**KAUNO TECHNOLOGIJOS UNIVERSITETAS**
**INFORMATIKOS FAKULTETAS**
**PROGRAMŲ INŽINERIJOS KATEDRA**

Andrius Garliauskas

# Procesorinio komponento bendrinimo tyrimas: sintezės aspektai

## Magistro baigiamasis darbas

Darbo vadovas:
doc. Giedrius Ziberkas

Kaunas, 2007

**KAUNO TECHNOLOGIJOS UNIVERSITETAS**
**INFORMATIKOS FAKULTETAS**
**PROGRAMŲ INŽINERIJOS KATEDRA**

Andrius Garliauskas

# Procesorinio komponento bendrinimo tyrimas: sintezės aspektai

## Magistro baigiamasis darbas

Recenzentas
Lekt. R.Damaševičius
2007-05-25

Darbo vadovas
doc. Giedrius Ziberkas
2007-05-25

Atliko
IFM-1/5 gr. Stud.
Andrius Garliauskas
2007-05-25

Kaunas, 2007

# Procesorinio komponento bendrinimo tyrimas: sintezės aspektai


## Santrauka

Šiame darbe tiriama, kaip kinta procesorinių komponentų (ALU įrenginio, duomenų registro, instrukcijų registro bei programos skaitiklio registro) techniniai parametrai keičiant procesoriaus apdorojamos informacijos dydį bitais. Su Synopsys programine įranga buvo susintezuoti komponentų VHDL programavimo kalba parašyti aprašai, taip gaunant apytikrius procesorinių komponentų techninius parametrus (kristalo plotas, schemai realizuoti reikalingas elementų kiekis, schemos vidinis galingumas). Šiam tikslui buvo susintezuoti procesoriniai komponentai, kurie apdorja 8, 16, 32 ir 64 bitų ilgio duomenis. Tai dažniausiai sutinkami duomenų ilgiai, su kuriais tenka susidurti dabartiniams procesoriams.

Iš gautų sintezės rezultatų padaryta išvada, kad keičiant apdorojamų duomenų ilgį bitais galima preliminariai numatyti susintezuotos schemos kristalo plotą, schemai realizuoti reikalingą loginių elementų kiekį, vidinį schemos galingumą. Buvo nustatyta, kad didėjant apdorojamos informacijos kiekiui, padidėja įrenginių, kurie galėtų apdoroti šią informaciją, kristalo plotas, jiems realizuoti reikalingas loginių elementų kiekis bei vidinis galingumas.

The research of generalization of CPU’s components: aspects of synthesis

Summary

This work examines how the change of processed data length influence the technical parameters of processor’s components such as arithmetic logic unit (ALU), data registers, instruction registers and program counter registers. It was done by using Synopsis software which enabled the synthesis of the needed components. The synthesis results showed information about occupied area, the number of cells and the internal voltage of the synthesised scheme. There were chosen the most common length of processed data (8, 16, 32 and 64 bit).

The results of synthesis showed, that it is possible to predict the results of synthesis by changing the length of the processed data. The longer word of information must be processed by components, the larger area is needed for implementation of the processor components, more logical element are needed to implement the components and the greater internal voltage of the scheme will be.

# Turinys

# 1. Įvadas

Vis daugiau žmogaus darbo jėgą keičia mašinos. Tai ypač naudinga tada, kai tuos darbus reikia atlikti agresyviose, žmogui kenksmingose aplinkose, arba tie darbai reikalauja sudėtingų skaičiavimų. Visą skaičiavimą mašinose atlieka procesorius. Kadangi darbo, kurį reikia atlikti mašinoms, specifika gali būti įvairi, nuo to priklauso ir procesoriaus architektūra. Projektuojant procesorių, reikia atsižvelgti į jam keliamus reikalavimus: pagaminimo kainą, kokį maksimalų plotą gali užimti kristalas, jo techninius parametrus. Kaina priklauso nuo to, kokie elementai ir koks kiekis įeina į procesorių.

Kaina taip pat priklauso nuo procesoriaus projektavimo sąnaudų. Projektavimas yra ilgas procesas, kuris užima daug laiko. Šiam procesui pagreitinti kuriami bendriniai komponentai (ne vien tik procesoriui), kurie gali būti naudojami daug kartų. Šie komponentai sukuriami taip, kad keičiant jų parametrus, juos galima pritaikyti įvairių sistemų projektavimui.

Elementų konfigūracijos priklauso nuo užduočių, kurias reikia atlikti, specifikos. Todėl yra sukurta didelė įvairovė procesorių, kurie vienas nuo kito skiriasi savo vidine architektūra bei juose esančiais elementais. Tie elementai vienas nuo kito gali skirtis bitų dydžiu, t.y. parametru, kuris nusako su kokio dydžio duomenimis tas elementas gali atlikti savo užduotį. Pavyzdžiui procesoriaus registrų dydį procesoriaus projektuotojai parenka pagal tai, su kokio dydžio duomenimis procesoriui reikės dirbti, bei kokia tų registrų kaina, kadangi kuo didesni registrai, tuo brangiau jie kainuoja. Taip pat nuo registrų dydžio gali priklausyti jų greitaveika, kas yra irgi labai svarbu. Šis darbas, pasinaudojus tam tikra programine įranga bei jos pagalba gautais procesorinių komponentų sintezės rezultatais, tiria, kaip kinta procesoriaus komponentų techniniai parametrai, keičiant jų apdorojamos informacijos bitų dydį.

# 2. Bendriniai komponentai

Vis labiau tobulėjant gamybos technologijoms, silicio kristalo talpumas dvigubėja kas 18 mėnesių. Tai leidžia kompanijoms kurti vis sudėtingesnes sistemas ir talpinti jas į vienlusčius kristalus (SOC). Tačiau tokių sudėtingų sistemų kūrimas užima vis daugiau laiko, o tai lėtina puslaidininkinės pramonės augimą. Šiai problemai išspręsti buvo pasirinkti trys būdai: platformos, daugkartinis panaudojimas, sintezė.

Naudojantis pirmuoju būdu, puslaidininkių prekiautojai tiekia universalias SOC platformas, kurios susideda iš vieno ar kelių pagrindinių procesorių ir iš daugelio mažesnių periferinių procesorių, kurie skirti įėjimų/išėjimų protokolų bei kodavimo/dekodavimo funkcijų valdymui.

Naudojantis daugkartiniu panaudojimu, silicio kristalo modeliai surenkami iš skirtingų blokų, kurie jau yra sukurti ir buvo panaudoti.

Naudojantis sintezės būdu, silicio kristalai susintezuojami, jų funkcijas aprašant bendrinėmis kalbomis (tokiomis kaip C), po to jas pritaikant tam tikrai programinei įrangai bei gamybos technologijoms.

## 2.1 Daugkartinis panaudojimas

Daugkartinis panaudojimas sukūrė atskirą puslaidininkinės pramonės šaką, kuri vadinama IP (Intelectual property – intelektualinė nuosavybė) verslu. Siauresne prasme, IP yra virtualus komponentas su gerai aprašytomis funkcijomis, naudojimo metodais bei įrankiais, kurie leistų naudotis šiuo komponentu. Kitaip tariant, virtualus komponentas yra sukurtas ir paruoštas naudojimui bendrinis komponentas, kuris gali būti naudojamas sistemos kūrimui.

Bendrinių komponentų daugkartinis panaudojimas pasireiškia tuo, kad jie, šiek tiek modifikuojant, naudojami įvairiose sistemose.

## 2.2 Daugkartinio panaudojimo problemos

### IP bibliotekos

Viena iš bendrinimo problemų yra kaip apibrėžti bendrinio komponento (IP) funkcionalumą, stilių ir tipą. IP gali būti kaip paruoštas vartojimui produktas arba kaip parametrizuojamas IP. Didėjantis IP neapibrėžtumas sukelia sunkumus optimizuojant, tikrinant ir testuojant IP.

Kita iškylanti problema yra originalaus kodo ar schemos pritaikymas daugkartiniam naudojimui. Kadangi dauguma projektuotojų savaip modifikuoja jam reikalingą komponentą, todėl IP turi pasižymėti išsiplečiamumu.

Problema taip pat iškyla IP pateikime. IP gali būti pateikiamas kaip techninis IP arba programinis IP. Griežtas IP yra nuspėjamas, bet sunkiai plintantis. T.y. jis yra lengvai charakterizuojamas ir naudojamas, bet jį sunku modifikuoti ir sunkiai pritaikomas skirtingose technologijose. Kita vertus „lengvas“ IP gali būti taikomas skirtingose technologijose, bet jo elgsena nėra lengvai nuspėjama, todėl prieš naudojimą jis turi būti sintezuojamas bei patikrinamas. Pagrindinis tikslas yra sukurti nuspėjamą IP, kuris tuo pačiu metu būtų lengvai pritaikomas skirtingose technologijose.

### Dokumentacija

Daugkartinio naudojimo IP kūrimas užima daug laiko. Didelę laiko dalį atima dokumentacijos ruošimas. Į dokumentaciją įtraukiama specifikacijos, vartojimo instrukcijos, modeliavimo ir pakartotinio naudojimo modeliai, testiniai rinkiniai. Vartotojo atžvilgiu gera dokumentacija turi būti glausta, išsami ir išbaigta.

### Kokybės patikimumas

Dauguma lustus gaminančių šiuolaikinių kompanijų vengia naudotis nepažįstamų tiekėjų siūlomais IP. Taip yra todėl, kad IP tiekėjų sukurti programiniai IP dažnai niekada nebuvo įdiegti į silicio kristalą, todėl nėra žinomas jų patikimumas bei veikimas.

Kita iškylanti problema yra tokia, kad sunku patikrinti ar IP veikia teisingai veikia su duotomis reikšmėmis bei parametrais.

Galų gale sunku patikrinti ar testavimo metodai bei testiniai rinkiniai, sukurti IP tiekėjų, gali gerai patikrinti IP veikimą kaip atskiro komponento ir komponento, kuris yra integruotas į sistemą.

**Standartizavimas**

Pagrindinis standartizavimo tikslas yra palengvinti IP daugkartinio panaudojimo procesą. Šiuolaikiniai IP vartotojai dažnai užtrunka keletą mėnesių IP vertinimui. Jeigu reikia įvertinti skirtingų tiekėjų sukurtus IP, užduotis tampa dar sudėtingesnė, kadangi skirtingi tiekėjai gali turėti skirtingus specifikacijos bei dokumentacijos stilius. Kad palyginti skirtingų tiekėjų siūlomus IP, vartotojui tenka atlikti eilę konvertavimo procedūrų, o tai labai vargina ir atima daug laiko.

**Tinkamo IP paieška**

Kad iš daugelio sukurtų IP schemų kūrėjai išsirinktų sau tinkamą, reikalingas mechanizmas reikiamo IP paieškai. Pirma, schemų kūrėjai turi žinoti kokios schemų alternatyvos egzistuoja ir kuris IP atitinka tas alternatyvas. Antra, schemų kūrėjai turi patikrinti skirtingų tiekėjų, bet panašaus funkcionalumo IP, kad galėtų išsirinkti sau labiausiai tinkamą.

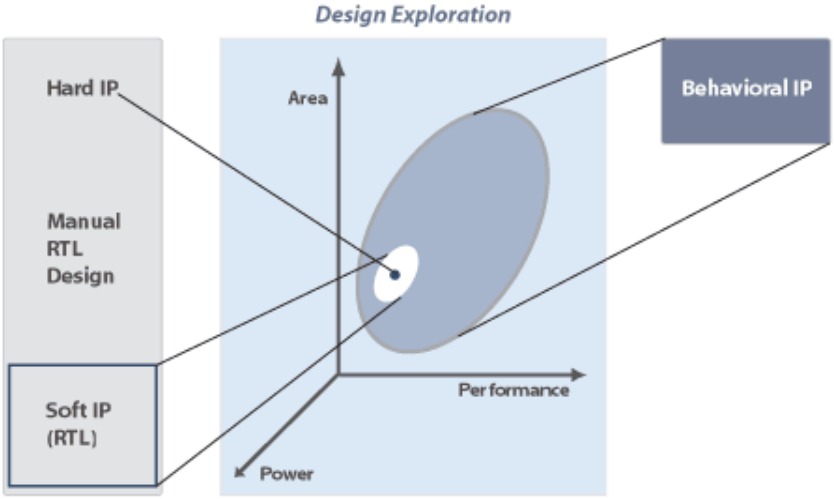
**Integravimas į sistemą**

Kad sėkmingai panaudoti jau sukurtą IP, schemų kūrėjai turi netik pilnai suprasti IP specifikacijas, bet jie taip pat priklauso nuo IP tiekėjų siūlomų IP palaikymo paslaugų, tokių kaip IP rinkinio papildymų tiekimas, testavimo įrankių bei jų modifikacijų papildymas.

## 2.3 Bendrinio komponento sintezavimas

Sukurtas bendrinis komponentas turi būti patikrinamas. Kad nereikėtų be reikalo eikvoti gamybai reikalingų resursų (sukurtas komponentas gali veikti netinkamai), sukurtas bendrinis komponentas yra sintezuojamas. Tai atlikti padeda speciali programinė įranga, kuri vadinama sintezatoriumi. Sintezatoriaus pagalba sistemų kūrėjai gali susintezuoti jiems reikalingą schemą bei ją ištirti. Tai leidžia greičiau sukurti bendrinius komponentus, kurie vėliau gali būti

panaudojami įvairių schemų komplektavimui. Kiekvieno tipo bendrinis komponentas, kuris naudoja aukštesnio lygio abstrakciją, padidina savo daugkartinį panaudojimą.




1 pav. Bendrinių komponentų paplitimas

„Techninis“ bendrinis komponentas („Hard IP“) yra platinamas fiziniame lygyje. Jis yra skirtas tam tikram technologiniam procesui, proceso mazgui. Šio komponento funkcionalumas yra griežtai fiksuotas. Dauguma bendrinių procesorių yra pateikiami šiame lygmenyje.

„Programinis“ bendrinis komponentas („Soft IP“) platinamas registrų perdavimo lygmenyje (RTL). Šio lygmens abstrakcija padidina bendrinio programinio komponento daugkartinio panaudojimo galimybes palyginus su techniniu komponentu. Deja, dėl sunkumų, iškylančių aprašant kanalų struktūras RTL lygmeniu, programinio bendrinio komponento daugkartinis panaudojimas yra gana ribotas.

Elgsenos bendrinis komponentas („Behavioral IP“) padidina abstrakcijos lygmenį ir smarkiai padidina daugkartinio panaudojimo sritį. Elgsenos komponentas aprašomas aukšto lygmens programavimo kalba SystemC. Kadangi SystemC yra C++ programavimo kalbos klasės biblioteka, kuri C++ kalbą papildo techniniais aprašais, todėl su ja galima aprašyti hierarchiją, protokolus ir t.t. Sistemos aprašas, naudojamas sintezavimui, neturi sinchronizavimo duomenų. Vietoj to, apribojimai, kuriuos nurodo sintezuotojas, verčia sintezatorių sukurti greitesnę, mažesnę ar mažiau energijos reikalaujančią RTL realizaciją.

1 lentelė. Bendrinių komponentų palyginimas

| Elgsenos, programinio ir techninio IP palyginimas | | | |
|---|---|---|---|
| | **Elgsenos IP** | **Programinis IP** | **Techninis IP** |
| **Abstrakcija** | Elgsenos | RTL | Fizinis |
| **Kalba** | SystemC | Verilog / VHDL | GDSII |
| **Proceso tech.** | Kintamas | Kintamas | Fiksuotas |
| **Greitis** | Kintamas | Ribotai kintamas | Fiksuotas |
| **Veikimas** | Parametrizuojamas | Parametrizuojamas ribotai | Fiksuotas |

# 3. Procesorius

Visi procesoriai, nepriklausomai nuo jų techninių skirtumų, skirti komandų sekos, vadinamos programa, vykdymui. Komanda vykdoma keturiais žingsniais: užkrovimas, dekodavimas, komandos vykdymas bei įrašymas į atmintį.

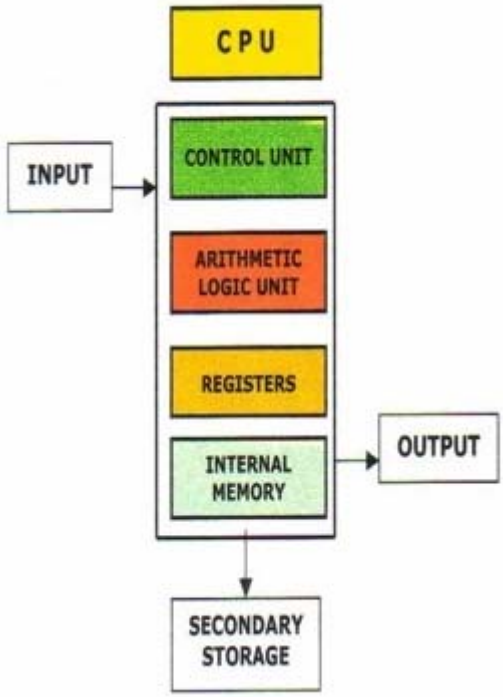
Pirmas žingsnis yra komandos užkrovimas. Šiuo žingsniu iš programos atmintinės yra paimama komanda, kurią reikia vykdyti. Šios komandos adresas yra saugomas programos skaitiklio registre. Tokiu budu procesoriui "nurodoma" kokią komandą reikia vykdyti.

Dekodavimo etape, komanda yra dekoduojama. Užkrautos komandos kodas sudarytas iš skaitmenų dvejetainiame sistemoje. Visa komanda yra suskaldoma į dalis, kurios turi skirtingas reikssmes. Viena tokios komandos dalis nusako kokią operaciją reikia atlikti. Likusioji komandos dalis gali nusakyti operandų reikšmes arba adresus, kur jie yra įrašyti.

Sekantis žingsnis yra komandos vykdymas. Šiame etape yra paleidžiamos atitinkamos procesoriaus dalys, kurios reikalingos operacijos atlikimui. Pačią operaciją atlieka procesoriaus kompnentas ALU (aritmetinis loginis įrenginys), kuriame yra elektroninės schemos, kurios leidžia atlikti atitinkamas operacijas. Į šio įrenginio įėjimus paduodami operandai su kuriais reikia atlikti operaciją. Išėjime yra gaunamas atliekamos operacijos rezultatas.

Paskutiniuoju žingsniu atliktos operacijos rezultatai įrašomi į atmintį. Dažniausiai tai buna procesoriaus vidiniai registrai, kuriuose talpinami duomenys tam, kad vykdant sekančias operacijas butų galima šiuos duomenis panaudoti, taip pagreitinant darbą. Atskirais atvejais rezultatai gali buti talpinami į kokią nors išorinę atmintį, kuri yra lėtesnė, bet pigesnė bei didesnė.

## 3.1 Procesoriaus pagrindinės dalys




2 pav. Procesoriaus sudėtinės dalys

2 pav. parodytos standartinio procesoriaus sudėtinės dalys.

Control unit – valdantysis modulis

Arithmetic Logic Unit – aritmetinis loginis modulis

Registers – registrai

Internal memory – vidinė atmintis

Input – įėjimas

Output – išėjimas

Secondary storage – išorinė atmintis

Valdantysis modulis kontroliuoja visos sistemos darbą.

Aritmetinis loginis modulis atlieka aritmetines operacijas (sudėtį, atimtį, daugybą, dalybą), bei logines operacijas (loginę sudėtį, loginę daugybą, neigimą).

Registruose talpinami duomenys su kuriais procesorius atlieka operacijas, bei pačios operacijos, kurias reikia atlikti

## 3.2 Valdantysis modulis

Valdantysis modulis – tai elektroninė schema, kuri valdo informacijos srautus procesoriuje, bei koordinuoja kitų procesoriuje esančių įrenginių darbą. Tai lyg „smegenys smegenyse“, kadangi valdantysis modulis kontroliuoja procesoriaus darbą, o pats procesorius kontroliuoja viso kompiuterio darbą.

Funkcijos, kurias atlieka valdantysis modulis, labai priklauso nuo vidinės procesoriaus architektūros, kadangi šis modulis realizuoja visos architektūros darbą. Įprastuose procesoriuose, kurie vykdo x86 komandas, valdantysis modulis valdo tokias užduotis kaip komandos užkrovimas, jos dekodavimas, komandos vykdymo valdymas bei rezultatų įrašymas į atmintį. RISC (reduced instruction set computer – sumažinto komandų rinkinio kompiuteris) branduolį turinčiuose procesoriuose valdantysis modulis atlieka žymiai daugiau operacijų. Jis valdo x86 komandų konvertavimą į RISC mikrokomandas, kitiems procesoriaus įrenginiams nurodo kuriuo laiko momentu reikia atlikti atitinkamą mikrokomandą, užtikrina, kad procesoriaus įrenginiams įvykdžius mikrokomandą, gauti rezultatai nukeliautų ten kur reikia.

Kai kuriuose procesoriuose valdantysis modulis gali buti padalintas į kelis modulius (planavimo modulis, kuris suplanuoja mikrokomandų vykdymo tvarką, rezultatų tvarkymo modulis, kuris reguliuoja rezultatų srautus), priklausomai nuo darbo, kurį reikia atlikti, sudėtingumo.

## 3.3 ALU įrenginys

Visas reikalingas operacijas atlieka ALU įrenginys. ALU įrenginys per įėjimo registrus gauna duomenis ( įėjimo registrai duomenys gauna tiesiogiai iš atminties). Šie duomenys yra apdorojami, t.y. su jais atliekami tam tikri veiksmai, o atliktos operacijos rezultatai talpinamas ALU išėjimo registruose. Vėliau iš šių registrų rezultatai perrašomi į atmintį.

ALU įrenginio darbą valdo valdymo modulis. Šis įrenginys nurodo kokią operaciją su duomenimis atlikti ALU įrenginiui, kada jam paimti duomenis ir kada perduoti atliktos operacijos rezultatus į atmintį.

Duomenys, kuriuos apdoroja ALU įrenginys, gali būti įvairaus ilgio. Kokio ilgio duomenis gali apdoroti ALU įrenginys priklauso nuo to, kokio bitų ilgio duomenis gali priimti ALU įrenginio įėjimai.

## 3.4 ALU atliekamos operacijos

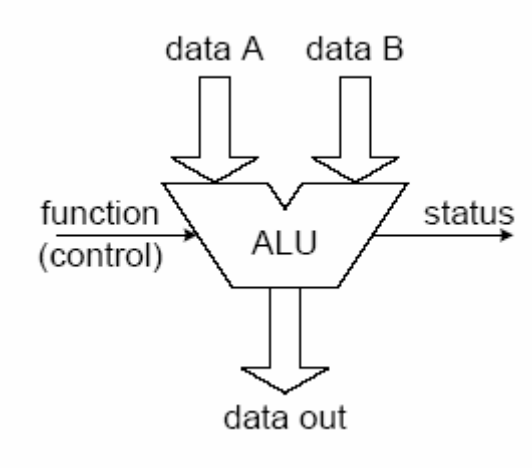
2 lentelėje pateikiamos ALU įrenginio atliekamos operacijos, operacijų kodai bei operacijų išraiškos:

2 lentelė. ALU atliekamos operacijos

| | | | **Output Data** | | **Output Flags** | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Mnemonic** | **Opcode** | **Description** | Result R | Temp_Out | Carry (C) | Zero (Z) | TC Overflow (V) | Negative (N) | Signed (S) | **Implementation** |
| ADD | 0 0 0 0 | Add without Carry | A + B | | Carry(A+B) | set if R==0 | MSB(A) MSB(B) /MSB(R) + /MSB(A) /MSB(B) MSB(R) | MSB | N xor V | addsub |
| ADC | 0 0 0 1 | Add with Carry | A + B + Carry | | Carry(A + B + Carry) | set if R==0 | MSB(A) MSB(B) /MSB(R) + /MSB(A) /MSB(B) MSB(R) | MSB | N xor V | addsub |
| SUB | 0 0 1 0 | Subtract without Carry | A - B | | Carry(A - B) | set if R==0 | MSB(A) /MSB(B) /MSB(R) + /MSB(A) MSB(B) MSB(R) | MSB | N xor V | addsub |
| SBC | 0 0 1 1 | Subtract with Carry | A - B - Carry | | Carry(A - B - Carry) | set if R==0 | MSB(A) /MSB(B) /MSB(R) + /MSB(A) MSB(B) MSB(R) | MSB | N xor V | addsub |
| MUL | 0 1 0 0 | Multiply Unsigned | LowWord(A * B) | HighWord(A * B) | MSB(Temp_Out) | set if (R==0 && Temp_Out==0) | | | N xor V | mult |
| MULS | 0 1 0 1 | Multiply Signed | LowWord(A * B) | HighWord(A * B) | MSB(Temp_Out) | set if (R==0 && Temp_Out==0) | | | N xor V | mult |
| DIV | 0 1 1 0 | Divide Signed | (Temp_In A) / B | | | set if R==0 | | | N xor V | divider |
| DIVEND | 0 1 1 1 | Divide Signed End | (Temp_In A) / B | | | set if R==0 | | | N xor V | divider |
| EOR | 1 0 0 0 | Exclusive OR | A xor B | | | set if R==0 | | MSB | N xor V | xor |
| AND | 1 0 0 1 | Logical AND | A and B | | | set if R==0 | | MSB | N xor V | and |
| OR | 1 0 1 0 | Logical OR | A or B | | | set if R==0 | | MSB | N xor V | or |
| NEG | 1 0 1 1 | Two's Complement | $0000 - B | | set if R!=0 | set if R==0 | | MSB | N xor V | addsub |
| SH | 1 1 0 0 | Shift | B << A | | | set if R==0 | | MSB | N xor V | ash |
| SHA | 1 1 0 1 | Arithmetic Shift | B << A | | | set if R==0 | | MSB | N xor V | ash |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| ROR | 1 1 1 0 | Rotate Right through Carry | ROR B, Carry | | | set if R==0 | | MSB | N xor V | ash |
| COM | 1 1 1 1 | One's Complement | $FFFF - B | | 1 (set) | set if R==0 | | MSB | N xor V | addsub |

## 3.5 ALU įrenginio principinė schema

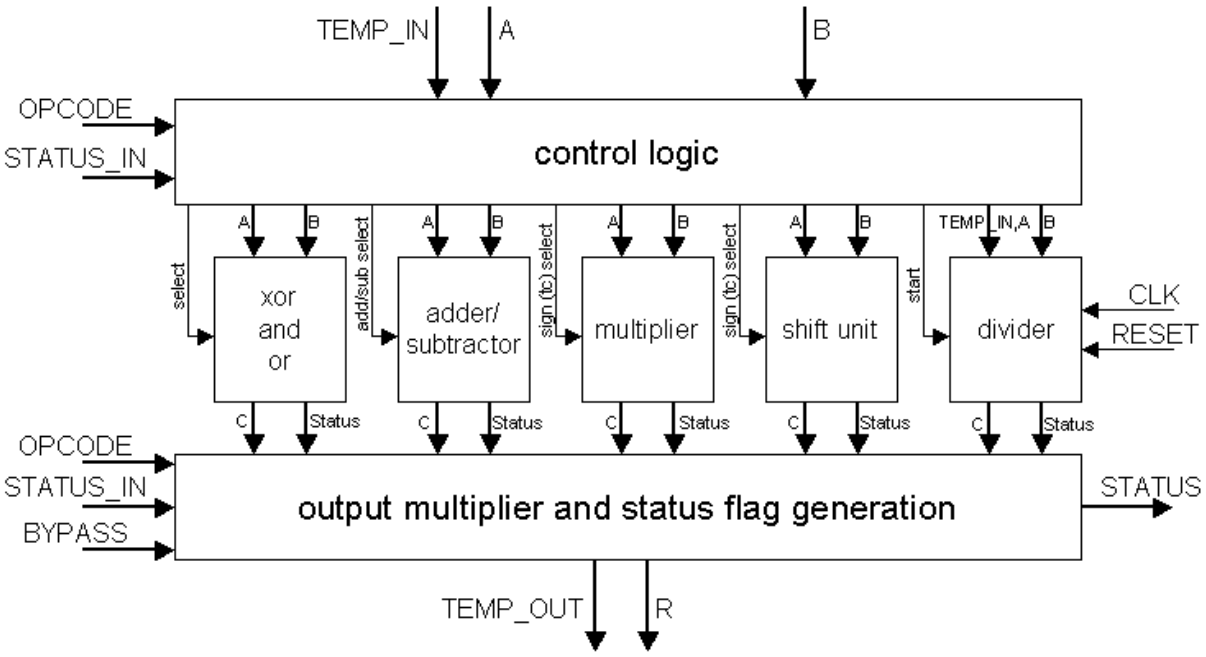



1 diagrama. Supaprastintas ALU modelis

1 diagramoje pavaizduotas supaprastintas procesoriaus įrenginio ALU modelis. Standartinis ALU įrenginys turi du duomenų įėjimus (A ir B) ir vieną išėjimą (out). Signalas control nurodo ALU įrenginiui kokią operaciją reikia atlikti. Signalas status nusako ALU įrenginio būseną (pvz. perpildymo atveju).

## 3.6 ALU įrenginio architektūra

ALU įrenginio sudėtinės dalys parodytos žemiau esančioje 2 diagramoje (prieiga per internetą <http://www.kip.uni-heidelberg.de/ti/TRD/alu/blockdiagram.html>):




2 diagrama. Sudėtinės ALU įrenginio dalys

Blokas control logic (valdymo logika) pasitelkdamas ALU įrenginio atliekamų operacijų kodus (opcode) bei būsenų vėliavėles sugeneruoja kontrolės signalus, kuriais išrenka atitinkamus modulius reikiamai operacijai įvykdyti. ALU įrenginys turi tokius modulius: loginės sudėties ir loginės daugybos modulis, aritmetinės sudėties ir atimties modulis, aritmetinės daugybos modulis, aritmetinės dalybos modulis ir postūmio modulis.

Dalybos (divider) modulis yra vienintelė ne kombinatorinė vaizduojamo modelio dalis. Išėjimo daugiklio ir būsenų vėliavėlių generavimo (output multiplier and status flag generation) bloke atliktos operacijos rezultatas ir būsenos vėliavėlė paimami pagal ALU įrenginio atliekamos operacijos kodą. Signalas BYPASS kartais naudojamas būsenų vėliavėlių pakeitimams panaikinti.

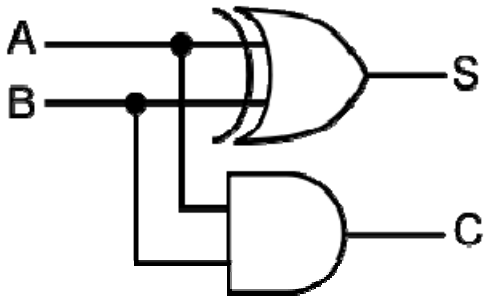
## 3.7 ALU įrenginio moduliai

Nuo to, kokio ilgio duomenis gali apdoroti ALU įrenginys, gali priklausyti jo greitaveika. Taip yra todėl, kad jeigu apdorojami duomenys yra ilgesni negu ALU įrenginys gali apdoroti, įrenginiui gali tekti atlikti papildomas operacijas, kad juos reikiamai apdorotų, o tai jau lėtina greitaveiką. Kad išspręsti šią problemą, buvo sukurti įvairūs algoritmai toms pačioms operacijos realizuoti. Todėl nors įvairūs ALU įrenginiai gali atlikti vienodas operacijas, bet būdai ,kaip tos operacijos atliekamos, gali skirtis.

### 3.7.1 Sumatorius

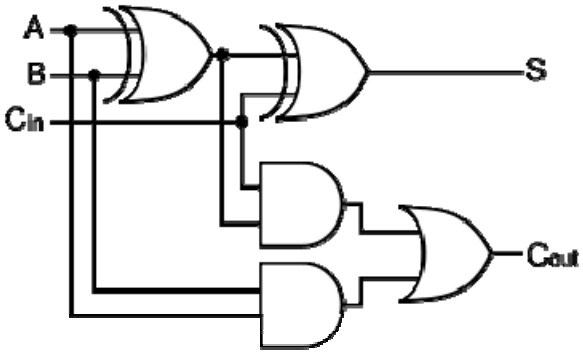
Sumatorius – tai ALU įrenginio modulis, kuris atlieka aritmetines sudėties operacijas. Šis modulis yra sudarytas iš kelių į grandinę sujungtų 1 bito sumatorių, kurie sudėties operaciją atlieka tik su vieno bito duomenimis. Šie 1 bito sumatoriai yra dviejų rūšių: nepilnas sumatorius (half adder) ir pilnas sumatorius.

Nepilnas sumatorius turi du duomenims skirtus įėjimus, pavadintus A ir B, ir du išėjimus: S – sumos ir C – pernašos. S yra rezultatas loginės operacijos „griežto arba“ (XOR), atliktos su A ir B. C išėjime gaunamas rezultatas, su duomenimis A ir B atlikus loginės daugybos (AND) operaciją. Šio sumatoriaus schema parodyta 3 paveikslėlyje:




3 pav. 1 bito nepilno sumatoriaus schema

Pilnas sumatorius turi tris įėjimus (A, B ir $C_{in}$) ir du išėjimus (S ir $C_{out}$). $C_{in}$ įėjimas ir $C_{out}$ išėjimas yra pernaša. Šio sumatoriaus schema parodyta 4 paveikslėlyje:




4 pav. 1 bito pilno sumatoriaus schema

Šis sumatorius nuo nepilno sumatoriaus skiriasi tuo, kad atlikdamas sudėties operaciją, jis įvertina pernašą, todėl sujungus kelis tokius sumatorius į grandinę, galima atlikti skaičiavimus su ilgesniais nei 1 bito duomenimis.

Sumatoriaus modulis sudaromas iš kelių į vieną grandinę sujungtų 1 bito sumatorių. Šiam moduliui realizuoti naudojami trys algoritmai: bangos pernašos (ripple carry), pernašos išsaugojimo (carry save) arba pernašos numatymo (carry look ahead).

*Bangos pernašos algoritmas*

Bangos pernašos algoritmas realizuojamas kelis 1 bito pilnus sumatorius sujungus į vientisą grandinę. Šis algoritmas taip pavadintas todėl, kad nuo pirmojo grandinėje esančio 1 bito sumatoriaus pernaša lyg banga sklinda link kitų grandinėje esančių sumatorių.

Bangos pernašos algoritmu pagrįsto sumatoriaus schema yra gana paprasta, todėl ji greitai projektuojama. Bet toks sumatorius yra gana lėtas, kadangi kiekvienas grandinėje esantis ą bito pilnas sumatorius turi palaukti kol prieš jį grandinėje esantis sumatorius paskaičiuos pernašą.

*Pernašos išsaugojimo algoritmas*

Hierarchinis pernašos išsaugojimo algoritmas yra modifikuotas pernašos išsaugojimo algoritmas. Šio algoritmo principas – kiekvienas rezultato bitas vienu metu paskaičiuojamas dviem būdais:

1. Be pernašos

**S(i) = A(i) + B(i) + 0**

2. Su pernaša

**S(i) = A(i) + B(i) + 1**

Tikroji pernaša (i – 1) padeda atrinkti kurias iš šių tarpinių sumų bei apskaičiuotą pernašą panaudoti kitam skaičiavimo etapui. Šis algoritmas yra greitesnis už bangos pernašos algoritmą.

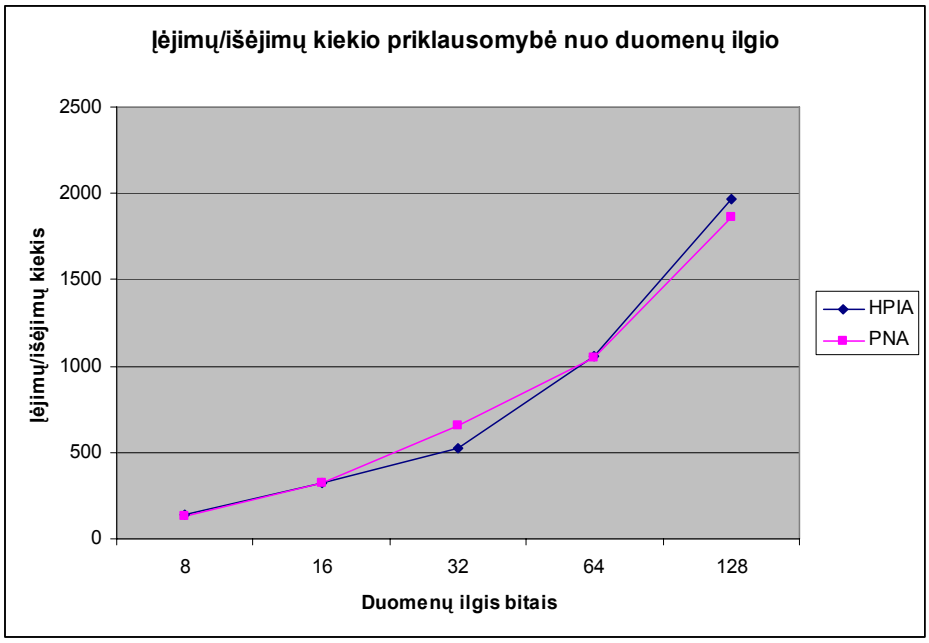
*Pernašos numatymo algoritmas*

Pernašos numatymo algoritmas buvo suprojektuotas sumavimo pagreitinimui. Kiekvienai bito pozicijai sukuriami Propagavimo (P) ir Generavimo (G) signalai, priklausomai nuo to, ar yra galimybė pernašai sklisti iš žemesnės bito pozicijos (jeigu bent vieno įėjimo reikšmė yra

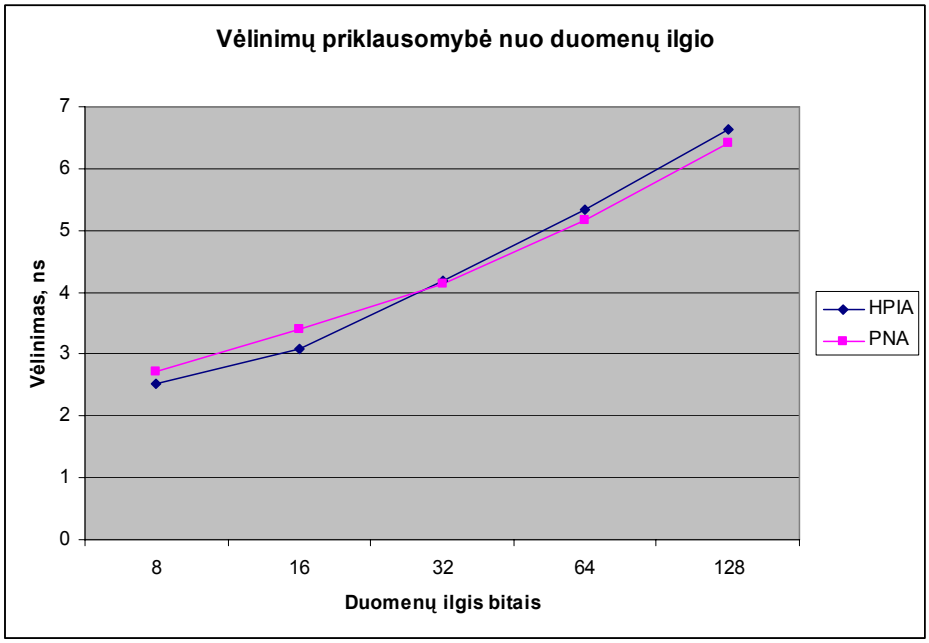
lygi „1“), ar pernaša yra sugeneruojama žemesnėje bito pozicijoje (abiejų įėjimų reikšmės lygios „1“), ar pernaša nebus gauta iš žemesnės bito pozicijos (abiejų įėjimų reikšmės lygios „0“). Dažniausiai P signalas yra nepilno 1 bito sumatoriaus suformuota suma, o G yra to paties sumatoriaus suformuota pernaša. Po to, kai sugeneruojami P ir G signalai, kiekvienai bito pozicijai sukuriamos pernašos.

*Palyginimas*

Žemiau pateiktoje 3 lentelėje bei 3 ir 4 diagramose palyginami hierarchinio pernašos išsaugojimo (HPIA) bei pernašos numatymo (PNA) algoritmais realizuotų sumatorių sintezės ir simuliacijos rezultatai (prieiga per internetą http://deversys.com/?action=download&id=32):




3 diagrama. Įėjimų/išėjimų kiekio priklausomybė nuo duomenų ilgio




4 diagrama. Vėlinimų priklausomybė nuo duomenų ilgio

3 lentelė. HPIA ir PNA sintezės rezultatų palyginimo lentelė

| | Hierarchinio pernašos išsaugojimo algoritmas | | Pernašos numatymo algoritmas | |
|---|---|---|---|---|
| Duomenų ilgis bitais | Įėjimų/išėjimų skaičius | Vėlinimas | Įėjimų/išėjimų skaičius | Vėlinimas |
| 8 | 143 | 2,51 | 127 | 2,72 |
| 16 | 327 | 3,09 | 327 | 3,40 |
| 32 | 527 | 4,18 | 655 | 4,14 |
| 64 | 1061 | 5,34 | 1053 | 5,16 |
| 128 | 1965 | 6,64 | 1865 | 6,41 |

Iš šių rezultatų matosi, kad duomenų ilgiui esant 8 arba 16 bitų, hierarchinio pernašos išsaugojimo algoritmu realizuotas sumatorius turi mažesnį vėlinimą negu sumatorius, realizuotas pernašos numatymo algoritmu. Bet kita vertus, pastarasis sumatorius, esant 8 bitų duomenų ilgiui, turi mažesnį įėjimų/išėjimų kiekį negu HPIA sumatorius.

Duomenų ilgiui esant 32 ir daugiau bitų, PNA sumatoriaus vėlinimas tampa mažesniu negu HPIA sumatoriaus, o įėjimų/išėjimų skaičius padidėja.

### 3.7.2 ALU atimties modulis

Paprastai ALU įrenginyje aritmetinės sudėties ir aritmetinės atimties operacijas atlieka vienas ir tas pats modulis, kadangi šiek tiek modifikavus sumatorių galima jį „priversti“ atlikti atimties operacijas.

Sumatorius atlieka operaciją S = A + B. Norint atlikti operaciją B – A, reikia kiekvieną A bitą invertuoti ir pridėti vienetą. Gautume tokią išraišką: S = B + (invertuotas A) + 1. Techniškai tai realizuojama prie kiekvieno 1 bito sumatoriaus A įėjimo pridedant dviejų įėjimų ir vieno išėjimo multipleksorių. Į vieną šio multipleksoriaus įėjima paduodamas invertuotas A, į kitą įėjimą paduodamas neinvertuotas A. Šio multipleksoriaus išėjimas yra sumatoriaus įėjimas A. Multipleksoriui padavus signalą D, nuo kurio reikšmės („1“ ar „0“) priklauso kokį įėjimą A iš multipleksoriaus gaus sumatorius: A ar invertuotą A. Po to yra atlieka aritmetinė sudėtis.

Kadangi aritmetinė atimtis realizuojama prie sumatoriaus pridėjus dar kelis elementus, tai praktiškai atimties operacijos greitaveika priklausys nuo to, kaip greitai sumatorius atliks aritmetinės sudėties operaciją.

### 3.7.3 Postūmio modulis

Postūmio modulis atlieka bitų postūmį į kairę arba į dešinę. Kadangi šis veiksmas nėra sudėtingas, nes paprasčiausiai duomenų bito pozicija perkeliama į kito bito poziciją (priklausomai nuo reikalingo postūmio, sekanti bito pozicija gali būti iš dešinės arba iš kairės), tai šios operacijos algoritmai vienas nuo kito nelabai skiriasi. Vienas nuo kito jie skiriasi tik tuo, kas yra įrašoma į atsilaisvinusį bitą ir kur yra įrašomas išstumtas bitas.

Kartais postūmio operacijos gali būti interpretuojamos kaip dalybos arba daugybos operacija. Dvejetainio skaičiaus postūmis į dešinę gali būti traktuojamas kaip dalybos operacija iš dviejų, o postūmis į kairę gali būti traktuojamas kaip skaičiaus daugybos operacija iš dviejų.

### 3.7.4 Dalybos ir daugybos moduliai

Dalybos ir daugybos moduliai ALU įrenginyje pasitaiko ne dažnai, kadangi jie yra sudėtingi ir brangūs. Todėl dalybos ir daugybos operacijoms atlikti panaudojami kiti ALU įrenginyje esantys moduliai (pavyzdžiui postūmio modulis).

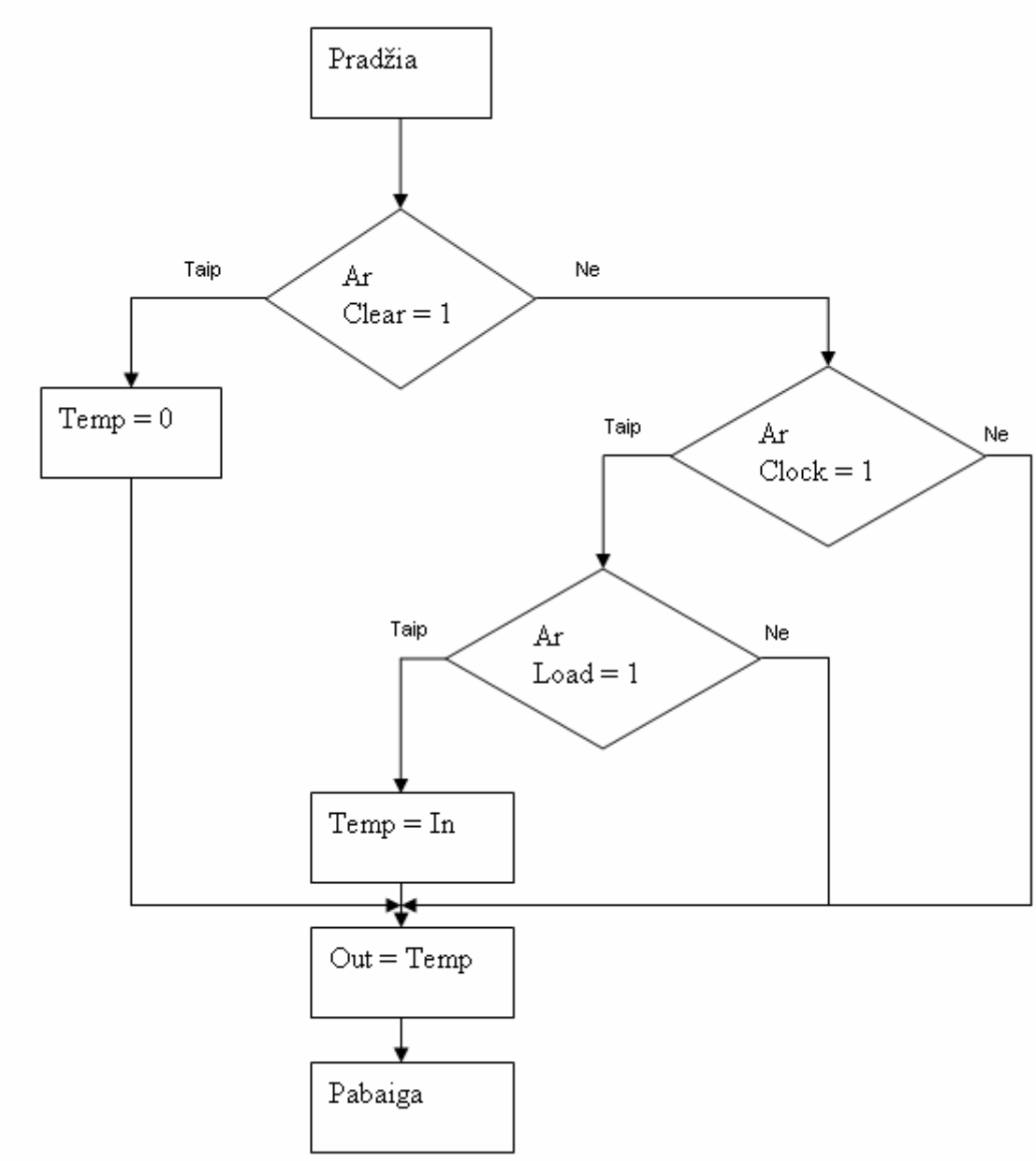
## 3.8 Registrai

Procesorius turi kelių tipų registrus – duomenų registrus, instrukcijų registrus, programos skaitiklio registrus. Duomenų registruose yra talpinami duomenys, su kuriais atliekamos tam tikros operacijos, arba saugomi tarpiniai rezultatai. Instrukcijų registruose įrašoma einamuoju laiku vykdoma operacija. Programos skaitiklio registre įrašoma sekančios operacijos adresas.

## 3.9 Registrų veikimo algoritmai

*Duomenų registrai*

Duomenų registrų veikimo algoritmas pateiktas žemiau esančioje 5 diagramoje:



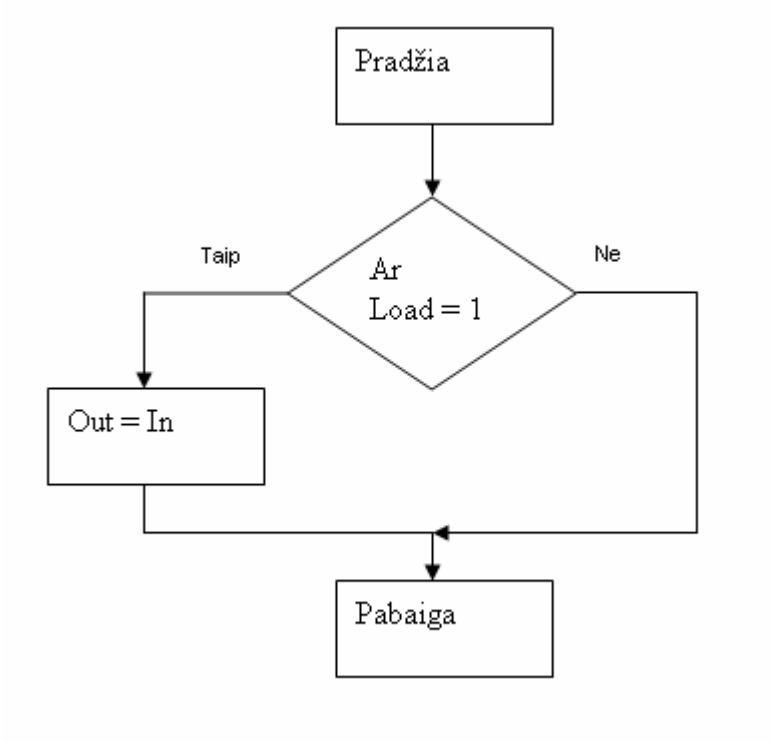

5 diagrama. Duomenų registro veikimo algoritmas

Duomenų registras iš valdančiojo modulio gali gauti signalą Clear, nuo kurio reikšmės priklauso ar registre esantys duomenys bus ištrinami, ar ne. Jeigu signalo Clear reikšmė bus lygi 1, tada iš registro duomenys bus ištrinami (į registrą įrašomi 0). Jeigu duomenų registras iš

valdančiojo modulio gauna signalą Load ir jo reikšmė lygi 1, tada į registrą yra įrašomi nauji duomenys. Jeigu signalo Clear reikšmė yra lygi 0 ir signalo Load reikšmė yra lygi 0, tada registras tiesiog saugo jame esančius duomenis, kuriuos iš jo gali nuskaityti kiti procesoriaus elementai.

*Instrukcijų registrai*

Instrukcijų registrų veikimo algoritmas pateiktas 6 diagramoje:

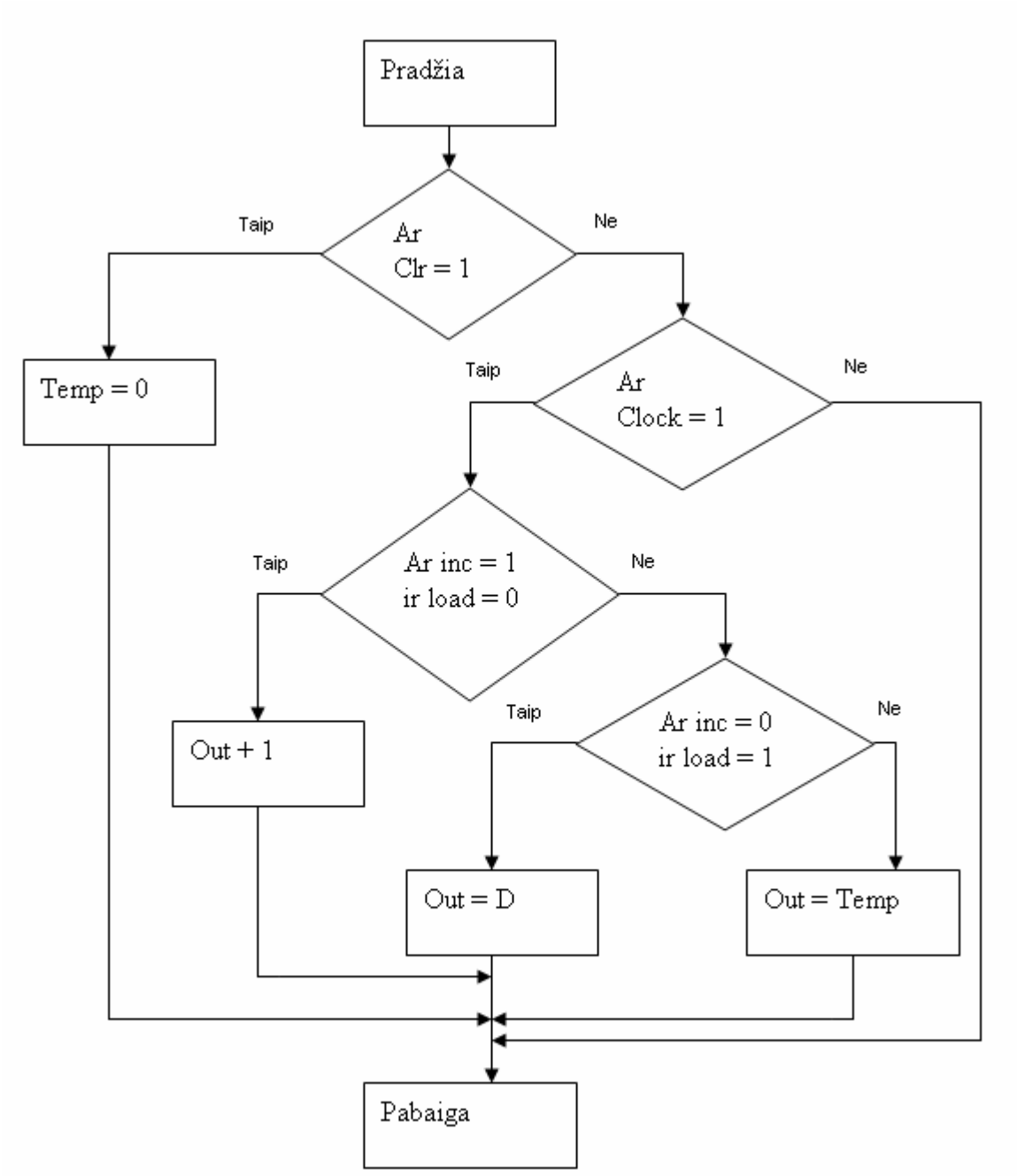



6 diagrama. Instrukcijų registrų veikimo algoritmas

Instrukcijų registre yra įrašoma operacija (arba jos adresas), kurią einamuoju laiku vykdo procesorius. Jeigu iš valdančiojo modulio instrukcijų registras gauna signalą Load ir jo reikšmė lygi 1, tada į šį registrą yra įrašoma nauja komanda (arba naujos komandos adresas), kurią turi atlikti procesoriaus aritmetinis logini įrenginys.

*Programos skaitiklio registrai*

Programos skaitiklio registrų veikimo algoritmas pateiktas 7 diagramoje:




7 diagrama. Programos skaitiklio registrų veikimo algoritmas

Programos skaitiklio registre įrašoma sekanti operacija (arba jos adresas), kuri bus vykdoma po einamuoju laiko momentu atliekamos operacijos. Jeigu registras iš valdančiojo modulio gauna signalą Clr ir jis būna lygus 1, tuomet registras yra užpildomas nuliais. Jeigu registras iš valdančiojo elemento gauna signalą inc kuris būna lygus 1 ir signalą load kuris būna

lygus 0, tuomet į registrą yra įrašoma sekanti mikroprogramos mikrokomanda. Jeigu registras iš valdančiojo modulio gauna signalą inc kurio reikšmė lygi 0 ir load signalą, kurio reikšmė lygi 1, tuomet i registrą įrašoma naujos mikroprogramos mikrokomanda. Visais kitais atvejais registras saugo mikrokomandą, kurią reikės procesoriui vykdyti kai bus atlikta einamuoju momentu vykdoma komanda.

# 4. Sintezė

Kad patikrinti procesoriaus komponentų parametrų kitimą keičiant jų įėjimų ar išėjimų bitų skaičių, komponentai buvo aprašyti VHDL programavimo kalboje ir, pasinaudojus Synopsys programine įranga, buvo susintezuoti bei gauti jų techniniai parametrai. Kadangi tiriama tik apdorojamų duomenų ilgio įtaka, todėl visi realioje aplinkoje procesorinių komponentų veikimą įtakojantys veiksniai laikomi visada vienodais, nepriklausomai nuo apdorojamų duomenų ilgio (pvz:. ar 8, ar 64 bitų ALU įrenginys, jis atlieka tik 4 operacijas).

## 4.1 ALU įrenginio sintezavimas

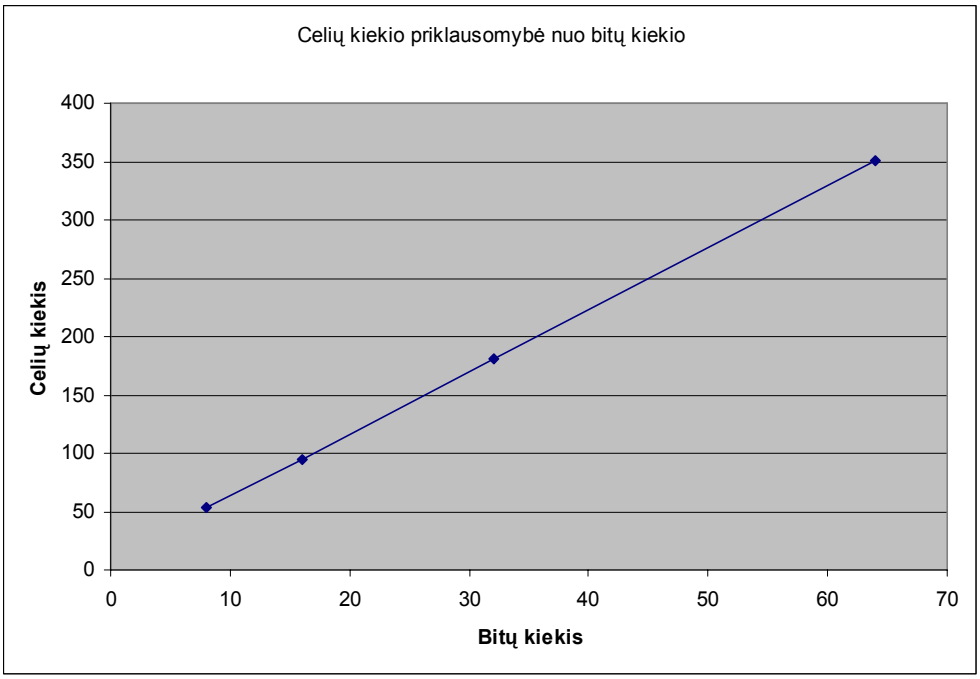
ALU įrenginiui bitų skaičius nusako su kokio dydžio duomenimis ALU įrenginys gali atlikti operacijas. Jeigu šio komponento įėjimai yra 8 bitų, tai jis gali atlikti operacijas su 8 bitų ilgio duomenimis. Todėl projektuojant ALU įrenginį atsižvelgiama į duomenų dydį, su kuriuo jam teks susidurti darbo metu.

ALU įrenginio sintezės rezultatų suvestinė pateikta 4 lentelėje (susintezuotos schemos pirmame priede 29 – 32 psl., VHDL aprašai – antrame priede 45 – 47 psl.):

4 lentelė. ALU įrenginio sintezės rezultatų suvestinė

| Bitų skaičius | Celių skaičius | Užimamas plotas | Vidinis galingumas |
|---|---|---|---|
| 8 | 53 | 215 | 219,9955 |
| 16 | 95 | 428 | 442,0952 |
| 32 | 181 | 854 | 885,1216 |
| 64 | 351 | 1707 | 1768,1 |

Iš 4 lentelės gauname tokias priklausomybes 8, 9 bei 10 diagramose:




8 diagrama. Celių skaičiaus priklausomybė nuo bitų skaičiaus

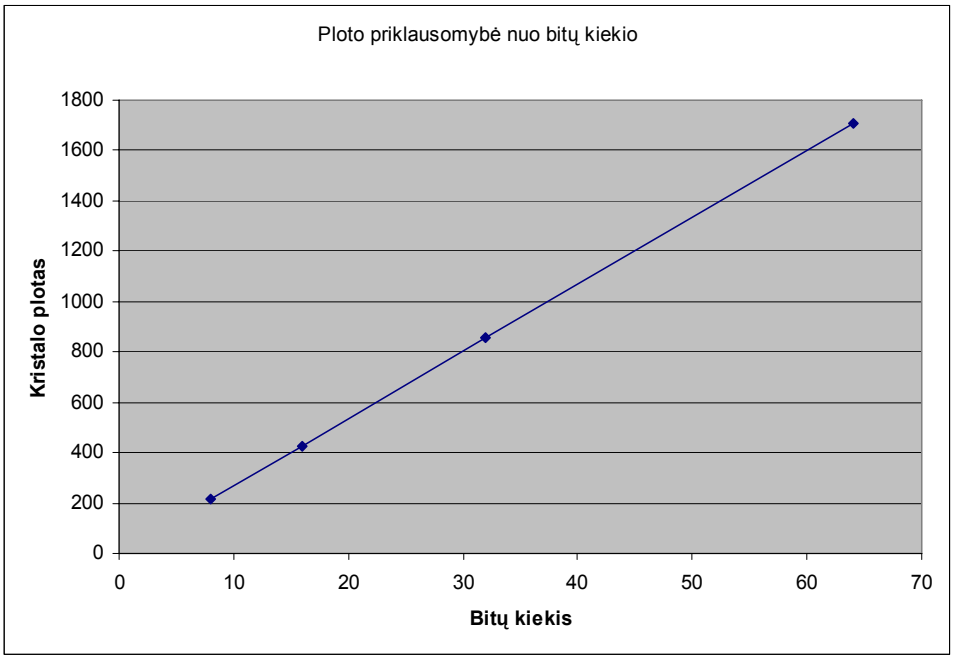
Pagal 8 diagramą gauname celių kiekio nuo bitų kiekio priklausomybės matematinę išraišką:

$$y = 5{,}3261x + 10{,}217$$

Į šią išraišką vietoj x įstatę bitų kiekį gauname tokias reikšmes:

5 lentelė. ALU celių priklausomybė nuo bitų

| Bitų skaičius | Celių skaičius |
|---|---|
| 8 | 52,8258 |
| 16 | 95,4346 |
| 32 | 180,6522 |
| 64 | 351,0874 |




9 diagrama. Kristalo ploto priklausomybė nuo bitų skaičiaus

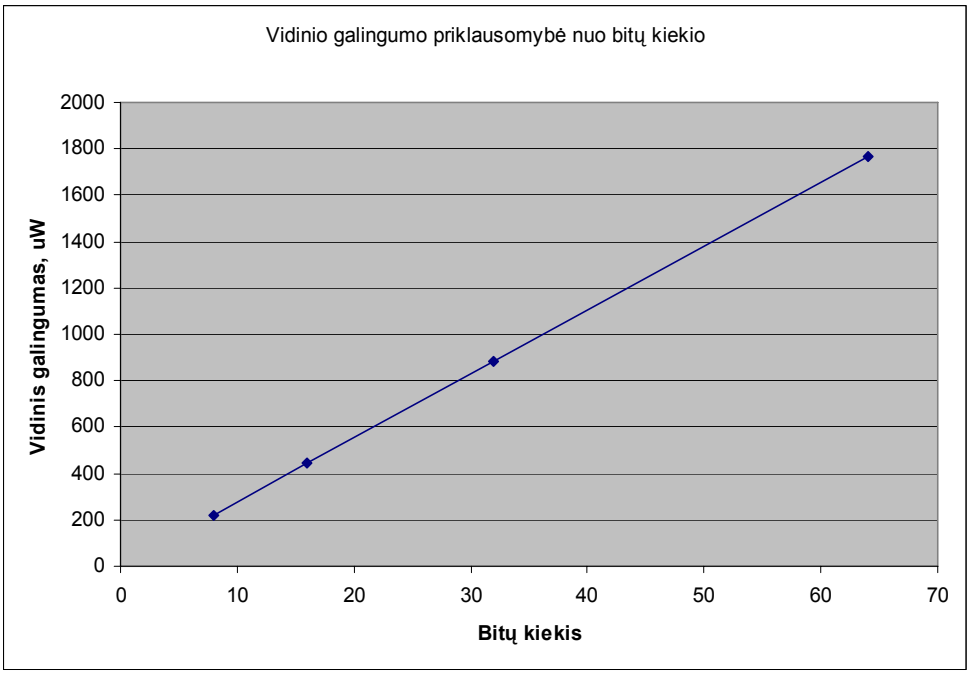
Pagal 9 diagramą gauname ploto nuo bitų kiekio priklausomybės matematinę išraišką:

$$y = 26,643x + 1,6957$$

Į šią išraišką vietoj x įstatę bitų kiekį gauname tokias reikšmes:

6 lentelė. ALU ploto priklausomybė nuo bitų

| Bitų skaičius | Kristalo plotas |
|---|---|
| 8 | 214,8397 |
| 16 | 427,9837 |
| 32 | 854,2171 |
| 64 | 1706,848 |




10 diagrama. Vidinio galingumo priklausomybė nuo bitų skaičiaus

Pagal 10 diagramą gauname vidinio galingumo nuo bitų kiekio priklausomybės matematinę išraišką:

$$y = 27{,}639x - 0{,}3525$$

Į šią išraišką vietoj x įstatę bitų kiekį gauname tokias reikšmes:

7 lentelė. ALU vidinio galingumo priklausomybė nuo bitų

| Bitų skaičius | Vidinis galingumas, uW |
|---|---|
| 8 | 220,7595 |
| 16 | 441,8715 |
| 32 | 854,0955 |
| 64 | 1768,544 |

Iš šių rezultatų galima pamatyti, kad didinant ALU įrenginio apdorojamos informacijos bitų skaičių, didėja ALU įrenginio užimamas plotas, jam realizuoti reikalingų loginių elementų kiekis bei vidinis galingumas. Didėjant elementų kiekiui, didėja ir ALU įrenginio bendras vėlinimas, todėl gali sumažėti šio procesoriaus komponento greitaveika.

Taip pat iš priklausomybių lentelių matome, kad naudojantis išvestomis formulėmis gauto reikšmės gaunamos panašios į sintezavimo rezultatus.

## 4.2 Duomenų registrų sintezavimas

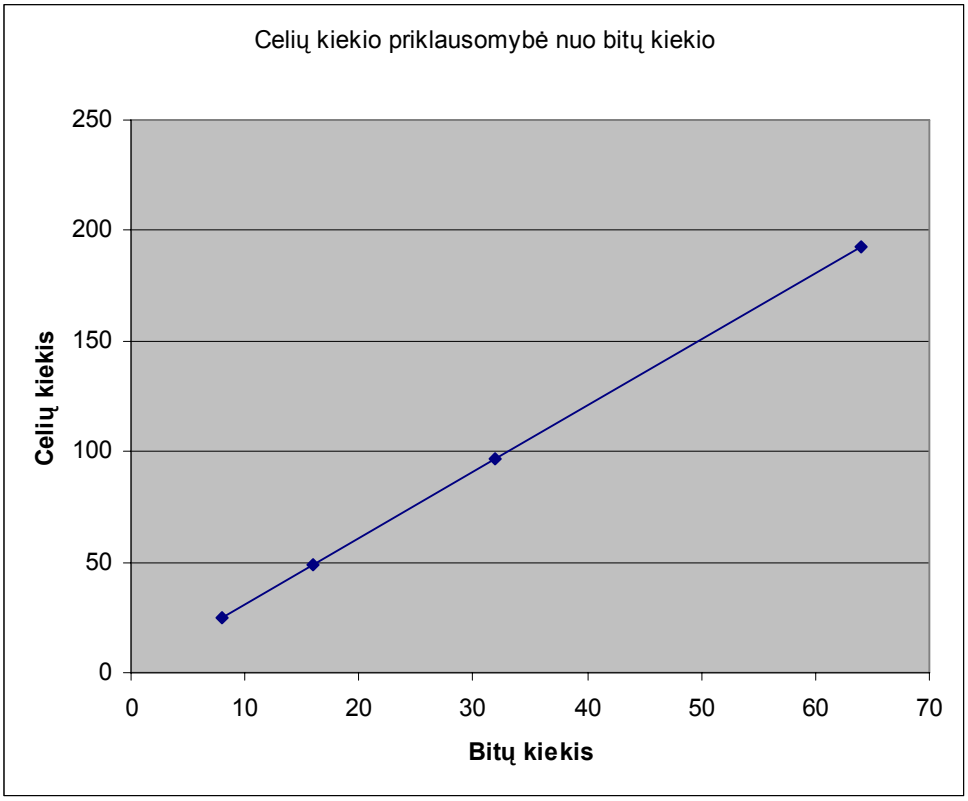
Duomenų registrų bitų skaičius nusako kokio dydžio duomenis registras gali saugoti. Didesnis saugomos informacijos kiekis įgalina procesorių dirbti su didesniais informacijos kiekiais (priklauso ir nuo kitų procesoriaus komponentų).

Duomenų registro sintezės rezultatų suvestinė pateikta 8 lentelėje (susintezuotos schemos pirmame priede 33 – 36 psl., VHDL aprašai – antrame priede 47 – 50 psl.):

8 lentelė. Duomenų registro sintezės rezultatų suvestinė

| Bitų skaičius | Celių skaičius | Užimamas plotas | Vidinis galingumas |
|---|---|---|---|
| 8 | 25 | 97 | 20,5765 |
| 16 | 49 | 193 | 41,0200 |
| 32 | 97 | 385 | 82,3685 |
| 64 | 193 | 769 | 163,8027 |

Iš šios lentelės gauname tokias priklausomybes, pateiktas 11, 12 ir 13 diagramose:




11 diagrama. Celių skaičiaus priklausomybė nuo bitų skaičiaus

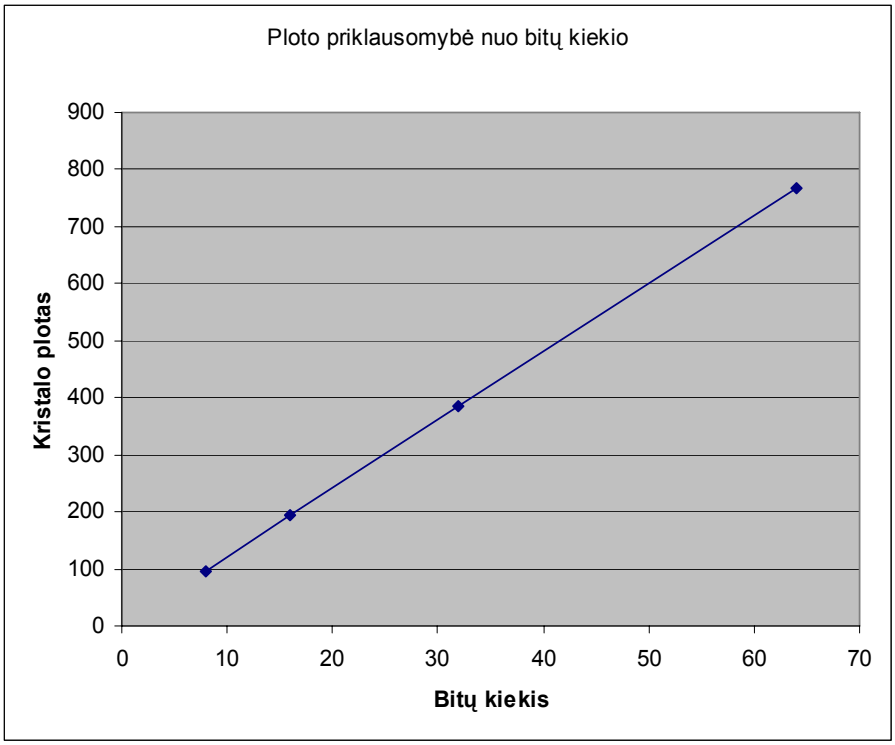
Pagal 11 diagramą gauname celių kiekio nuo bitų kiekio priklausomybės matematinę išraišką:

$$y = 3x + 1$$

Į šią išraišką vietoj x įstatę bitų kiekį gauname tokias reikšmes:

9 lentelė. Duomenų registro celių priklausomybė nuo bitų

| Bitų skaičius | Celių skaičius |
|---|---|
| 8 | 25 |
| 16 | 49 |
| 32 | 97 |
| 64 | 193 |




12 diagrama. Kristalo ploto priklausomybė nuo bitų skaičiaus

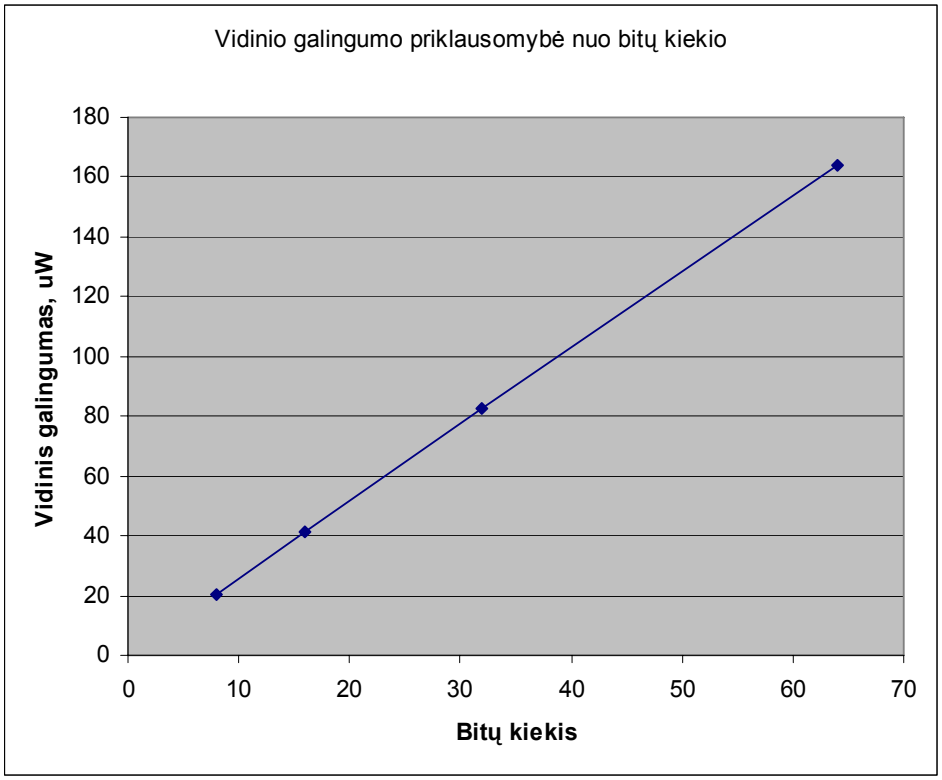
Pagal 12 diagramą gauname ploto nuo bitų kiekio priklausomybės matematinę išraišką:

$$y = 12x + 1$$

Į šią išraišką vietoj x įstatę bitų kiekį gauname tokias reikšmes:

10 lentelė. Duomenų registro ploto priklausomybė nuo bitų

| Bitų skaičius | Kristalo plotas |
|---|---|
| 8 | 97 |
| 16 | 193 |
| 32 | 385 |
| 64 | 769 |




13 diagrama. Vidinio galingumo priklausomybė nuo bitų skaičiaus

Pagal 13 diagramą gauname vidinio galingumo nuo bitų kiekio priklausomybės matematinę išraišką:

$$y = 2{,}5582x + 0{,}1963$$

Į šią išraišką vietoj x įstatę bitų kiekį gauname tokias reikšmes:

11 lentelė. Duomenų registro vidinio galingumo priklausomybė nuo bitų

| Bitų skaičius | Vidinis galingumas, uW |
|---|---|
| 8 | 20,6619 |
| 16 | 41,1275 |
| 32 | 82,0587 |
| 64 | 163,9211 |

Iš šių rezultatų galima pamatyti kaip kinta duomenų registrų parametrai keičiant talpinamos informacijos bitų skaičių. Didinant bitų skaičių, didėja registrų celių kiekis, užimamas plotas, vidinis galingumas.

Taip pat iš priklausomybių lentelių matome, kad naudojantis išvestomis formulėmis gauto reikšmės gaunamos panašios į sintezavimo rezultatus.

## 4.3 Instrukcijų registrų sintezavimas

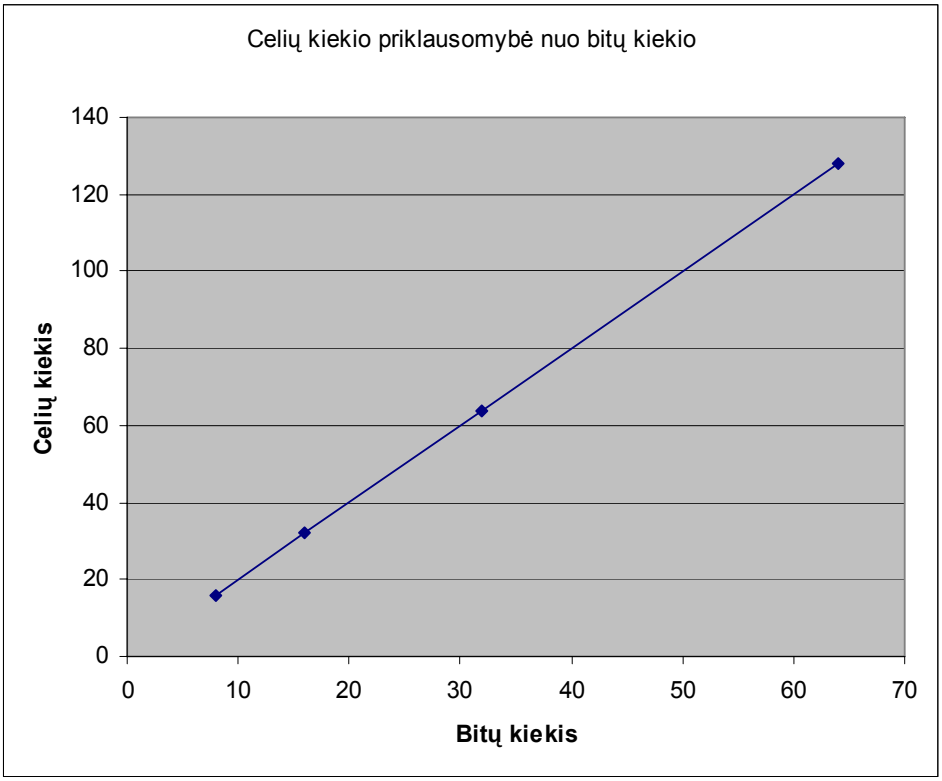
Instrukcijų registrams bitų skaičiaus keitimas nusako kiek skirtingų operacijų gali atlikti procesorius. Jeigu instrukcijų registras yra 3 bitų, tai reiškia, kad procesorius gali atlikti $2^3$ (8) skirtingų operacijų.

Instrukcijų registro sintezės rezultatų suvestinė pateikta 12 lentelėje (susintezuotos schemos pirmame priede 37 – 40 psl., VHDL aprašai – antrame priede 50 – 51 psl.):

12 lentelė. Instrukcijų registro sintezės rezultatų suvestinė

| Bitų skaičius | Celių skaičius | Užimamas plotas | Vidinis galingumas |
|---|---|---|---|
| 8 | 16 | 80 | 14,2557 |
| 16 | 32 | 160 | 28,7272 |
| 32 | 64 | 320 | 56,9564 |
| 64 | 128 | 640 | 113,1720 |

Iš šios lentelės gauname priklausomybes, pateiktas 14, 15 ir 16 diagramose:




14 diagrama. Celių skaičiaus priklausomybė nuo bitų skaičiaus

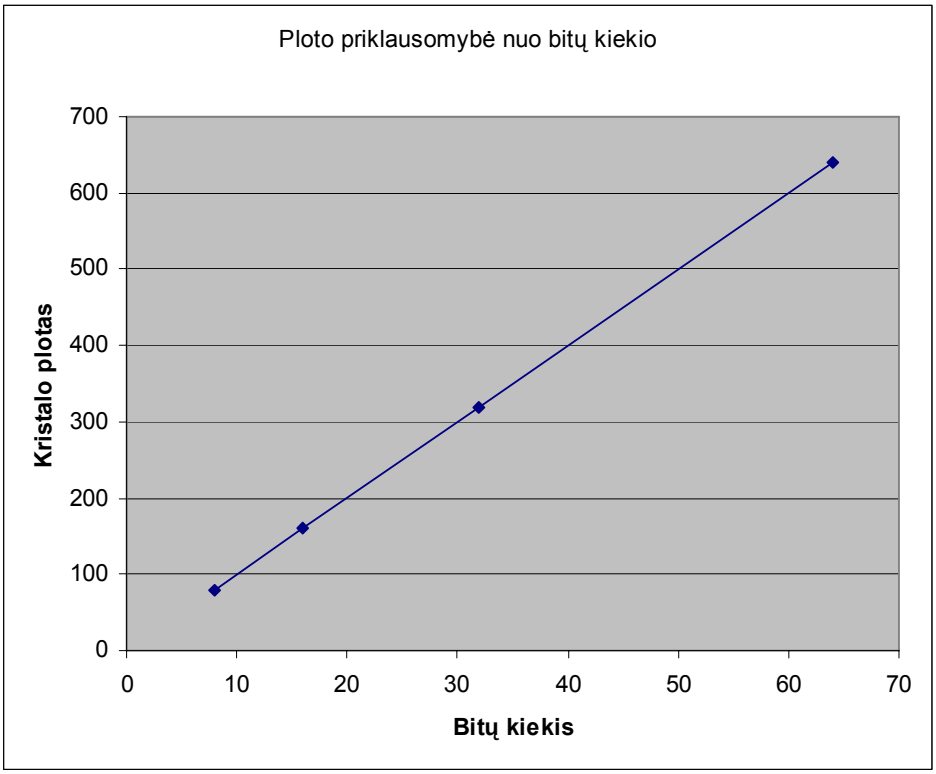
Pagal 14 diagramą gauname celių kiekio nuo bitų kiekio priklausomybės matematinę išraišką:

$$y = 2x$$

Į šią išraišką vietoj x įstatę bitų kiekį gauname tokias reikšmes:

13 lentelė. Instrukcijų registro celių priklausomybė nuo bitų

| Bitų skaičius | Celių skaičius |
|---|---|
| 8 | 16 |
| 16 | 32 |
| 32 | 64 |
| 64 | 128 |




15 diagrama. Kristalo ploto priklausomybė nuo bitų skaičiaus

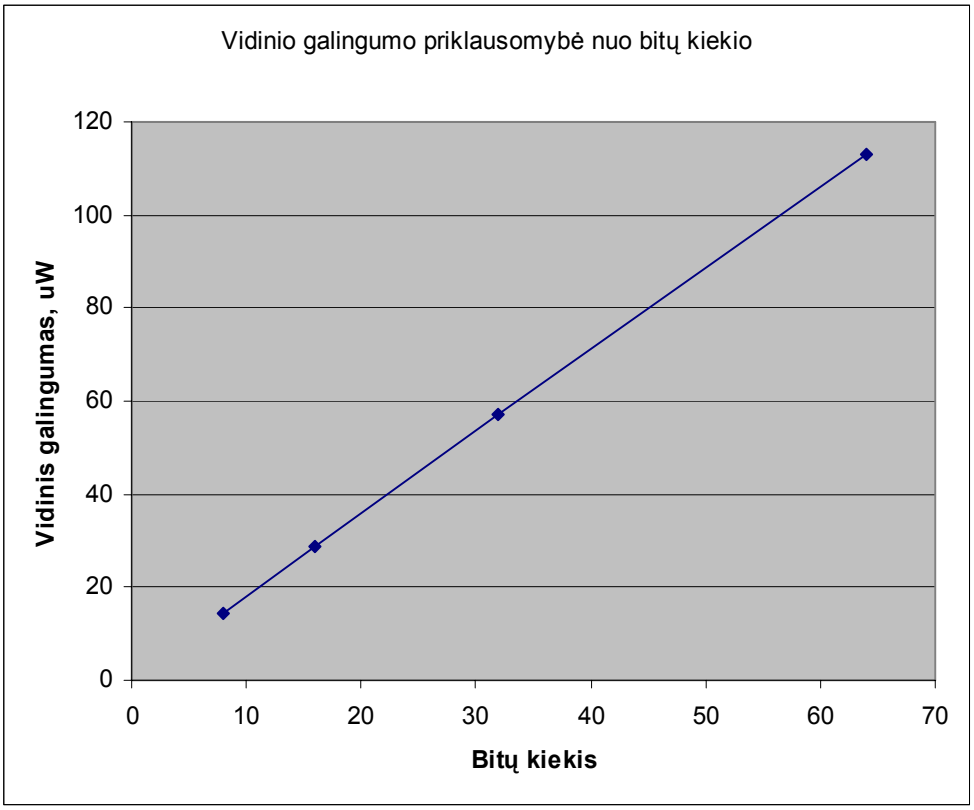
Pagal 15 diagramą gauname ploto nuo bitų kiekio priklausomybės matematinę išraišką:

$$y = 10x$$

Į šią išraišką vietoj x įstatę bitų kiekį gauname tokias reikšmes:

14 lentelė. Instrukcijų registro ploto priklausomybė nuo bitų

| Bitų skaičius | Kristalo plotas |
|---|---|
| 8 | 80 |
| 16 | 160 |
| 32 | 320 |
| 64 | 640 |




16 diagrama. Vidinio galingumo priklausomybė nuo bitų skaičiaus

Pagal 16 diagramą gauname vidinio galingumo nuo bitų kiekio priklausomybės matematinę išraišką:

$$y = 1{,}7641x + 0{,}3547$$

Į šią išraišką vietoj x įstatę bitų kiekį gauname tokias reikšmes:

15 lentelė. Instrukcijų registro vidinio galingumo priklausomybė nuo bitų

| Bitų skaičius | Vidinis galingumas, uW |
|---|---|
| 8 | 14,4675 |
| 16 | 28,5803 |
| 32 | 56,8059 |
| 64 | 113,2571 |

Iš šių rezultatų matome, kad didinant instrukcijų registrų talpinamų bitų skaičių, padidėja jo užimamas plotas,jam realizuoti reikalingas loginių elementų kiekis bei vidinis galingumas.

Taip pat iš priklausomybių lentelių matome, kad naudojantis išvestomis formulėmis gauto reikšmės gaunamos panašios į sintezavimo rezultatus.

## 4.4 Programos skaitiklio registro sintezavimas

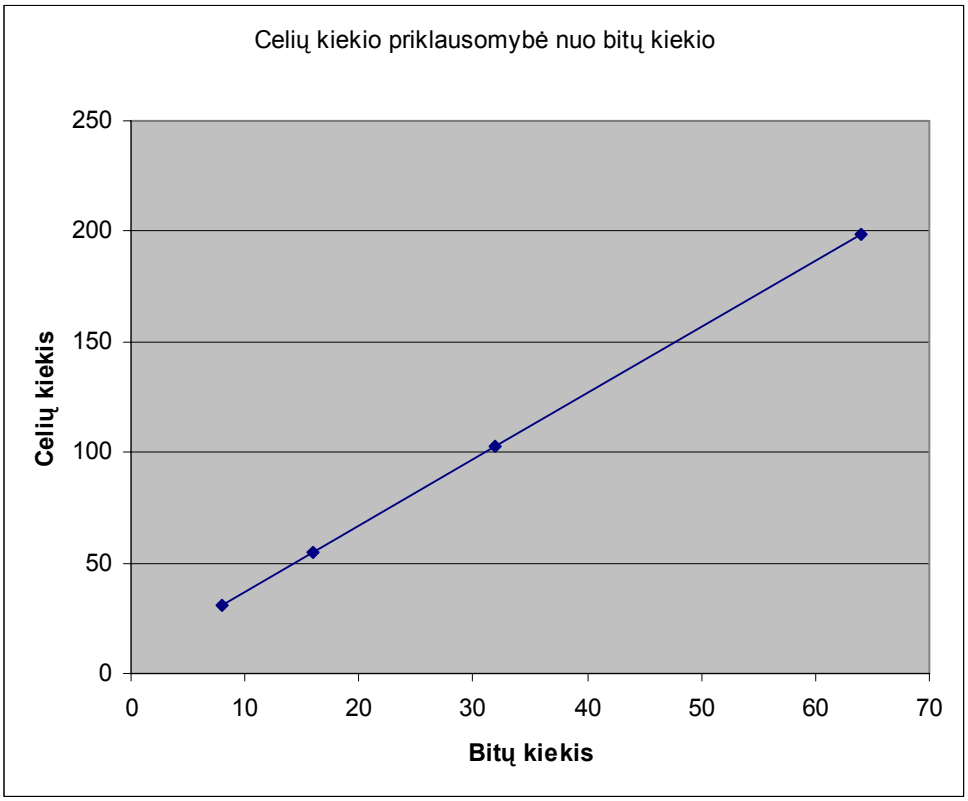
Programos skaitiklio registro talpinamų bitų skaičius (kaip ir instrukcijų registrui) nusako kiek skirtingų operacijų gali atlikti procesorius.

Programų skaitiklio registro sintezavimo rezultatų suvestinė pateikta 16 lentelėje (susintezuotos schemos pirmame priede 41 – 44 psl., VHDL aprašai – antrame priede 51 – 53 psl.):

16 lentelė. Programų skaitiklio registro sintezavimo rezultatų suvestinė

| Bitų skaičius | Celių skaičius | Užimamas plotas | Vidinis galingumas |
|---|---|---|---|
| 8 | 31 | 142 | 46,8515 |
| 16 | 55 | 282 | 87,8122 |
| 32 | 103 | 562 | 168,7212 |
| 64 | 199 | 1122 | 331,6020 |

Iš šios lentelės gauname priklausomybes, pateiktas 17, 18 ir 19 diagramose:




17 diagrama. Celių skaičiaus priklausomybė nuo bitų skaičiaus

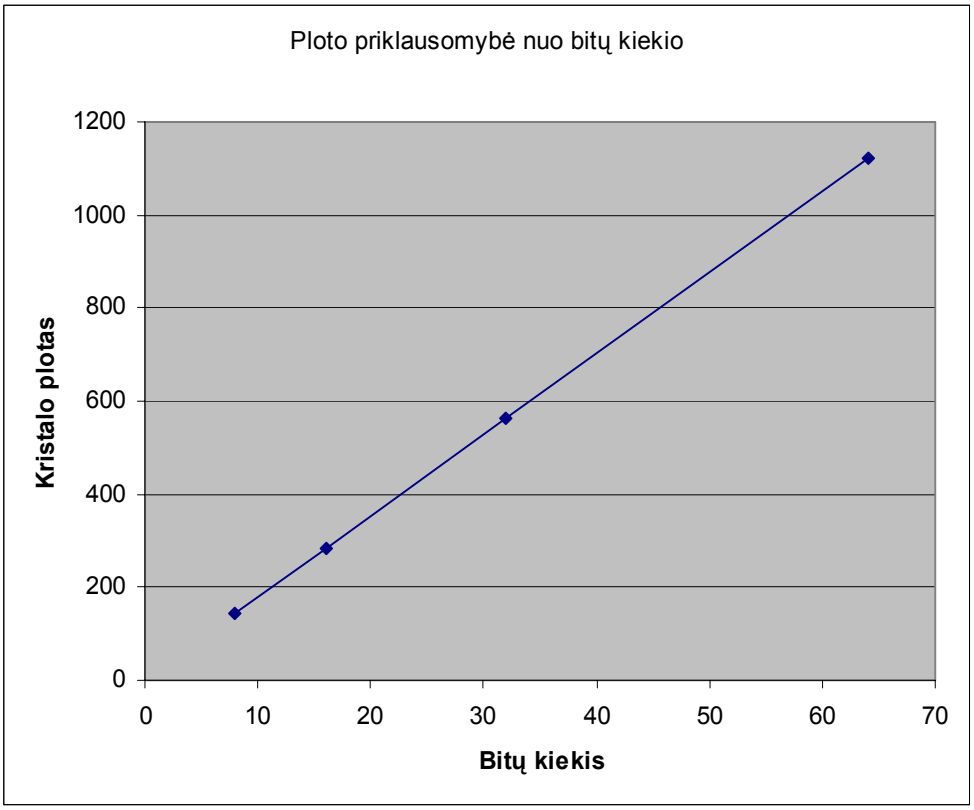
Pagal 17 diagramą gauname celių kiekio nuo bitų kiekio priklausomybės matematinę išraišką:

$$y = 3x + 7$$

Į šią išraišką vietoj x įstatę bitų kiekį gauname tokias reikšmes:

17 lentelė. Programos skaitiklio registro celių priklausomybė nuo bitų

| Bitų skaičius | Celių skaičius |
|---|---|
| 8 | 31 |
| 16 | 55 |
| 32 | 103 |
| 64 | 199 |




18 diagrama. Kristalo ploto priklausomybė nuo bitų skaičiaus

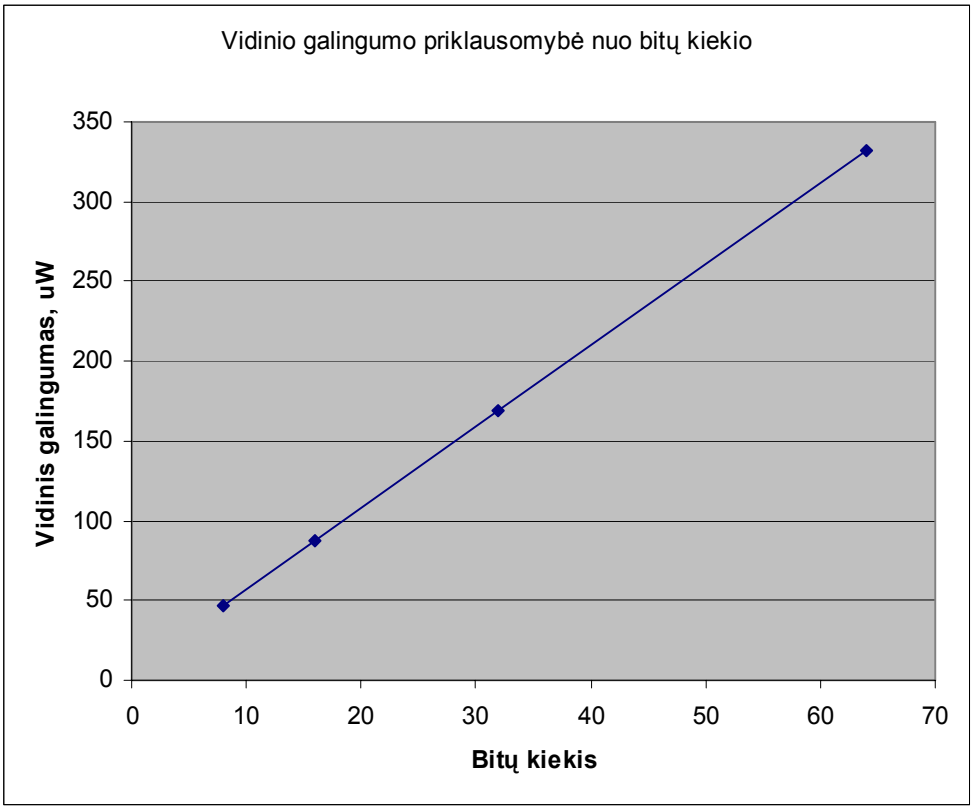
Pagal 18 diagramą gauname ploto nuo bitų kiekio priklausomybės matematinę išraišką:

$$y = 17{,}5x + 2$$

Į šią išraišką vietoj x įstatę bitų kiekį gauname tokias reikšmes:

18 lentelė. Programos skaitiklio registro ploto priklausomybė nuo bitų

| Bitų skaičius | Kristalo plotas |
|---|---|
| 8 | 142 |
| 16 | 282 |
| 32 | 562 |
| 64 | 1122 |




19 diagrama. Vidinio galingumo priklausomybė nuo bitų skaičiaus

Pagal 19 diagramą gauname vidinio galingumo nuo bitų kiekio priklausomybės matematinę išraišką:

$$y = 5{,}0825x + 6{,}2716$$

Į šią išraišką vietoj x įstatę bitų kiekį gauname tokias reikšmes:

19 lentelė. Programos skaitiklio registro vidinio galingumo priklausomybė nuo bitų

| Bitų skaičius | Vidinis galingumas, uW |
|---|---|
| 8 | 46,9316 |
| 16 | 87,5916 |
| 32 | 168,9116 |
| 64 | 331,5516 |

Iš šių rezultatų matome kaip didėja programos skaitiklio registro parametrai, padidinus jo talpumą. Didinant bitų skaičių, padidėja užimamas plotas, jam realizuoti reikalingų loginių elementų kiekis bei vidinis galingumas.

Taip pat iš priklausomybių lentelių matome, kad naudojantis išvestomis formulėmis gauto reikšmės gaunamos panašios į sintezavimo rezultatus.

## 4.5 Gautų formulių tikrinimas

Kad patikrinti, kaip gautos formulės veikia su nestandartiniais parametrais, buvo susintezuoti tiriamieji procesoriaus komponentai taip, kad jie galėtų apdoroti 24 bitų ilgio duomenis. Tai nedažnai pasitaikantis informacinio žodžio ilgis.

*24 bitų ALU įrenginys*

ALU įrenginio celių kiekiui, kristalo plotui bei vidiniam galingumui gauti buvo išvestos šios formulės:

- y = 5,3261x+10,217  celių kiekis
- y = 26,643x + 1,6957  kristalo plotas
- y = 27,639x - 0,3525  vidinis galingumas

Čia x – bitų skaičius.

Pasinaudojus formulėmis gauti tokie rezultatai:

- Celių kiekis  138,0434
- Kristalo plotas  641,1277
- Vidinis galingumas  662,9835

Aprašius ALU įrenginį VHDL kalboje ir susintezavus gavosi tokie rezultatai:

- Celių kiekis  139
- Kristalo plotas  644
- Vidinis galingumas  664

*24 bitų duomenų registrai*

Duomenų registro celių kiekiui, kristalo plotui bei vidiniam galingumui gauti buvo išvestos šios formulės:

- y = 3x + 1  celių kiekis
- y = 12x + 1  kristalo plotas
- y = 2,5582x + 0,1963  vidinis galingumas

Čia x – bitų skaičius.

Pasinaudojus formulėmis gauti tokie rezultatai:

- Celių kiekis  73

- Kristalo plotas 289
- Vidinis galingumas 61,5931

Aprašius duomenų registrą VHDL kalboje ir susintezavus gavosi tokie rezultatai:

- Celių kiekis 73
- Kristalo plotas 289
- Vidinis galingumas 61,4241

*24 bitų instrukcijų registrai*

Instrukcijų registro celių kiekiui, kristalo plotui bei vidiniam galingumui gauti buvo išvestos šios formulės:

- $y = 2x$ celių kiekis
- $y = 10x$ kristalo plotas
- $y = 1,7641x + 0,3547$ vidinis galingumas

Čia x – bitų skaičius.

Pasinaudojus formulėmis gauti tokie rezultatai:

- Celių kiekis 48
- Kristalo plotas 240
- Vidinis galingumas 42,6931

Aprašius instrukcijų registrą VHDL kalboje ir susintezavus gavosi tokie rezultatai:

- Celių kiekis 48
- Kristalo plotas 240
- Vidinis galingumas 43,2362

*24 bitų programos skaitiklio registrai*

Programos skaitiklio registro celių kiekiui, kristalo plotui bei vidiniam galingumui gauti buvo išvestos šios formulės:

- $y = 3x + 7$ celių kiekis
- $y = 17,5x + 2$ kristalo plotas
- $y = 5,0825x + 6,2716$ vidinis galingumas

Čia x – bitų skaičius.

Pasinaudojus formulėmis gauti tokie rezultatai:

- Celių kiekis 79
- Kristalo plotas 422
- Vidinis galingumas 128,2616

Aprašius programos skaitiklio registrą VHDL kalboje ir susintezavus gavosi tokie rezultatai:

- Celių kiekis 79
- Kristalo plotas 422
- Vidinis galingumas 126,1112

# 5. Išvados

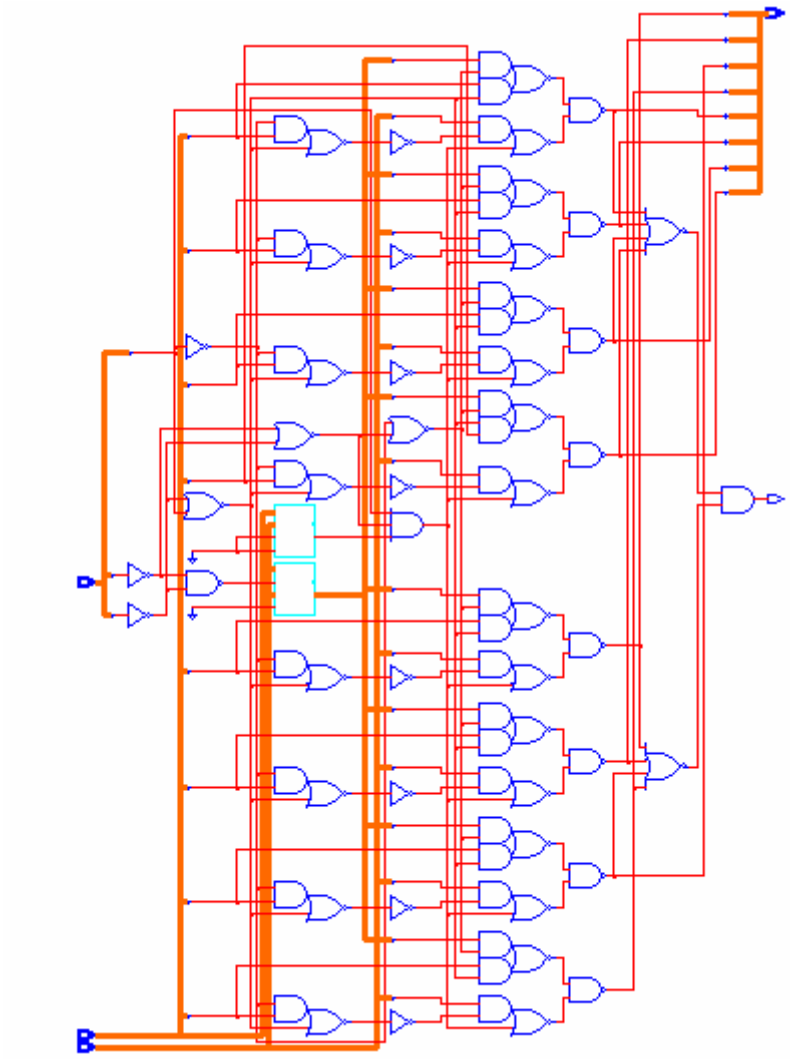
1) Iš sintezės rezultatų galima teigti, kad, žinant apdorojamų duomenų ilgį bitais, galima preliminariai nusakyti procesorinių komponentų sintezavimo rezultatus, tokius kaip susintezuoto kristalo užimamą plotą, kristalui reikalingų elementų kiekį, energijos sąnaudas. Juos, nors ir su paklaida, galima apskaičiuoti su išvestomis formulėmis.
2) Iš gautų sintezės rezultatų galima pamatyti, kad didinant duomenų bitų skaičių, kurį reikia procesoriui apdoroti, didėja procesoriaus komponentų, kurie galėtų apdoroti atitinkamo ilgio duomenis, užimamas plotas, jiems realizuoti reikalingų loginių elementų skaičius bei vidinis galingumas. Dėl šių priežasčių didėja procesoriaus pagaminimo kaina. Kita iškylanti problema yra ta, kad padaugėjus loginių elementų kiekiui, didėja viso procesoriaus vėlinimas. Todėl svarbu atsižvelgti į tai, su kokio ilgio duomenimis susidurs projektuojamas procesorius.

# 6. Literatūra


1. G. Ziberkas. The Development of Generic Components in VHDL. // Information Technology & Control. ISSN 1392-124X, Kaunas: Technologija, 2000, No. 2(15), p. 51–58.
2. Daniel D. Gajski, University of California, Irvine, USA; Allen C.-H. Wu, Tsing Hua University, Taiwan, ROC; Viraphol Chaiyakul, Y Explorations Inc., USA; Shojiro Mori, Toshiba Corp., Japan; Tom Nukiyama, NEC Corp., Japan; Pierre Bricaud, Mentor Graphics Corp., USA. Essential Issues for IP Reuse
3. Design reuse and retargeting. Prieiga per internetą: <http://www.forteds.com/solutions/behavioralip.asp>
4. A VHDL TUTORIAL, Developed by Syed Yawar Ali Shah, Supervisor: Dr. Asim J. Alkhalili October, 1999. Department of Electrical and Computer Engineering Concordia University, Montreal. Prieiga per internetą: <http://users.encs.concordia.ca/~asim/COEN_6501/tools/vhdl_yawar.html>
5. HCSA adder and Generic ALU based on HCSA: Overview. Vladimir V.Erokhin, PhD. Prieiga per internetą: <http://www.opencores.org/projects.cgi/web/hsca_adder/overview>
6. Ruprecht-Karls-Universitat Heidelberg. Prieiga per internetą: <http://www.kip.uni-heidelberg.de/ti/TRD/alu/architecture.html>
7. http://en.wikipedia.org/wiki/Main_Page
8. http://deversys.com/?action=download&id=32

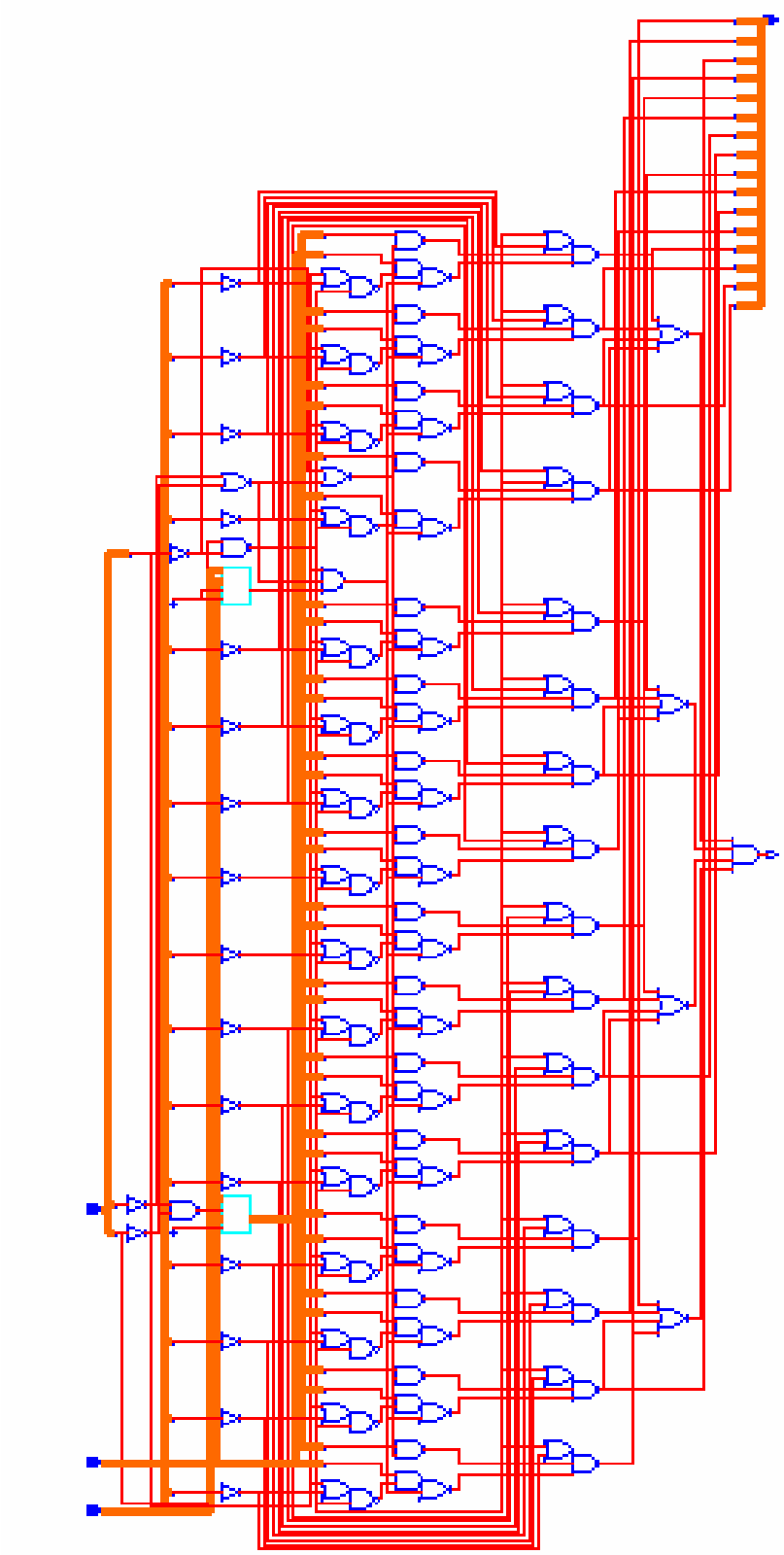
# 7. Priedas I

8 bitų ALU įrenginys



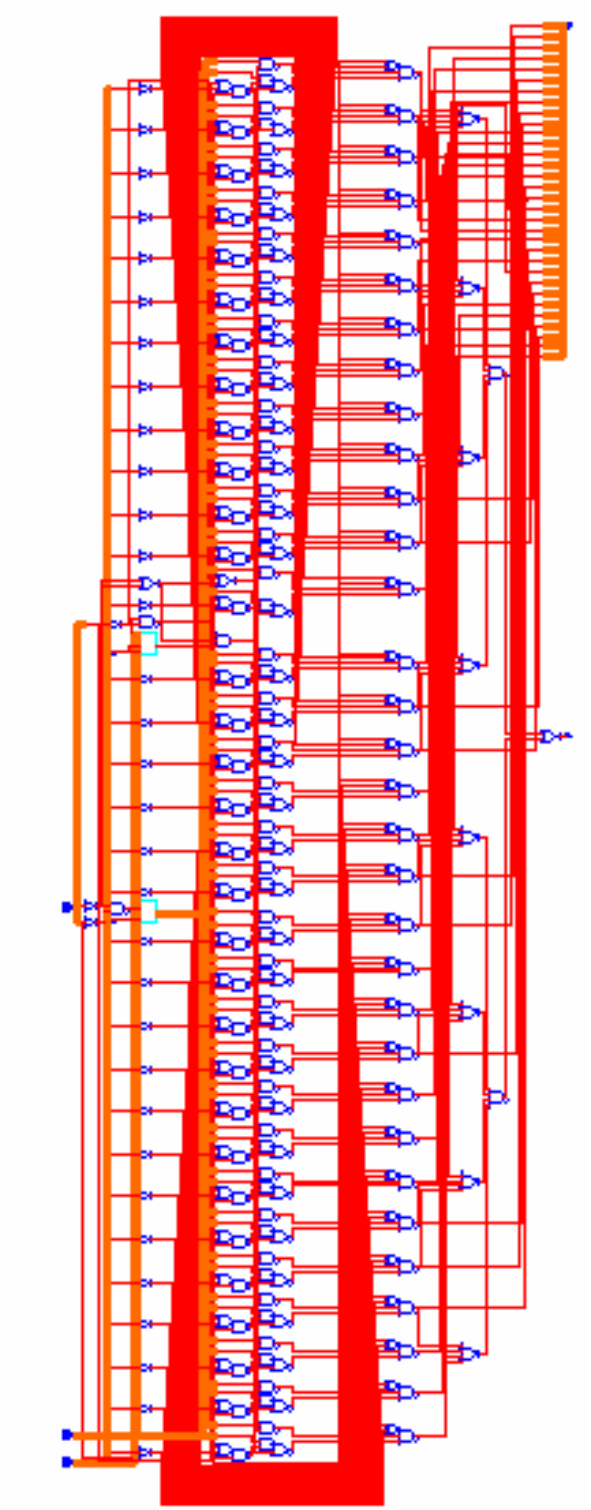
Celių skaičius 53
Užimamas plotas 215
Vidinis galingumas 219.9955 uW

16 bitų ALU įrenginys



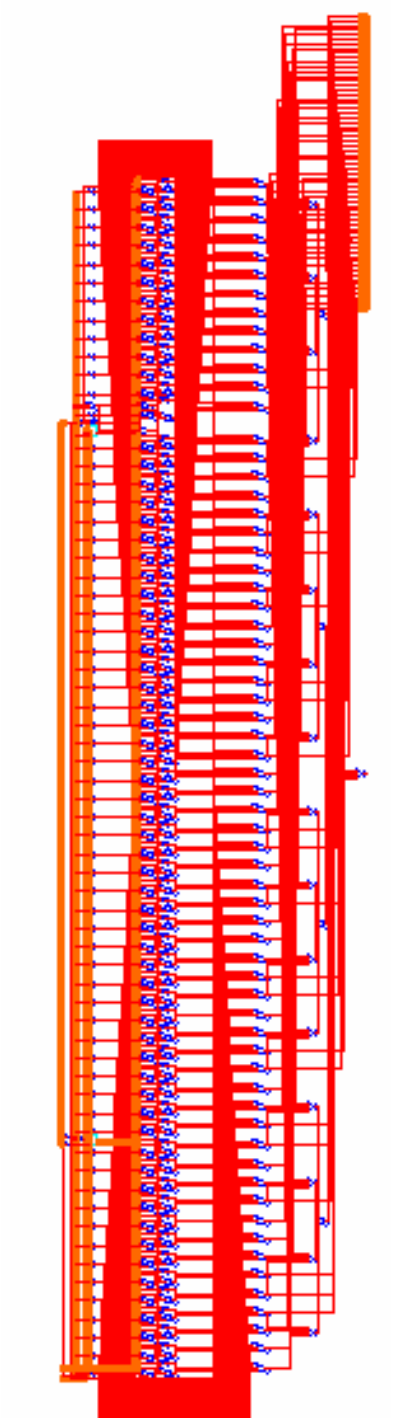
Celių skaičius 95
Užimamas plotas 428
Vidinis galingumas 442.0952 uW

32 bitų ALU įrenginys



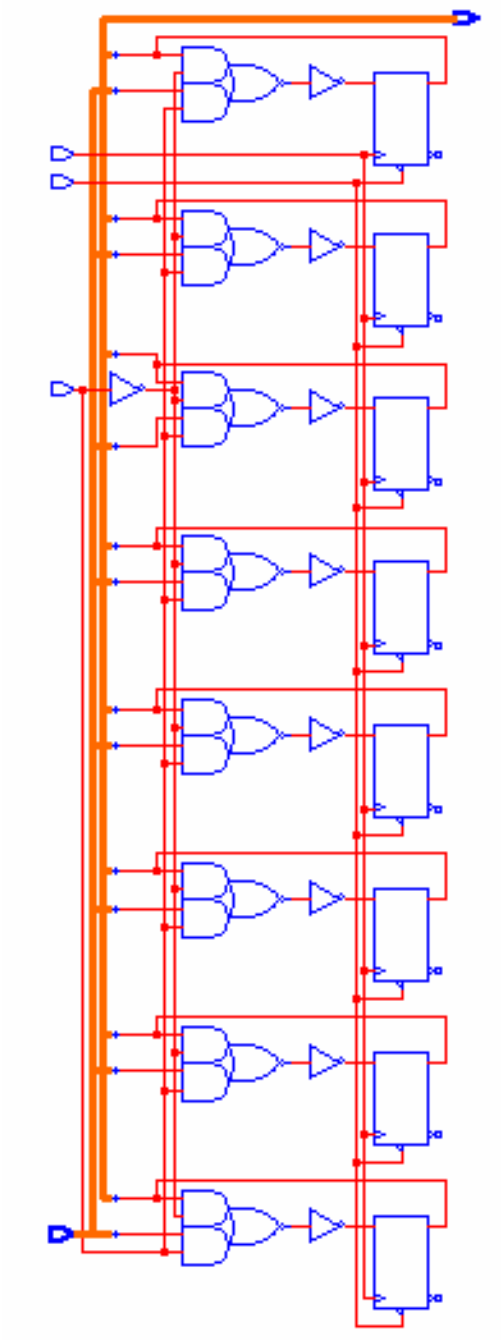
Celių skaičius 181
Užimamas plotas 854
Vidinis galingumas 885.1216 uW

64 bitų ALU įrenginys



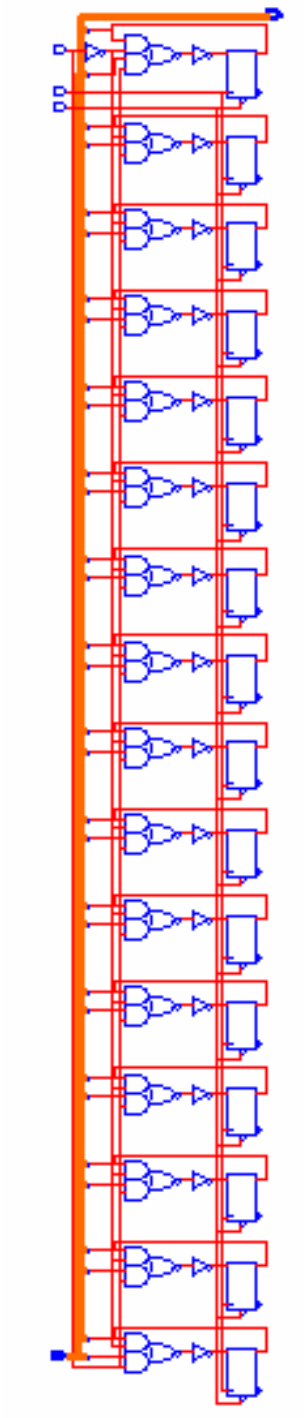
Celių skaičius  351
Užimamas plotas  1707
Vidinis galingumas  1.7681 mW

8 bitų duomenų registrai



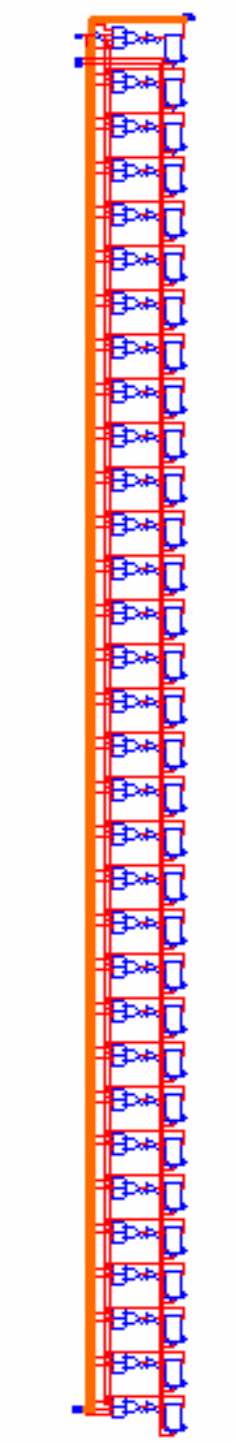
Celių skaičius  25
Užimamas plotas  97
Vidinis galingumas  20.5765 uW

16 bitų duomenų registrai



Celių skaičius  49
Užimamas plotas  193
Vidinis galingumas  41.0200 uW

32 bitų duomenų registrai

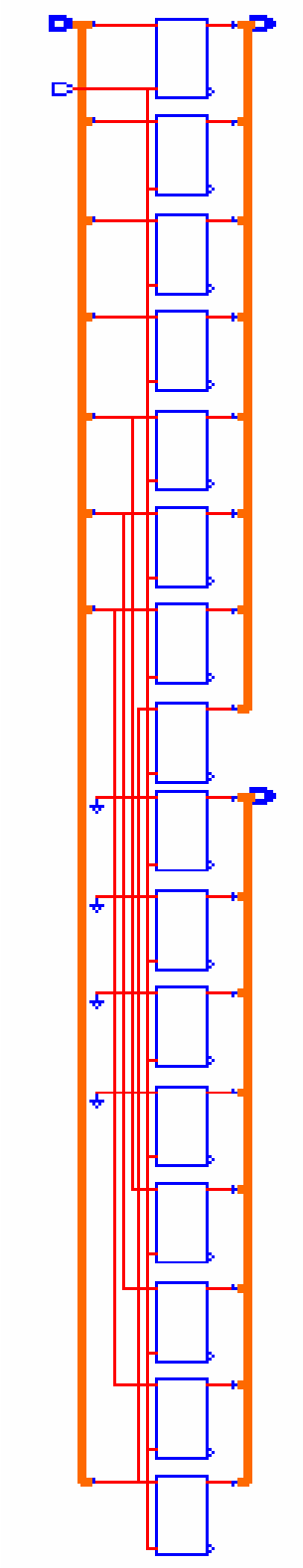


Celių skaičius  97
Užimamas plotas  385
Vidinis galingumas  82.3685 uW

64 bitų duomenų registrai

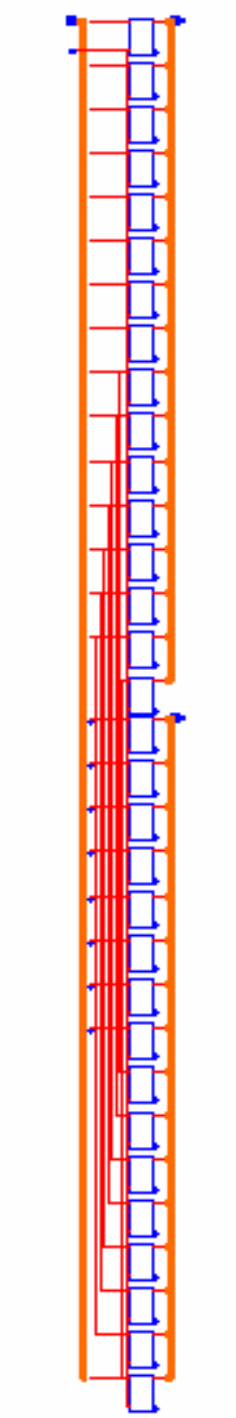
Celių skaičius 193
Užimamas plotas 769
Vidinis galingumas 163.8027 uW

8 bitų instrukcijų registrai



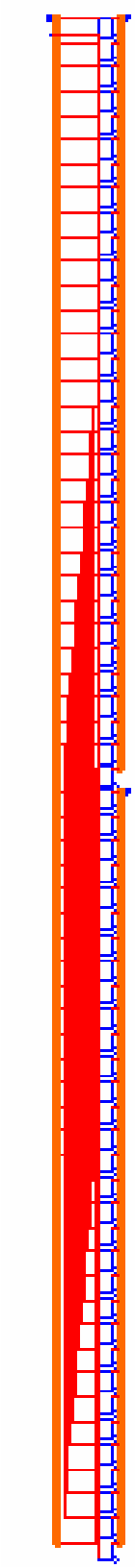
Celių skaičius  16
Užimamas plotas  80
Vidinis galingumas  14.2557 uW

16 bitų duomenų registrai



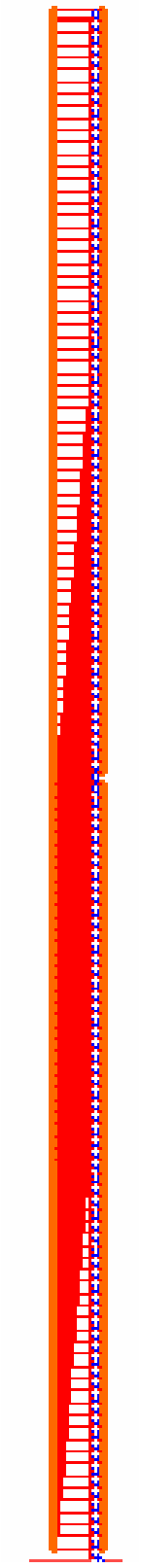
Celių skaičius  32
Užimamas plotas  160
Vidinis galingumas  28.7272 uW

32 bitų instrukcijų registrai



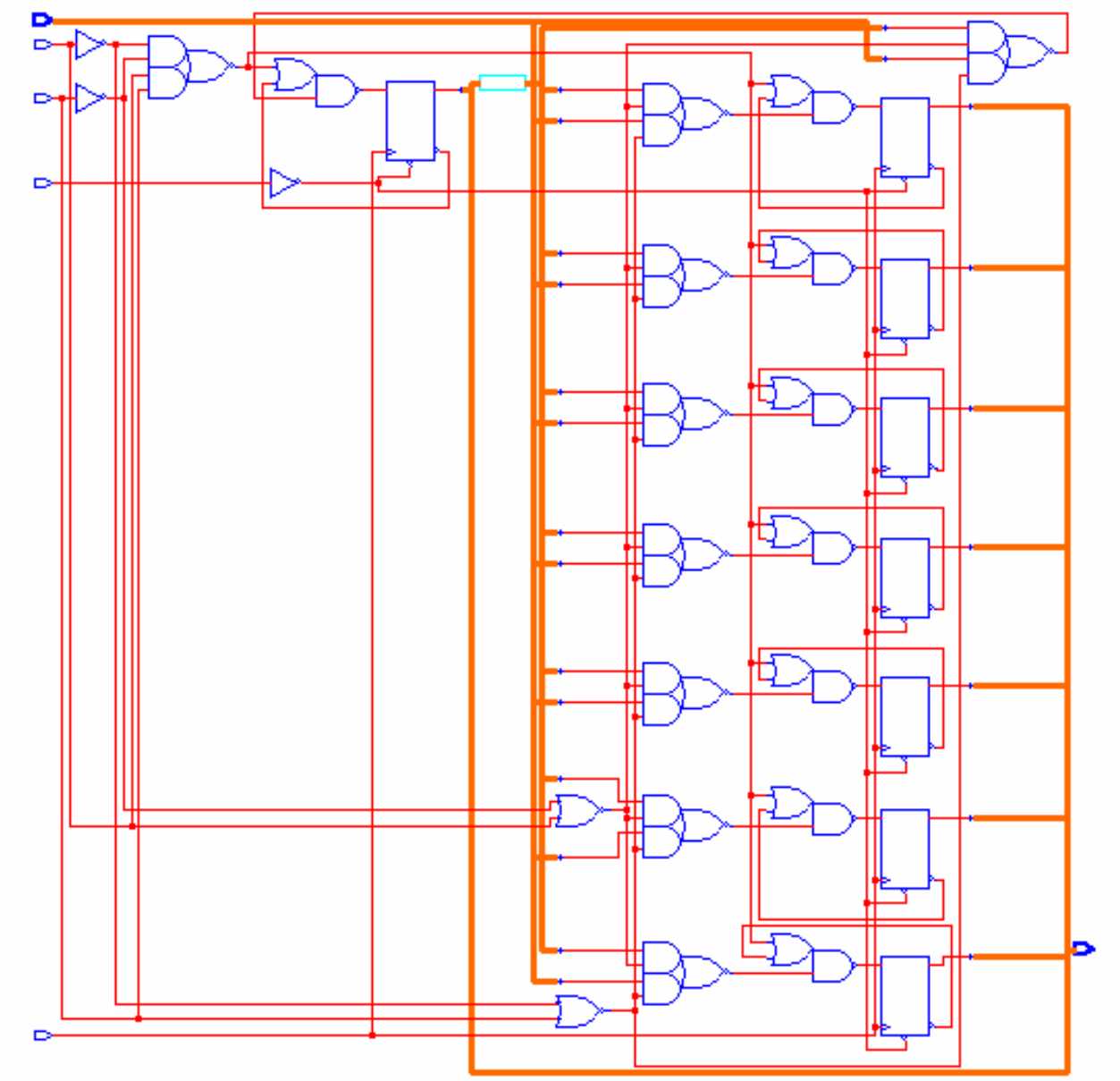
Celių skaičius  64
Užimamas plotas  320
Vidinis galingumas  56.9564 uW

64 bitų instrukcijų registrai



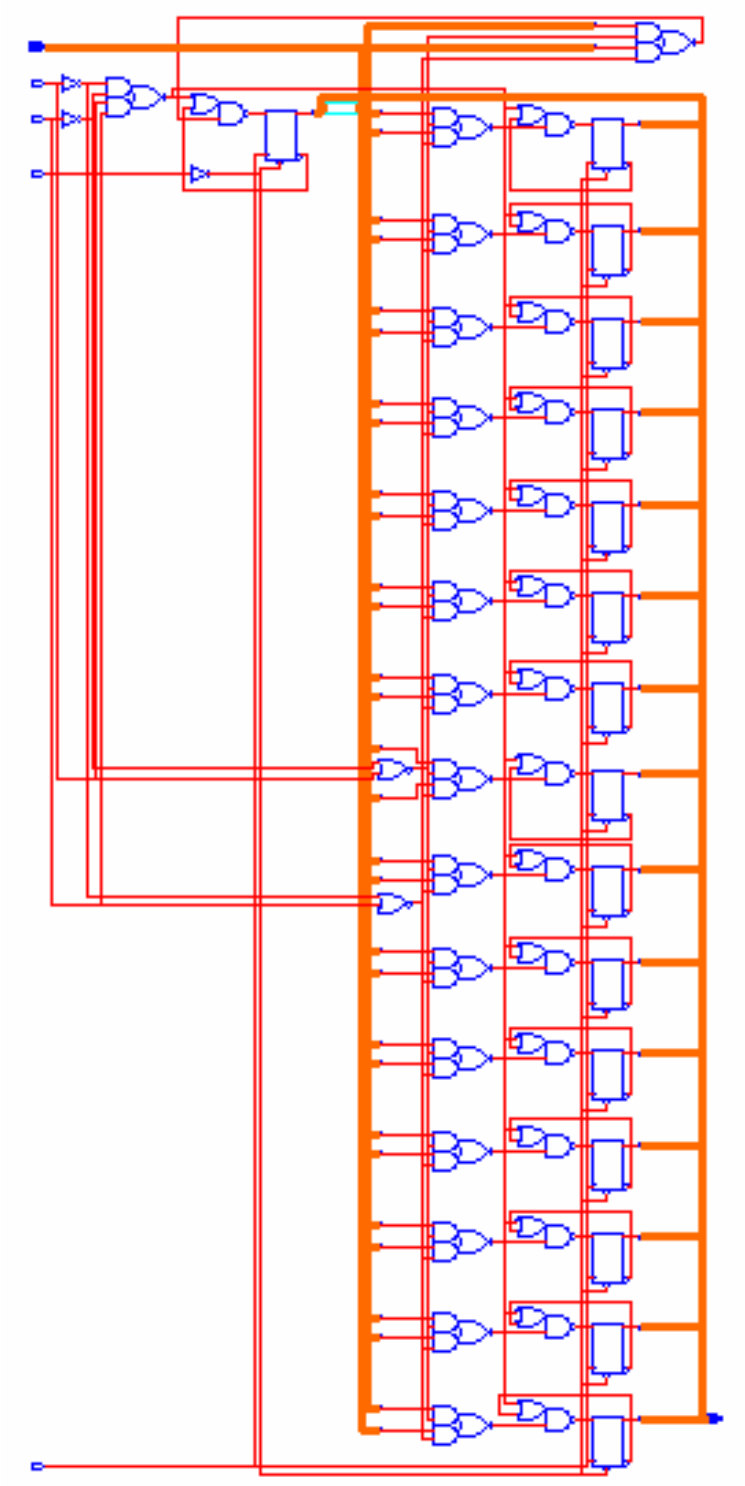
Celių skaičius  128
Užimamas plotas  640
Vidinis galingumas  113.1720 uW

8 bitų programos skaitiklio registras



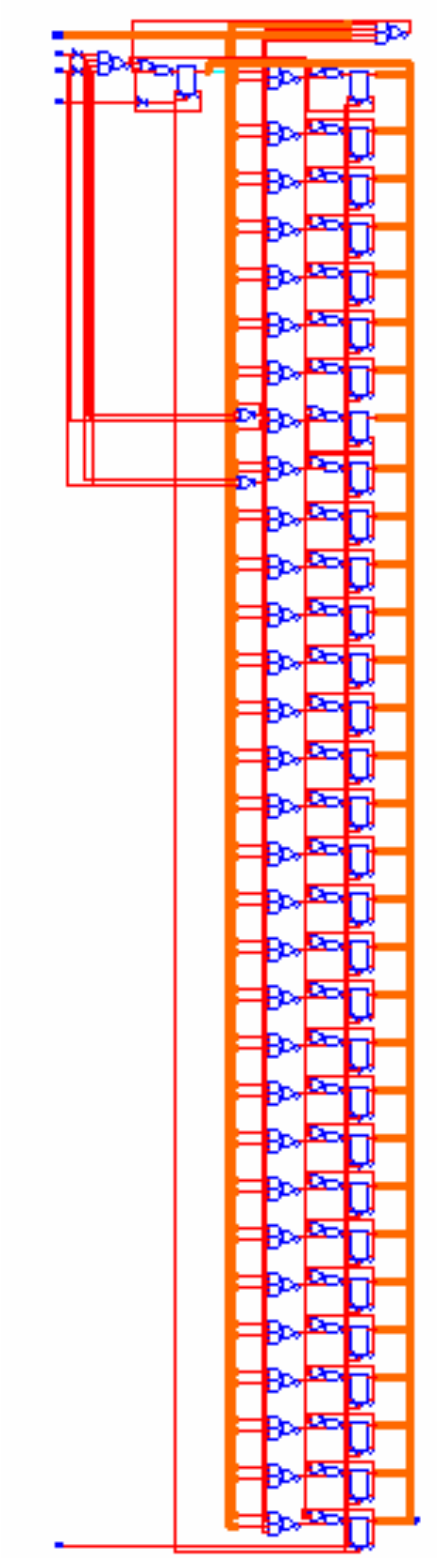
Celių skaičius 31
Užimamas plotas 142
Vidinis galingumas 46.8515 uW

16 bitų programos skaitiklio registras



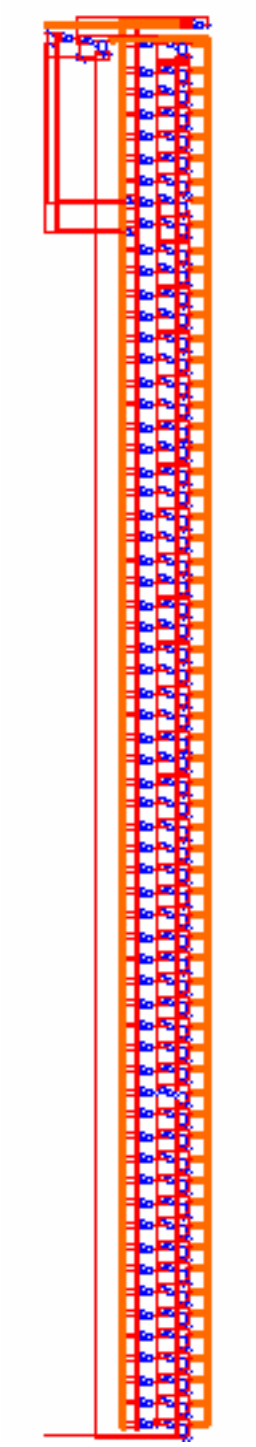
Celių skaičius 55
Užimamas plotas 282
Vidinis galingumas 87.8122 uW

32 bitų programos skaitiklio registras



Celių skaičius 103
Užimamas plotas 562
Vidinis galingumas 168.7212 uW

64 bitų programos skaitiklio registras



Celių skaičius 199
Užimamas plotas 1122
Vidinis galingumas 331.6020 uW

# 8. Priedas II

Tiriamų komponentų VHDL aprašai

## 8 bitų ALU

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

ENTITY alu8bit IS
port(a, b : in std_logic_vector(7 downto 0);
        op : in std_logic_vector(2 downto 0);
        zero : out std_logic;
   f : out std_logic_vector(7 downto 0));
END alu8bit;

architecture behave of alu8bit is
begin
process(op)
variable temp: std_logic_vector(7 downto 0);
begin
case op is
        when "000" =>
                temp := a and b;
        when "100" =>
                temp := a and b;
        when "001" =>
                temp := a or b;
        when "101" =>
                temp := a or b;
        when "010" =>
                temp := a + b;
        when "110" =>
                temp := a - b;
        when "111" =>
                if a < b then
                temp := "11111111";
                else
                temp := "00000000";
                end if;
        when others =>
                temp := a - b;
end case;
if temp="00000000" then
zero <= '1';
else
zero <= '0';
end if;
f <= temp;
end process;
end behave;
```

## 16 bitų ALU

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

ENTITY alu16bit IS
port(a, b : in std_logic_vector(15 downto 0);
        op : in std_logic_vector(2 downto 0);
        zero : out std_logic;
   f : out std_logic_vector(15 downto 0));
END alu16bit;

architecture behave of alu16bit is
```

```
begin
process(op)
variable temp: std_logic_vector(15 downto 0);
begin
case op is
        when "000" =>
                temp := a and b;
        when "100" =>
                temp := a and b;
        when "001" =>
                temp := a or b;
        when "101" =>
                temp := a or b;
        when "010" =>
                temp := a + b;
        when "110" =>
                temp := a - b;
        when "111" =>
                if a < b then
                temp := "1111111111111111";
                else
                temp := "0000000000000000";
                end if;
        when others =>
                temp := a - b;
end case;
if temp="0000000000000000" then
zero <= '1';
else
zero <= '0';
end if;
f <= temp;
end process;
end behave;
```

## 32 bitų ALU

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

ENTITY alu32bit IS
port(a, b : in std_logic_vector(31 downto 0);
        op : in std_logic_vector(2 downto 0);
        zero : out std_logic;
   f : out std_logic_vector(31 downto 0));
END alu32bit;

architecture behave of alu32bit is
begin
process(op)
variable temp: std_logic_vector(31 downto 0);
begin
case op is
        when "000" =>
                temp := a and b;
        when "100" =>
                temp := a and b;
        when "001" =>
                temp := a or b;
        when "101" =>
                temp := a or b;
        when "010" =>
                temp := a + b;
        when "110" =>
                temp := a - b;
        when "111" =>
                if a < b then
                temp :=
"11111111111111111111111111111111";
                else
                temp :=
"00000000000000000000000000000000";
```

```
                                        end if;
                when others =>
                                        temp := a - b;
        end case;
        if temp="00000000000000000000000000000000" then
        zero <= '1';
        else
        zero <= '0';
        end if;
        f <= temp;
        end process;
end behave;
```

## 64 bitų ALU

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

ENTITY alu64bit IS
        port(a, b : in std_logic_vector(63 downto 0);
                op : in std_logic_vector(2 downto 0);
                zero : out std_logic;
           f : out std_logic_vector(63 downto 0));
END alu64bit;

architecture behave of alu64bit is
begin
        process(op)
        variable temp: std_logic_vector(63 downto 0);
        begin
        case op is
                when "000" =>
                                        temp := a and b;
                when "100" =>
                                        temp := a and b;
                when "001" =>
                                        temp := a or b;
                when "101" =>
                                        temp := a or b;
                when "010" =>
                                        temp := a + b;
                when "110" =>
                                        temp := a - b;
                when "111" =>
                                        if a < b then
                                        temp :=
"11111111111111111111111111111111111111111111111111111111111111111";
                                        else
                                        temp :=
"00000000000000000000000000000000000000000000000000000000000000000";
                                        end if;
                when others =>
                                        temp := a - b;
        end case;
        if
temp="00000000000000000000000000000000000000000000000000000000000000000" then
        zero <= '1';
        else
        zero <= '0';
        end if;
        f <= temp;
        end process;
end behave;
```

## 8 bitų duomenų registrai

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

--------------------------------------------------
```

```
entity reg is

generic(n: natural :=8);
port(                                   I:          in std_logic_vector(n-1 downto 0);
                                        clock:      in std_logic;
                                        load:       in std_logic;
                                        clear:      in std_logic;
                                        Q:          out std_logic_vector(n-1 downto 0)
);
end reg;

-------------------------------------------------------

architecture behv of reg is

   signal Q_tmp: std_logic_vector(n-1 downto 0);

begin

   process(I, clock, load, clear)
   begin

                                        if clear = '0' then

         Q_tmp <= (Q_tmp'range => '0');

                                        elsif (clock='1' and clock'event) then
                                           if load = '1' then
                                                 Q_tmp <= I;
                                           end if;
                                        end if;

   end process;


   Q <= Q_tmp;

end behv;
```

## 16 bitų duomenų registrai

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

-------------------------------------------------------

entity reg is

generic(n: natural :=16);
port(                                   I:          in std_logic_vector(n-1 downto 0);
                                        clock:      in std_logic;
                                        load:       in std_logic;
                                        clear:      in std_logic;
                                        Q:          out std_logic_vector(n-1 downto 0)
);
end reg;

-------------------------------------------------------

architecture behv of reg is

   signal Q_tmp: std_logic_vector(n-1 downto 0);

begin

   process(I, clock, load, clear)
   begin

                                        if clear = '0' then

         Q_tmp <= (Q_tmp'range => '0');

                                        elsif (clock='1' and clock'event) then
                                           if load = '1' then
```

```
                    Q_tmp <= I;
                end if;
            end if;

    end process;


    Q <= Q_tmp;

end behv;
```

## 32 bitų duomenų registrai

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

---------------------------------------------------

entity reg is

generic(n: natural :=32);
port(                                   I:      in std_logic_vector(n-1 downto 0);
                                        clock:  in std_logic;
                                        load:   in std_logic;
                                        clear:  in std_logic;
                                        Q:      out std_logic_vector(n-1 downto 0)
);
end reg;

---------------------------------------------------

architecture behv of reg is

    signal Q_tmp: std_logic_vector(n-1 downto 0);

begin

    process(I, clock, load, clear)
    begin

                                        if clear = '0' then

        Q_tmp <= (Q_tmp'range => '0');

                                        elsif (clock='1' and clock'event) then
                                            if load = '1' then
                                                Q_tmp <= I;
                                            end if;
                                        end if;

    end process;


    Q <= Q_tmp;

end behv;
```

## 64 bitų duomenų registrai

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

---------------------------------------------------

entity reg is

generic(n: natural :=64);
port(                                   I:      in std_logic_vector(n-1 downto 0);
                                        clock:  in std_logic;
                                        load:   in std_logic;
                                        clear:  in std_logic;
                                        Q:      out std_logic_vector(n-1 downto 0)
```

```
);
end reg;

-------------------------------------------------------

architecture behv of reg is

   signal Q_tmp: std_logic_vector(n-1 downto 0);

begin

   process(I, clock, load, clear)
   begin

                                                        if clear = '0' then

         Q_tmp <= (Q_tmp'range => '0');

                                                        elsif (clock='1' and clock'event) then
                                                           if load = '1' then
                                                                 Q_tmp <= I;
                                                           end if;
                                                        end if;

   end process;


   Q <= Q_tmp;

end behv;
```

## 8 bitų instrukcijų registrai

```
Library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity IR is
port(                                                   IRin:      in std_logic_vector(7 downto 0);
                                                        IRld:      in std_logic;
                                                        dir_addr: out std_logic_vector(7 downto 0);
                                                        IRout:     out std_logic_vector(7 downto 0)

);
end IR;

architecture behv of IR is

begin
  process(IRld, IRin)
  begin
    if IRld = '1' then
                                                        IRout <= IRin;
                                                        dir_addr <= "0000" & IRin(3 downto 0);
    end if;
  end process;
end behv;
```

## 16 bitų instrukcijų registrai

```
Library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity IR is
port(                                                   IRin:      in std_logic_vector(15 downto 0);
                                                        IRld:      in std_logic;
                                                        dir_addr: out std_logic_vector(15 downto 0);
                                                        IRout:     out std_logic_vector(15 downto 0)

);
end IR;

architecture behv of IR is
```

```
begin
  process(IRld, IRin)
  begin
    if IRld = '1' then
                                                    IRout <= IRin;
                                                    dir_addr <= "00000000" & IRin(7 downto 0);
    end if;
  end process;
end behv;
```

## 32 bitų instrukcijų registrai

```
Library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity IR is
port(                                               IRin:       in std_logic_vector(31 downto 0);
                                                    IRld:       in std_logic;
                                                    dir_addr: out std_logic_vector(31 downto 0);
                                                    IRout:      out std_logic_vector(31 downto 0)

);
end IR;

architecture behv of IR is

begin
  process(IRld, IRin)
  begin
    if IRld = '1' then
                                                    IRout <= IRin;
                                                    dir_addr <= "0000000000000000" & IRin(15 downto 0);
    end if;
  end process;
end behv;
```

## 64 bitų instrukcijų registrai

```
Library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity IR is
port(                                               IRin:       in std_logic_vector(63 downto 0);
                                                    IRld:       in std_logic;
                                                    dir_addr: out std_logic_vector(63 downto 0);
                                                    IRout:      out std_logic_vector(63 downto 0)

);
end IR;

architecture behv of IR is

begin
  process(IRld, IRin)
  begin
    if IRld = '1' then
                                                    IRout <= IRin;
                                                    dir_addr <= "00000000000000000000000000000000" & IRin(31 downto 0);
    end if;
  end process;
end behv;
```

## 8 bitų programos skaitiklio registrai

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
```

```
use IEEE.std_logic_unsigned.all;
Entity PC is
Port (q_out : buffer std_logic_vector(7 downto 0);
--q_out : inout std_logic_vector(7 downto 0);
clk, clr : in std_logic;
D : in std_logic_vector(7 downto 0);
load, inc : in std_logic);
end entity;
architecture pc_arch of PC is
signal d_in : std_logic_vector(7 downto 0);
begin
it5: process (clk, clr)
begin
if (clr='1') then
q_out <= (others=>'0');
elsif (clk'event and clk='1') then
if ((inc='1') and (load='0')) then
q_out <= (q_out+1);
elsif ((load='1') and (inc='0')) then
q_out <= D;
else q_out <= q_out;
end if;
end if;
end process;
end architecture;
```

## 16 bitų programos skaitiklio registrai

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

use IEEE.std_logic_unsigned.all;
Entity PC is
Port (q_out : buffer std_logic_vector(15 downto 0);
--q_out : inout std_logic_vector(15 downto 0);
clk, clr : in std_logic;
D : in std_logic_vector(15 downto 0);
load, inc : in std_logic);
end entity;
architecture pc_arch of PC is
signal d_in : std_logic_vector(15 downto 0);
begin
it5: process (clk, clr)
begin
if (clr='1') then
q_out <= (others=>'0');
elsif (clk'event and clk='1') then
if ((inc='1') and (load='0')) then
q_out <= (q_out+1);
elsif ((load='1') and (inc='0')) then
q_out <= D;
else q_out <= q_out;
end if;
end if;
end process;
end architecture;
```

## 32 bitų programos skaitiklio registrai

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

use IEEE.std_logic_unsigned.all;
Entity PC is
Port (q_out : buffer std_logic_vector(31 downto 0);
--q_out : inout std_logic_vector(31 downto 0);
clk, clr : in std_logic;
D : in std_logic_vector(31 downto 0);
load, inc : in std_logic);
end entity;
architecture pc_arch of PC is
```

```
signal d_in : std_logic_vector(31 downto 0);
begin
it5: process (clk, clr)
begin
if (clr='1') then
q_out <= (others=>'0');
elsif (clk'event and clk='1') then
if ((inc='1') and (load='0')) then
q_out <= (q_out+1);
elsif ((load='1') and (inc='0')) then
q_out <= D;
else q_out <= q_out;
end if;
end if;
end process;
end architecture;
```

## 64 bitų programos skaitiklio registrai

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

use IEEE.std_logic_unsigned.all;
Entity PC is
Port (q_out : buffer std_logic_vector(63 downto 0);
--q_out : inout std_logic_vector(63 downto 0);
clk, clr : in std_logic;
D : in std_logic_vector(63 downto 0);
load, inc : in std_logic);
end entity;
architecture pc_arch of PC is
signal d_in : std_logic_vector(63 downto 0);
begin
it5: process (clk, clr)
begin
if (clr='1') then
q_out <= (others=>'0');
elsif (clk'event and clk='1') then
if ((inc='1') and (load='0')) then
q_out <= (q_out+1);
elsif ((load='1') and (inc='0')) then
q_out <= D;
else q_out <= q_out;
end if;
end if;
end process;
end architecture;
```