momentu (Visos Gsm_LPC_Analysis schemos modeliavimo trukmė 141000 ns sinchrosignalo periodui esant 20 ns).

Nesėkmingo bandymo gauti rezultatus, atitinkančius pradinę funkciją (VHDL RTL aprašo modeliavimo rezultatus), priežastys:

- Rašant pradinį modeliuojamą (vėliau sintezuojamą) VHDL RTL aprašą, pagrindinis dėmesys buvo kreipiamas į gaunamus rezultatus. Pagrindinis tikslas buvo gauti rezultatus, atitinkančius C kalba realizuotos programos.
- Nebuvo kreipiamas dėmesys į laiko planavimą (ar įtaisai suspėja atlikti visus reikiamus veiksmus per tam tikrą laiką pvz. sinchrosignalo periodą)
- Nebuvo kreipiamas dėmesys kokią įtaką galutiniam variantui turės duomenų tipų konvertavimo funkcijų (to_signed, to_integer) naudojimas.
- Nebuvo kreipiamas dėmesys ar korektiškai bus atliekamos postūmio operacijos.

Modeliuojant pradinį VHDL aprašą, minėtos problemos paprastai neiškyla. Po loginės sintezės elgsenos aprašas pervedamas į struktūrinį. Visi faktoriai, į kuriuos modeliuojant galima nekreipti dėmesio, po loginės sintezės tampa būtini.

Ieškant nesėkmingo gate level aprašo modeliavimo priežasties, tenka dar kartą modeliuoti pradinį VHDL aprašą ir patikrinti, kokie veiksmai buvo vykdomi 6500 ns laiko momentu. Pagal laikines diagramas pastebima, jog Autoccorrelation baigtinis automatas pereidamas iš būsenos 5 į 6 nustoja veikti. Žemiau pateikiamas minėto automato elgsenos aprašo fragmentas:

```
when 5 => -- scale_2
          busy <= '1';
          en_ref <= '0';
          dataOut <= 0;
          dataOutL <= x"00000000";
```

```
gsm_abs_in <= 0;
gsm_norm_in <= x"00000000";
if scalauto > 0 and scalauto <= 4 then
        rden <= '1';
        wren <= '0';
        addr <= count;
        rdenL <= '0';
        wrenL <= '0';
        addrL <= 0;
        gsm_mult_r_in_a <= dataIn; -- is atminties
        tempv := to_signed(16384,32);
        case scalauto is
        when 2 =>
                tempv := '0' & tempv(31 downto 1);
        when 3 =>
                tempv := "00" & tempv(31 downto 2);
        when 4 =>
                tempv := "000" & tempv(31 downto 3);
        when others =>
                tempv := tempv;
        end case;
        gsm_mult_r_in_b <= to_integer(tempv);
        state <= scale_3;
else
        rden <= '0';
        wren <= '0';
        addr <= 0;
        rdenL <= '1';
        wrenL <= '0';
        addrL <= 0;
        gsm_mult_r_in_a <= 0;
        gsm_mult_r_in_b <= 0;
        count := 0; -- nuresetinam counteri
        countL := 0;
        state <= compute_l_acf;
end if;
```

Fragmentas (fragmentų yra ir daugiau) analizuojamas siekiant rasti nekorektiškas RTL aprašo vietas.

Pagal pateiktą fragmentą, klaidingos vietos gali būti: sąlygos sakinys (`if scalauto > 0 and scalauto <= 4 then`) arba postūmio operacijos išreikštos per masyvų fragmentų sujungimą (`tempv := "000" & tempv(31 downto 3);`). Paminėtos priežastys galimos, bet mažai tikėtinos. Reikia analizuoti visų signalų formavimą dar nuo pradinių automato būsenų.

Analizuojant minėto automato laikines diagramas, gautas modeliuojant gate-level aparašą, tikrinamos nuosekliai visos reikiamos operacijos pagal modeliuojamą VHDL aprašą. Autocorrelation automatas pagal algoritmą pirmiausiai turi rasti didžiausią reikšmę, saugomą masyvę (atminties komponente) so. Stebint laikines diagramas, pastebima, kad kintamojo max reikšmė jau 70ns laiko momentu tampa XXXX (neatpažinta), todėl joks tolimesnis įtaiso darbas neįmanomas.

Atliekant tolimesnį komponento tyrimą, redaguojamas pradinis aprašas, bandant pataisyti blogai sintezės metu interpretuojamas VHDL aprašo vietas. Po kiekvieno VHDL aprašo pakeitimo iš naujo atliekama loginė sintezė, ventilių lygio aprašo pritaikymas modeliavimui bei

modeliavimas. Kiekvieno tokio modeliavimo metu fiksuojamos nekorektiškai veikiančios ar neveikiančios algoritmo vietos, kol ventilių lygio aprašo laikinės diagramos sutampa su modeliuojamo VHDL aprašo laikinėmis diagramomis.

Toliau eksperimentas atliekamas keičiant modeliavimo dažnį (pradinis dažnis 50 MHz, takto periodas 20 ns).

6.3 Tyrimo išvados

- Eksperimento rezultatai parodo, jog rankinis toks procesas reikalauja pastangų, laiko sąnaudų ir yra sudėtingas. Jeigu ir pavyksta iš vieno ar iš kelių bandymų realizuoti modeliuojamą VHDL aprašą, tai dar nereiškia, jog jis bus sintezuojamas ir po sintezės gautas ventilių lygio aprašas bus modeliuojamas.
- Eksperimento rezultatai įrodo, jog C aprašas turi būti apribotas, turi būti vengiama rodyklės tipo kintamųjų, kurie įneša maišaties projektuojant įtaiso elgseną.

7. IŠVADOS

- Automatiniai RTL generavimo įrankiai pramonėje nėra paplitę, daugiau naudojami kaip mokslinių tyrimų objektai. HLS įrankiai automatiniam RTL generavimui naudoja daug sudėtingų algoritmų, todėl tokių įrankių sukūrimas reikalauja daug kaštų ir tokių įrankių, skirtų profesionaliam naudojimui, kaina yra ganėtinai didelė.
- Alternatyvūs HLS įrankiai pigesni, tačiau turi įvairių apribojimų, kurie leidžia HLS panaudoti tik tam tikriems projektuojamo įtaiso fragmentams.
- Siaurą HLS įrankių naudojimą lemia ir prieštaringos specialistų nuomonės. Daugelis aparatūrinės įrangos inžinierių mano, jog geras specialistas rankiniu būdu RTL aprašys geriau ir optimaliau.
- Atliekant eksperimentą, C algoritmas buvo perrašomas į VDL. Susidurta su įvairiomis problemomis: rodyklės tipo kintamieji, adresų aritmetika, nuoseklūs C sakiniai, sudėtingos operacijos, laiko taktų paskaičiavimas, ciklas cikle.
- Siekiant, kad HLS įrankiai būtų plačiau naudojami (būtų panaudojami ir nekomerciniai įrankiai), pradiniam C/C++ aprašui turi būti keliami atitinkami reikalavimai: nenaudoti rekursijos, rodyklės tipo kintamųjų, supaprastinti algoritmo operacijas, net ir programinės įrangos inžinierius rašantis aukšto lygio C/C++ aprašą, turi mąstyti, kokiu principu įtaisas vykdys nurodytus veiksmus.

8. LITERATŪROS SĄRAŠAS

[1]. R. A. Bergamaschi, R. A. O'Connor, L. Stok, M. Z. Moricz, S. Prakash, A. Kuehlmann, D. S. Rao; High-level synthesis in an industrial environment; 1995 <http://domino.research.ibm.com/tchjr/journalindex.nsf/4ac37cf0bdc4dd6a85256547004d47e1/d3d8095dc2eb3b9e85256bfa0067f9a9!OpenDocument>

[2]. Philipe Coussy, Adam Morawiec; High-Level Synthesis from Algorithm to Digital Circuit; 2008 <http://books.google.lt/books?id=lEWRGB5JlHkC&dq=high+level+synthesis&source=gbs_navlinks_s>

[3]. Vocal Technologies Ltd; GSM Speech Coders; 2009 <http://www.vocal.com/speech_coders/gsm_coders.html>

[4]. Thierry Turletti; The GSM Spech Coding; 1996; <http://www.tns.lcs.mit.edu/~turletti/gsm-overview/node6.html>

[5]. Jutta Degener; GSM 06.10 lossy speech compression; 2009 <http://www.quut.com/gsm/>

[6]. Connexions; Linear Predictive Coding in Voice Conversion; 2004 <http://cnx.org/content/m12473/latest/>

[7]. CHStone; A Suite of Benchmark Programs for C-based High-Level Synthesys; 2009 <http://www.ertl.jp/chstone/>

[8]. S.Ahuja, S. K. Shukla, S. T. Gurumani, C. Spackman; „Hardware Coprocessor Synthesys form an ANSI C Specification“; 2009

[9]. Labstic; GAUT - High-Level Synthesis tool**;** <http://www-labsticc.univ-ubs.fr/www-gaut/>

[10]. Design Flow through the SPARK Framework ; University Of California, San Diego; 2003; <http://mesl.ucsd.edu/spark/methodology.shtml>

[11]. Cadence; C-to-Silicon Compiler; 2008; <http://www.cadence.com/rl/Resources/datasheets/C2Silicon_ds.pdf>

[12]. Agility Design Solution; Agility SC Compiler academic edition; 2008; <http://www.msc.rl.ac.uk/europractice/vendors/agility_sc_datasheet_01000_hq_screen.pdf>

[13]. Mentor Graphics; Catapult C Synthesis; 2009 <http://www.saros.co.uk/images/upload/file/pdf/Catapult_DS.pdf>

[14]. High-Level Synthesis; <http://www.ida.liu.se/~petel/SysSyn/lect3.frm.pdf>

[15]. CebaTech; C2R Compiler Overview; 2007 <http://www.cebatech.com/assets/files/C2R%20Compiler%20DataSheet%20final%20022208f.pdf>

[16]. John P. Elliott; Understanding behavioral synthesis; 2000; p10-13

9. PRIEDAI

9.1 GSM LPC komponento C kalbos aprašas

lpc.c

```
/*
+--------------------------------------------------------------------------+
| CHStone : a suite of benchmark programs for C-based High-Level Synthesis |
| ======================================================================== |
|                                                                          |
| * Collected and Modified : Y. Hara, H. Tomiyama, S. Honda,               |
|                            H. Takada and K. Ishii                        |
|                            Nagoya University, Japan                      |
|                                                                          |
| * Remark :                                                               |
|    1. This source code is modified to unify the formats of the benchmark |
|       programs in CHStone.                                              |
|    2. Test vectors are added for CHStone.                                |
|    3. If "main_result" is 0 at the end of the program, the program is    |
|       correctly executed.                                                |
|    4. Please follow the copyright of each benchmark program.            |
+--------------------------------------------------------------------------+
*/
/*
 * Copyright 1992 by Jutta Degener and Carsten Bormann, Technische
 * Universitaet Berlin.  See the accompanying file "COPYRIGHT" for
 * details.  THERE IS ABSOLUTELY NO WARRANTY FOR THIS SOFTWARE.
 */

#include "private.h"
#include "add.c"


/*
 *  4.2.4 .. 4.2.7 LPC ANALYSIS SECTION
 */

/* 4.2.4 */


void
Autocorrelation (word * s /* [0..159]      IN/OUT  */, longword * L_ACF /*
[0..8]       OUT     */)
/*
 *  The goal is to compute the array L_ACF[k].  The signal s[i] must
 *  be scaled in order to avoid an overflow situation.
 */
{
  register int k, i;

  word temp, smax, scalauto, n;
  word *sp, sl;
```

```
  /*  Search for the maximum.
       */
  smax = 0;
  for (k = 0; k <= 159; k++)
  {
      temp = GSM_ABS (s[k]);
      if (temp > smax)
        smax = temp;
  }

  /*  Computation of the scaling factor.
   */
  if (smax == 0)
     scalauto = 0;
  else
     scalauto = 4 - gsm_norm ((longword) smax << 16); /* sub(4,..) */

  if (scalauto > 0 && scalauto <= 4)
  {
       n = scalauto; // n [1..4]           kad cia patekti galimos gsm_norm
reiksmes  0, 1, 2, 3
     for (k = 0; k <= 159; k++)
           s[k] = GSM_MULT_R (s[k], 16384 >> (n - 1));  // 2^14 >> (n-1)
  }

  /*  Compute the L_ACF[..].
   */
  {
    sp = s;
    sl = *sp;

#define STEP(k)    L_ACF[k] += ((longword)sl * sp[ -(k) ]);

#define NEXTI      sl = *++sp
    for (k = 9; k >= 0; k--)
         L_ACF[k] = 0;

    STEP (0);
    NEXTI;
    STEP (0);
    STEP (1);
    NEXTI;
    STEP (0);
    STEP (1);
    STEP (2);
    NEXTI;
    STEP (0);
    STEP (1);
    STEP (2);
    STEP (3);
    NEXTI;
    STEP (0);
    STEP (1);
    STEP (2);
    STEP (3);
    STEP (4);
    NEXTI;
    STEP (0);
    STEP (1);
```

```
    STEP (2);
    STEP (3);
    STEP (4);
    STEP (5);
    NEXTI;
    STEP (0);
    STEP (1);
    STEP (2);
    STEP (3);
    STEP (4);
    STEP (5);
    STEP (6);
    NEXTI;
    STEP (0);
    STEP (1);
    STEP (2);
    STEP (3);
    STEP (4);
    STEP (5);
    STEP (6);
    STEP (7);

    for (i = 8; i <= 159; i++)
      {

      NEXTI;

      STEP (0);
      STEP (1);
      STEP (2);
      STEP (3);
      STEP (4);
      STEP (5);
      STEP (6);
      STEP (7);
      STEP (8);
      }

    for (k = 9; k >= 0; k--)
      L_ACF[k] <<= 1;

  }
  /*   Rescaling of the array s[0..159]
   */
  if (scalauto > 0)
      for (k = 160; k >= 0; k--)
      *s++ <<= scalauto;
}

/* 4.2.5 */

void
Reflection_coefficients (longword * L_ACF /* 0...8           IN        */,
register word * r /* 0...7        OUT     */)
{
  register int i, m, n;
  register word temp;
  register longword ltmp;
  word ACF[9];                /* 0..8 */
```

```
word P[9];                    /* 0..8 */
word K[9];                    /* 2..8 */

/*  Schur recursion with 16 bits arithmetic.
 */

if (L_ACF[0] == 0)
  {
    for (i = 8; *r++ = 0; i--);
    return;
  }

temp = gsm_norm (L_ACF[0]);
for (i = 0; i <= 8; i++)
  ACF[i] = SASR (L_ACF[i] << temp, 16);

/*   Initialize array P[..] and K[..] for the recursion.
 */

for (i = 1; i <= 7; i++)
  K[i] = ACF[i];
for (i = 0; i <= 8; i++)
  P[i] = ACF[i];

/*   Compute reflection coefficients
 */
for (n = 1; n <= 8; n++, r++)
  {

    temp = P[1];
    temp = GSM_ABS (temp);
    if (P[0] < temp)
    {
      for (i = n; i <= 8; i++)
        *r++ = 0;
      return;
    }

    *r = gsm_div (temp, P[0]);

    if (P[1] > 0)
    *r = -*r;          /* r[n] = sub(0, r[n]) */
    if (n == 8)
    return;

    /*  Schur recursion
     */
    temp = GSM_MULT_R (P[1], *r);
    P[0] = GSM_ADD (P[0], temp);

    for (m = 1; m <= 8 - n; m++)
    {
      temp = GSM_MULT_R (K[m], *r);
      P[m] = GSM_ADD (P[m + 1], temp);

      temp = GSM_MULT_R (P[m + 1], *r);
      K[m] = GSM_ADD (K[m], temp);
    }
  }
```

```
}

/* 4.2.6 */

void
Transformation_to_Log_Area_Ratios (register word * r /* 0..7    IN/OUT */ )
/*
 *  The following scaling for r[..] and LAR[..] has been used:
 *
 *  r[..]   = integer( real_r[..]*32768. ); -1 <= real_r < 1.
 *  LAR[..] = integer( real_LAR[..] * 16384 );
 *  with -1.625 <= real_LAR <= 1.625
 */
{
  register word temp;
  register int i;


  /* Computation of the LAR[0..7] from the r[0..7]
   */
  for (i = 1; i <= 8; i++, r++)
    {

      temp = *r;
      temp = GSM_ABS (temp);

      if (temp < 22118)
        temp >>= 1;
      else if (temp < 31130)
        temp -= 11059;
      else
      {
        temp -= 26112;
        temp <<= 2;
      }

      *r = *r < 0 ? -temp : temp;
    }
}

/* 4.2.7 */

void
Quantization_and_coding (register word * LAR /* [0..7]       IN/OUT  */ )
{
  register word temp;
  longword ltmp;


  /*  This procedure needs four tables; the following equations
   *  give the optimum scaling for the constants:
   *
   *  A[0..7] = integer( real_A[0..7] * 1024 )
   *  B[0..7] = integer( real_B[0..7] *  512 )
   *  MAC[0..7] = maximum of the LARc[0..7]
   *  MIC[0..7] = minimum of the LARc[0..7]
   */

#     undef STEP
```

```
#     define      STEP( A, B, MAC, MIC )        \
            temp = GSM_MULT( A,   *LAR ); \
            temp = GSM_ADD(  temp,   B ); \
            temp = GSM_ADD(  temp, 256 ); \
            temp = SASR(     temp,   9 ); \
            *LAR  =  temp>MAC ? MAC - MIC : (temp<MIC ? 0 : temp - MIC); \
            LAR++;

  STEP (20480, 0, 31, -32);
  STEP (20480, 0, 31, -32);
  STEP (20480, 2048, 15, -16);
  STEP (20480, -2560, 15, -16);

  STEP (13964, 94, 7, -8);
  STEP (15360, -1792, 7, -8);
  STEP (8534, -341, 3, -4);
  STEP (9036, -1144, 3, -4);

#     undef STEP
}

void
Gsm_LPC_Analysis (word * s /* 0..159 signals          IN/OUT   */,   word *
LARc /* 0..7   LARc's        OUT     */)
{
  longword L_ACF[9];

  Autocorrelation (s, L_ACF);
  Reflection_coefficients (L_ACF, LARc);
  Transformation_to_Log_Area_Ratios (LARc);
  Quantization_and_coding (LARc);
}
```

add.c

```
/*
+------------------------------------------------------------------------+
| CHStone : a suite of benchmark programs for C-based High-Level Synthesis |
| ====================================================================== |
|                                                                        |
| * Collected and Modified : Y. Hara, H. Tomiyama, S. Honda,             |
|                            H. Takada and K. Ishii                      |
|                            Nagoya University, Japan                    |
|                                                                        |
| * Remark :                                                             |
|    1. This source code is modified to unify the formats of the benchmark |
|       programs in CHStone.                                             |
|    2. Test vectors are added for CHStone.                              |
|    3. If "main_result" is 0 at the end of the program, the program is  |
|       correctly executed.                                              |
|    4. Please follow the copyright of each benchmark program.           |
+------------------------------------------------------------------------+
*/
/*
 * Copyright 1992 by Jutta Degener and Carsten Bormann, Technische
 * Universitaet Berlin.  See the accompanying file "COPYRIGHT" for
```

```
 * details.  THERE IS ABSOLUTELY NO WARRANTY FOR THIS SOFTWARE.
 */

/*  $Header:  /home/kbs/jutta/src/gsm/gsm-1.0/src/RCS/add.c,v  1.5  1994/12/30
22:35:09 jutta Exp $ */

/*
 *  See private.h for the more commonly used macro versions.
 */

#define     saturate(x)       \
      ((x) < MIN_WORD ? MIN_WORD : (x) > MAX_WORD ? MAX_WORD: (x))

word
gsm_add (word a, word b)
{
  longword sum;
  sum = (longword) a + (longword) b;
  return saturate (sum);
}

word
gsm_mult (word a, word b)
{
  if (a == MIN_WORD && b == MIN_WORD)
    return MAX_WORD;
  else
    return SASR ((longword) a * (longword) b, 15);
}

word
gsm_mult_r (word a, word b)
{
  longword prod;
  if (b == MIN_WORD && a == MIN_WORD)
    return MAX_WORD;
  else
    {
      prod = (longword) a *(longword) b + 16384; // 2^14
      prod >>= 15;
      return prod & 0xFFFF;
    }
}

word
gsm_abs (word a)
{
  return a < 0 ? (a == MIN_WORD ? MAX_WORD : -a) : a;
}

const unsigned char bitoff[256] = {
  8, 7, 6, 6, 5, 5, 5, 5, 4, 4, 4, 4, 4, 4, 4, 4,
  3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
  2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
  2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
```

```
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}; // sito masyvo reiksmes
parodo pirmo reiksminio bito pozicija 8 bitu intervale
  // pvz 255 - 1111 1111 - pirmo reiksminio (is kaires i desine) pozicija 0
  //       127 -  0111 1111 - -//- pozicija 1

word
gsm_norm (longword a)
/*
 * the number of left shifts needed to normalize the 32 bit
 * variable L_var1 for positive values on the interval
 *
 * with minimum of
 * minimum of 1073741824  (01000000000000000000000000000000) and
 * maximum of 2147483647  (01111111111111111111111111111111)
 *
 *
 * and for negative values on the interval with
 * minimum of -2147483648 (-10000000000000000000000000000000) and
 * maximum of -1073741824 (-1000000000000000000000000000000).
 *
 * in order to normalize the result, the following
 * operation must be done: L_norm_var1 = L_var1 << norm( L_var1 );
 *
 * (That's 'ffs', only from the left, not the right..)
 */
{

  if (a < 0)
  {
    if (a <= -1073741824)
         return 0;
    a = ~a;
  }
  return
     a & 0xffff0000 ? // ar a skaiciaus reiksminiai bitai vyresniuose 16 bitu
is 32,  buvo atlikta 16 postumiu i kaire
         (a & 0xff000000 ? // jei a skaiciaus reiksminiai bitai vyresniuose 8
bituose is 32
            -1 + bitoff[0xFF & (a >> 24)]
          : 7 + bitoff[0xFF & (a >> 16)]
         )
       : (a & 0xff00 ?
             15 + bitoff[0xFF & (a >> 8)]
                 : 23 + bitoff[0xFF & a]
             );
}

word
gsm_div (word num, word denum)
{
  longword L_num, L_denum;
  word div = 0;
```

```
  int k = 15;

  L_num = num;
  L_denum = denum;
  /* The parameter num sometimes becomes zero.
   * Although this is explicitly guarded against in 4.2.5,
   * we assume that the result should then be zero as well.
   */

  if (num == 0)
    return 0;

  while (k--)
    {
      div <<= 1;
      L_num <<= 1;

      if (L_num >= L_denum)
      {
        L_num -= L_denum;
        div++;
      }
    }

  return div;
}
```