ХАРКІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
РАДІОЕЛЕКТРОНІКИ

На правах рукопису

Обрізан Володимир Ігорович

УДК 658: 512.011: 681.326: 519.713

Мультиверсний паралельний синтез цифрових структур на основі SystemC специфікації

05.13.05 – комп'ютерні системи і компоненти

Дисертація на здобуття наукового ступеня
кандидата технічних наук

Цей примірник дисертації ідентичний за змістом
з іншими примірниками, що подані до спеціалізованої
вченої ради Д 64.052.01

Учений секретар спеціалізованої
вченої ради Д 64.052.01 О. А. Винокурова

Науковий керівник:

доктор технічних наук, проф.

Хаханов Володимир Іванович

Харків – 2017

АНОТАЦІЯ

Обрізан Володимир Ігорович. Кваліфікаційна наукова праця на правах рукопису – Дисертація на здобуття наукового ступеня кандидата технічних наук (доктора філософії) за спеціальністю 05.13.05 «комп'ютерні системи та компоненти». – Харківський національний університет радіоелектроніки, Міністерство освіти і науки України, Харків, 2017.

Мета дослідження – суттєве зменшення часу проектування обчислювальних архітектур і підвищення якості цифрових виробів шляхом мультіверсного синтезу структури цифрового вироби на основі заданої специфікації в середовищі SystemC (C++) і автоматичному підборі функціональних компонентів за рахунок паралельного синтезу і верифікації архітектурних рішень системного рівня відповідно до запропонованої метрики.

Задачі:

1. Огляд існуючих моделей, методів, алгоритмів і програмних засобів створення цифрових систем на кристалах.

2. Розробка структур даних для опису функціональних примітивів системного рівня, орієнтованих на використання семантичних і синтаксичних конструкцій мови C ++ і SystemC з метою забезпечення паралельного синтезу і верифікації архітектурних рішень.

3. Розробка методу синтезу інтерфейсних структур і протоколів виконання транзакцій RT-рівня на основі аналізу специфікації SoC-архітектури системного рівня, яка використовує стандартну шину Wishbone обміну даними між функціональними модулями.

4. Розробка методу синтезу RTL-моделей функціональностей шляхом перетворення C++ і SystemC-описів цифрових блоків системного рівня в алгоритми і структури даних автоматної моделі Мура, що задана синтезованою підмножиною мовних конструкцій VHDL.

5. Розробка методу мультиверсного синтезу керуючих і операційних автоматів, орієнтованих на архітектурні рішення в метриці, що мінімізує час виконання функціональності за рахунок розпаралелювання операцій при обмеженні на апаратні витрати.

6. Програмна реалізація моделей і методів мультиверсної розробки операційних пристроїв в рамках інтегрованої системи проектування функціональних і архітектурних рішень SoC на основі використання продуктів верифікації та синтезу компаній Aldec і Xilinx.

Сутність ринково-орієнтованого науково-технічного дослідження полягає в мультіверсному проектуванні архітектури цифрового виробу на основі заданої специфікації в середовищі SystemC (C++) і автоматичному виборі синтезованих функціональних структур з метою істотного зменшення часу створення проекту і підвищенні виходу придатної продукції за рахунок паралельного синтезу та верифікації архітектурних рішень системного рівня відповідно до запропонованої метрики. Основна інноваційна ідея – паралельний автоматичний синтез квазіоптимальної архітектури відповідно до запропонованої специфікації і метрики на основі підбору синтезованих функціональних структур.

Наукова новизна отриманих результатів.

1. **Вперше** запропоновано метод синтезу інтерфейсних структур і протоколів виконання транзакцій RT-рівня на основі аналізу специфікації SoC-архітектури системного рівня, який характеризується використанням двобічної стандартної шини Wishbone обміну даними між функціональними модулями, що дозволяє здійснювати мультиверсне створення компонентів цифрових систем на кристалах.

2. **Вперше** запропоновано метод синтезу RTL-моделей функціональностей, який характеризується однозначним перетворенням C++ і SystemC-описів цифрових блоків системного рівня в алгоритми і структури даних автоматної моделі Мура, заданої синтезованою підмножиною мовних

конструкцій VHDL, що дає можливість істотно зменшити час виконання процесів проектування, тестування і верифікації.

3. **Удосконалено** структури даних для опису функціональних примітивів системного рівня, які відрізняються орієнтацією на використання семантичних і синтаксичних конструкцій мови C ++ і SystemC, що дозволяє здійснювати паралельний синтез і верифікацію архітектурних рішень.

4. **Удосконалено** метод мультиверсного синтезу керуючих і операційних автоматів, орієнтованих на архітектурні рішення в метриці, яка відрізняється мінімальним часом виконання функціональності за рахунок розпаралелювання операцій при обмеженні на апаратні витрати, що дозволило збільшити ефективність засобів автоматизованого проектування цифрових виробів.

Практична значущість отриманих результатів.

1. Розроблено програмні засоби для реалізації моделей і методів мультиверсного створення операційних пристроїв в рамках інтегрованої системи проектування функціональних і архітектурних рішень SoC на основі використання продуктів верифікації та синтезу компаній Aldec і Xilinx.

2. Проведено тестування і верифікацію програмних модулів мультиверсної розробки операційних пристроїв в рамках інтегрованої системи проектування функціональних і архітектурних рішень SoC на десяти прикладах реалізації промислово-орієнтованих функціональних блоків.

Практична реалізація полягає в розробці локальних і серверних програм, розміщених на хмарних сервісах Amazon Web Services. Система виконана за мікросервісною архітектурою, яка має властивості масштабованості, надійності, можливість розробляти та оновлювати модулі системи незалежно один від одного. Клієнтську програму реалізовано на мові C++, вона має графічний інтерфейс користувача з підсвічуванням синтаксису. Дані експерименту були отримані в результаті роботи 10 інженерів над одним проектом. Оцінювалися такі характеристики: часові витрати на проектування, швидкодія, енергоспоживання, площа на кристалі.

Було вибрано три найпоширеніші реалізації: послідовна, паралельна, конвейєрна. Результатом експерименту отримано, що в разі адитивної оцінці автоматичний метод краще в 1,3 раз. При мультипликативній оцінці автоматичний метод краще в 133,2 разу. В роботі показано, що значення ручного методу практично наближаються до нуля, в той час як в автоматичному методі значення наближаються до 1, що доводить те, що мультиверсний метод дійсно ефективніше.

Результати дисертаційної роботи відображені у 36 друкованих працях: 16 статей, серед яких 13 у наукових журналах, що входять до «Переліків наукових фахових видань України» (з них 12 – у міжнародних наукометричних базах), 3 статті в міжнародних наукових журналах за кордоном (з них 2 – в міжнародній наукометричній базі Scopus); а також 20 публікацій у міжнародних наукових конференціях (з них 8 за кордоном, 11 входять до наукометричної бази Scopus). Здобувач має 15 публікацій, що входять до наукометричної бази Scopus, та має індекс Хірша h=2.

СПИСОК ОПУБЛІКОВАНИХ РОБІТ ЗА ТЕМОЮ ДИСЕРТАЦІЇ

Список публікацій здобувача, в яких опубліковані основні наукові результати дисертації:

1.Хаханов В.И. Технология моделирования и синтеза тестов для сложных цифровых систем / В.И. Хаханов, К.В. Колесников, А.Н. Парфентий, И.В. Хаханова, В.И. Обризан, О.В. Мельникова // Радиоэлектроника и информатика. – 2003. – №1. – С. 70-78. (Входить до міжнародних наукометричних баз Index Copernicus, Google Scholar, OECSP, OAJI, Scholar Steer, SIS, Cyberleninka, CiteFactor, TIU Hannover, I2OR).

2.Hahanov V. Advanced Software Tools For Fault Simulation And Test Generation / V. Hahanov, A. Egorov, O. Melnikova, V. Obrizan, E. Kamenuka, O. Krapchunova, O. Guz // Радиоэлектроника и информатика. – 2003. – №3. – С. 77-81. (Входить до міжнародних наукометричних баз Index Copernicus, Google Scholar, OECSP, OAJI, Scholar Steer, SIS, Cyberleninka, CiteFactor, TIU Hannover, I2OR).

3.Obrizan V. A Method of High-Level Synthesis and Verification with SystemC Language / V. Obrizan // Радиоэлектроника и информатика. – 2010. – №4. – С. 47-50. (Входить до міжнародних наукометричних баз Index Copernicus, Google Scholar, OECSP, OAJI, Scholar Steer, SIS, Cyberleninka, CiteFactor, TIU Hannover, I2OR).

4.Obrizan V. Matrix-Model for Diagnosing SoC HDL-Code / V. Obrizan, I. Yemelyanov, V. Hahanov, E. Litvinova // Radioelektroniks and informatics. – 2013. – №1. – P.12-19. (Входить до міжнародних наукометричних баз Index Copernicus, Google Scholar, OECSP, OAJI, Scholar Steer, SIS, Cyberleninka**,** CiteFactor**,** TIU Hannover, I2OR).

5.Обризан В.И. Метрика для анализа Big Data / В.И. Обризан, А.С. Мищенко, В.И. Хаханов, Tamer Bani Amer // Радиоэлектроника и информатика. – 2014. – № 2. – С. 26-29. (Входить до міжнародних наукометричних баз Index Copernicus, Google Scholar, OECSP, OAJI, Scholar Steer, SIS, Cyberleninka, CiteFactor**,** TIU Hannover, I2OR).

6.Обризан В.И. Киберфизические системы как технологии киберуправления (аналитический обзор) / В.И. Обризан, А.С. Мищенко, В.И. Хаханов, И.В. Филиппенко // Радиоэлектроника и информатика. – 2014. – № 1. – С. 39-45. (Входить до міжнародних наукометричних баз Index Copernicus, Google Scholar, OECSP, OAJI, Scholar Steer, SIS, Cyberleninka, CiteFactor, TIU Hannover, I2OR).

7.Обризан В.И. Инфраструктура проектирования SOC для метода мультиверсного синтеза / В.И. Обризан // Радиоэлектроника и информатика. – 2016. – №2. – С. 48-60. (Входить до міжнародних наукометричних баз Index Copernicus, Google Scholar, OECSP, OAJI, Scholar Steer, SIS, Cyberleninka, CiteFactor, TIU Hannover, I2OR).

8.Hahanov V. High Performance Fault Simulation For Digital Systems / V. Hahanov, G. Krivoulya, I. Hahanova, O. Melnikova, V. Obrizan // International Scientific Journal of Computing. **–** 2003. **–** Vol. 2, Issue 2. – P. 114-121. (Входить

до міжнародних наукометричних баз Index Copernicus, Norwegian Social Science Data Services (NSD), Google Scholar, Vernadsky National Library of Ukraine).

9.Hahanov V. Algebra-logical diagnosis model for SoC F-IP / V. Hahanov, V. Obrizan, E. Litvinova, Man K.L. // WSEAS Transactions on Circuits and Systems. – 2008. – No 7. – P. 708-717. (Входить до міжнародних наукометричних баз Scopus, Elsevier, Google Scholar).

10.Hahanov V. Embedded method of SoC diagnosis / V. Hahanov, E. Litvinova, V. Obrizan, W. Gharibi // Elektronika ir Elektrotechnika. – 2008. – No8. – P. 3-8. (Входить до міжнародних наукометричних баз Scopus, Thomson Reuters (ISI), Web of Knowledge Citation Databases, Science Citation Index Expanded (SCIE); Journal Citation Reports (JCR); INSPEC; VINITI; EBSCO Publishing).

11.Хаханов В.И. Обзор международного рынка электронных технологий / В.И. Хаханов, В.И. Обризан, О.В. Мельникова // Вестник НТУ «ХПИ». – Серия: Информатика и моделирование. – 2004. – Вып. 46. – С.111-115.

12.Хаханов В.И. Assert-метод верификации цифровых систем на основе стандарта IEEE 1500 SECT / В.И. Хаханов, В.В. Елисеев, В.И. Обризан // АСУ и приборы автоматики. – 2005. – Вып. 132. – С. 93–105. (Входить до міжнародних наукометричних баз Google Scholar, Cyberleninka).

13.Хаханов В.И. Иерархическое тестирование программно-технических комплексов / В.И. Хаханов, В.В. Елисеев, В.И. Обризан // АСУ и приборы автоматики. – Харьков, 2006. – Вып. 134. – С. 93–102. (Входить до міжнародних наукометричних баз Google Scholar, Cyberleninka).

14.Михтонюк С.В. Архитектурная модель масштабируемой системы асинхронной обработки больших объемов данных / С.В. Михтонюк, Р.С. Хван, В.И. Обризан // АСУ и приборы автоматики. – 2008. – Вып. 142. – С. 13-17. (Входить до міжнародних наукометричних баз Google Scholar, Cyberleninka).

15.Хаханов В.И. MQT-автомат для анализа больших данных / В.И. Хаханов, В.И. Обризан, С.А. Зайченко, И.В. Хаханов // АСУ и приборы автоматики. – 2014. – Вып. 168. – С. 64-72. (Входить до міжнародних наукометричних баз Google Scholar, Cyberleninka).

Результати, які засвідчують апробацію матеріалів дисертації:

16.Hahanov V.I. Sigetest – fault simulation and test generation for digital designs / V.I. Hahanov, D.M. Gorbunov, Y.V. Miroshnychenko, O.V. Melnikova, V.I. Obrizan, E.A. Kamenuka // Современные технологии проектирования систем на микросхемах программируемой логики. – 23 сентября 2003. – Харьков. – С. 50-53.

17.Hahanov V. High performance Fault Simulation for Digital Systems / V. Hahanov, O. Melnikova, V. Obrizan, I. Hahanova // Proc. of the Euromicro Symposium on Digital Systems Design. – Turkey. Belek-Antalya. – 2003. – P. 15-16. (Входить до міжнародних наукометричних баз Scopus, IEEE Xplore).

18.Мирошниченко Я.В. Система моделирования неисправностей для дискретных устройств / Я.В. Мирошниченко, О.В. Мельникова, В.И. Обризан / Материалы 7-го молодежного форума «Радиоэлектроника и молодежь в XXI веке». – Украина, Харьков: ХНУРЭ. – 2003. – С. 463.

19.Hahanov V.I. New Features of Deductive Fault Simulation / V.I. Hahanov, V.I. Obrizan, A.V. Kiyaszhenko, I.A. Pobezhenko // Proc. of the 2nd East-West Design and Test Workshop 2004. – September 23-26, 2004. – Alushta. – 2004. – P. 274-280.

20.Шабанов-Кушнаренко Ю.П. Параллелизм мозгоподобных вычислений / Ю.П. Шабанов-Кушнаренко, В.И. Обризан // Материалы 5-го Международного научно-практического семинара «Высокопроизводи-тельные параллельные вычисления на кластерных системах». – 22-25 ноября 2005. – Харьков. – С. 196- 203.

21.Hahanov V.I. High-performance deductive fault simulation method / V.I. Hahanov, I. Hahanova, V.I. Obrizan // Proc. of the 10th IEEE European Test Sym-

posium. – May 22-25, 2005. – Estonia. – Tallinn, 2005. – P. 91-96. (Входить до міжнародних наукометричних баз Scopus, IEEE Xplore).

22.Hahanov V. Hierarchical Testing of Complex Digital Systems / V. Hahanov, V. Obrizan, V. Yeliseev, W. Ghribi // Proc. of the International Conference “Modern Problems of Radio Engineering, Telecommunications and Computer Science (TCSET'2006)”. – February 28 – March 4, 2006. – Slavske, Lviv, Ukraine. 2006. – P. 426-429. (Входить до міжнародних наукометричних баз Scopus, IEEE Xplore).

23.Hahanov V. Verification of digital system by a new asserting mechanism based on IEEE 1500 SECT standard / V. Hahanov, V. Obrizan, I. Hahanova, E. Fomina // Proc. of the International Conference MIXDES 2006. – June 22-24, 2006. – Gdunia, Poland. – 2006. – P. 544-548. (Входить до міжнародних наукометричних баз Scopus, IEEE Xplore).

24.Хаханов В.И. SIGETEST – система моделирования тестов проверки неисправностей цифровых устройств / В.И. Хаханов, В.И. Обризан, Я.В. Мирошниченко, О.В. Мельникова // Каталог аннотаций и разработок по материалам первого украинско-китайского форума «Наука – производство». – Харьков: ХНУРЭ, 2007. – С. 25–26.

25.Hahanov V. Hardware Simulation and Verification Technologies / V. Hahanov, A. Hahanova, V. Obrizan, W. Ghribi // Proc. of the IEEE East-West Design and Test Symposium. – September 7-10, 2007. – Yerevan, Armenia. – Yerevan, 2007. – P. 739-744.

26.Hahanov V. Technologies for hardware simulation and verification / V. Hahanov, A. Hahanova, V. Obrizan, K. Zaharov // Proc. of the International Conference Modern Problems of Radio Engineering, Telecommunications and Computer Science. – TCSET'2008. – February 19-23, 2008. – Slavske, Lviv, Ukraine. – 2008. – P. 560-564. (Входить до міжнародних наукометричних баз Scopus, IEEE Xplore).

27.Hahanov V. Testing challenges of SoC hardware-software components / V. Hahanov, V. Obrizan, S. Miroshnichenko, A. Gorobets // Proc. of the IEEE

East-West Design and Test International Symposium. – October 9-12, 2008. – Lviv, Ukraine. – 2008. – P. 149-154. (Входить до міжнародних наукометричних баз Scopus, IEEE Xplore).

28.Obrizan V. A method for automatic generation of an RTL-interface from a C++ description / V. Obrizan // Proc. of the East-West Design & Test Symposium (EWDTS) 2010. – 17-20 Sept. 2010. – St. Petersburg, Russia. – P.186-189. (Входить до міжнародних наукометричних баз Scopus, IEEE Xplore).

29.Hahanova I.V. Transaction level model of embedded processor for vector-logical analysis / I.V. Hahanova, V. Obrizan, A. Adamov, D. Shcherbin // Proc. of the East-West Design & Test Symposium. – 27-30 Sept. 2013. – Rostov-on-Don, Russia. – 4 p. (Входить до міжнародних наукометричних баз Scopus, IEEE Xplore).

30.Hahanova Yu. Metric for Analyzing Big Data / Yu. Hahanova, I. Yemelyanov, V. Obrizan, D. Krulevska, M. Skorobogatiy, A. Hahanova // Матеріали XIII Міжнародної науково-технічної конференции CADSM 2015 «Досвід розробки та застосування приладнати-технологічних САПР в мікроелектроніці». 24-27 лютого 2015. Львів – Поляна. – С.81-83. (Входить до міжнародних наукометричних баз Scopus, IEEE Xplore).

31.Обризан В.И. Мультиарендные облачные сервисы / В.И. Обризан, Ю.В. Ломова // Материалы XIX Международного молодежного форума «Радиоэлектроника и молодежь в XXI веке». – Украина, Харьков: ХНУРЭ. – 20-22 апреля 2015. – Ч. 5. – С.34-35.

32.Obrizan V. Multiversion parallel synthesis of digital structures based on SystemC specification / V.Obrizan; T. Soklakova // Proc. of the IEEE East-West Design & Test Symposium (EWDTS). – 2016 – Yerevan, Armenia. – 6p. (Входить до міжнародних наукометричних баз Scopus, IEEE Xplore).

Публікації, які додатково відображають наукові результати дисертації:

33.Гайдук С.М. Сферический мультипроцессор PRUS для решения булевых уравнений / С.М. Гайдук, В.И. Хаханов, В.И. Обризан, Е.А. Каменюка // Радиоэлектроника и информатика. – Харьков, 2004. – № 4 (29). – С. 107–

116. (Входить до міжнародних наукометричних баз Index Copernicus, Google Scholar, OECSP, OAJI, Scholar Steer, SIS, Cyberleninka, CiteFactor, TIU Hannover, I2OR).

34.Hyduke S. PRUS – Spherical Multiprocessor for Computation of Boolean equations / S. Hyduke, V.I. Hahanov, V.I. Obrizan, Wade Ghribi // Proceedings of the 8th International Conference CADSM 2005. – Ukraine. – Lviv, 2005. – P. 41-48.

35.Hyduke S.M. PRUS – Processor Network for Digital Circuit Implementation / S.M. Hyduke, V.I. Hahanov, V.I. Obrizan, O. Guz // Proc. of the 8th Euromicro Conference on Digital System Design. – August 30 – September 3, 2005. – Porto, Portugal. – 2005. – P. 239-242. (Входить до міжнародних наукометричних баз Scopus, IEEE Xplore).

36.Шевченко А.А. Математические модели описания потенциально опасных объектов газотранспортной промышлености / А.А. Шевченко, О.А. Довгошея, В.И. Обризан, Д.Н. Красильников // Материалы 2-го Международного радиоэлектронного форума «Прикладная радиоэлектроника». – 19-23 сентября 2005. – Украина. – С. 272- 276.

Ключові слова: мультиверсний синтез, специфікація, інтерфейсні структури, операційний пристрій, верифікація, цифрові системи на кристалах.

ABSTRACT

Obrizan Vladimir Igorevich. Multi population-inverted parallel synthesis of digital structures based on SystemC specification. – Manuscript. PhD thesis (candidate degree of technical sciences) in speciality 05.13.05 – Computer Systems and Components. – Kharkiv National University of Radio Electronics, Ministry of Education and Science of Ukraine, Kharkiv, 2017

The purpose of the research – a significant reduction time in design of computing architectures and increasing quality of digital products by multi population-inverted synthesis structure of the digital products based on a predetermined specification in SystemC environments (C ++) and automatic selection of functional components by parallel synthesis and verification of system-level architectural decisions in accordance the proposed metric.

The main results:

1) For the first time proposed a method for the synthesis of interface structures and protocols perform transactions RT-level on the basis of the analysis of SoC-architecture system-level specification, which is characterized by the use of bi-directional standard Wishbone bus communication between functional modules.

2) For the first time proposed a method of synthesizing the RTL-model functionality, which is characterized by the transformation uniquely C ++ - and SystemC-system level descriptions of digital blocks in algorithms and data structures automaton model given Moore synthesizable subset of VHDL language constructs.

3) Obtained improved data structure for describing the functional primitive system-level, oriented to usage of semantic and syntax of C ++ and the SystemC, which differ in terms of providing for parallel synthesis and verification of architectural solutions.

4) Improved method multi population-inverted synthesis of control and operating machines, focused on the architectural solutions in the metric, which is charac-

terized by minimal runtime functionality by parallelizing operations while limiting on hardware costs.

Analysis was carried out automatic and manual method by experiment, where an automatic method – a method derived in this paper. The experimental data were obtained in the investigation work of 10 engineers on a project. Evaluated the following characteristics: time spent on design, performance, power consumption, an area on the chip. the three most common implementation were selected: sequential, parallel, pipelined. Experimental results are obtained in the case of additive evaluation of an automatic method for better times 1.301292012.

List of publications

List of publications, in which the main scientific results of the thesis have been published:

1.Хаханов В.И. Технология моделирования и синтеза тестов для сложных цифровых систем / В.И. Хаханов, К.В. Колесников, А.Н. Парфентий, И.В. Хаханова, В.И. Обризан, О.В. Мельникова // Радиоэлектроника и информатика. – 2003. – №1. – С. 70-78. (Входить до міжнародних наукометричних баз Index Copernicus, Google Scholar, OECSP, OAJI, Scholar Steer, SIS, Cyberleninka, CiteFactor, TIU Hannover, I2OR).

2.Hahanov V. Advanced Software Tools For Fault Simulation And Test Generation / V. Hahanov, A. Egorov, O. Melnikova, V. Obrizan, E. Kamenuka, O. Krapchunova, O. Guz // Радиоэлектроника и информатика. – 2003. – №3. – С. 77-81. (Входить до міжнародних наукометричних баз Index Copernicus, Google Scholar, OECSP, OAJI, Scholar Steer, SIS, Cyberleninka, CiteFactor, TIU Hannover, I2OR).

3.Obrizan V. A Method of High-Level Synthesis and Verification with SystemC Language / V. Obrizan // Радиоэлектроника и информатика. – 2010. – №4. – С. 47-50. (Входить до міжнародних наукометричних баз Index Copernicus, Google Scholar, OECSP, OAJI, Scholar Steer, SIS, Cyberleninka, CiteFactor, TIU Hannover, I2OR).

4.Obrizan V. Matrix-Model for Diagnosing SoC HDL-Code / V. Obrizan, I. Yemelyanov, V. Hahanov, E. Litvinova // Radioelektroniks and informatics. – 2013. – №1. – P.12-19. (Входить до міжнародних наукометричних баз Index Copernicus, Google Scholar, OECSP, OAJI, Scholar Steer, SIS, Cyberleninka, CiteFactor, TIU Hannover, I2OR).

5.Обризан В.И. Метрика для анализа Big Data / В.И. Обризан, А.С. Мищенко, В.И. Хаханов, Tamer Bani Amer // Радиоэлектроника и информатика. – 2014. – № 2. – С. 26-29. (Входить до міжнародних наукометричних баз Index Copernicus, Google Scholar, OECSP, OAJI, Scholar Steer, SIS, Cyberleninka, CiteFactor, TIU Hannover, I2OR).

6.Обризан В.И. Киберфизические системы как технологии киберуправления (аналитический обзор) / В.И. Обризан, А.С. Мищенко, В.И. Хаханов, И.В. Филиппенко // Радиоэлектроника и информатика. – 2014. – № 1. – С. 39-45. (Входить до міжнародних наукометричних баз Index Copernicus, Google Scholar, OECSP, OAJI, Scholar Steer, SIS, Cyberleninka, CiteFactor, TIU Hannover, I2OR).

7.Обризан В.И. Инфраструктура проектирования SOC для метода мультиверсного синтеза / В.И. Обризан // Радиоэлектроника и информатика. – 2016. – №2. – С. 48-60. (Входить до міжнародних наукометричних баз Index Copernicus, Google Scholar, OECSP, OAJI, Scholar Steer, SIS, Cyberleninka, CiteFactor, TIU Hannover, I2OR).

8.Hahanov V. High Performance Fault Simulation For Digital Systems / V. Hahanov, G. Krivoulya, I. Hahanova, O. Melnikova, V. Obrizan // International Scientific Journal of Computing. – 2003. – Vol. 2, Issue 2. – P. 114-121. (Входить до міжнародних наукометричних баз Index Copernicus, Norwegian Social Science Data Services (NSD), Google Scholar, Vernadsky National Library of Ukraine).

9.Hahanov V. Algebra-logical diagnosis model for SoC F-IP / V. Hahanov, V. Obrizan, E. Litvinova, Man K.L. // WSEAS Transactions on Circuits and

Systems. – 2008. – No 7. – P. 708-717. (Входить до міжнародних наукометричних баз Scopus, Elsevier, Google Scholar).

10.Hahanov V. Embedded method of SoC diagnosis / V. Hahanov, E. Litvinova, V. Obrizan, W. Gharibi // Elektronika ir Elektrotechnika. – 2008. – No8. – P. 3-8. (Входить до міжнародних наукометричних баз Scopus, Thomson Reuters (ISI), Web of Knowledge Citation Databases, Science Citation Index Expanded (SCIE); Journal Citation Reports (JCR); INSPEC; VINITI; EBSCO Publishing).

11.Хаханов В.И. Обзор международного рынка электронных технологий / В.И. Хаханов, В.И. Обризан, О.В. Мельникова // Вестник НТУ «ХПИ». – Серия: Информатика и моделирование. – 2004. – Вып. 46. – С.111-115.

12.Хаханов В.И. Assert-метод верификации цифровых систем на основе стандарта IEEE 1500 SECT / В.И. Хаханов, В.В. Елисеев, В.И. Обризан // АСУ и приборы автоматики. – 2005. – Вып. 132. – С. 93–105. (Входить до міжнародних наукометричних баз Google Scholar, Cyberleninka).

13.Хаханов В.И. Иерархическое тестирование программно-технических комплексов / В.И. Хаханов, В.В. Елисеев, В.И. Обризан // АСУ и приборы автоматики. – Харьков, 2006. – Вып. 134. – С. 93–102. (Входить до міжнародних наукометричних баз Google Scholar, Cyberleninka).

14.Михтонюк С.В. Архитектурная модель масштабируемой системы асинхронной обработки больших объемов данных / С.В. Михтонюк, Р.С. Хван, В.И. Обризан // АСУ и приборы автоматики. – 2008. – Вып. 142. – С. 13-17. (Входить до міжнародних наукометричних баз Google Scholar, Cyberleninka).

15.Хаханов В.И. MQT-автомат для анализа больших данных / В.И. Хаханов, В.И. Обризан, С.А. Зайченко, И.В. Хаханов // АСУ и приборы автоматики. – 2014. – Вып. 168. – С. 64-72. (Входить до міжнародних наукометричних баз Google Scholar, Cyberleninka).

Results, which confirm approbation of the thesis:

16.Hahanov V.I. Sigetest – fault simulation and test generation for digital designs / V.I. Hahanov, D.M. Gorbunov, Y.V. Miroshnychenko, O.V. Melnikova, V.I. Obrizan, E.A. Kamenuka // Современные технологии проектирования систем на микросхемах программируемой логики. – 23 сентября 2003. – Харьков. – С. 50-53.

17.Hahanov V. High performance Fault Simulation for Digital Systems / V. Hahanov, O. Melnikova, V. Obrizan, I. Hahanova // Proc. of the Euromicro Symposium on Digital Systems Design. – Turkey. Belek-Antalya. – 2003. – P. 15-16. (Входить до міжнародних наукометричних баз Scopus, IEEE Xplore).

18.Мирошниченко Я.В. Система моделирования неисправностей для дискретных устройств / Я.В. Мирошниченко, О.В. Мельникова, В.И. Обризан / Материалы 7-го молодежного форума «Радиоэлектроника и молодежь в XXI веке». – Украина, Харьков: ХНУРЭ. – 2003. – С. 463.

19.Hahanov V.I. New Features of Deductive Fault Simulation / V.I. Hahanov, V.I. Obrizan, A.V. Kiyaszhenko, I.A. Pobezhenko // Proc. of the 2nd East-West Design and Test Workshop 2004. – September 23-26, 2004. – Alushta. – 2004. – P. 274-280.

20.Шабанов-Кушнаренко Ю.П. Параллелизм мозгоподобных вычислений / Ю.П. Шабанов-Кушнаренко, В.И. Обризан // Материалы 5-го Международного научно-практического семинара «Высокопроизводи-тельные параллельные вычисления на кластерных системах». – 22-25 ноября 2005. – Харьков. – С. 196- 203.

21.Hahanov V.I. High-performance deductive fault simulation method / V.I. Hahanov, I. Hahanova, V.I. Obrizan // Proc. of the 10th IEEE European Test Symposium. – May 22-25, 2005. – Estonia. – Tallinn, 2005. – P. 91-96. (Входить до міжнародних наукометричних баз Scopus, IEEE Xplore).

22.Hahanov V. Hierarchical Testing of Complex Digital Systems / V. Hahanov, V. Obrizan, V. Yeliseev, W. Ghribi // Proc. of the International Conference “Modern Problems of Radio Engineering, Telecommunications and Computer Science (TCSET'2006)”. – February 28 – March 4, 2006. – Slavske,

Lviv, Ukraine. 2006. – P. 426-429. (Входить до міжнародних наукометричних баз Scopus, IEEE Xplore).

23.Hahanov V. Verification of digital system by a new asserting mechanism based on IEEE 1500 SECT standard / V. Hahanov, V. Obrizan, I. Hahanova, E. Fomina // Proc. of the International Conference MIXDES 2006. – June 22-24, 2006. – Gdunia, Poland. – 2006. – P. 544-548. (Входить до міжнародних наукометричних баз Scopus, IEEE Xplore).

24.Хаханов В.И. SIGETEST – система моделирования тестов проверки неисправностей цифровых устройств / В.И. Хаханов, В.И. Обризан, Я.В. Мирошниченко, О.В. Мельникова // Каталог аннотаций и разработок по материалам первого украинско-китайского форума «Наука – производство». – Харьков: ХНУРЭ, 2007. – С. 25–26.

25.Hahanov V. Hardware Simulation and Verification Technologies / V. Hahanov, A. Hahanova, V. Obrizan, W. Ghribi // Proc. of the IEEE East-West Design and Test Symposium. – September 7-10, 2007. – Yerevan, Armenia. – Yerevan, 2007. – P. 739-744.

26.Hahanov V. Technologies for hardware simulation and verification / V. Hahanov, A. Hahanova, V. Obrizan, K. Zaharov // Proc. of the International Conference Modern Problems of Radio Engineering, Telecommunications and Computer Science. – TCSET'2008. – February 19-23, 2008. – Slavske, Lviv, Ukraine. – 2008. – P. 560-564. (Входить до міжнародних наукометричних баз Scopus, IEEE Xplore).

27.Hahanov V. Testing challenges of SoC hardware-software components / V. Hahanov, V. Obrizan, S. Miroshnichenko, A. Gorobets // Proc. of the IEEE East-West Design and Test International Symposium. – October 9-12, 2008. – Lviv, Ukraine. – 2008. – P. 149-154. (Входить до міжнародних наукометричних баз Scopus, IEEE Xplore).

28.Obrizan V. A method for automatic generation of an RTL-interface from a C++ description / V. Obrizan // Proc. of the East-West Design & Test Symposium

(EWDTS) 2010. – 17-20 Sept. 2010. – St. Petersburg, Russia. – P.186-189. (Входить до міжнародних наукометричних баз Scopus, IEEE Xplore).

29.Hahanova I.V. Transaction level model of embedded processor for vector-logical analysis / I.V. Hahanova, V. Obrizan, A. Adamov, D. Shcherbin // Proc. of the East-West Design & Test Symposium. – 27-30 Sept. 2013. – Rostov-on-Don, Russia. – 4 p. (Входить до міжнародних наукометричних баз Scopus, IEEE Xplore).

30.Hahanova Yu. Metric for Analyzing Big Data / Yu. Hahanova, I. Yemelyanov, V. Obrizan, D. Krulevska, M. Skorobogatiy, A. Hahanova // Матеріали XIII Міжнародної науково-технічної конференции CADSM 2015 «Досвід розробки та застосування приладнати-технологічних САПР в мікроелектроніці ». 24-27 лютого 2015. Львів – Поляна. – С.81-83. (Входить до міжнародних наукометричних баз Scopus, IEEE Xplore).

31.Обризан В.И. Мультиарендные облачные сервисы / В.И. Обризан, Ю.В. Ломова // Материалы XIX Международного молодежного форума «Радиоэлектроника и молодежь в XXI веке». – Украина, Харьков: ХНУРЭ. – 20-22 апреля 2015. – Ч. 5. – С.34-35.

32.Obrizan V. Multiversion parallel synthesis of digital structures based on SystemC specification / V.Obrizan; T. Soklakova // Proc. of the IEEE East-West Design & Test Symposium (EWDTS). – 2016 – Yerevan, Armenia. – 6p. (Входить до міжнародних наукометричних баз Scopus, IEEE Xplore).

Publications that additionally reflect the scientific results of the thesis:

33.Гайдук С.М. Сферический мультипроцессор PRUS для решения булевых уравнений / С.М. Гайдук, В.И. Хаханов, В.И. Обризан, Е.А. Каменюка // Радиоэлектроника и информатика. – Харьков, 2004. – № 4 (29). – С. 107–116. (Входить до міжнародних наукометричних баз Index Copernicus, Google Scholar, OECSP, OAJI, Scholar Steer, SIS, Cyberleninka, CiteFactor, TIU Hannover, I2OR).

34.Hyduke S. PRUS – Spherical Multiprocessor for Computation of Boolean equations / S. Hyduke, V.I. Hahanov, V.I. Obrizan, Wade Ghribi // Proceedings of

the 8th International Conference CADSM 2005. – Ukraine. – Lviv, 2005. – P. 41-48.

35.Hyduke S.M. PRUS – Processor Network for Digital Circuit Implementation / S.M. Hyduke, V.I. Hahanov, V.I. Obrizan, O. Guz // Proc. of the 8th Euromicro Conference on Digital System Design. – August 30 – September 3, 2005. – Porto, Portugal. – 2005. – P. 239-242. (Входить до міжнародних наукометричних баз Scopus, IEEE Xplore).

36.Шевченко А.А. Математические модели описания потенциально опасных объектов газотранспортной промышлености / А.А. Шевченко, О.А. Довгошея, В.И. Обризан, Д.Н. Красильников // Материалы 2-го Международного радиоэлектронного форума «Прикладная радиоэлектроника». – 19-23 сентября 2005. – Украина. – С. 272- 276.

Key words: multiversion synthesis, specification, interface structure, operating device, verification, digital system-on-chip.

ЗМІСТ

ВСТУП

Створення цифрових систем на кристалах має ключову роль для розвитку комп'ютерних технологій, що використовуються в економіці, соціології, медицині, бізнесі, промисловості, освіті, які формують кіберекосистему планети: розподілені центри великих даних, Internet of Things, Internet of Everything, кіберфізичні системи, cloud-mobile computing, service computing.

Важливо відзначити експоненціальне зростання не тільки продуктивності комп'ютерних систем, але і продуктивності інженерів, завдяки створенню хмарних сервісів проектування апаратних і програмних додатків, інтегрованих в поняття комп'ютинг (рис. 1). Комп'ютинг – процес досягнення поставленої мети шляхом використання ресурсів, механізмів управління та виконання в циклічно замкнутій системі з заданими відносинами і результуючою продукцією, сигналами моніторингу та актюації. Розвиток хмарних сервісів не має на меті скоротити робочі місця, а надати інженерам інструментарій для виконання високотехнологічних робіт в будь-якій точці земної кулі зі створення якісних програмно-апаратних додатків, що мінімізує час виходу придатної продукції на ринок.

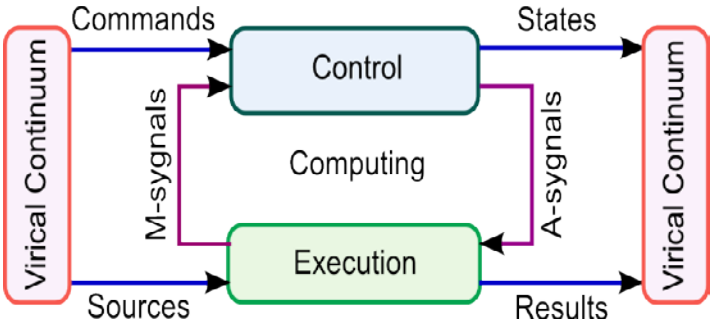


Рис. 1. Комп'ютинг для створення цифрових систем на кристалах

Технології хмарних сервісів по створенню цифрових систем на кристалах системного рівня дозволяють створювати виконувані специфікації – опис проекту на функціональному рівні, який можна верифікувати у віддале-

ному online-режимі. На функціональному рівні специфікація відображає питання «що робити?», а не як робити, таким чином вона не містить вимог до структури або архітектури цільової платформи. При цьому компонентами створення деякої цифрової системи виступають вже сервіси, а не функціональні елементи. Така заміна дає суттєвий виграш у часі проектування, який визначається швидкодією процедур верифікації та тестування, за рахунок варіювання параметрами системного, RT і вентильного рівнів, домагаючись кращих характеристик реалізації цифрового виробу на кристалі.

Проблеми проектування, тестування і верифікації цифрових систем на кристалах описані в публікаціях вчених: Y. Zorian, M.Abramovichi, J. Bergeron, Z. Navabi, A. Jerraya, DB Armstrong, P. Prinetto, J. Abraham, H. Fujiwara, T. Nishida, X. Wang, А.І. Петренко, Р. Убар, А. Ivanov, А.М. Романкевич, Д.В. Сперанський, П.П. Пархоменко, Ю.В. Малишенко, В.Н. Ярмолік, В. П. Чіпуліс, J.P.Roth, А.Ю.Матросова, Самвел Шукурян, Ю.А. Скобцов, М.Ф. Коровай, В.С. Харченко, Л.В. Дербуновіч, Р. Шейнаускас, Н. Євтушенко, Р. Базилевич, А. Матросова, В. Мелікян. Значний внесок у розвиток теорії проектування комп'ютерних сервісів зробили вчені Самарій Баранов, Сергій Майоров, Геннадій Новіков, Віктор Глушков, Деніел Гайським, Роберт Кун, Алекс Орайлоглу, Раеш Гупта, Ахмед Джерайя, Яник Бергерон, Хідео Фуджівара, Айріш Померанц, Раймунд Убар, В'ячеслав Харченко.

Зв'язок роботи з науковими програмами, планами, темами. Розробка розділів дисертації здійснювалася відповідно до планів НДР і договорів, що виконувалися на кафедрі АПОТ Харківського національного університету радіоелектроніки в період з 2011 року:

1) Договір про дружбу і співробітництво між ХНУРЕ та корпорацією «Aldec Inc.» (USA) № 04 від 01.11.2011 року;

2) Держбюджетна НДР «Теорія й проектування енергозберігаючіх цифрових обчислювальних систем на кристалах, что моделюють и підсилюють функціональні можливості людини, д/б № 232, 2009, №ДР 0109U001646;

3) Держбюджетна НДР «Мультипроцесорна система пошуку, розпізнавання та прийняття рішень для інформаційної комп'ютерної екосистеми», д/б № 269 (2011-2013), №ДР 0111U002956;

4) Держбюджетна науково-дослідна фундаментальна робота №268 «Персональний віртуальний кіберкомп'ютер та інфраструктура аналізу кіберпростору» 01.01.2012-31.12.2014 № ДР 0112U000209;

5) Держбюджетна науково-дослідна фундаментальна робота №297 "Кіберфізична система – «Розумне хмарне управління транспортом (Cyber Physical System – Smart Cloud Traffic Control)» 01.01.2015-31.12.2017 № 0115U-000712 від 04.03.2015.

6) Проект 530785-TEMPUS-1-2012-1-PL-TEMPUS-JPCR «Curricula Development for New Specialization: Master of Engineering in Microsystems Design (MastMEMS)» сумісно з університетом «Львівська політехніка», Київським національнім університетом, технічним університетом м. Лодзь (Польща), Ліонським університетом (Франція), Університетом м. Ільменау (Німеччина), Університетом м. Павія (Італія) на 2012 - 2016 рр.

7) Educating the Next generation experts in Cyber Security: the new EU-recognized Master's program (ENGENSEC) 544455-TEMPUS-1-2013-1-SE-TEMPUS-JPCR (01 Dec 2013 - 30 Nov 2016).

Автор дисертації при виконанні зазначених договорів і програм брав участь, як розробник системного рівня архітектури цифрових виробів на кристалах і програміст, при створенні моделей функціональних примітивів і методів верифікації. Він також є виконавцем в C ++ і Verilog-кодуванні програмно-апаратних компонентів системи верифікації HDL-кодів на основі IEEE стандартів, інтегрованих з програмним забезпеченням компанії Aldec.

Сутність ринково-орієнтованого науково-технічного дослідження полягає у мультиверсному проектуванні архітектури цифрового виробу на основі заданої специфікації в середовищі SystemC (C++) і автоматичному виборі синтезованих функціональних структур з метою істотного зменшення часу створення проекту і підвищенні виходу придатної продукції за рахунок

паралельного синтезу та верифікації архітектурних рішень системного рівня відповідно до запропонованої метрики. Основна інноваційна ідея – паралельний автоматичний синтез квазіоптимальної архітектури відповідно до запропонованої специфікації і метрики, регулюючої підбір синтезованих функціональних структур.

Мета дослідження – суттєве зменшення часу проектування обчислювальних архітектур і підвищення якості цифрових виробів шляхом мультиверсного синтезу структури цифрового виробу на основі заданої специфікації в середовищі SystemC (C++) і автоматичному підборі функціональних компонентів за рахунок паралельного синтезу і верифікації архітектурних рішень системного рівня відповідно до запропонованої метрики.

Об'єкт дослідження – процеси паралельного синтезу і верифікації цифрових структур.

Предмет дослідження – моделі, методи та інфраструктура для мультиверсного паралельного синтезу цифрових структур на основі SystemC специфікації.

Задачі:

1. Огляд існуючих моделей, методів, алгоритмів і програмних засобів створення цифрових систем на кристалах.

2. Розробка структур даних для опису функціональних примітивів системного рівня, орієнтованих на використання семантичних і синтаксичних конструкцій мови C ++ і SystemC з метою забезпечення паралельного синтезу і верифікації архітектурних рішень.

3. Розробка методу синтезу інтерфейсних структур і протоколів виконання транзакцій RT-рівня на основі аналізу специфікації SoC-архітектури системного рівня, яка використовує стандартну шину Wishbone обміну даними між функціональними модулями.

4. Розробка методу синтезу RTL-моделей функціональностей шляхом перетворення C++ і SystemC-описів цифрових блоків системного рівня в

алгоритми і структури даних автоматної моделі Мура, що задана синтезованою підмножиною мовних конструкцій VHDL.

5. Розробка методу мультиверсного синтезу керуючих і операційних автоматів, орієнтованих на архітектурні рішення в метриці, що мінімізує час виконання функціональності за рахунок розпаралелювання операцій при обмеженні на апаратні витрати.

6. Програмна реалізація моделей і методів мультиверсної розробки операційних пристроїв в рамках інтегрованої системи проектування функціональних і архітектурних рішень SoC на основі використання продуктів верифікації та синтезу компаній Aldec і Xilinx.

Наукова новизна отриманих результатів.

1. Вперше запропоновано метод синтезу інтерфейсних структур і протоколів виконання транзакцій RT-рівня на основі аналізу специфікації SoC-архітектури системного рівня, який характеризується використанням двобічної стандартної шини Wishbone обміну даними між функціональними модулями, що дозволяє здійснювати мультиверсне створення компонентів цифрових систем на кристалах.

2. Вперше запропоновано метод синтезу RTL-моделей функціональностей, який характеризується однозначним перетворенням C++ і SystemC-описів цифрових блоків системного рівня в алгоритми і структури даних автоматної моделі Мура, заданої синтезованою підмножиною мовних конструкцій VHDL, що дає можливість істотно зменшити час виконання процесів проектування, тестування і верифікації.

3. Удосконалено структури даних для опису функціональних примітивів системного рівня, які відрізняються орієнтацією на використання семантичних і синтаксичних конструкцій мови C ++ і SystemC, що дозволяє здійснювати паралельний синтез і верифікацію архітектурних рішень.

4. Удосконалено метод мультиверсного синтезу керуючих і операційних автоматів, орієнтованих на архітектурні рішення в метриці, яка відрізняється мінімальним часом виконання функціональності за рахунок

розпаралелювання операцій при обмеженні на апаратні витрати, що дозволило збільшити ефективність засобів автоматизованого проектування цифрових виробів.

Практична значущість отриманих результатів.

1. Розроблено програмні засоби для реалізації моделей і методів мультиверсного створення операційних пристроїв в рамках інтегрованої системи проектування функціональних і архітектурних рішень SoC на основі використання продуктів верифікації та синтезу компаній Aldec і Xilinx.

2. Проведено тестування і верифікацію програмних модулів мультиверсної розробки операційних пристроїв в рамках інтегрованої системи проектування функціональних і архітектурних рішень SoC на десяти прикладах реалізації промислово-орієнтованих функціональних блоків.

Особистий внесок здобувача. Всі наукові і практичні результати отримані автором особисто. У роботах, опублікованих зі співавторами, здобувачеві належать: [1] – структури даних для опису функціональних примітивів системного рівня; [2] – розробка програмних модулів системи SIGETEST для моделювання несправностей та генерації тестів; [4] – програмна реалізація методу верифікації HDL-коду; [5] – приклади застосування метрики аналізу Big Data для прикладних архітектурних рішень; [6] – огляд застосування хмарних сервісів в кіберфізичній системі управління ресурсами; [8] – програмна реалізація дедуктивно-паралельного методу моделювання несправностей; [9] – приклад використання алгебро-логічної моделі вбудованого діагностування несправностей; [10] – програма реалізації алгебро-логічного методу діагностування; [11] – огляд технологій проектування систем на кристалах; [12] – застосування методу верифікації цифрових систем на основі стандарту IEEE 1500 SECT; [13] – ієрархічне тестування програмно-технічних комплексів; [14] – рефлексивна модель подання даних; [15] – MQT-автомат для аналізу великих даних; [16] – sigetest-fault моделювання та випробування покоління для цифрових пристроїв; [17] – висока продуктивність моделювання несправностей для

цифрових систем; [18] – система моделювання несправностей для дискретних пристроїв; [19] – нові можливості дедуктивного моделювання несправностей; [20] – паралелізм мозкоподібних обчислень; [21] – високопродуктивний дедуктивний метод моделювання несправностей; [22] – ієрархічне тестування складних цифрових систем; [23] – перевірка цифрової системи за допомогою нового механізму на основі IEEE 1500; [24] – SIGETEST-система моделювання тестів перевірки несправностей цифрових пристроїв; [25] – апаратні засоби моделювання і верифікації технології; [26] – технології для апаратного моделювання та верифікації; [27] – тестування проблеми SoC апаратно-програмних компонентів; [28] – спосіб автоматичної генерації RTL-інтерфейсу з описом C++; [29] – транзакційна модель мікропроцесора для векторно-логічного аналізу; [30] – метрика для обробки великих даних; [31] – мультіарендні хмарні сервіси; [32] – багатоваріантність паралельного синтезу цифрових структур на основі специфікації SystemC; [33] – тестування сферичного мультипроцесору PRUS для вирішення булевих рівнянь; [34] – PRUS-сферичні мультипроцесори для обчислень булевих рівнянь; [35] – процесорна структура для реалізації цифрової схеми; [36] – математичні моделі опису об'єктів газотранспортної промисловості.

Апробація результатів дисертації. Результати роботи були представлені та обговорені на 20 конференціях: 1) IEEE East-West Design and Test Symposium, 2004 (Алушта, Україна), 2007, 2016 (Єреван, Вірменія), 2008 (Львів, Україна), 2010 року (Санкт-Петербург, Росія), 2013 (Ростов-на-Дону, РФ); 2) Современные технологии проектирования систем на микросхемах программируемой логики, 2003, (Харків, Україна); 3) Молодіжний радіоелектронний форум «Радіоелектроніка та молодь у XXI сторіччі», 2003, 2015 (Харків, Україна); 4) Міжнародний науково-практичний семінар «Высокопроизводительные параллельные вычисления на кластерных системах», 2005, (Харків, Україна); 5) Міжнародний радіоелектронний форум «Прикладна радіоелектроніка», 2005, (Харків, Україна); 6) Advanced Compact Modeling Workshop, 2005 (Porto, Portugal); 7) Euromicro Symposium on Digital

Systems Design, 2003 (Белек-Анталія, Туреччина); 8) The Experience of designing and Application of CAD Systems in Microelectronics, International Conference, 2005, 2015 (Поляна, Україна); 9) International Conference Mixdes Design of Integrated Circuits and Systems, 2006 (Gdynia, Poland); 10) 10th IEEE European Test Symposium, 2005, (Tallinn, Estonia); 11) Українсько-китайський форум «Наука – виробництво», 2007, (Харків, Україна); 12) Modern Problems of Radio Engineering, Telecommunications and Computer Science, 2006, 2008 (Slavske, Lviv, Ukraine).

Публікації. Результати дисертаційної роботи відображені у 36 друкованих працях: 16 статей, серед яких 13 у наукових журналах, що входять до «Переліків наукових фахових видань України» (з них 12 – у міжнародних наукометричних базах), 3 статті в міжнародних наукових журналах за кордоном (з них 2 – в міжнародній наукометричній базі Scopus); а також 20 публікацій у міжнародних наукових конференціях (з них 8 за кордоном, 11 входять до наукометричної бази Scopus). Здобувач має 15 публікацій, що входять до наукометричної бази Scopus, та має індекс Хірша h=2.

СПИСОК ОПУБЛІКОВАНИХ РОБІТ ЗА ТЕМОЮ ДИСЕРТАЦІЇ

Список публікацій здобувача, в яких опубліковані основні наукові результати дисертації:

37.Хаханов В.И. Технология моделирования и синтеза тестов для сложных цифровых систем / В.И. Хаханов, К.В. Колесников, А.Н. Парфентий, И.В. Хаханова, В.И. Обризан, О.В. Мельникова // Радиоэлектроника и информатика. – 2003. – №1. – С. 70-78. (Входить до міжнародних наукометричних баз Index Copernicus, Google Scholar, OECSP, OAJI, Scholar Steer, SIS, Cyberleninka, CiteFactor, TIU Hannover, I2OR).

38.Hahanov V. Advanced Software Tools For Fault Simulation And Test Generation / V. Hahanov, A. Egorov, O. Melnikova, V. Obrizan, E. Kamenuka, O. Krapchunova, O. Guz // Радиоэлектроника и информатика. – 2003. – №3. – С. 77-81. (Входить до міжнародних наукометричних баз Index Copernicus,

Google Scholar, OECSP, OAJI, Scholar Steer, SIS, Cyberleninka, CiteFactor, TIU Hannover, I2OR).

39.Obrizan V. A Method of High-Level Synthesis and Verification with SystemC Language / V. Obrizan // Радиоэлектроника и информатика. – 2010. – №4. – С. 47-50. (Входить до міжнародних наукометричних баз Index Copernicus, Google Scholar, OECSP, OAJI, Scholar Steer, SIS, Cyberleninka, CiteFactor, TIU Hannover, I2OR).

40.Obrizan V. Matrix-Model for Diagnosing SoC HDL-Code / V. Obrizan, I. Yemelyanov, V. Hahanov, E. Litvinova // Radioelektroniks and informatics. – 2013. – №1. – P.12-19. (Входить до міжнародних наукометричних баз Index Copernicus, Google Scholar, OECSP, OAJI, Scholar Steer, SIS, Cyberleninka**, CiteFactor,** TIU Hannover, I2OR).

41.Обризан В.И. Метрика для анализа Big Data / В.И. Обризан, А.С. Мищенко, В.И. Хаханов, Tamer Bani Amer // Радиоэлектроника и информатика. – 2014. – № 2. – С. 26-29. (Входить до міжнародних наукометричних баз Index Copernicus, Google Scholar, OECSP, OAJI, Scholar Steer, SIS, Cyberleninka, CiteFactor**,** TIU Hannover, I2OR).

42.Обризан В.И. Киберфизические системы как технологии киберуправления (аналитический обзор) / В.И. Обризан, А.С. Мищенко, В.И. Хаханов, И.В. Филиппенко // Радиоэлектроника и информатика. – 2014. – № 1. – С. 39-45. (Входить до міжнародних наукометричних баз Index Copernicus, Google Scholar, OECSP, OAJI, Scholar Steer, SIS, Cyberleninka, CiteFactor, TIU Hannover, I2OR).

43.Обризан В.И. Инфраструктура проектирования SOC для метода мультиверсного синтеза / В.И. Обризан // Радиоэлектроника и информатика. – 2016. – №2. – С. 48-60. (Входить до міжнародних наукометричних баз Index Copernicus, Google Scholar, OECSP, OAJI, Scholar Steer, SIS, Cyberleninka, CiteFactor, TIU Hannover, I2OR).

44.Hahanov V. High Performance Fault Simulation For Digital Systems / V. Hahanov, G. Krivoulya, I. Hahanova, O. Melnikova, V. Obrizan // International

Scientific Journal of Computing. – 2003. – Vol. 2, Issue 2. – P. 114-121. (Входить до міжнародних наукометричних баз Index Copernicus, Norwegian Social Science Data Services (NSD), Google Scholar, Vernadsky National Library of Ukraine).

45.Hahanov V. Algebra-logical diagnosis model for SoC F-IP / V. Hahanov, V. Obrizan, E. Litvinova, Man K.L. // WSEAS Transactions on Circuits and Systems. – 2008. – No 7. – P. 708-717. (Входить до міжнародних наукометричних баз Scopus, Elsevier, Google Scholar).

46.Hahanov V. Embedded method of SoC diagnosis / V. Hahanov, E. Litvinova, V. Obrizan, W. Gharibi // Elektronika ir Elektrotechnika. – 2008. – No8. – P. 3-8. (Входить до міжнародних наукометричних баз Scopus, Thomson Reuters (ISI), Web of Knowledge Citation Databases, Science Citation Index Expanded (SCIE); Journal Citation Reports (JCR); INSPEC; VINITI; EBSCO Publishing).

47.Хаханов В.И. Обзор международного рынка электронных технологий / В.И. Хаханов, В.И. Обризан, О.В. Мельникова // Вестник НТУ «ХПИ». – Серия: Информатика и моделирование. – 2004. – Вып. 46. – С.111-115.

48.Хаханов В.И. Assert-метод верификации цифровых систем на основе стандарта IEEE 1500 SECT / В.И. Хаханов, В.В. Елисеев, В.И. Обризан // АСУ и приборы автоматики. – 2005. – Вып. 132. – С. 93–105. (Входить до міжнародних наукометричних баз Google Scholar, Cyberleninka).

49.Хаханов В.И. Иерархическое тестирование программно-технических комплексов / В.И. Хаханов, В.В. Елисеев, В.И. Обризан // АСУ и приборы автоматики. – Харьков, 2006. – Вып. 134. – С. 93–102. (Входить до міжнародних наукометричних баз Google Scholar, Cyberleninka).

50.Михтонюк С.В. Архитектурная модель масштабируемой системы асинхронной обработки больших объемов данных / С.В. Михтонюк, Р.С. Хван, В.И. Обризан // АСУ и приборы автоматики. – 2008. – Вып. 142. – С.

13-17. (Входить до міжнародних наукометричних баз Google Scholar, Cyberleninka).

51.Хаханов В.И. MQT-автомат для анализа больших данных / В.И. Хаханов, В.И. Обризан, С.А. Зайченко, И.В. Хаханов // АСУ и приборы автоматики. – 2014. – Вып. 168. – С. 64-72. (Входить до міжнародних наукометричних баз Google Scholar, Cyberleninka).

Результати, які засвідчують апробацію матеріалів дисертації:

52.Hahanov V.I. Sigetest – fault simulation and test generation for digital designs / V.I. Hahanov, D.M. Gorbunov, Y.V. Miroshnychenko, O.V. Melnikova, V.I. Obrizan, E.A. Kamenuka // Современные технологии проектирования систем на микросхемах программируемой логики. – 23 сентября 2003. – Харьков. – С. 50-53.

53.Hahanov V. High performance Fault Simulation for Digital Systems / V. Hahanov, O. Melnikova, V. Obrizan, I. Hahanova // Proc. of the Euromicro Symposium on Digital Systems Design. – Turkey. Belek-Antalya. – 2003. – P. 15-16. (Входить до міжнародних наукометричних баз Scopus, IEEE Xplore).

54.Мирошниченко Я.В. Система моделирования неисправностей для дискретных устройств / Я.В. Мирошниченко, О.В. Мельникова, В.И. Обризан / Материалы 7-го молодежного форума «Радиоэлектроника и молодежь в XXI веке». – Украина, Харьков: ХНУРЭ. – 2003. – С. 463.

55.Hahanov V.I. New Features of Deductive Fault Simulation / V.I. Hahanov, V.I. Obrizan, A.V. Kiyaszhenko, I.A. Pobezhenko // Proc. of the 2nd East-West Design and Test Workshop 2004. – September 23-26, 2004. – Alushta. – 2004. – P. 274-280.

56.Шабанов-Кушнаренко Ю.П. Параллелизм мозгоподобных вычислений / Ю.П. Шабанов-Кушнаренко, В.И. Обризан // Материалы 5-го Международного научно-практического семинара «Высокопроизводи-тельные параллельные вычисления на кластерных системах». – 22-25 ноября 2005. – Харьков. – С. 196- 203.

57.Hahanov V.I. High-performance deductive fault simulation method / V.I. Hahanov, I. Hahanova, V.I. Obrizan // Proc. of the 10th IEEE European Test Symposium. – May 22-25, 2005. – Estonia. – Tallinn, 2005. – P. 91-96. (Входить до міжнародних наукометричних баз Scopus, IEEE Xplore).

58.Hahanov V. Hierarchical Testing of Complex Digital Systems / V. Hahanov, V. Obrizan, V. Yeliseev, W. Ghribi // Proc. of the International Conference "Modern Problems of Radio Engineering, Telecommunications and Computer Science (TCSET'2006)". – February 28 – March 4, 2006. – Slavske, Lviv, Ukraine. 2006. – P. 426-429. (Входить до міжнародних наукометричних баз Scopus, IEEE Xplore).

59.Hahanov V. Verification of digital system by a new asserting mechanism based on IEEE 1500 SECT standard / V. Hahanov, V. Obrizan, I. Hahanova, E. Fomina // Proc. of the International Conference MIXDES 2006. – June 22-24, 2006. – Gdunia, Poland. – 2006. – P. 544-548. (Входить до міжнародних наукометричних баз Scopus, IEEE Xplore).

60.Хаханов В.И. SIGETEST – система моделирования тестов проверки неисправностей цифровых устройств / В.И. Хаханов, В.И. Обризан, Я.В. Мирошниченко, О.В. Мельникова // Каталог аннотаций и разработок по материалам первого украинско-китайского форума «Наука – производство». – Харьков: ХНУРЭ, 2007. – С. 25–26.

61.Hahanov V. Hardware Simulation and Verification Technologies / V. Hahanov, A. Hahanova, V. Obrizan, W. Ghribi // Proc. of the IEEE East-West Design and Test Symposium. – September 7-10, 2007. – Yerevan, Armenia. – Yerevan, 2007. – P. 739-744.

62.Hahanov V. Technologies for hardware simulation and verification / V. Hahanov, A. Hahanova, V. Obrizan, K. Zaharov // Proc. of the International Conference Modern Problems of Radio Engineering, Telecommunications and Computer Science. – TCSET'2008. – February 19-23, 2008. – Slavske, Lviv, Ukraine. – 2008. – P. 560-564. (Входить до міжнародних наукометричних баз Scopus, IEEE Xplore).

63.Hahanov V. Testing challenges of SoC hardware-software components / V. Hahanov, V. Obrizan, S. Miroshnichenko, A. Gorobets // Proc. of the IEEE East-West Design and Test International Symposium. – October 9-12, 2008. – Lviv, Ukraine. – 2008. – P. 149-154. (Входить до міжнародних наукометричних баз Scopus, IEEE Xplore).

64.Obrizan V. A method for automatic generation of an RTL-interface from a C++ description / V. Obrizan // Proc. of the East-West Design & Test Symposium (EWDTS) 2010. – 17-20 Sept. 2010. – St. Petersburg, Russia. – P.186-189. (Входить до міжнародних наукометричних баз Scopus, IEEE Xplore).

65.Hahanova I.V. Transaction level model of embedded processor for vector-logical analysis / I.V. Hahanova, V. Obrizan, A. Adamov, D. Shcherbin // Proc. of the East-West Design & Test Symposium. – 27-30 Sept. 2013. – Rostov-on-Don, Russia. – 4 p. (Входить до міжнародних наукометричних баз Scopus, IEEE Xplore).

66.Hahanova Yu. Metric for Analyzing Big Data / Yu. Hahanova, I. Yemelyanov, V. Obrizan, D. Krulevska, M. Skorobogatiy, A. Hahanova // Матеріали XIII Міжнародної науково-технічної конференции CADSM 2015 «Досвід розробки та застосування приладнати-технологічних САПР в мікроелектроніці ». 24-27 лютого 2015. Львів – Поляна. – С.81-83. (Входить до міжнародних наукометричних баз Scopus, IEEE Xplore).

67.Обризан В.И. Мультиарендные облачные сервисы / В.И. Обризан, Ю.В. Ломова // Материалы XIX Международного молодежного форума «Радиоэлектроника и молодежь в XXI веке». – Украина, Харьков: ХНУРЭ. – 20-22 апреля 2015. – Ч. 5. – С.34-35.

68.Obrizan V. Multiversion parallel synthesis of digital structures based on SystemC specification / V.Obrizan; T. Soklakova // Proc. of the IEEE East-West Design & Test Symposium (EWDTS). – 2016 – Yerevan, Armenia. – 6p. (Входить до міжнародних наукометричних баз Scopus, IEEE Xplore).

Публікації, які додатково відображають наукові результати дисертації:

69.Гайдук С.М. Сферический мультипроцессор PRUS для решения булевых уравнений / С.М. Гайдук, В.И. Хаханов, В.И. Обризан, Е.А. Каменюка // Радиоэлектроника и информатика. – Харьков, 2004. – № 4 (29). – С. 107–116. (Входить до міжнародних наукометричних баз Index Copernicus, Google Scholar, OECSP, OAJI, Scholar Steer, SIS, Cyberleninka, CiteFactor, TIU Hannover, I2OR).

70.Hyduke S. PRUS – Spherical Multiprocessor for Computation of Boolean equations / S. Hyduke, V.I. Hahanov, V.I. Obrizan, Wade Ghribi // Proceedings of the 8th International Conference CADSM 2005. – Ukraine. – Lviv, 2005. – P. 41-48.

71.Hyduke S.M. PRUS – Processor Network for Digital Circuit Implementation / S.M. Hyduke, V.I. Hahanov, V.I. Obrizan, O. Guz // Proc. of the 8th Euromicro Conference on Digital System Design. – August 30 – September 3, 2005. – Porto, Portugal. – 2005. – P. 239-242. (Входить до міжнародних наукометричних баз Scopus, IEEE Xplore).

72.Шевченко А.А. Математические модели описания потенциально опасных объектов газотранспортной промышлености / А.А. Шевченко, О.А. Довгошея, В.И. Обризан, Д.Н. Красильников // Материалы 2-го Международного радиоэлектронного форума «Прикладная радиоэлектроника». – 19-23 сентября 2005. – Украина. – С. 272-276.

Результати дисертаційної роботи впроваджено у компанії ООО ALDEC-КТС (довідка від 15.04.2016) та навчальному процесі кафедри АПОТ ХНУРЕ (довідка від 12.05.2016).

РОЗДІЛ 1
РОЗВИТОК І СУЧАСНИЙ СТАН ЗАСОБІВ АВТОМАТИЗОВАНОГО ПРОЕКТУВАННЯ

Мета – показати розвиток моделей, методів, алгоритмів і програмних засобів створення цифрових систем на кристалах в часі і просторі. Визначити вузькі місця і переваги найцікавіших моделей і методів, опублікованих в спеціальній літературі: матеріали конференцій і журнали. Сформулювати мету і завдання дослідження, орієнтовані на усунення вузьких місць і використання найбільш ефективних існуючих рішень для розробки теоретичних основ і практичних засобів системно-орієнтованого проектування обчислювальних пристроїв.

1.1. Ретроспектива автоматизації проектування електроніки

Перші повідомлення про автоматизацію проектування електроніки відносяться до 1956 року, коли був використаний аналоговий комп'ютер для моделювання цифрового фільтра [1]. У 1957 році, в університеті Ілінойса була використана комп'ютерна програма для розрахунків характеристик тригера на основі транзисторів [2]. У [3] повідомляється, що в 1958 році була розроблена програма, яка синтезувала фізичні характеристики логічної схеми, при цьому результати, які раніше виходили за місяць ручних обчислень, були отримані протягом дня з використанням комп'ютера. У 1963 році розроблена і випробувана комп'ютерна програма для синтезу мінімальних дворівневих багатовиходових логічних функцій [4]. Складність тестової схеми – 4 змінних, 5 функцій, 20 термів ДНФ – обмежується доступною оперативною пам'яттю і часом на розв’язок. Наголошується також, що при ручному проектуванні інженер отримав рішення на 4% оптимальніше комп'ютерного. В цьому ж році розроблені компілятори булевих рівнянь [5, 6], які автоматично оптимізували час моделювання схеми.

У 1964 році в США був створений комітет з автоматизації проектування [7]. Він був створений урядовими, освітніми та комерційними організаціями з метою спільного вирішення проблем, обміну інформацією, розвитком ідей в новій галузі – автоматизації проектування електроніки. Щорічні збори цього комітету пізніше отримали назву конференції по автоматизації проектування. На першій зустрічі було підкреслено важливий економічний ефект галузі [8, 9]: скорочення термінів розробки, а також вартості виробництва, зниження ризиків.

У 1965 році були застосовані перші прості мови для опису і моделювання логічних схем, які дозволили істотно скоротити час їх аналізу: замість декількох тижнів лабораторних тестів досить було двох-трьох хвилин комп'ютерного моделювання [10, 11]. Можливості програми аналізу схеми не перевищували 100 дискретних елементів. У цьому ж році, в [12] Улріх описує алгоритм подієвого интерпретативного моделювання для цифрових схем з урахуванням часових затримок, що є важливим кроком до підвищення адекватності результатів аналізу. В [13] відзначається, що з використанням програм автоматизації проектування вартість розробки скоротилася більш ніж на половину, і зросла точність і надійність виробу.

У 1965 році Гордон Мур, будучи співробітником Феірчайлд Семікондактор, сформував емпіричний закон розвитку інтегральних технологій, який пізніше отримав його ім'я [14]. Муром було зроблено спостереження за кілька років, що кількість елементів інтегральної схеми подвоюється приблизно кожні два роки (рис. 1.1). Чудово те, що закон працює і досі.

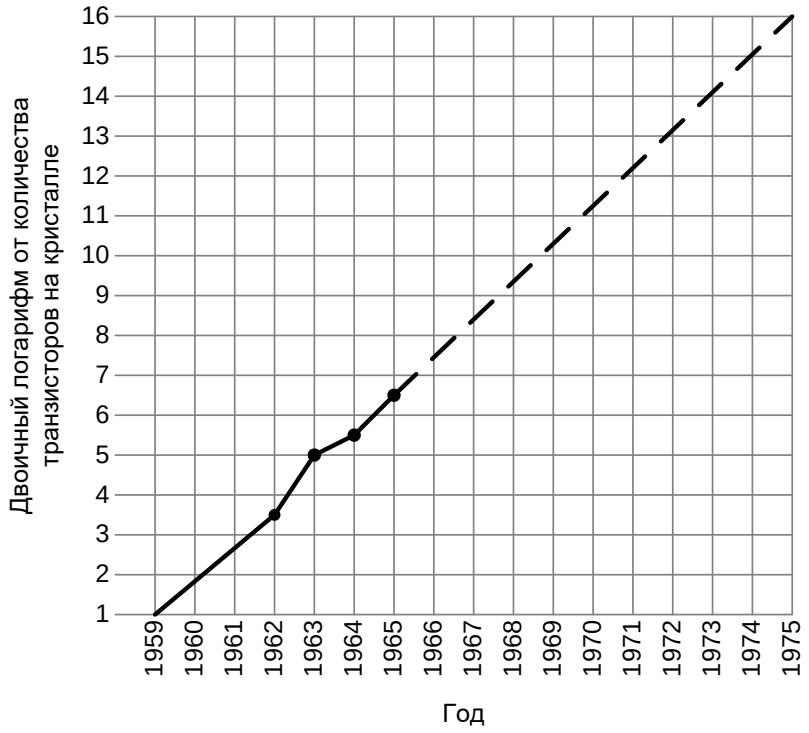


Рис. 1.1. Ілюстрація закону Мура (редакція 1965 г.) – кількість транзисторів на кристалі буде подвоюватися приблизно кожні два роки

У 1966 році була використана комп'ютерна програма для розміщення і трасування логічних елементів на друкованих платах [15], При цьому часові витрати склали 20 хвилин для 50 елементів і одна година для 100 елементів. У 1966 році зустрічається перша згадка [16] про використання елементів штучного інтелекту (еволюційних алгоритмів) для автоматизації: мінімізація довжини сполучних проводів, а також побудова поведінкової моделі по заданим вхідним і вихідним значенням логічної функції.

У 1971 році випущений мікропроцесор Інтел 4004, що містить 2300 транзисторів.

У 1983 році опубліковані результати американської програми VHSIC - Very High Speed Integrated Circtuits [17]. Було запропоновано використання мови VHDL як мови опису апаратури. Перша версія стандарту була прийнята суспільством IEEE у 1987 році під номером 1076. Виходили також версії стандарту з істотними змінами у 1993 і 2008 роках. Програма була настільки

вдала, що мова опису апаратури VHDL до сьогоднішнього дня користується популярністю.

Взимку 1983-1984 років Філ Мурба і Прабху Гоел розробили мову опису апаратури Verilog. Початкові права на винахід належали фірмі Gateway Design Automation, потім в 1990 році були куплені фірмою Cadence Design Automation. У 1995 році товариство IEEE затвердило стандарт цієї мови. Нові версії стандартів також затверджувалися в 2001 і 2005 роках.

Поведінковий опис може бути зроблений на декількох рівнях [18, 19, 20]: як алгоритм на мові високого рівня, як набір інструкцій для мікропроцесора, на рівні регістрових передач мікроархітектури або як система булевих рівнянь. Структурний опис також може бути виконано на різних рівнях, в залежності від складності компонентів: взаємодія процесора і пам'яті на системному рівні, реєстрові файли, АЛУ, лічильники і шини на мікроархітектурному рівні, вентилі і тригери на логічному рівні, і транзистори, контакти і провідники на рівні топології кристалла (рис. 1.2).

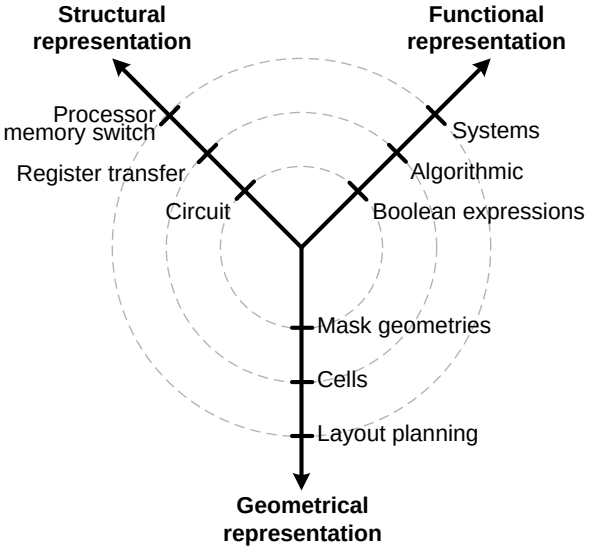


Рис. 1.2. Y-діаграма Гайского-Куна відображає тристороннє представлення проекту і різних рівнів абстракції

На системному рівні проектування використовуються транзакції - виклики функцій, щоб змоделювати взаємодію між блоками системи [21], на відміну від моделей рівня реєстрових передач, де використовуються сигнали і шини. Модель на рівні транзакцій виконується в кілька разів швидше, ніж на рівні регістрових передач. Наприклад, для того щоб змоделювати читання з пам'яті на системному рівні досить викликати одну функцію, а на рівні регістрових передач потрібно передати послідовність різних сигналів. Системні моделі на рівні транзакцій знайшли широке застосування в верифікації проектів [x].

1.2. Огляд високорівневих мов проектування

Для розробки вбудованих програмно-апаратних систем необхідні нові мови опису, відмінні від традиційних МОА VHDL і Verilog. З використанням типових МОА поділ на програмну і апаратну підсистеми робиться на ранніх етапах проектування, тому що ці частини розробляються незалежно один від одного. Пізніше зміна в це рішення можна внести з великими часовими і матеріальними витратами на переробку. Використання єдиної високорівневої мови, заснованої на мові C++, дозволить уникнути цієї проблеми.

На сьогоднішній день є велика кількість високорівневих мов опису систем. Серед них: C++, SystemC, SpecC, Handel-C, HardwareC [22]. Розглянемо ці мови більш докладно.

C ++ – мова програмування загального призначення, заснована на мові C [23, 24]. На відміну від C, мова C ++ підтримує більш складні типи даних, класи, шаблони, простору імен, що вбудовуються функції, перевантаження операторів, перевантаження функцій, посилання, стандартну бібліотеку шаблонів. Мова підтримує об'єктно-орієнтовану модель програмування, що дозволяє йому з допомогою класів описувати предмети з будь-якої області знання, наприклад, з цифрової електроніки: логічне значення, сигнал, регістр, шина, модулі. Існує два шляхи застосування мови C ++ для моделювання

апаратури: розширення синтаксису (HardwareC, SpecC, StreamIt [25]) і використання додаткових бібліотек класів (SystemC, TIPSY [26]). У першому випадку, необхідна розробка додаткових компіляторів і програм синтезу, які розпізнають новий синтаксис. У другому випадку, необхідно вводити додаткові класи, які реалізують поняття з предметної області [27]. Мова C ++ не має вбудованих можливостей для моделювання паралельного виконання і часових затримок.

SystemC (читається «систем сі») – мова проектування і верифікації моделей системного рівня, реалізований у вигляді бібліотеки C ++ з відкритим вихідним кодом [28, 29]. Бібліотека включає в себе ядро подієвого моделювання, що дозволяє отримати виконувану специфікацію проектованої системи. Мова застосовується для побудови транзакційних і поведінкових моделей, а також для високорівневого синтезу. SystemC використовує ряд понять, які роблять його схожим з традиційними мовами опису апаратури VHDL і Verilog: інтерфейси, процеси, сигнали, наповненості, ієрархія модулів. Мова SystemC не вносить обмежень на використання мови C ++ для опису моделей. Таким чином, можна його використовувати одночасно з іншими бібліотеками, наприклад, для роботи з графічним інтерфейсом користувача. Бібліотека SystemC є розвитком проекту Scenic фірми Synopsys [30].

На рис. 1.3 показана загальна архітектура мови SystemC [31]. Фундамент бібліотеки SystemC – стандарт мови C ++. На його основі побудовано ядро мови SystemC, яке представлено декількома абстракціями: модуль, порт, процес, інтерфейс, канал, подія. Верхні шари реалізовані за допомогою нижніх шарів. У свою чергу, нижні шари при необхідності можуть бути використані без верхніх шарів.

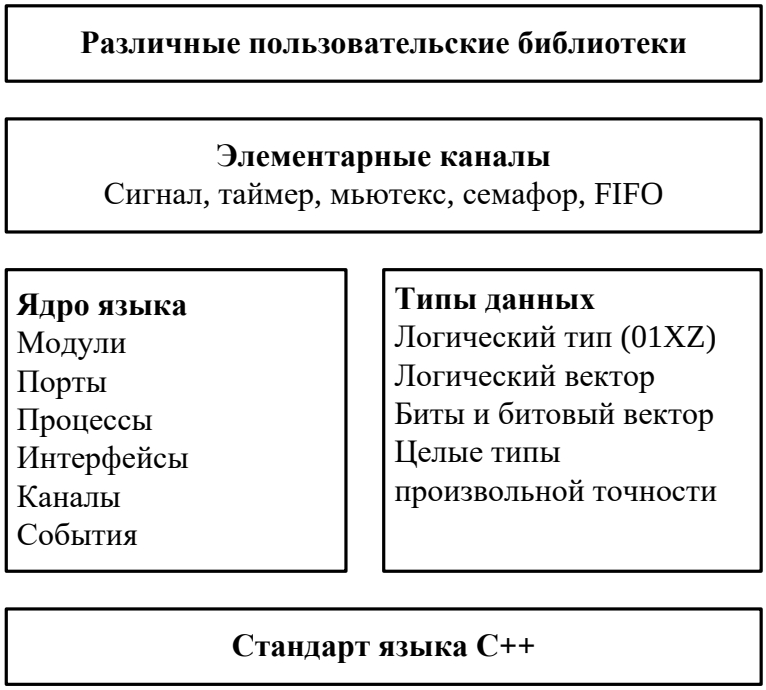


Рис. 1.3. Багатошарова архітектура бібліотеки SystemC

Клас *sc_module* реалізує модуль – основний клас бібліотеки, є будівельним блоком для проекту на SystemC. У мові VHDL йому відповідає поняття *entity*, а в мові Verilog поняття *module*. Всі модулі проекту повинні бути успадковані від цього класу. Кожен модуль містить декілька процесів, які виконуються паралельно. Процес повинен бути зареєстрований в ядрі моделювання. Мова C ++ не має вбудованих засобів для реалізації паралельного виконання, тому спільне виконання процесів досягається за рахунок ядра подієвого моделювання бібліотеки SystemC. Для процесу визначається список чутливості – список сигналів, при зміні значення яких, має активовано процес.

Бібліотека надає різні типи даних, властиві моделям сигналів в апаратурі. Клас *sc_logic* є один біт, який має чотири значення: "0" – логічний нуль, '1' – логічна одиниця, 'X' – невизначений стан, 'Z' – стан високого імпедансу. Клас *sc_lv* є логічний вектор, елементами якого є значення класу *sc_logic*. У бібліотеці визначено звичайний бітовий тип *sc_bit*, що має два значення: "0" і "1", також визначено бітовий вектор *sc_bv*.

Для синхронізації роботи паралельних процесів реалізовані різні класи елементарних каналів: *sc_mutex, sc_semaphore, sc_fifo. sc_mutex* – канал, який реалізує м'ютекс (англ. *mutex*, від *mutual exclusion* – обопільний виняток). Об'єкт класу *sc_mutex* дозволяє різним процесом забезпечити синхронний доступ до ресурсу. Використання цього класу гарантує, що тільки один процес матиме доступ до будь-якого ресурсу в один час. Для м'ютекса доступні методи «заблокувати», «перевірити блокування», «розблокувати». Типове використання м'ютекса при проектуванні системи – арбітр шини даних, який контролює доступ декількох пристроїв до однієї шині; *sc_semaphore* – канал, який реалізує семафор. Об'єкт класу *sc_semaphore*, на відміну від м'ютекса, дозволяє забезпечити одночасний доступ до загального ресурсу наперед заданій кількості процесів.

Мова SystemC знаходить широке застосування в різних завданнях проектування цифрових систем. В роботі [32] повідомляється про досвід поведінкового синтезу моделі шини PCI на мові SystemC за допомогою програмних продуктів фірми Synopsys. Вказується обмеження на синтезовану підмножину мови. Підтримуються типи SystemC: *sc_bit, sc_bv, sc_int, sc_bigint, sc_biguint,* і типи мови C ++: *bool, int, long, short*. Підтримується тільки процес типу *sc_cthread*, в якому список чутливості не містить ніяких інших сигналів, крім глобального тактового сигналу. В роботі [33] показано використання мови SystemC для перевірки коректності вихідної моделі системи за допомогою кінцевих автоматів. В роботі [34] повідомляється про використання SystemC для аналізу змагання сигналів в цифрових схемах. В роботі [35] описана модель шини AMBA [36] на мові SystemC. Розроблено необхідні інтерфейси, які приховують в собі деталі протоколу для обміну даних на шині. З використанням такої моделі можна домогтися моделювання послідовного читання або запису через шину за один такт, замість десятків тактів, якщо моделювати на рівні сигналів. В роботі [37] описується синтезовані класи – розширення до існуючих в SystemC. При синтезі відбувається відображення полів об'єкта в бітовий вектор. Доступ до таких полів

здійснюється за допомогою доступу до діапазону цього вектора. Конструктори, функції-члени і деструктори перетворюються в звичайні функції, які оперують над бітовими векторами відповідних об'єктів. Схожий підхід застосовується в [38], з тією відмінністю, що при синтезі підтримуються тільки об'єкти.

У лістингу 1.1 показаний вихідний код моделі суматора на мові SystemC. Модель має два входи a і b типу int, вихід sum типу int. У конструкторі модуля (класу) відбувається реєстрація процесу, асоційованого з цим модулем, в ядрі моделювання SystemC. Процес виконує необхідну логіку: суму двох вхідних значень. Також в конструкторі реєстрація списку чутливості процесу void do_add ().

Лістинг. 1.1. Оригінальний текст суматора на мові SystemC

```
1.   // підключення заголовки бібліотеки SystemC
2.   #include "systemc.h"
3.
4.   // визначення модуля (класу)
5.   SC_MODULE (adder)
6.   {
7.   // порти
8.   sc_in <int> a, b;
9.   sc_out <int> sum;
10.
11.  // процес
12.  void do_add ()
13.  {
14.  sum = a + b;
15.  }
16.
17.  // конструктор модуля
18.  SC_CTOR (adder)
19.  {
20.  // реєстрація процесу do_add в ядрі моделювання
21.  SC_METHOD (do_add);
22.
23.  // список чутливості процесу do_add
24.  sensitive << a << b;
25.  }
26.  };
```

Розроблено кілька чорнових версій стандарту на синтезовану підмножину мови SystemC. На момент написання дисертації остання версія 1,3 від

27 серпня 2009 года [39]. Крім синтезованої підмножини, цей стандарт визначає вимоги до написання синтезованого коду. В синтезовану підмножину входить:

—модулі, декларовані з використанням макро SC_MODULE, а також пряме успадкування від класу *sc_module* або від класу, що має в предках *sc_module* (модуль може мати тільки один конструктор);

—порти: *sc_in, sc_out, sc_inout*;

—інтегральні типи даних C ++: *bool, unsigned char, signed char, char, unsigned short, signed short, unsigned int, signed int, unsigned long, signed long, unsigned long long, signed long long* (бітова ширина цих типів визначається відповідно до популярних C ++ компіляторів);

—підтримка статичних масивів;

—підтримка статичних покажчиків;

—посилання;

—перерахування.

Стандартом не тримається

—числа з плаваючою точкою;

—динамічні покажчики, а також операції з покажчиками;

—функції main і sc_main не синтезуються. Функція sc_main може використовуватися для визначення параметрів шаблонів або конструкторів;

—new, delete, new [], delete [];

—виключення і їх обробка.

Наступні конструкції повинні розпізнаватися системою синтезу, але ігноруватися:

—створення об'єкта типу sc_trace_file;

—виклик функції sc_trace;

—виклик функції printf;

—використання оператора << для потоку cout.

Таким чином, стандарт визначає необхідний мінімум, які повинні підтримувати всі програмні продукти, які претендують на звання систем синтезу SystemC. Проте, виробники в праві самостійно розширювати синтезовані підмножини за рамки, зазначені в стандарті.

Мова SpecC [40] (Читається «спек-сі») - розширення до мови С, за допомогою якого можна описувати програмно-апаратні системи. Мова SpecC має синтаксичні конструкції, які підтримують структурну ієрархію, паралелізм, обмін повідомленнями, синхронізацію, зміну станів автомата, обробку винятків і вимір часу.

Опис моделі на мові HardwareC складається з множини взаємодіючих процесів. Всі процеси виконуються паралельно. Кожен процес починає роботу заново після завершення роботи. Всередині процесу можна описати послідовні і паралельні операції. Мова має синтаксис, подібний до мови С, не має покажчиків [41].

1.3. Засоби високорівневого синтезу

Фірма Cadence пропонує рішення C-to-Silicon Compiler [42], яке дозволяє автоматично генерувати код на мові Verilog на рівні регістрових передач з C / C ++ / SystemC описів. Програма реалізує кілька технологій: оптимізація керуючого і операційного автоматів, інкрементальний синтез, можливість апаратного прискорення верифікації, визначення часових затримок і необхідної площі на кристалі. Система підтримує базу даних з версіями проекту за мірою розробки, що дозволяє інженеру робити експерименти, з можливістю відкотитися на попередню придатну версію, в разі невдалого проектного рішення. Програма зберігає вихідні тексти моделей і обмеження на реалізацію окремо, що дозволяє неодноразово використовувати вже готові моделі в інших проектах, змінивши обмеження.

Фірма Synopsys розробила програму System Studio [43], яка дозволяє проектувати програмно-апаратні системи на високому рівні абстракції. Є

підтримка великої бібліотеки, що включає в себе більше 2000 моделей для додатків обробки сигналів. Є можливість інтеграції з функціями, написаними на мовах C / C ++. Програма дозволяє отримувати моделі, написані на мові SystemC. Є підтримка спільного моделювання з моделями на мовах VHDL і Verilog. Обмеження на синтез C ++ і SystemC коду: оголошення класів, успадкування класів, робота з динамічною пам'яттю (оператори new, delete), обробка виключень, рекурсивний виклик функції, перевантаження функцій, вбудовані C ++ функції, віртуальні функції, статичні члени, перевантаження операторів, шаблони. Типи C ++ і SystemC, які не придатні для синтезу: числа з плаваючою точкою (float, double), sc_fixed, sc_ufixed, sc_fix, sc_ufix, покажчики, файл, потоки введення / виводу.

Фірма Mentor Graphics пропонує програму Catapult C Synthesis [44], яка обробляє вхідний файл на мові ANSI C ++, синтезує з нього опис на рівні регістрових передач у 20 разів швидше, ніж традиційні ручні методи. Є підтримка об'єктно-орієнтованого програмування і шаблонів, що дозволяє неодноразове використання параметрізуємих моделей пристроїв. Система має свої власні вбудовані типи для реалізації цілочисельних і речових типів даних. Це зроблено з метою оптимізації при реалізації інтегральної схеми, в тих випадках, коли використання 64- і 32-розрядних регістрів і шин є надмірною. Ці типи даних є параметризуємими, і програма синтезу підлаштовує розрядність для кожного конкретного випадку. Програма автоматично синтезує інтерфейси для сполучення моделі на мові C ++ з моделями на MOA VHDL або Verilog. Програма також дозволяє використовувати єдиний тест, як для вихідного алгоритму на мові C ++, так і для синтезованої RTL-моделі, що скорочує час на верифікацію. Для оптимізації проектів використовуються такі методи: розгортка і конвейєризація циклів, злиття циклів, розділення і злиття модулів пам'яті. Програма Catapult C була успішно використана для більш ніж 200 замовлених мікросхем і ПЛІС [45].

Фірма Y Explorations, Inc. пропонує програму eXCite [46]. Вхідним форматом для цієї програми є мова ANSI C. Програма здійснює синтез про-

ектів, застосовуючи згортку багатовимірних циклів і конвейєризацію циклів (за допомогою директив) і всього пристрою для кращих часових характеристик. Програма також дозволяє вказати обмеження на пропускну здатність і продуктивність. Користувач за допомогою директив може вказати число використовуваних біт для вбудованих типів даних мови С, щоб отримати оптимальний результат синтезу. eXCite Professional дозволяє використовувати готові логічні блоки в проекті, що істотно скорочує час на розробку і знижує ризики.

Фірма Forte Design Systems пропонує рішення Cynthesizer для високорівневого синтезу [47]. Програма забезпечує продуктивність інженера в два мільйони логічних вентилів на рік, у порівнянні від 200 000 вентилів на рік, проектуючи на рівні регістрових передач. Програма приймає на вході моделі, описані на мові С ++ без часових характеристик, а на виході виробляє опис моделі на рівні регістрових передач. Є можливість генерації інтерфейсів до моделей на МОА для спільної верифікації. Cynthesizer також дозволяє синтезувати моделі на рівні транзакцій. Програма інтегрована з бібліотекою логічних блоків на поведінковому рівні (behavioral IP), використання яких істотно скорочує час на розробку проекту і верифікацію. Cynthesizer може інтегруватися з програмою SystemCoDesigner, розробленої в університеті Нюрнберга (Німеччина) [48]. Ця програма дозволяє синтезувати набір різних проектних рішень для різної елементної бази, наприклад, з різними типами шини, елементів пам'яті або процесорів. Далі інженер, оцінивши компроміс між різними параметрами – швидкодією, вартістю, надійністю, енергоспоживанням – вибирає оптимальний варіант реалізації програмно-апаратної системи на кристалі. SystemCoDesigner також дозволяє робити прототипи на FPGA-кристалі, що значно скорочує час верифікації проекту і підвищує адекватність отриманих результатів. Автори повідомляють про істотне (в кілька разів) скорочення витрат часу на розробку проекту, а також збільшення швидкодії.

Фірма SystemCrafter пропонує програму синтезу SystemCrafter SC [49], яка перетворює модель, описану за допомогою підмножини мов C ++ / SystemC в синтезований VHDL або Verilog код. Речові типи даних не підтримуються через коштовну апаратну реалізацію. Розподіл системи на програмні і апаратні частини здійснюється в ручному режимі. Як і в інших програмах високорівневого синтезу, SystemCrafter дозволяє повторно використовувати тести на всіх рівнях проектування.

Система SPARK [50], розроблена в Каліфорнійському університеті, використовується для високорівневого синтезу. Вона обробляє вхідний файл на мові C, і перетворює його в VHDL модель на рівні регістрових передач. Програма виконує деякі перетворення вихідного коду: розгортку циклів, обчислення констант в тілі циклу, винесення інваріантів циклу. Обмеження системи: немає конвейєризації, немає підтримки покажчиків, немає підтримки рекурсивних функцій, немає підтримки багатовимірних масивів, немає підтримки користувальницьких типів даних (класи), немає підтримки тернарного оператора a?b: c.

Фірма NEC пропонує систему CyberWorkBench [51, 52], яка призначена для ефективного проектування HBIC. Для цього проекти повинні бути описані на мові C, C ++, SystemC. Система надає засоби для поведінкового синтезу, програмно-апаратного моделювання, формальної верифікації. Програма поведінкового синтезу генерує VHDL або Verilog модель на рівні регістрових передач, яка може бути далі використана для перетворення в модель на рівні вентилів. Система також синтезує інтерфейси до шин, наприклад, AMBA-AHB. Є можливість конвертувати RTL-моделі з VHDL або Verilog назад в C ++ і SystemC, щоб збільшити швидкість моделювання. Для валідації проекту використовуються методи формальної верифікації, які перевіряють еквівалентність вихідного поведінкового опису на мові C / C ++ і отриманого RTL-рішення.

Система ROCCC (riverside optimizing configurable computing compiler), розроблена в Каліфорнійському університеті (Ріверсайд, США [53]), – оп-

тимізуючий компілятор C ++ і Фортрана в VHDL на рівні регістрових передач. Відмінність цієї системи від стандартного компілятора C або Фортрана полягає в тому, що крім потоку інструкцій, вона генерує архітектуру замовного мікропроцесора. На етапі оптимізації виробляються згортки і розгортки циклів, конвейєризація циклів з метою підвищення швидкодії і мінімізації використовуваних ресурсів. На етапі логічного синтезу є можливість конфігурувати розміри шин даних між блоками мікропроцесора, кількість регістрів, кількість ступенів конвеєра, а також згенерувати нестандартні виконавчі блоки. Система призначена для генерації проектів для ПЛІС. Повідомляється, що отримані програмно-апаратні реалізації працюють від десятків до тисяч разів швидше, ніж реалізації на традиційних мікропроцесорах. У вихідному коді заборонено використовувати покажчики, оператори break і continue. Крім того, всі адреси пам'яті повинні бути обчислюваними на момент компіляції, що істотно знижує множину придатних вихідних файлів.

Фірма Хьюлет-Пакард розробила систему Піко-Н [54, 55]. Система використовується для автоматизації високорівневого синтезу непрограмованих апаратних прискорювачів – функціональних блоків з жорсткою логікою. Система обробляє вкладені цикли на мові C і генерує апаратну модель на мові VHDL у вигляді систолічного масиву з високим ступенем паралелізму, а також керуючий автомат, інтерфейс до ОЗУ і до периферійних пристроїв. Використовується спеціальна підсистема, яка вибирає оптимальний варіант з простору проектних рішень. Вона варіює наступними параметрами, які можуть бути обмежені інженером: кількість процесорів в масиві, топологія зв'язків, пропускна здатність шини даних. Процес трансляції складається з наступних етапів: аналіз вихідного коду, призначення ітерацій на процесори, трансформація та оптимізація циклу, оптимізація потоку операцій і розмірності даних, синтез процесора, синтез системи, генерація вихідної VHDL-моделі і оцінка витрат апаратури. Обмеження на вхіднуі модель: межі циклу повинні бути константними, посилання на пам'ять повинні бути зв'язковими,

залежності змінних між ітераціями повинні бути однорідними – мати постійну відстань, незалежну від розміру масиву.

Фірма CebaTech розробила компілятор C2R [56], який транслює вихідний опис на ANSI C в код на Верілогі рівня реєстрових передач, придатний для синтезу. У коді можливе використання спеціальних директив компілятору, які дозволяють генерувати різні мікроархітектурні рішення. Компілятор підтримує структури, масиви, покажчики, різні проекти: як для ПЛІС, жорсткої логіки і систем на кристалах. Недоліки компілятора: немає підтримки мови C ++ і бібліотеки SystemC.

Розглянемо, яким чином споживачі проявляли інтерес до засобів високорівневого синтезу. На рис. 1.4 показані обсяги продаж програмних продуктів для синтезу за період 1994-2007 року [57]. Можна чітко виділити наступні періоди розвитку:

а) до 1994 року алгоритмічний і поведінковий синтез існував у вигляді науково-дослідних розробок і не мав істотного комерційного успіху;

б) в період з 1994 по 1996 рік спостерігалося незначне зростання продажів дослідних зразків;

в) з 1997 по 1999 рік на ринку домінує Synopsys Behavioral Compier, Cadence, Mentor Graphics, але продукти не виправдали очікувань користувачів, що призвело до зниження обсягів продажів;

г) з 2001 по 2003 рік спостерігається рецесія і недовіра користувачів до програм високорівневого синтезу;

г) з 2004 року на ринку з'явилися компілятори нового покоління, які повернули інтерес користувачів до високорівневого синтезу. У 2008 році лідерами ринку є фірми Mentor Graphics і Forte Design Systems.

За прогнозами аналітиків [58], до 2013 року обсяг ринку програм високорівневого синтезу може досягти 100 мільйонів доларів.

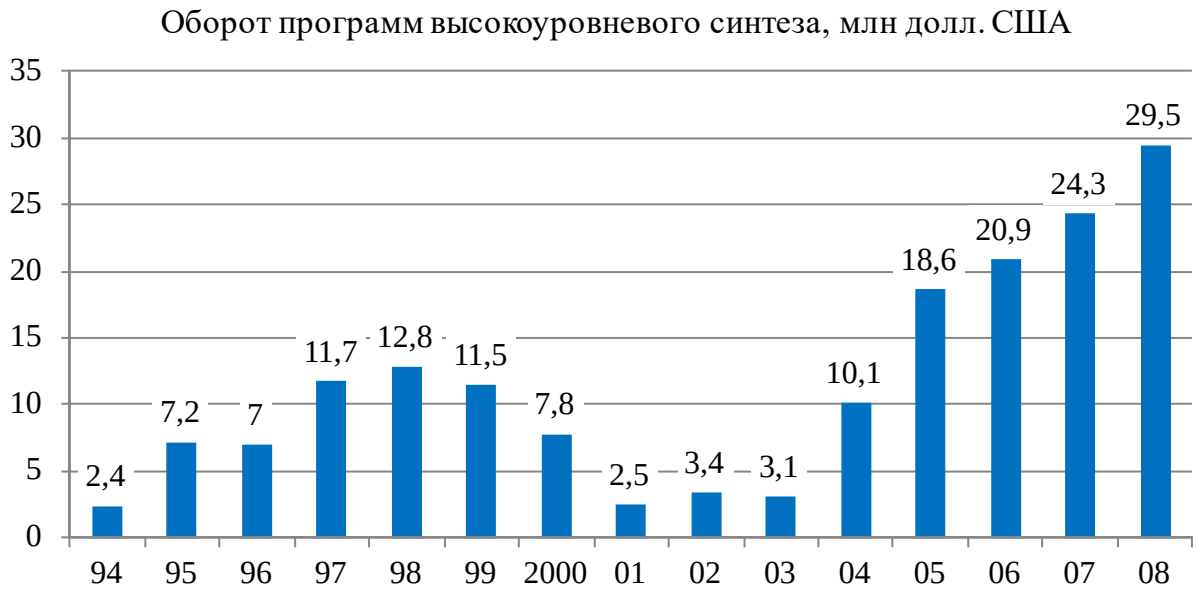


Рис. 1.4 Статистика продажів програм високорівневого синтезу [59, 60]

1.4. Характеристика системного рівня проектування

Системний рівень проектування характеризується спадним підходом, коли на перших етапах створюється та налагоджується модель на досить високому рівні абстракції (специфікації, алгоритми), а потім окремі компоненти відображаються на більш низький рівень абстракції, аж до фізичної реалізації (топологія кристала). Системний рівень представляється сукупність об'єктів, які в свою чергу є системами. Іншими словами, на системному рівні нас цікавить питання «Що проект робить?», а не «Як проект фізично реалізований»?

Систему S можна виразити наступною трійкою:

$$S = < X, Y, Z >,$$

де X – множина вхідних змінних, Y – множина вихідних змінних, Z – множина внутрішніх функцій.

Відмінні риси системного рівня проектування:

—висока продуктивність інженерів [61];

—використання мов високого рівня для опису системи [62];

—спільне проектування програмних і апаратних модулів;

—використання готових протестованих логічних блоків (IP core) [63, 64, 101] з множини, що надається різними виробниками;

—абстракція від технології реалізації системи [102].

У табл. 1.1 показані рівні опису системи і відповідні їм цільові архітектури. Топологія кристала – найнижчий рівень, системний рівень – самий верхній. Тут можна простежити, як зростає від низу до верху гнучкість проекту і продуктивність інженера. Наприклад, проект, представлений у вигляді топології кристала, придатний до реалізації тільки на мікросхемі конкретної технології. Якщо технологія змінюється, то повторно використовувати такий проект неможливо. Проект на вентильному рівні (система булевих функцій) швидше розробляти і верифікувати, ніж топологію кристала, і його можна повторно використовувати для мікросхем різних технологій. І тільки системний рівень дає можливість для спільного програмно-апаратного проектування для різних платформ.

Таблиця 1.1 Відповідність рівня опису та цільової архітектури

рівень опису	**цільова архітектура**
Системний рівень опису (SystemC, C ++)	Мікропроцесор, FPGA, ASIC
Мова програмування (C ++)	мікропроцесор
Рівень реєстрових передач (VHDL, Verilog)	FPGA, ASIC різних технологій
вентильний рівень	ASIC різних технологій
транзисторний рівень	ASIC різних технологій
топологія кристала	ASIC конкретної технології

Системний рівень проектування знижує ризики виробництва за рахунок використання протестованих компонентів і верифікації на ранній стадії.

Виділяють кілька етапів високорівневого проектування. Перший – етап функціональної специфікації [65]. На цьому етапі ще невідомо, як буде ре-

алізована та чи інша підсистема, у вигляді програмного коду або апаратури. Система описується в вигляді множини процесів, які пов'язані між собою каналами або сигналами [66]. Якщо система описується за допомогою мови C++, то така специфікація може бути виконана і верифікована перед переходом на наступний етап проектування.

На етапі компіляції здійснюється автоматичне перетворення специфікації системи на мовах C++ і SystemC у внутрішні моделі, придатні для виконання наступних етапів.

Для побудови внутрішньої моделі пристрою застосовують кілька описів: граф потоків даних (DFG, data flow graph) і граф керуючої логіки (CFG, control flow graph). Може застосовуватися також змішаний граф даних і керуючої логіки (CDFG, control and data flow graph).

В [67] повідомляється про використання анотованих мереж Петрі для проміжного представлення SystemC-моделі.

Третій етап високорівневого проектування – це розподіл системи на програмну і апаратну частини, в результаті якого виходить архітектурна специфікація. Разом зі специфікацією системи надходять вимоги на продуктивність і вартість кінцевого виробу. Найчастіше, 100% програмне рішення виявляється занадто повільним, а 100% апаратна реалізація – надмірно дорогою [68]. Практика показує, що в більшості випадків тільки суміш програмних і апаратних блоків задовольняє умовам вартості і продуктивності (рис. 1.5).

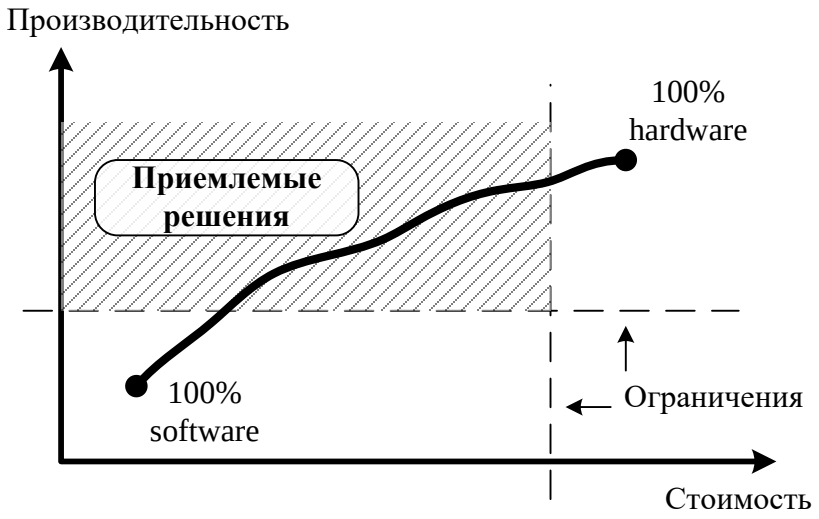


Рис. 1.5. Компроміс між 100% програмним і 100% апаратним рішеннями: змішана система, яка задовольняє обмеженням [х вище]

Насправді, проект має набагато більше альтернатив, ніж можливість реалізації окремого блоку в програмному або апаратному забезпеченні [103]. Серед параметрів реалізації можна виділити наступні: логічна конфігурація проекту, логічна організація блоку, тип інфраструктури на кристалі, технологія реалізації системи.

Логічна конфігурація проекту визначає, в якому вигляді буде представлений блок: програмному або апаратному. Кількість конфігурацій k для проекту з n блоків дорівнює $k = 2^n$.

Кожен логічний блок характеризується як мінімум двома параметрами: швидкодією і обсягом витрачених ресурсів (площа на кристалі або пам'ять процесора). *Логічна організація блоку* визначає, з якою метою блок був спроектований: максимальної швидкодії, яка зазвичай супроводжується структурною надмірністю, або мінімальною затратою ресурсів.

Тип інфраструктури на кристалі визначає використовувану шину або топологію мережі на кристалі, а також типи використовуваних вбудованих процесорів і блоків пам'яті. Шина або мережа характеризується пропускною здатністю. Процесори характеризуються вартістю, швидкодією, кількістю займаних ресурсів на кристалі.

Технологія реалізації визначається типом використовуваної апаратури: ПЛІС або кристали жорсткої логіки. Технології характеризуються вартістю, швидкодією, ємністю, енергоспоживанням, складністю виготовлення та налагодження [104].

Простір проектних рішень – множина всіх доступних альтернатив реалізації проекту. Кожна точка цього простору має значення характеристик реалізації проекту (рис. 1.6).

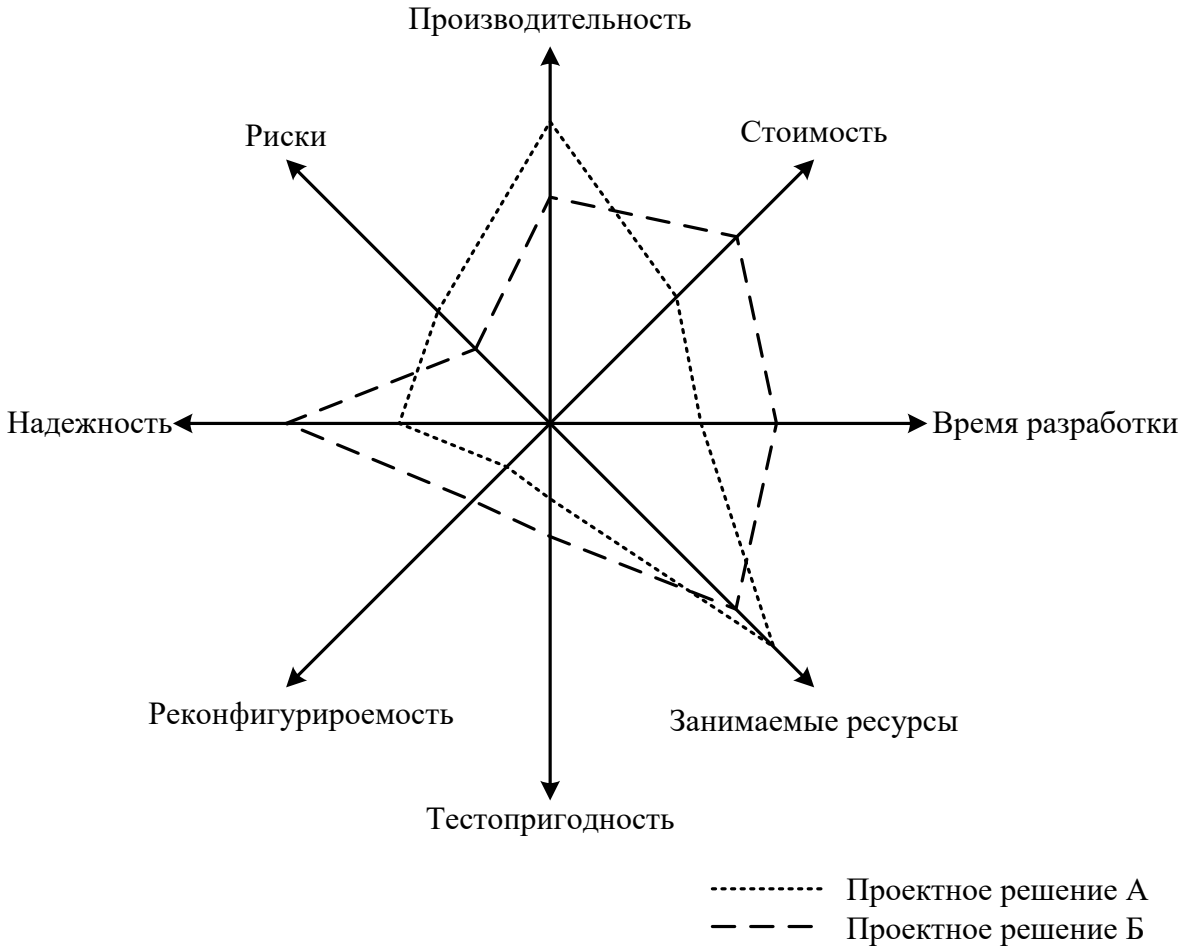


Рис. 1.6. Роза характеристик – графічний спосіб аналізу двох проектних рішень

В роботі [69] розглядається адаптивний маршрут проектування вбудованої системи. Перед розробниками стояло завдання: як скоротити вартість цифрового виробу за рахунок зміни логічної конфігурації проекту? Складається список всіх блоків, який поділяється на три групи: а) блоки, які мо-

жуть бути реалізовані тільки в апаратурі; б) блоки, які можуть бути реалізовані тільки в програмному забезпеченні; в) блоки, які можуть бути реалізовані як програмно, так і апаратно. Передбачається, що вартість реалізації блоку програмним способом дорівнює нулю, а вартість апаратного блоку - його собівартості. Фірма-виробник відстежує ситуацію на ринку, і на кожній ітерації змінює конфігурацію проекту: переводить апаратні блоки в програмні. В результаті знижується вартість кінцевого виробу. У роботі не розглядається компроміс між вартістю реалізації та іншими характеристиками, наприклад, продуктивністю компонента.

У зв'язку з тим, що необхідно здійснювати взаємодію між програмними і апаратними блоками, вводиться чотири моделі розподілу [70].

Модель із загальною однопортовою пам'яттю. У цій моделі всі змінні розміщуються в загальній однопортовій пам'яті, всі процеси отримують доступ через єдину загальну шину.

Модель з локальною пам'яттю і загальною однопортовою пам'яттю. У цій моделі всі локальні змінні розміщуються в локальній пам'яті, а глобальні змінні – в загальній. Процеси отримують доступ до локальних змінних через відповідні локальні шини, а до загальних – через єдину загальну шину. Після такого розподілу кількість шин в системі дорівнює p + 1, де p – кількість підсистем.

Модель з локальною пам'яттю і загальною багатопортовою пам'яттю. У цій моделі все локальні змінні розміщуються в локальній пам'яті, глобальні змінні – в глобальній пам'яті. Процеси отримують доступ до локальної пам'яті за відповідними шинам, а також до глобальної пам'яті по виділеним шинам. Максимальна кількість шин дорівнює $p + p \times p$, де p – кількість підсистем.

Модель з локальною пам'яттю і інтерфейсом до шини даних. У цій моделі всі змінні розміщуються з локальної пам'яті. Процеси отримують доступ до локальних змінних по відповідним шинам. Процес повинен використовувати інтерфейс до шини даних, якщо необхідно отримати доступ до ло-

кальної змінної, яка розміщена в локальній пам'яті іншої підсистеми. Максимальна кількість шин після розподілу дорівнює $2 \times p + 1$, де p – кількість підсистем після розподілу.

В роботі [71] описується модель взаємодії програмних і апаратних підсистем, заснована на неорієнтованому графі $G = (V, E)$, $V = \{v_1, \ldots, v_n\}, s, h : V$, і $c : E$. $s(v_i)$ (або s_i) і $h(v_i)$ (або h_i) визначають вартість програмного або апаратного рішення вузла відповідно, а (або визначає вартість обміну повідомленнями між вузлами, якщо вони знаходяться в різних контекстах.

Після етапу розподілу здійснюється синтез інтерфейсів між програмними і апаратними частинами [72]. Завдання визначається наступним чином: створити інтерфейс між двома процесами, які мають певні, але різні протоколи [73] (рис. 1.7). В [74] Вирішується завдання для двох різних компонентів, для яких дана тимчасова діаграма. Проводиться також оптимізація цього інтерфейсу. Якщо розробляється інтерфейс до апаратного блоку, то він буде представлений у вигляді логічної схеми, яка адаптує один протокол до іншого. Якщо розробляється інтерфейс до програмного блоку, то він буде представлений у вигляді спеціального драйвера.

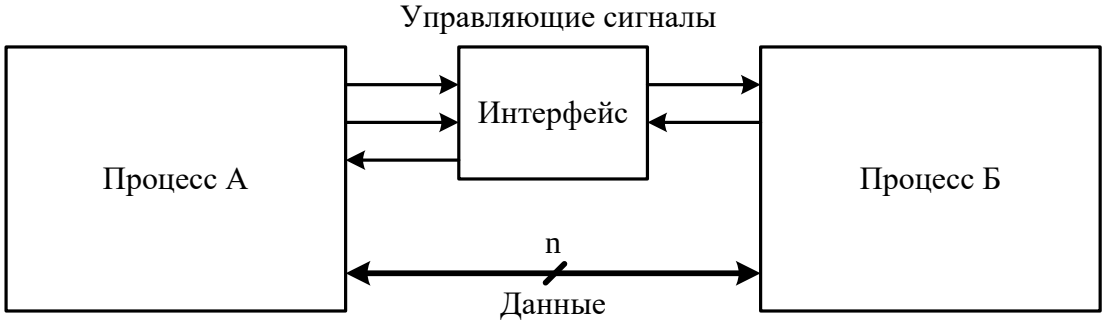


Рис. 1.7. Блок-інтерфейс погоджує протоколи між двома процесами

Існують різні рівні інтерфейсів [75]: електричний, логічний, послідовний, часовий, транзакційний, пакетний, рівень повідомлень. У [76] розглядається метод, заснований на високорівневих драйверах, які приховують деталі реалізації шини від розподіленого програмного забезпечення у вбудо-

ваній системі. Тут драйвер має ієрархічну структуру, і може бути визначений на спеціальній мові Ріа, який дозволяє визначити компоненти, як колекцію інтерфейсів, інтерфейси, як колекцію портів, порти, сигнали. В роботі [77] повідомляється про спеціальній формі записи протоколу, на основі якого синтезується апаратний інтерфейс. На рис. 1.8 показано опис протоколу роботи суматора: на першому такті роботи значення a і b передаються на входи суматора, а на четвертому такті значення c може бути лічено з його виходів.

$$\mathrm{t_{ADD}} = ((?\,\mathrm{a} \parallel ?\,\mathrm{b}) \rhd^{3} !\,\mathrm{c})$$

Рис. 1.8. Протокол роботи суматора

В [78] показаний метод синтезу інтерфейсів, заснований на моделях цифрових автоматів, за допомогою яких описано протокол передачі даних. В [79] запропонована програма Поляріс, яка генерує інтерфейси між апаратними блоками на мові опису апаратури.

На цьому етапі синтезу інтерфейсів також відбувається спільне програмно-апаратне моделювання для забезпечення правильності роботи моделі.

Третій етап високорівневого проектування – синтез підсистем. На етапі синтезу здійснюється синтез керуючого і операційного автоматів, вибір базових логічних блоків, планування виконання операцій.

Черговий етап – оптимізація, застосовується для отримання більш вигідних апаратних і програмних реалізацій.

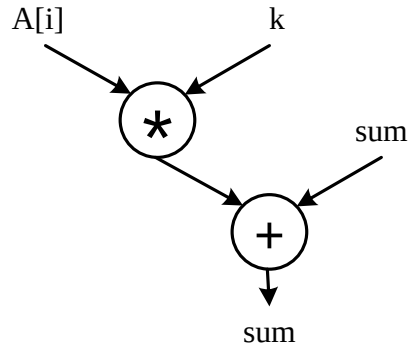
Для оптимізації циклів використовується кілька методів: згортка циклів, розгортка циклів, конвейеризація циклів, обчислення інваріантів циклу, зрушення ітерацій циклу [80].

Метод зсуву ітерацій циклу полягає в переміщенні деяких операторів на початку тіла циклу однієї ітерації в кінець тіла циклу попередньої ітерації. В результаті такої трансформації виходить оптимальне планування ресурсів апаратури, що дає виграш у швидкодії пристрою.

При використанні методу розгортки циклу кількість ітерацій скорочується в n разів. При цьому в тілі циклу необхідно виконати ітерації, кратні $i = 0,1,..n-1$ (рис. 1.9). При програмній реалізації цей метод дозволяє скоротити час виконання за рахунок скорочення кількості накладних інструкцій на організацію циклу. При апаратній реалізації цей метод може бути використаний для розпаралелювання обчислення операцій тіла циклу. Розгортка циклів застосовується при пошуку компромісу між займаними ресурсами, які мінімальні, коли цикл повністю згорнуто, і максимальні, коли цикл повністю розгорнуто. Але в першому випадку обчислення здійснюються послідовно, а в другому випадку можливе паралельне і конвейєрне обчислення, що істотно збільшує швидкість виконання.

а)

```
for (int i = 0; i < m; i++)
    sum += A[i] * k;
```



б)

```
for (int i = 0; i < m; i += 2)
{
    sum += A[i]   * k;
    sum += A[i+1] * k;
}
```

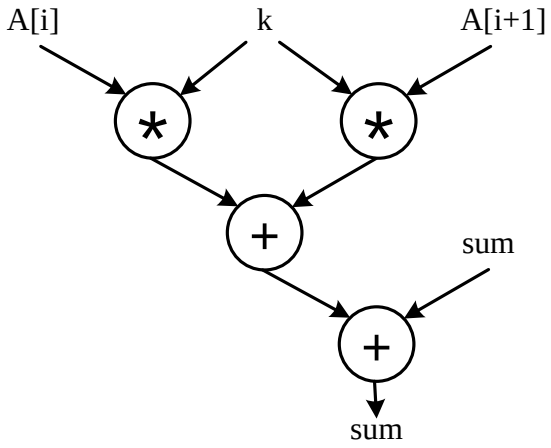


Рис. 1.9. Метод розгортки циклів: а – вихідний цикл; б – розгорнутий цикл

В [81] розглядаються моделі розрахунку періоду сигналу при розгортці циклів на різну кількість ітерацій. Повідомляється про більш ніж 25% виграші в продуктивності.

Первага використання мов високого рівня при описі систем полягає в тому, що можна швидко розробити і налагодити алгоритм або прототип системи. Тут може виникнути проблема трансляції опису на C ++ / SytemC в синтезований VHDL / Verilog код (рис. 1.10). Через відсутність засобів високорівневого синтезу цей перехід повинен бути виконаний вручну. У цьому випадку потрібні істотні витрати часу: від двох до шести місяців. Зазвичай, при ручному перекодуванні виконуються наступні етапи [82]: отримання інформаційного та керуючого графів з вихідного опису на C++, оптимізація (вбудовування функцій, розгортка циклів, обчислення константних виразів), конвейєризація, виділення апаратних ресурсів, планування виконання. Такий процес підданий ризикам: термін і якість ручного перекладу заздалегідь точно передбачити не можна.

Причини, які перешкоджають автоматичної трансляції з C ++ / SystemC в VHDL або Verilog-код:

—відсутність лінгвістичних аналізаторів мови C ++, орієнтованих на високорівневий синтез;

—відсутність підтримки бібліотеки SystemC;

—недостатньо потужна множина синтезованих конструкцій, яка підтримується тим або іншим програмним продуктом;

—використання бібліотек класів і функцій, які не мають синтезованих аналогів;

—нездатність програми синтезу розділити систему на апаратні (синтезовані) і програмні (компільовані) модулі;

—великі витрати часу і оперативної пам'яті на виконання алгоритмів синтезу.

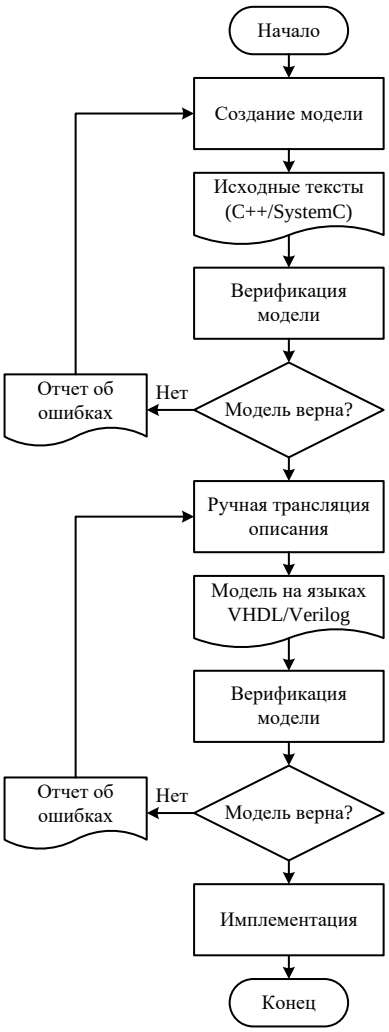


Рис. 1.10. Процес проектування з ручною трансляцією опису

1.5. Висновки до розділу 1

Системи автоматизованого проектування мікроелектроніки дозволяють інженерам створювати надскладні рішення за дуже короткий термін і з високою точністю. Значне скорочення часу виходу продукту на ринок і його вартості досягається за рахунок наступних чинників:

—використання САПР мікроелектроніки на кожному кроці проектування: введення, моделювання, верифікація, синтез, тестування;

—використання високорівневої мови проектування SystemC з широкими виразними можливостями C ++;

— повторне використання готових і протестованих логічних блоків при проектуванні моделі пристрою;

— висока швидкість і точність методів, реалізованих в програмному забезпеченні.

Можна виділити наступні актуальні проблеми:

— відсутність синтезованих моделей системного рівня;

— складність логічного синтезу з високорівневого опису на мовах C ++ / SystemC в модель рівня реєстрових передач;

— великі витрати часу на ручному перекодуванні з C ++ / SystemC в VHDL / Verilog моделі;

— відсутність коштів аналізу простору проектних рішень для програмно-апаратної реалізації пристрою.

Таким чином, можна визначити подальший розвиток моделей і методів автоматизованого проектування.

Функція мети при автоматизованому проектуванні цифрових пристроїв зводиться до мінімізації різниці параметрів моделі і параметрів, заявлених у вимогах до пристрою:

$$P^S = \{p_0^S, p_1^S, ..., p_i^S, ..., p_n^S\};$$
$$P^A = \{p_0^A, p_1^A, ..., p_i^A, ..., p_n^A\};$$
$$\Delta p_i = p_i^S - p_i^A;$$
$$\sum_{i=0}^{N} \Delta p_i \to 0.$$

Тут PS – множина параметрів пристрою, заявлених в специфікації, PA – множина параметрів пристрою, отриманих в результаті автоматизованого синтезу, Δpi – різниця i-того параметра специфікації і моделі пристрою.

1.6 Постановка мети і завдань наукового дослідження

Аналітичний огляд проектування, тестування і верифікації цифрових систем на кристалах орієнтований на визначення вузьких місць і переваг найцікавіших моделей і методів, опублікованих в спеціальній літературі: матеріали конференцій і журнали. За результатами огляду було сформульовано сутність, мета і завдання дослідження.

Сутність ринково-орієнтованого науково-технічного дослідження полягає в мультіверсному проектуванні архітектури цифрового виробу на основі заданої специфікації в середовищі SystemC (C++) і автоматичному виборі синтезованих функціональних структур з метою істотного зменшення часу створення проекту і підвищенні виходу придатної продукції за рахунок паралельного синтезу та верифікації архітектурних рішень системного рівня відповідно до запропонованої метрики. Основна інноваційна ідея – паралельний автоматичний синтез квазіоптимальної архітектури відповідно до запропонованої специфікації і метрики на основі підбору синтезованих функціональних структур.

Мета дослідження – суттєве зменшення часу проектування обчислювальних архітектур і підвищення якості цифрових виробів шляхом мультиверсного синтезу структури цифрового виробу на основі заданої специфікації в середовищі SystemC (C++) і автоматичному підборі функціональних компонентів за рахунок паралельного синтезу і верифікації архітектурних рішень системного рівня відповідно до запропонованої метрики.

Задачі:

1. Огляд існуючих моделей, методів, алгоритмів і програмних засобів створення цифрових систем на кристалах.

2. Розробка структур даних для опису функціональних примітивів системного рівня, орієнтованих на використання семантичних і синтаксичних

конструкцій мови C ++ і SystemC з метою забезпечення паралельного синтезу і верифікації архітектурних рішень.

3. Розробка методу синтезу інтерфейсних структур і протоколів виконання транзакцій RT-рівня на основі аналізу специфікації SoC-архітектури системного рівня, яка використовує стандартну шину Wishbone обміну даними між функціональними модулями.

4. Розробка методу синтезу RTL-моделей функціональностей шляхом перетворення C++ і SystemC-описів цифрових блоків системного рівня в алгоритми і структури даних автоматної моделі Мура, що задана синтезованою підмножиною мовних конструкцій VHDL.

5. Розробка методу мультиверсного синтезу керуючих і операційних автоматів, орієнтованих на архітектурні рішення в метриці, що мінімізує час виконання функціональності за рахунок розпаралелювання операцій при обмеженні на апаратні витрати.

6. Програмна реалізація моделей і методів мультиверсної розробки операційних пристроїв в рамках інтегрованої системи проектування функціональних і архітектурних рішень SoC на основі використання продуктів верифікації та синтезу компаній Aldec і Xilinx.

Економічний ефект від реалізації дисертаційного дослідження полягає в підвищенні швидкодії програмних і апаратних засобів синтезу і верифікації цифрових пристроїв за рахунок мультіверсності або опосередкованого збільшення програмно-апаратної надмірності використовуваної в процесі проектування обчислювальних виробів:

$$
\begin{aligned}
&E = F(L, T, H),\\
&Z = \min[L, T]\Big|_{(H \le 0{,}05F;\ Y \ge 95\%)},\\
&Y = (1-P)^{n};\\
&L = 1 - Y^{(1-k)} = 1 - (1-P)^{n(1-k)};\\
&T = \frac{(1-k)\times H^{s}}{H^{s}+H^{a}};\ H = \frac{H^{a}}{H^{s}+H^{a}},
\end{aligned}
$$

де L – параметр виходу придатної продукції Y, що залежить від якості процесу проектування k, ймовірності P існування некоректних рішень і числа несинтезованих блоків n. Час синтезу і верифікації залежить від якості мовних компонентів архітектури k, помноженої на число блоків, поділеної на загальну кількість функціональних примітивів специфікації. Мультіверсна надмірність H для створення пристрою залежить від числа повторень проекту, поділеній на загальну апаратну складність.

РОЗДІЛ 2
МЕТОДИ СИСТЕМНОГО І АРХІТЕКТУРНОГО СИНТЕЗУ

У розділі представлені методи системного і архітектурного синтезу компонентів цифрових систем на кристалах і інтерфейсних блоків, програм-но-апаратної декомпозиції. У циклі проектування SoC важливим є системний підхід, який полягає в: чіткої організація інформаційних потоків, повній відповідності міжнародним промисловим стандартам IEEE, ISO та інших профільних організацій. Використання стандартів IEEE 1500, 1149 необхідно для успішної інтеграції з програмами продуктами моделювання, верифікації, логічного синтезу, трасування і розміщення [111].

2.1 Постановка завдання синтезу

Мета – розробка методів синтезу інтерфейсних структур і протоколів виконання транзакцій RTL-рівня, а також RTL-моделі функціональності шляхом використання HDL-опису цифрових блоків, для подальшого мультиверсного створення компонентів цифрових систем на кристалах.

Завдання:

1. Розробка методу синтезу інтерфейсних структур і протоколів виконання транзакцій RT-рівня на основі аналізу специфікації SoC-архітектури системного рівня, яка використовує стандартну шину Wishbone обміну даними між функціональними модулями.

2. Розробка методу синтезу RTL-моделей функціональностей шляхом перетворення C++ і SystemC-описів цифрових блоків системного рівня в алгоритми і структури даних автоматної моделі Мура, що задана синтезованою підмножиною мовних конструкцій VHDL.

3. Створення інфраструктури проектування і верифікації компонентів цифрових систем на кристалах з метою перевірки алгоритмів перетворення специфікації з системного рівня на рівень RTL-опису.

4. Синтез різноманітних варіантів трансформування типових мовних конструкцій в моделі RT-рівня і подальша їх оцінка з позиції швидкодії та апаратних витрат.

Процес синтезу – це відображення специфікації на множину заданих примітивів, пов'язаних між собою у часі і просторі. При синтезі відбувається перетворення опису з більш високого рівня на більш низький.

Нехай S = {S1, S2, ..., Si, ... Sn} – множина функціональних і не функціональних вимог, визначених у специфікації, P = {P1, P2, ..., Pj, ..., Pk} – множина бібліотечних примітивів. На цих множинах визначається відношення $M \subset P \times S$ таке, що $m_{ij} \in M$ і $m_{ij} \in \{0,1\}$. Це відношення представлено в табличній формі:

	S_1	S_2	...	S_n
P_1	M_{11}	M_{12}	...	M_{1n}
P_2	M_{21}	M_{22}	...	M_{2n}
...	...	...	...	...
P_k	M_{k1}	M_{k2}	...	M_{kn}

(2.1)

Таким чином, одиниця на перетині стовпця і та рядку j означає, що бібліотечний примітив Pj задовольняє вимогу специфікації Si.

Тут завдання синтезу зводиться до покриття 100% вимог специфікації мінімальною кількістю бібліотечних примітивів.

2.3 Синтез інтерфейсних блоків

Модель системного рівня може не мати однозначного інтерфейсу на логічному рівні. Важливе завдання високорівневого синтезу – визначення (деталізація) інтерфейсу моделі, щоб її можна було однозначно представити на логічному рівні [105, 106].

Нехай F – функція, визначена за допомогою високорівневого стилю опису, X = (x1, x2, ..., xi) – вектор аргументів функції, Y = (y1, y2, ..., yk) –

вектор значень функції. Тоді в загальному випадку синтезована функція має вигляд:

$$Y = F(X). \quad (2.2)$$

Зазвичай високорівневі моделі не мають поняття модельного часу: всі обчислення відбуваються на одному умовному такті роботи. Моделі ж на логічному рівні працюють за тактами. Таким чином, можна виділити дві підзадачі: а) визначення відповідності аргументів функції на системному рівні і сигналів на логічному рівні; б) визначення протоколу взаємодії з синтезованою функцією. Тут протокол – це послідовність зміни інформаційних і керуючих сигналів схеми, яка призводить до виконання синтезованої функції.

Виконувана консольна програма на C++ має одну точку входу – функцію main, яка за стандартом має тільки два прототипи: *int main ()* та *int main (int argc, char * argv [])*, де *argc* – кількість аргументів командного рядка, *argv* – масив аргументів. Аргументи передаються у вигляді рядка символів. Змінити кількість і тип аргументів не можна, тому пропонується вважати модулями верхнього рівня ті функції, які викликаються з функції *main* або функції, зазначені користувачем. У цьому випадку функція *main* може використовуватися для тестування розроблюваних модулів: подавати вхідні тести і контролювати повернені значення.

У вихідної моделі на мові C++ інтерфейс може бути заданий декількома способами [107]:

— прототип функції;

— модуль SystemC;

— інший спосіб, відмінний від попередніх.

Під прототипом функції розуміється вираз виду:

```
return_type function_name (argument_list);
```

де *return_type* – тип значення, що повертається, *function_name* – ім'я функції, *argument_list* – список аргументів, що включає тип і ім'я аргументу. Функція може мати такі типи значення, що повертається: вбудовані типи в C ++, а також посилання або покажчики на вбудовані і призначені для користувача типи. Якщо тип значення – *void*, то мається на увазі відсутність значення, що повертається.

У табл. 2.1 показано відповідність між вбудованими типами C++ і синтезованими типами мови VHDL. Розмірність векторів обрана таким чином, щоб мати сумісність з популярними C++ компіляторами: Microsoft Visual Studio C++ Compiler, Intel C++ Compiler, g ++. Це зроблено для того, щоб забезпечити коректні результати при верифікації – сумісність результатів моделювання до і після синтезу. Слід звернути увагу (див. табл. 2.1) на перелічувальний тип *bool*, який може приймати два значення: *true* (істина) або *false* (неправда). Через особливості архітектури x86-сумісних мікропроцесорів, у всіх компіляторах тип *bool* займає один байт пам'яті. Будь-яке нульове значення інтерпретується, як «істина», а нульове – як «неправда».

Таблиця 2.1. Відповідність C ++ і VHDL типів

Тип C ++	Розмір, біт	Тип VHDL
char	8	std_logic_vector (7 downto 0)
unsigned char	8	std_logic_vector (7 downto 0)
signed char	8	std_logic_vector (7 downto 0)
bool	8	std_logic_vector (7 downto 0)
unsigned int	32	std_logic_vector (31 downto 0)
int	32	std_logic_vector (31 downto 0)
unsigned short int	16	std_logic_vector (15 downto 0)
unsigned long int	32	std_logic_vector (31 downto 0)
long int	32	std_logic_vector (31 downto 0)

short int	16	std_logic_vector (15 downto 0)
wchar_t	-	немає відповідності (не синтезується)
float	16	std_logic_vector (15 downto 0)
double	32	std_logic_vector (31 downto 0)
long long	64	std_logic_vector (63 downto 0)
long double	64	std_logic_vector (63 downto 0)
void	-	немає відповідності
* T (покажчик на T)	32	std_logic_vector (31 downto 0)

Таким чином, у розпорядженні програми синтезу є типи шириною 8, 16, 32 і 64 біта без проміжних варіантів. Ці типи будуть синтезовані в регістри і шини відповідного розміру. Це може привести до невиправданої витрати апаратних ресурсів в тому випадку, якщо область використовуваних значень змінної значно вужче, ніж можливість регістра. Наприклад, якщо потрібно закодувати 10 значень, то можна використовувати змінну типу *char*, яка займає 8 біт і може приймати 256 значень. Але для кодування 10 значень достатньо всього лише 4 біта. Отже, половина регістра використовуватися не буде. У загальному випадку, якщо потрібно закодувати n значень, то кількість необхідних біт k дорівнюватиме:

$$k = \lceil \log_2 n \rceil. \qquad (2.2)$$

Для вирішення цієї проблеми в SystemC представлені дві множини типів для роботи з арифметичними даними з довільною точністю: від 1 до 64 біта, і від 64 біта і вище. Класи sc_int (integer, знаковий тип) і sc_uint (unsigned integer, беззнаковий тип) дозволяють моделювати цілі арифметичні дані в діапазоні 1-64 біт. Ці типи даних рекомендується застосовувати тільки при синтезі, де необхідна однозначність результатів моделювання і синтезу.

Виключно для моделювання краще використовувати вбудований тип C++ *int*, який моделюється істотно швидше. Класи *sc_bigint* і *sc_biguint* застосовуються для моделювання великих чисел, ніж вбудовані типи і *sc_int* / *sc_uint*. Недолік цих типів – вони мають більш складну організацію, для них потрібно більше часу на моделювання. Для даних, ширина яких менше 64 біт, краще використовувати *sc_int*.

Існує кілька способів передачі аргументів на функцію: за значенням, через покажчик, за посиланням. Функція C++ може повернути тільки одне значення через оператор *return*. Проте, функція може змінити об'єкти, які були передані за посиланням або через покажчик.

Поняттям «функція, що викликає» і «викликаєма функція» в апаратурі відповідають поняття «ведучий модуль» і «ведений модуль» [108]. Ведучий модуль готує аргументи для передачі у ведений модуль, ініціює керуючі сигнали і чекає сповіщальний сигнал про завершення. Ведений модуль очікує керуючий сигнал «початок обчислень», зчитує аргументи, робить обчислення і оповіщає ведений модуль про завершення. UML-діаграма послідовності такого протоколу показана на рис. 2.1.

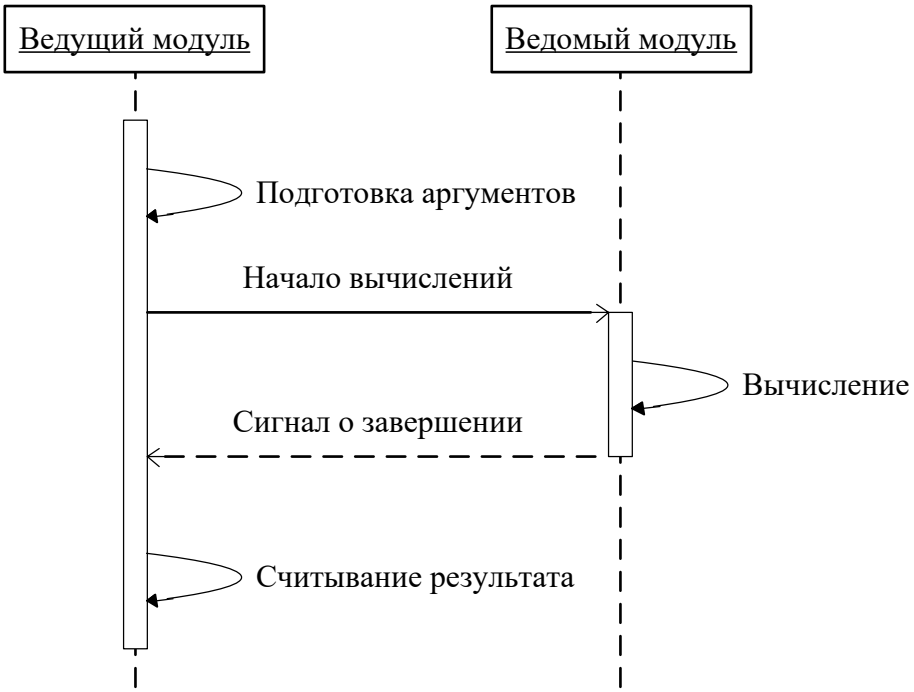


Рис. 2.1. UML-діаграма послідовності взаємодії ведучого і відомого модулів

При передачі аргументу за значенням, та чи інша змінна копіюється на стек, що викликається. В апаратурі це означає, що аргументи зчитуються з вхідних сигналів моделі і в деяких випадках можуть зберігатися у внутрішніх регістрах. Це необхідно в тих випадках, коли значення вхідних змінних можуть змінитися під час обчислень. На рис. 2.2 показаний інтерфейс функції *find_gcf* – пошук найбільшого загального кратного. Ця функція приймає два 32-бітних аргументи і повертає 32-бітове значення [109].

```
int find_gcf(int a, int b);
```

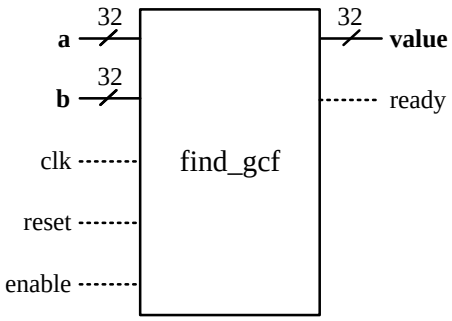


Рис. 2.2. Прототип функції *find_gcf* і відповідний інтерфейс модуля

Передача аргументу через покажчик означає, що замість значення об'єкта передається його адреса в загальній пам'яті. Модулю необхідно зчитати значення об'єкта з пам'яті. В інтерфейсі модуля на рис. 2.2 немає необхідних інформаційних та керуючих портів для доступу до оперативної пам'яті [110]. Щоб вирішити цю проблему, інтерфейс моделі повинен містити наступні порти (див. рис. 2.1):

—вхідна шина даних для передачі адреси об'єкта (a);

—вхідна і вихідна шини даних для доступу до пам'яті (*data_in*, *data_out*);

—вихідна шина результату (*value*);

—вихідні сигнали (*read / write*).

На рис. 2.3 показаний інтерфейс моделі для відповідного прототипу функції. У загальному випадку кількість вхідних шин повинна відповідати кількості переданих параметрів, а ширина – розмірності переданих аргументів. У прототипі функції *f1* явно зазначені тільки вхідний сигнал *a* – аргумент функції, який є покажчиком, і вихідний сигнал *value* – значення, що повертається. Решта сигналів (позначені пунктирними лініями) є допоміжними, і додаються в інтерфейс моделі при синтезі. Шини *data_in* і *data_out* підключаються до зовнішньої магістралі або безпосередньо до блоку пам'яті. Розрядність шини дорівнює 8 бітам, тому що використовується C++ тип *char*. Шина addr підключається до відповідного адресного входу блоку пам'яті. Через цю шину модель адресує елементи пам'яті. *Clk, reset, enable, read / write, ready* – відповідні службові та керуючі сигнали.

```
int f1 (char *a);
```

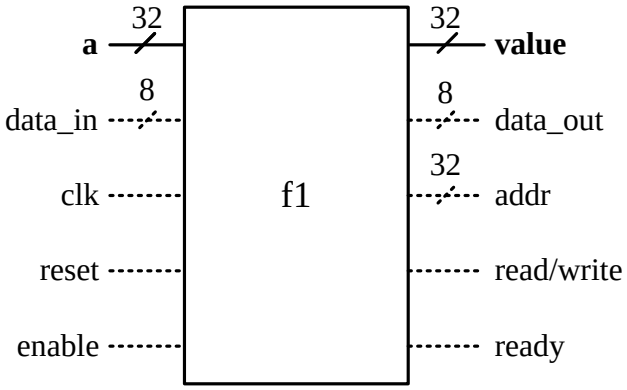


Рис. 2.3. Прототип функції *f1* і відповідний інтерфейс модуля

Як може бути синтезований виклик функції в апаратурні елементи? Якщо функція є комбінаційною, тобто на одні й ті ж аргументи функція повертає одні і ті ж значення (математична функція, *pure*-функція в мові VHDL), то така функція може бути представлена комбінаційним операційним блоком. Час обчислення такої функції має бути менше, ніж період одного такту роботи операційного пристрою. Якщо функцію неможливо уявити комбінаційним блоком, то вона повинна бути синтезована як послідовна, тобто буде обчислюватися за кілька тактів. У деяких випадках кількість тактів, за яке функція завершить свою роботу, невідомо. Для вирішення цієї проблеми потрібно розробити протокол виклику такої послідовної функції. На рис. 2.4 показана ГСА виклику функції. Алгоритм складається з наступних кроків.

1.Підготувати параметри функції – виставити значення регістрів, що безпосередньо прилягають до інтерфейсу модуля.

2.Встановити сигнал *enable* модуля в одиницю.

3.Чекати, до тих пір, поки функція не завершить свою роботу (сигнал ready = 1).

4.Встановити сигнал *enable* модуля в нуль.

5.Вважати вихідні значення функції в зовнішні регістри.

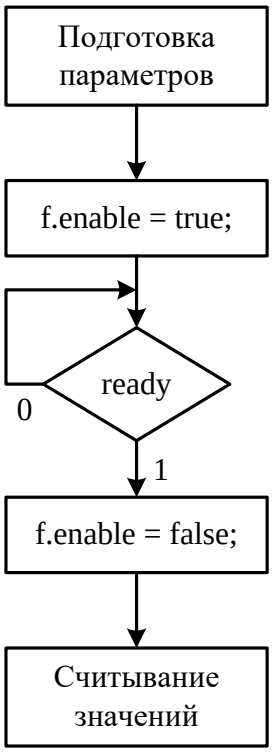


Рис. 2.4. Змістовна ГСА виклику функції

Для підтримки цього протоколу потрібно зміни в ГСА на стороні модуля, який представляє функцію, що викликається. Алгоритм роботи пристрою показаний нижче.

1. Вважати сигнал *enable*. Якщо він дорівнює одиниці, то перейти до п. 2. Якщо він дорівнює нулю, то повторити п. 1.
2. Обчислити значення функції.
3. Записати значення функції в вихідний регістр.
4. Виставити сигнал *ready* в одиницю на один такт.
5. Скинути сигнал *ready* в нуль.

На рис. 2.5 показана змістовна ГСА для підтримки протоколу.

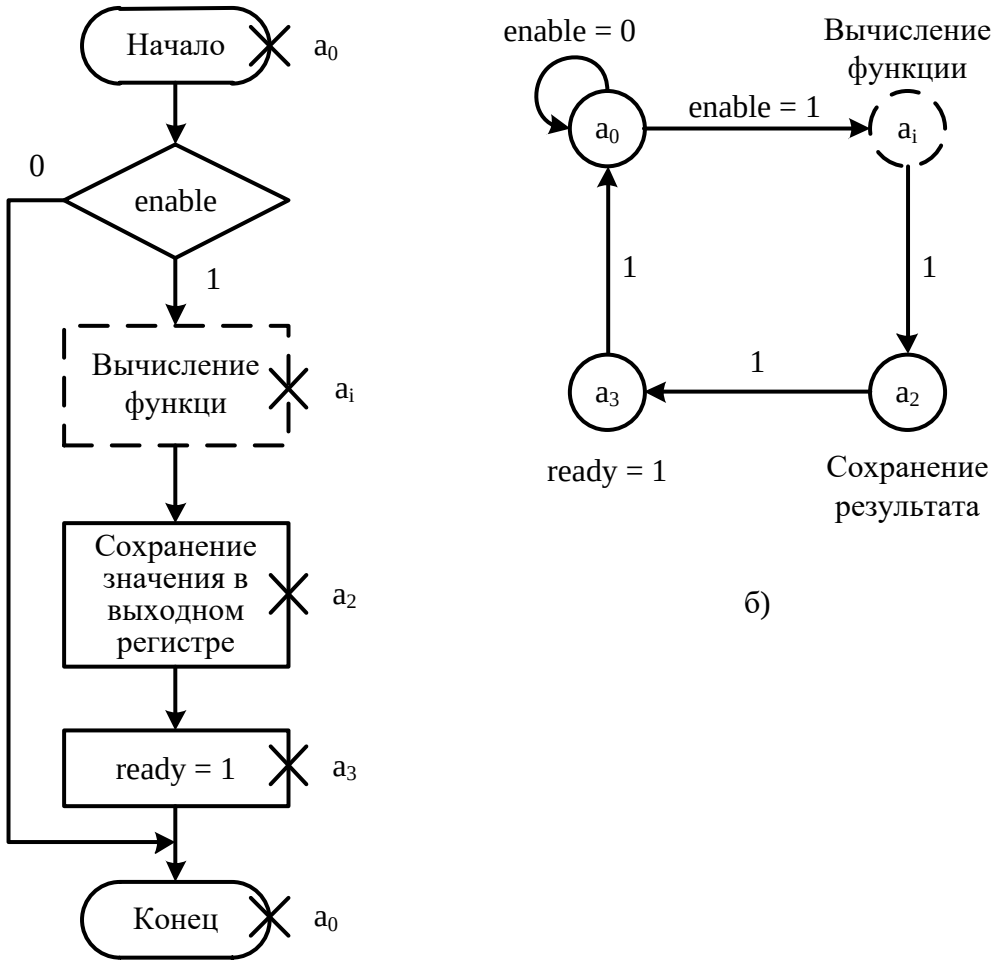


Рис. 2.5. Змістовна ГСА (а) і граф переходів автомата Мура (б) підтримки протоколу з боку модуля, що реалізує функцію *f1*

В результаті структурного синтезу виходить комбінація операційного і керуючого автоматів, як показано на рис. 2.6. Всі сигнали можна умовно розділити на наступні групи:

—сповіщальні, які передають дані в пристрій і з нього: *a*, *b*, *output_value* (всі розрядністю 32 біта);

—глобальні керуючі сигнали: сигнал синхронізації *clock* і сигнал скидання *reset*;

—протокол – сигнали, специфічні для конкретного протоколу: *enable* – дозвіл рахунку, *ready* – сигналізує про завершення рахунку. При реалізації

різних протоколів обміну даними, кількість і значення цих сигналів може відрізнятися;

—керуючі сигнали, які визначають активацію тих чи інших мікроопераций в операційному автоматі: *y1*, *y2*, ..., *yn*;

—інформуючі сигнали, які сповіщають керуючий автомат про стан тих чи інших умов в операційному автоматі: x1, x2, ..., xm.

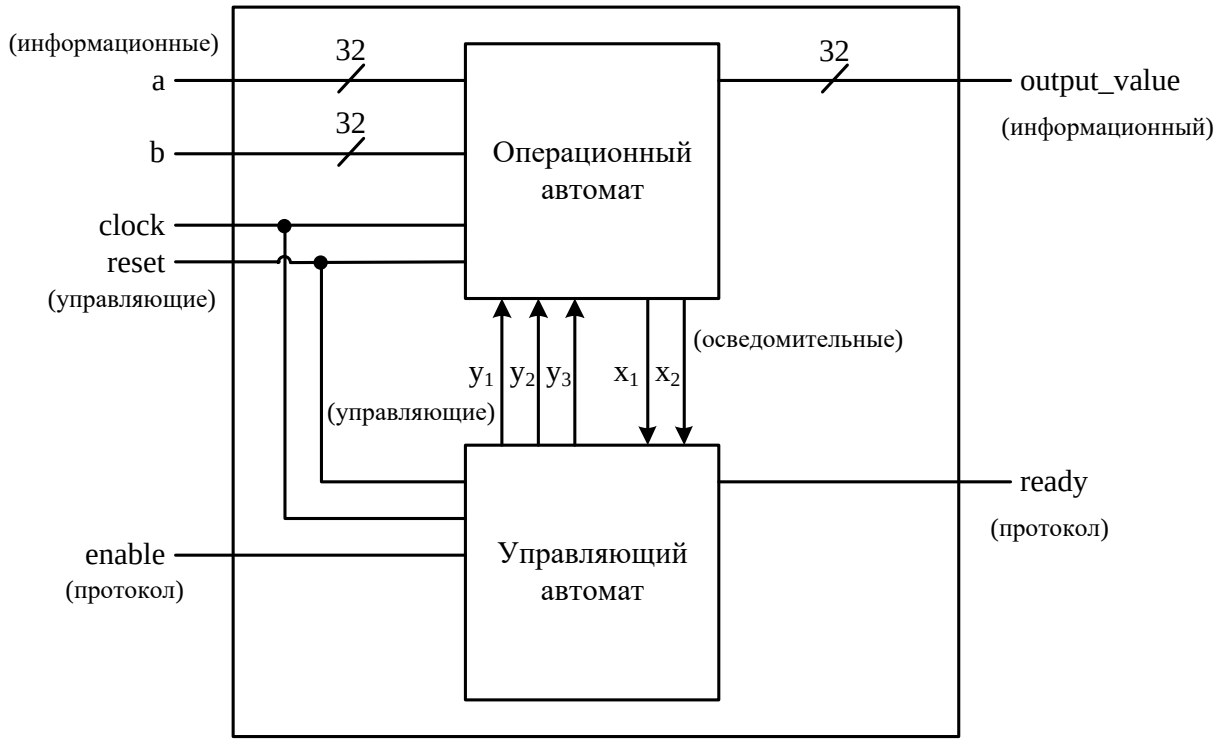


Рис. 2.6. Композиція керуючого і операційних автоматів

На рис. 2.6 показана часова діаграма обміну даними (значення вхідних і вихідних шин обрані випадково). На першому такті провідний модуль (який ініціює виклик функції) виставляє на входах відомого модуля (який обробляє виклик функції) необхідні аргументи. На цьому ж такті провідний модуль встановлює сигнал *enable* в одиницю, сигналізуючи про те, що аргументи валідність і відучий модуль може починати обчислення. На другому такті ведений модуль вважає аргументи з входів, і почне обчислення. На шостому такті роботи ведений модуль виставить значення функції на шину *value* і встановить сигнал *ready* в одиницю, сигналізуючи про те, що обчислення за-

вершені і ведучий модуль може зчитувати їх з виходів модуля. На сьомому такті роботи провідний модуль скине сигнал *enable*, ведений модуль скине сигнал *ready*. Система готова до наступного циклу виклику функції.

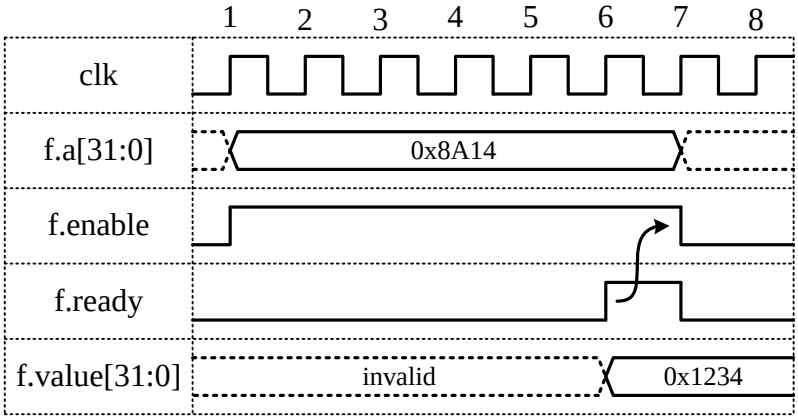


Рис. 2.7. Часова діаграма виклику функції

Час на виклик функції може бути оптимізовано за допомогою певної трансформації вихідного кода C++ програми. Така трансформація називається вбудовуванням функції (function inlining): тіло функції підставляється на місце її виклику, тим самим скорочується кількість тактів на синхронізацію [112].

Застосування такого спеціалізованого інтерфейсу має ряд недоліків. По-перше, з кількістю параметрів функції збільшується кількість інтерфейсних портів. Так, для функції з п'ятьма параметрами типу *int* буде потрібно 5*32=160 вхідних сигналів. Стандартом C++ не обмежується кількість переданих аргументів і їх розміри. Іншим важливим недоліком є те, що для функцій з різним набором параметрів інтерфейс модуля буде різним, що істотно ускладнює інтеграцію таких функцій в систему на кристалі. Рішенням цих проблем є використання стандартної шини для з'єднання модулів в системі на кристалі.

Розглянемо синтез інтерфейсів для популярного стандарту шини системи на кристалі Wishbone [x83]. Цей стандарт передбачає різні сполучення модулів в системі:

—з'єднання «точка-точка»;

—конвеєрне з'єднання (для обробки потоку даних);

—з'єднання «багато до багатьох» через загальну шину;

—з'єднання «багато до багатьох» через комутатор;

—з'єднання в складі мережі на кристалі [84].

Перерахуємо переваги стандарту Wishbone. Перш за все, це відкрита специфікація, що не передбачає грошових відрахувань правовласнику за її використання. Завдяки цьому, шина отримала поширення серед проектів цифрових систем з відкритим вихідним кодом (open source hardware) [113].

На рис. 2.8 показані інтерфейси ведучого і веденого модулів шини Wishbone, а також їх міжз'єднання.

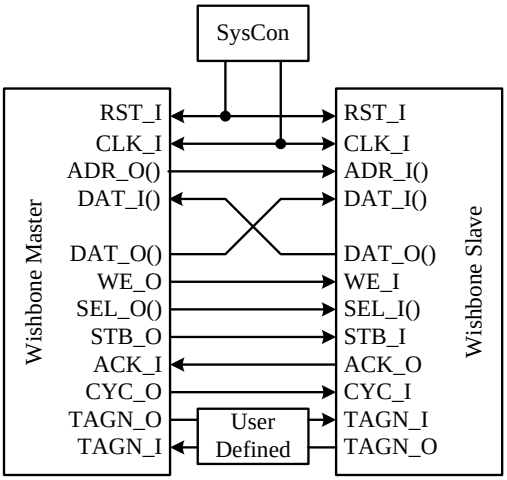


Рис. 2.8. Інтерфейси ведучого (зліва) і веденого (праворуч) модулів на шині Wishbone

У табл. 2.2. дано опис інтерфейсним сигналам протоколу Wishbone.

У разі об'єднання множини модулів в системі через загальну шину або комутатор, необхідно ввести відображення адресного простору на множину модулів в системі. Системний арбітр в залежності від пріоритету ведучого пристрою, а також в залежності від адреси, комутує пару ведучих і ведених пристроїв. На рис. 2.9 схематично показано одне з можливих відображень адресного простору. Весь адресний простір від адреси 000016 до FFFF16

(6553510) розділений між різними модулями. Наприклад, ПЗУ має адреси 000016-0FFF16; ОЗУ має адреси 100016-9FFF16; модуль 1 має адреси A00016-AFFF16; останній модуль N має адреси F00016-FFFF16. Програма синтезу системи на кристалі автоматично призначає адресні простори різних модулів в залежності від кількості адресованих осередків або регістрів.

Таблиця 2.2. Опис сигналів інтерфейсів модулів в системі Wishbone

Сигнал	Опис
CLK_I	Сигнал синхронізації. Всі операції зчитування даних відбуваються по передньому фронту цього сигналу.
RST_I	Скидання. За високого рівня цього сигналу логіка інтерфейсів скидається в початковий стан.
ADR_O	Шина адреси. Може бути 8, 16, 32 або 64 біт. Розрядність визначається об'ємом пам'яті, що адресується.
DAT_I, DAT_O	Шини даних, які прямують у двох напрямках: від ведучого модуля до веденого, і в зворотному напрямку. Можуть бути 8, 16, 32 або 64 біт.
WE_O	Керуючий сигнал ведучого модуля, який визначає тип операції: читання (низький рівень) або запис (високий).
SEL_O	Керуюча шина, яка визначає, який з байтів повинен бути зчитаний в шині даних. Для цього використовується позиційний код. Так, якщо шина даних має розмір 32 біта, але розмірність шини SEL_O буде [3: 0]. Якщо передається тільки один байт, наприклад, третій, то SEL_O матиме значення "0100".
STB_O	Строб операції. Ведений інтерфейс виконує операції тільки якщо цей сигнал встановлений в високий рівень.
ACK_I, ACK_O	Сповіщальний сигнал. Встановивши цей сигнал в високий рівень, ведене пристрій підтверджує, що дані були успішно прочитані або записані.
CYC_O	Сповіщальний сигнал. Встановивши цей сигнал в високий рівень, ведучий пристрій повідомляє, що розпочато цикл записи або читання з веденим пристроєм.
TAGN_O, TAG_I	Допоміжні шини, які можуть використовуватися, наприклад, для передачі знаків парності, або інших керуючих команд між ведучим і веденим пристроями. Необов'язкові сигнали.

Шина Wishbone передбачає використання часткового дешифрування адреси в інтерфейсі веденого модуля. Таким чином, кожен модуль дешифрує тільки використовуваний діапазон адрес. Наприклад, якщо ведений модуль використовує чотири внутрішніх адресованих регістра, то дешифрувати потрібно всього лише два адресних біта. Це призводить до суттєвої економії апаратних ресурсів, а також до більш високошвидкісний дешифрування адреси.

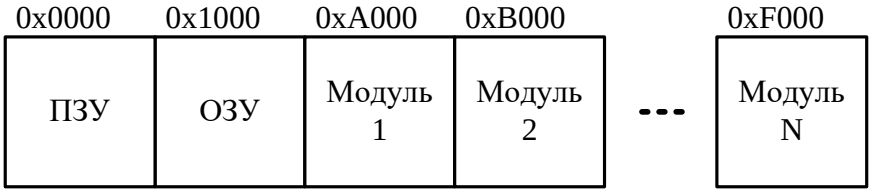


Рис. 2.9. Відображення адресного простору на множини модулів в системі

Шина Wishbone являє шину даних, розмір якої може бути від 8 до 64 біт, але завжди кратний одному байту. Таким чином, для передачі параметрів використовується послідовне завантаження значень у внутрішню пам'ять відомого модуля. Для цього використовується спеціальна адресація регістрів, які розташовані в інтерфейсі веденого модуля. Розглянемо приклад функції, яка приймає два параметри типу *long* (64 біта), і повертає значення типу *int* (32 біта). На рис. 2.10 показано відображення адресного простору модуля, що реалізує дану функцію, на внутрішні регістри. Регістр R1 використовується для зберігання значення функції. Він має розмір 32 біта і займає адреси 0-3. Регістри R2 і R3 мають розмір 64 біта кожен і займають адреси 4-19. Якщо шина даних має розмір менший, ніж адресується регістр, то потрібно кілька циклів запису. Так, якщо розмір шини даних дорівнює 32 бітам, то буде потрібно два цикли записи для регістра R2: за адресами 4 і 8.

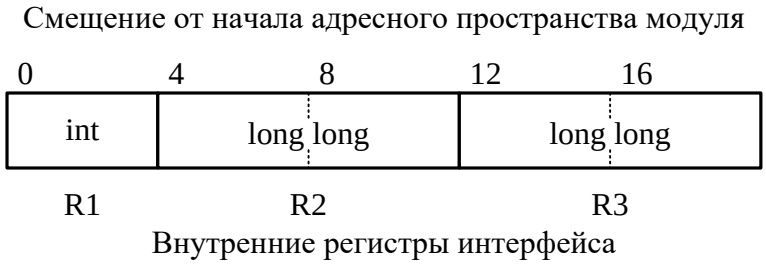


Рис. 2.10. Схема відображення адресного простору модуля на внутрішні регістри

У тому випадку, якщо розмір шини менше 32 біт, то потрібно кілька циклів читання результату з регістра R1. Можна розрахувати кількість тактів, необхідних для передачі параметрів. Нехай Pi – вектор параметрів функції, i=1..n, де n – кількість параметрів, переданих у функцію; S(Pi) – розмір параметра Pi в байтах; M – розрядність шини даних в байтах. Тоді кількість тактів l, необхідних для передачі параметрів, буде дорівнює значенню виразу:

$$l = \sum_{i=1}^{n} \left\lceil \frac{S(P_i)}{M} \right\rceil. \qquad (2.2)$$

У цьому випадку передбачається, що всі параметри передаються, що вирівнюються по межі ширини шини даних. Наприклад, якщо ширина шини даних M дорівнює 16 біт, і передається чотири параметри типу char (1 байт), то буде потрібно чотири такту записи. Видно, що за чотири такту можна передати вісім байт, але по факту передається лише чотири. Обчислимо ефективність e використання шини:

$$e = \frac{\sum_{i=1}^{n} S(P_i)}{l \times M} \times 100\%. \qquad (2.3)$$

У нашому прикладі n = 2, P1 = 8 біт, P2 = 8 біт, M = 16 біт, l = 2 такту, ефективність e = 16/32 = 50%. Можна застосувати спосіб, при якому параметри будуть упаковані, т. Е. За один такт роботи шини будуть передаватися

кілька параметрів. Тоді кількість тактів l, необхідних для передачі параметрів, буде дорівнює значенню:

$$l = \left\lceil \frac{\sum_{i=1}^{n} S(P_i)}{M} \right\rceil . \qquad (2.4)$$

На рис. 2.11 показані алгоритми роботи ведучого і веденого інтерфейсів в системі Wishbone. На рис. 2.11,б в стані *a3* показаний виклик функції *f1* з параметром *par*. Відповідно до цієї ГСА обчислення функції займає рівно один такт роботи автомата. Це може бути справедливо лише для високорівневих моделей без відсутності модельного часу. Наприклад, в моделі на мові SystemC значення функції обчислюється на тому ж дельта-циклі моделювання, коли був здійснений виклик. У системах на структурному рівні опису, такого можна домогтися тільки для комбінаційних функцій. Для автоматів, які потребують більшу кількість станів для завершення, необхідно змінити ГСА з рис. 2.11,а наступним чином. На місці стану *a3* необхідно підставити ГСА функції *f,* що викликається. Сигнал *ack* слід встановити в одиницю в останній операторній вершині ГСА функції *f1*.

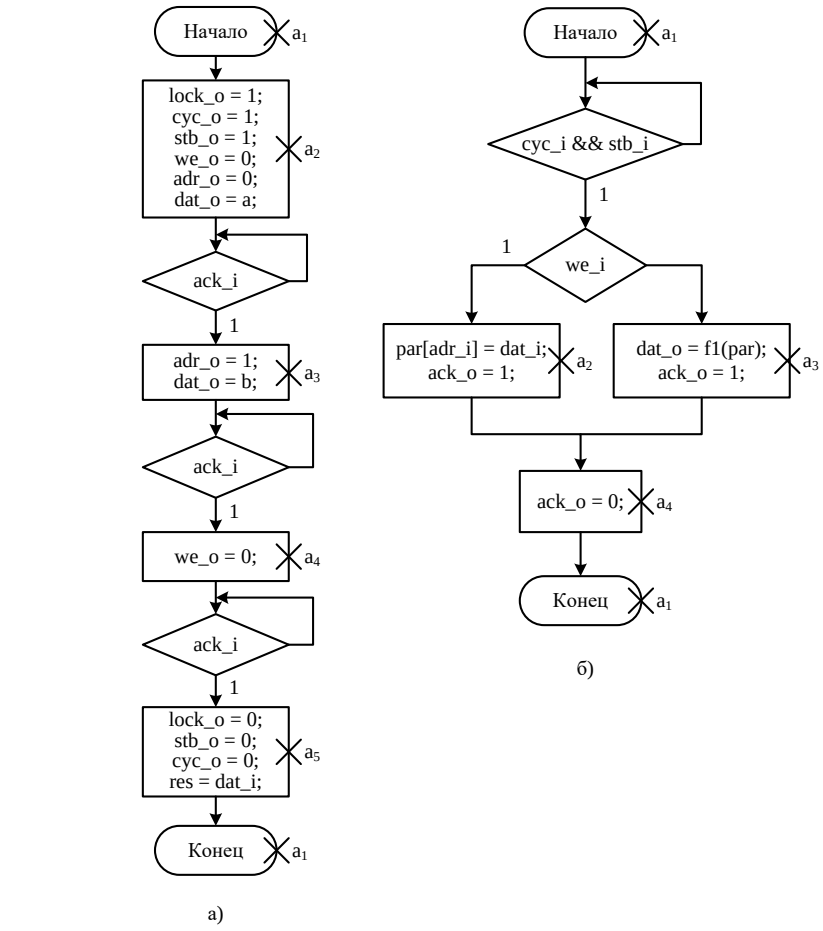


Рис. 2.11. Алгоритми роботи ведучого (а) і веденого (б) інтерфейсів в системі Wishbone

Автомати Мура для алгоритмів (див. рис. 2.11) показані на рис. 2.12.

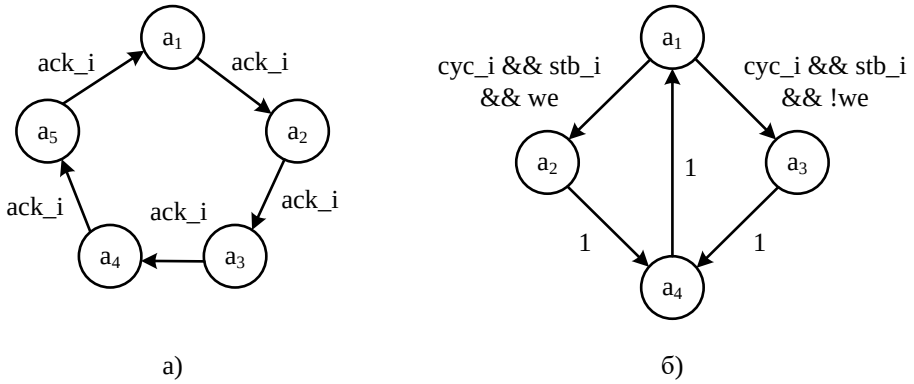


Рис. 2.12. Автомати Мура для алгоритмів ведучого (а) і веденого (б) модулів

Для автоматів, представлених на рис. 2.13, можуть бути отримані структурні еквіваленти. Таким чином, завдання отримання структурної організації інтерфейсів ведучого і веденого модулів вирішена. На рис. 2.9 пока-

зана часова діаграма виклику функції знаходження найбільшого спільного дільника. Передбачається, що адресний простір модуля починається з адреси 0xA000. За адресою 0xA001 передається значення 100, за адресою 0xA002 передається значення 15. При читанні значення за адресою 0xA000 отримуємо результат роботи функції: 5.

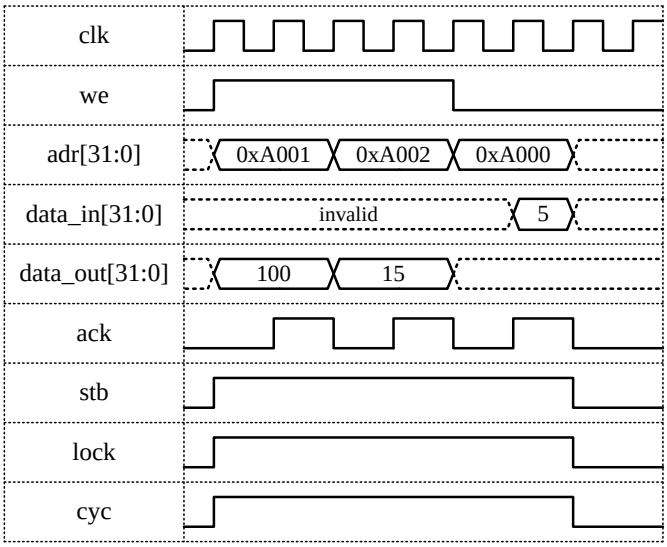


Рис. 2.13. Часова діаграма робочого циклу виклику функції

Цей робочий цикл може бути скорочений на кілька тактів роботи, якщо починати обчислювати значення функції не під час читання значення з модуля, а під час запису параметрів.

2.4 Синтез різних конструкцій мови C ++

Розглянемо синтез різних синтаксичних конструкцій мови C ++, які запропоновані в стандарті синтезу SystemC, як синтезовані. Слід зазначити, що стандарт дає вказівки, що має бути синтезовано, а що ні, але це не дає будь-яких методичних вказівок, про те яким чином повинен здійснюватися синтез, а також що має вийти в результаті.

Синтез статичних об'єктів. До статичних належать об'єкти, розмір яких відомий на момент компіляції вихідного тексту програми. Приклади статичних об'єктів: змінні, покажчики, константи, рядки, статичні масиви.

Статичні об'єкти можуть бути розміщені в елементах динамічної або статичної пам'яті (регістрах) [114, 115]. Тут слід розглянути компроміс між витраченими апаратними ресурсами і швидкодією. Для реалізації реєстрової пам'яті потрібно більше витрат апаратури, ніж для реалізації блоків динамічної пам'яті. З іншого боку, статична пам'ять працює швидше, ніж динамічна, по ряду причин. По-перше, час реакції статичної пам'яті вище, ніж час реакції динамічної. По-друге, до блоку статичної пам'яті можна організувати паралельний доступ. До блоку динамічної пам'яті можна організувати лише одночасний доступ лише до істотно обмеженій кількості осередків. Таким чином, об'єкти C ++ слід відображати в блоки статичної пам'яті, якщо потрібна реалізація алгоритмів, де буде використовуватися паралельна обробка даних.

Синтез покажчиків. Покажчик – це змінна, яка містить в собі адресу іншої змінної в пам'яті. Розмір покажчика p в бітах можна виразити через залежність від обсягу пам'яті N байт, де розташовуються об'єкти:

$$p = \lceil \log_2 N \rceil. \qquad (2.4)$$

Покажчик може бути синтезований, як індексний регістр, що містить адресу змінної, регістр даних і блок пам'яті. Операція «розіменування» може бути синтезована, як регістр даних, розмір якого, відповідає розміру об'єкта. На рис. 2.14 показана структурна модель пристрою, відповідного вказівнику C++ [116].

Тут IP – індексний регістр, РД – регістр даних. Мультиплексор може працювати в двох режимах: а) завантаження значення з ОЗУ в регістр даних; б) завантаження значення з іншого джерела. АЛУ використовується для завантаження адреси об'єкта з ОЗУ, а також для виконання арифметичних операцій з покажчиком (що вже виходить за рамки стандарту на синтез SystemC, тим самим розширюючи синтезується підмножина мови C ++) [117].

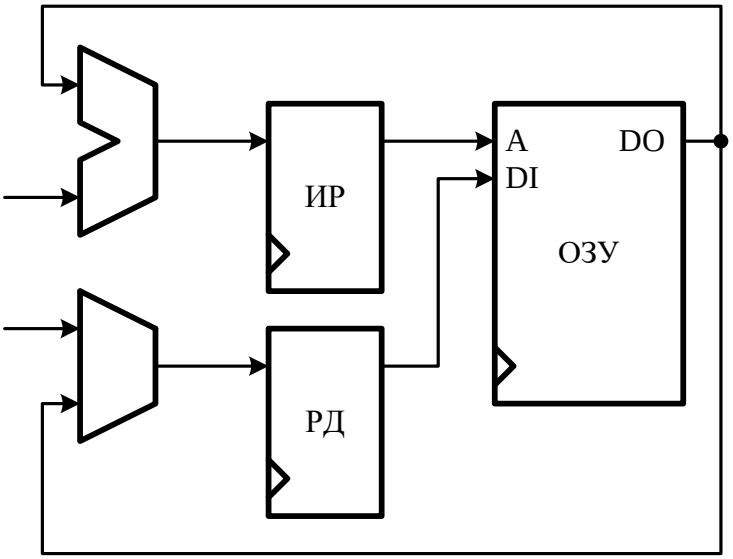


Рис. 2.14. Структурна модель, що відповідна вказівнику C ++

Синтез умовних операторів. У мові доступні три типи умовних конструкцій:

а) *condition? expr1: expr2* – тернарний оператор;

б) *if-then-else* оператор;

в) *switch-case* оператор.

Відмінність тернарного оператора від *if-then-else* оператора полягає в тому, що перший повертає вираз, а значить, результат може бути використаний в інших виразах. *If-then-else* конструкція – оператор і не може бути використана, як аргумент виразу. Відмінність *switch-case* оператора в тому, що він підтримує більше двох альтернатив. У лістингу 2.1 показаний приклад умовного оператора *if-then-else* на мові C ++.

Лістинг 2.1. Умовний оператор if-then-else на мові C ++.

```
1.   if (Condition)
2.       statement1 ();
3.   else
4.       statement2 ();
```

Цьому фрагменту коду відповідає змістовна граф-схема алгоритму, показана на рис. 2.15. Такий умовний оператор синтезується в мультиплексор,

який активує ту чи іншу мікрокоманду (набір мікрооперацій), в залежності від значення сигналу condition.

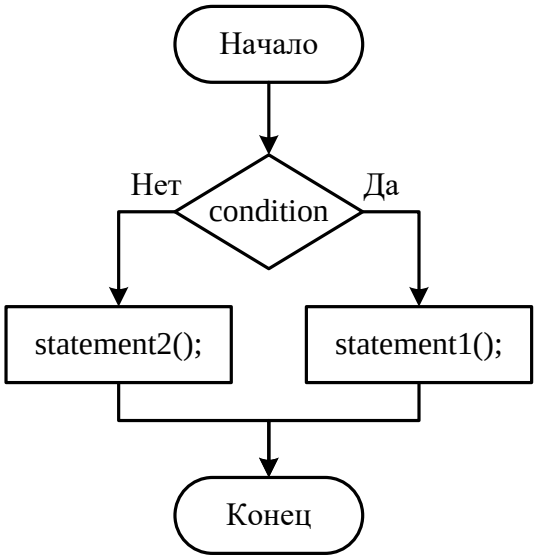
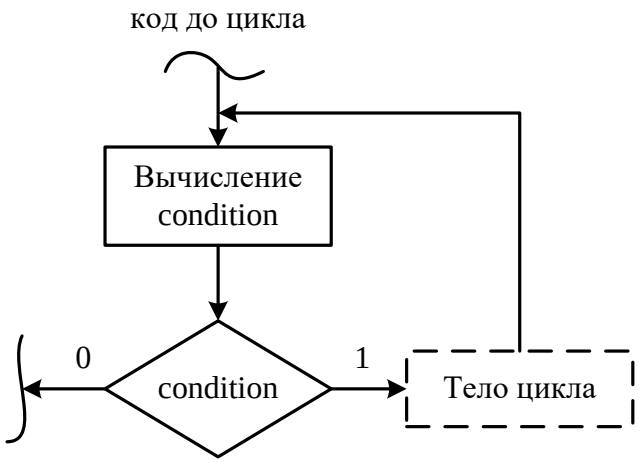


Рис. 2.15. Змістовна ГСА умовного оператора

Синтез циклів. Цикли з відомою кількістю ітерацій на момент компіляції можуть бути легко замінені паралельної структурою, яка включає в себе стільки екземплярів тіла циклу, скільки було ітерацій в оригінальному циклі [118]. Така трансформація називається повною розгорткою циклу. Але однією з проблем сучасних програм високорівневого синтезу є синтез циклів з невідомою кількістю ітерацій. Легко показати, що подібні конструкції можуть бути синтезовані з використанням прикладної теорії цифрових автоматів [119].

На рис. 2.16 показано відповідність між шаблоном коду циклу *while* на C++ і ГСА. Видно, значення *condition* буде обчислено хоча б один раз на вході в цикл. Тіло ж циклу може бути не виконано жодного разу. Такий цикл називають «цикл з умовою входу».

```
// код до цикла
while (condition)
{
    // тело цикла
}
// код после цикла
```

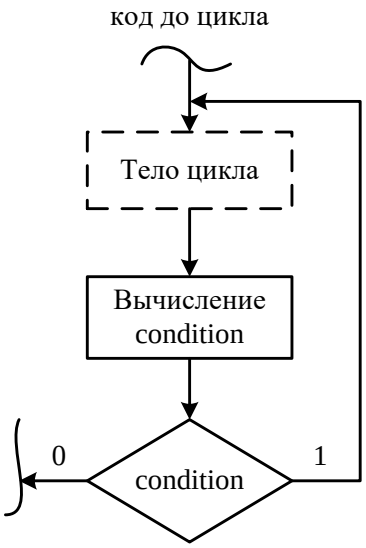


а) б)

Рис. 2.16. Шаблон циклу *while* (а) і відповідна ГСА (б)

На рис. 2.16 показано відповідність між шаблоном циклу *do-while* на мові С++ і ГСА. Показано, що в незалежності від умови *condition* тіло циклу виконується хоча б один раз. Такий цикл називають «цикл з умовою виходу».

```
// код до цикла
do
{
    // тело цикла
}
while (condition)
// код после цикла
```



а) б)

Рис. 2.17. Шаблон циклу *do-while* (а) і відповідна ГСА (б)

Розглянемо приклад програми (див. лістинг 2.2), що використовує цикл *while*. Цикл вирішує завдання приведення всіх знаків в тексті до верхнього

регістру. Програма перебирає всі символи в рядку. Умовою виходу з такого циклу є те, що черговий символ рядку дорівнює нулю.

Лістинг 2.2. C ++ програма для перекладу тексту в верхній регістр

```
 5.   int main ()
 6.   {
 7.       char str [] = "Where do you want to go today?";
 8.       char * s = str;
 9.       while (* s)
10.       {
11.           if (* s> = 'a' && * s <= 'z')
12.               * S - = 32;
13.           s ++;
14.       }
15.       return 0;
16.   }
```

Визначимо необхідні ресурси операційного автомата для виконання обчислень. Нам необхідний індексний регістр IR, який буде містити покажчик s, регістр даних DR. Перелік мікрооперацій Y, їх мнемоніки, а також опису, показані в табл. 2.4.

Таблиця 2.4. Список микроопераций

Y	Мнемоніка	Опис
y1	`MOV DR, [IR]`	Читання даних з пам'яті за адресою, що міститься в регістрі IR, в регістр DR.
y2	`MOV [IR], DR`	Запис даних з регістра DR в пам'ять за адресою, що міститься в регістрі IR.
y3	`SUB DR, 32`	Зменшити вміст регістра DR на 32.
y4	`INC IR`	Збільшити вміст регістра IR на 1.

Оскільки мова C++ є алгоритмічною мовою, який визначає послідовність дій, значить, будь-яку програму на цій мові можна представити у вигляді змістовної граф-схеми алгоритму. Змістовна ГСА для лістингу 2.4 показана на рис. 2.18.

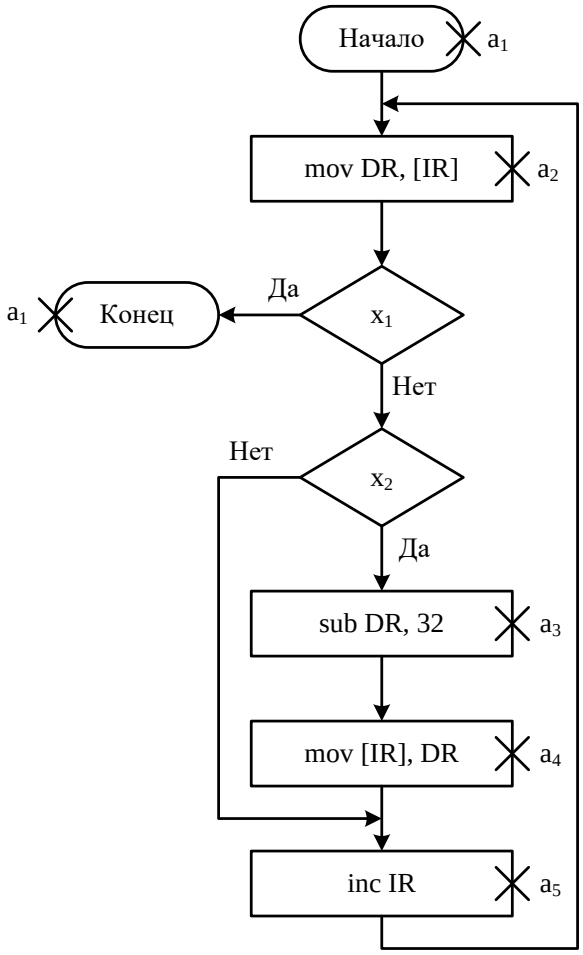


Рис. 2.18. Змістовна ГСА для лістингу 2.1

Після розмітки автомат Мура має п'ять станів: a1, a2, a3, a4, a5 (були відзначені початкова і кінцева вершини, а також всі операційні). Граф автомата Мура для ГСА на рис. 2.15 показаний на рис. 2.19. Функції умовних вершин визначені, як показано в табл. 2.5.

Таблиця 2.5. Список логічних умов

X	Логічна умова	Опис
x1	`* S == 0`	Символ рядка дорівнює нулю.
x2	`* S> = 'a' && * s <= 'z'`	Символ – мала літера латинського алфавіту.

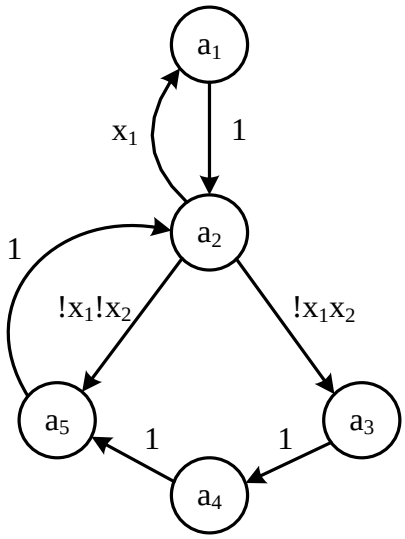


Рис. 2.19. Граф автомата Мура для ГСА рис. 2.15

Структурна таблиця мікропрограмного автомата для рис. 2.19 показана в табл. 2.7. Тут a_m – початковий стан; a_s – стан переходу; Y (a_s) – вихідний сигнал в стані as; K (a_m) – код вихідного стану; K (a_s) – код стану переходу; X (a_m, a_s) – вхідний сигнал на переході (a_m, a_s). Для зручності оператори C ++ закодовані вихідними сигналами y_1 ... y_4.

Таблиця 2.7. Структурна таблиця мікропрограмного автомата Мура

am	K (am)	as, Y (as)	K (as)	X (am, as)
a2	010	a_1, (-)	001	x1
a1 a5	001 100	a2, y1	010	1 1
a2	010	a3, y3	011	! x1x2
a3	011	a4, y2	100	1
a2 a4	010 100	a5, y4	101	! X1! X2 1

Для структурного синтезу використовується канонічне зображення автомата: блок пам'яті, який містить значення поточного стану, і комбінаційна

схема, яка виробляє значення функції збудження пам'яті автомата і значень входів Y.

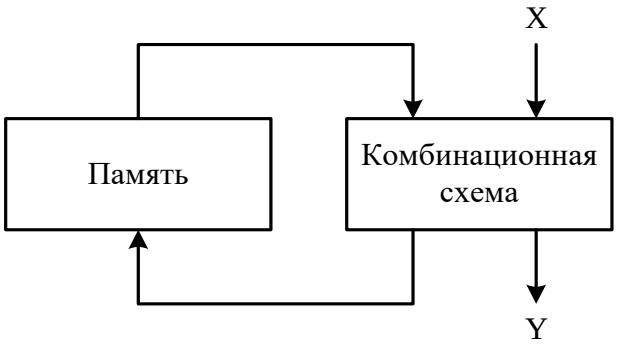


Рис. 2.20. Структура керуючого автомата

Після отримання таблиць переходів автомата і вихідних функцій необхідно згенерувати трьох-процесний шаблон автомата на мовах опису апаратури VHDL або Verilog. Такий шаблон містить в собі три процеси:

—процес, який реалізує функцію переходів (вироблення коду наступного стану);

—процес, який реалізує вихідну функцію;

—процес, який реалізує пам'ять автомата.

Така МОА-модель трьох-процесного шаблону придатна для програм логічного синтезу, наприклад: Synplicity Synplify, Xilinx ISE, Alter Quartus.

На рис. 2.21 показана об'єктна структурно-поведінкова модель цифрового пристрою. Вершина ієрархії – клас CurcuitObject, який служить абстракцією для будь-якого класу в моделі.

Клас Module представляє абстракцію поняття «модуль» і містить в собі: ім'я, список зовнішніх портів (клас Port), список структурних елементів модуля (клас StructureElement), список декларацій.

Клас StructureElement представляє абстракцію поняття «структурний елемент». Від цього класу успадковуються такі класи:

—Process – процес, який характеризується списком чутливості, а також тілом процесу (клас ProcessBody);

—DataFlow – безперервний потік даних, який характеризується призначається сигналом і функцією потоку [120];

—Instance – екземпляр модуля, який характеризується ім’ям модуля, ім’ям екземпляра, а також відображенням сигналів схеми на порти модуля.

Клас Function – абстракція поняття «функція», яка характеризується списком аргументів і типом операції (перелік арифметичних або логічних операцій).

Таким чином, розроблена модель дозволяє відобразити як структурні особливості схеми (класи Module і Instance), так і поведінкові (класи Process і DataFlow) [121].

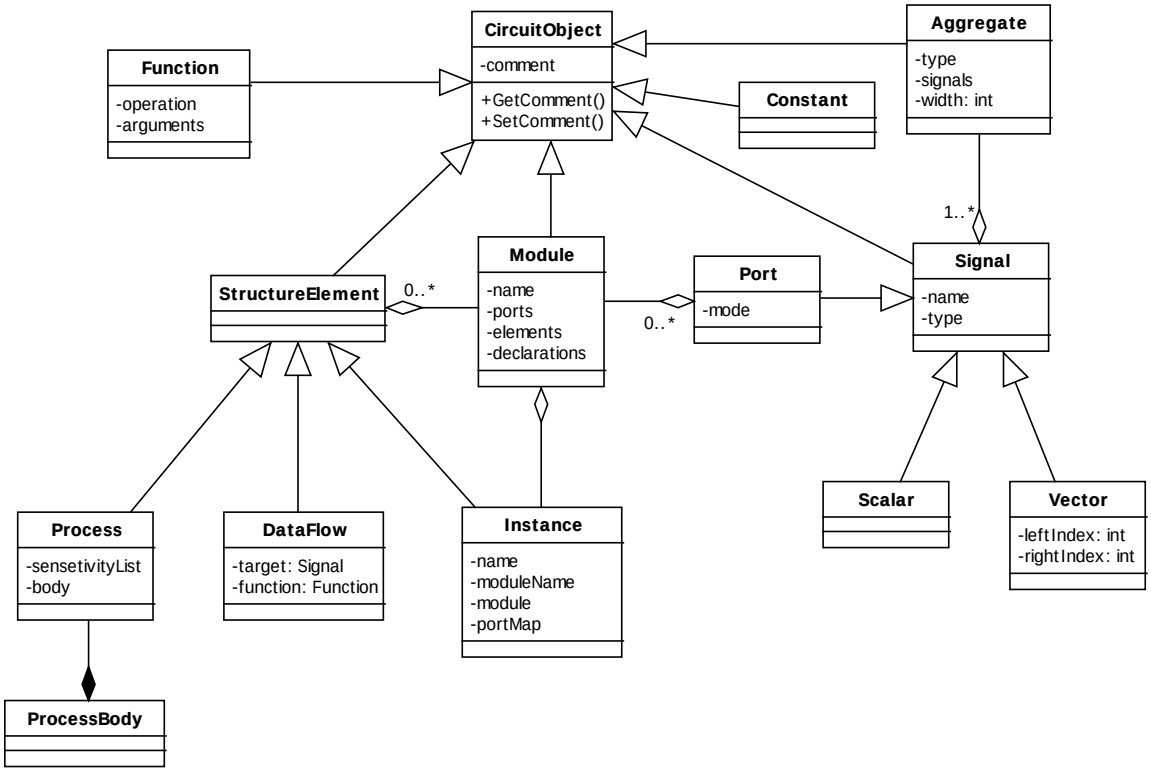


Рис. 2.21. Об’єктна структурно-поведінкова модель цифрового пристрою

2.5 Висновки до розділу 2

1. Розроблено структури даних і метод синтезу інтерфейсних структур і протоколів виконання транзакцій RT-рівня на основі аналізу специфікації SoC-архітектури системного рівня, яка використовує стандартну шину Wishbone обміну даними між функціональними модулями.

2. Розроблено структури даних і метод синтезу RTL-моделей функціональностей шляхом перетворення C++ і SystemC-описів цифрових блоків системного рівня в алгоритми і структури даних автоматної моделі Мура, що задана синтезованою підмножиною мовних конструкцій VHDL.

3. Створено інфраструктуру проектування і верифікації компонентів цифрових систем на кристалах з метою перевірки алгоритмів перетворення специфікації з системного рівня на рівень RTL-опису яка дає можливість істотно зменшити час виконання всіх процесів проектування, тестування і верифікації.

4. Представлені результати синтезу типових структур мовних конструкцій в моделі RT-рівня і подальша їх оцінка з позиції швидкодії та апаратних витрат, які використовуються як бібліотечні описи для вибору оптимальних рішень в процесі створення обчислювальних пристроїв.

5. Здійснено тестування і верифікація конструктивних рішень примітивів, використовуваних для проектування цифрових систем на кристалах на основі промислових систем моделювання Aldec Active-HDL, Riviera, Xilinx Web Pack.

РОЗДІЛ 3
МОДЕЛІ ЛОГІЧНИХ БЛОКІВ СИСТЕМНОГО РІВНЯ

В даному розділі вирішуються завдання розробки моделей і структур даних опису логічних блоків системного рівня, засновані на використанні мов опису апаратури System-C, VHDL, Verilog, які можуть бути використані як бібліотечні примітиви для створення IP-core цифрових систем на кристалах.

Мета розділу – створення квазіоптимальних синтезованих моделей функціональних модулів і їх подальше імплементація в RTL-код, використовуваних далі при синтезі функціонально-складних IP-core цифрових систем на кристалах.

Завдання:

1. Розробити структури даних для опису функціональних примітивів системного рівня, орієнтованих на використання семантичних і синтаксичних конструкцій мови C++ і SystemC з метою забезпечення паралельного синтезу і верифікації архітектурних рішень.

2. Програмна реалізація моделей і методів мультиверсної розробки операційних пристроїв в рамках інтегрованої системи проектування функціональних і архітектурних рішень SoC на основі використання продуктів верифікації та синтезу компаній Aldec та Xilinx.

Підвищуючи продуктивність інженерів, можна домогтися істотного скорочення витрат і часу на доведення проекту до використання його на ринку електронних технологій. Існує декілька способів підвищення продуктивності праці інженера-проектувальника в процесі створення обчислювальних пристроїв.

Перший спосіб [122] – використання мов опису систем високого рівня, такого як C++ і SystemC. Такі мови мають високий рівень абстракції, що дозволяє швидко описувати об’єкти різної природи. Використання прийомів об'єктно-орієнтованого проектування дозволяє створювати складні, масшта-

бовані і тиражовані системи в короткий термін з великою питомою кількістю логічних вентилів на один рядок коду. Відбувається також підвищення рівня абстракції моделей: від аналогових з безперервними значеннями напруги до цифрових моделей на вентильному рівні з подіями перемикання логічних сигналів, далі – до моделей на рівні регістрових передач, де операції вже відбуваються з цілими машинними словами. Останній доступний рівень абстракції сигналу – це транзакції запису / читання між вузлами цифрової системи на кристалі. Одна така транзакція може включати в себе роботу декількох шин і керуючих сигналів, а також десятки тактів моделювання на двійковому рівні. При цьому виграш в швидкості моделювання становить десятки разів.

Другий спосіб [123] – використання сучасних інструментальних засобів: компіляторів, генераторів тестів, формальних верифікаторів. Компілятори використовують внутрішні оптимізації для отримання кращих часових характеристик і мінімальних витрат ресурсів кристала (мається на увазі площа кристала або кількість використовуваних логічних блоків). Прикладом таких оптимізацій може служити автоматична конвейеризація, розгортки (згортки) або розпаралелювання циклів, при якому виходить більш швидка реалізація цифрової системи.

Третій спосіб [124] – використання методичного забезпечення. Розроблено сукупність шаблонів проектів – підходів до вирішення того чи іншого завдання в певному контексті умов. Розроблено сукупність стандартів проектування на арифметичні операції, інтерфейси, синтезовані підмножини мов опису апаратури.

Четвертий спосіб [125] – використання бібліотек (IP-Core) моделей, які містять готові протестовані рішення: алгоритми, структури даних високого рівня. В теперішній час бібліотеки включають як прості моделі: типи даних, арифметика, пам’ять, так і складні: аудіо / відео компоненти, сигнальні процесори, контролери, процесори загального призначення. Використовуючи бібліотечні компоненти, по-перше, скорочується час на проектування, а, по-

друге, скорочується час на верифікацію проекту, тому що бібліотечні елементи протестовані заздалегідь і неодноразово були впроваджені в промислові проекти. Таким чином, бібліотечні компоненти можуть займати до 90% площі одержуваного кристала, інші 10% – визначена користувачем логіка. На рис. 3.1 показана типова система на кристалі, де ADC – аналого-цифровий перетворювач, PCI – інтерфейс до шини PCI, SRAM – статична ОЗУ, CPU – центральний процесор, DSP – цифровий сигнальний процесор, ROM – ПЗУ, MPEG – відеокодек, DRAM – динамічна ОЗУ, UDL (user defined logic) – логіка, визначена користувачем.

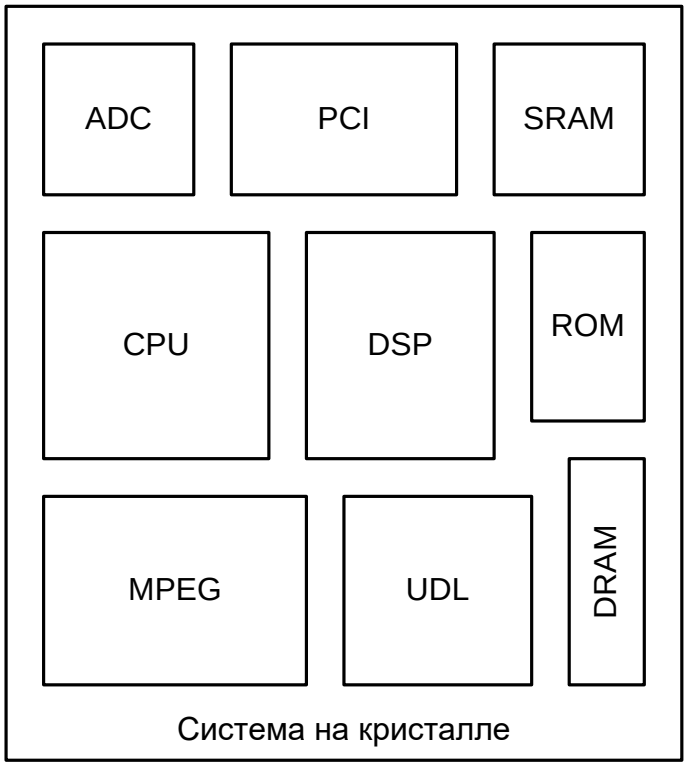


Рис. 3.1. Різноманіття компонентів в системі на кристалі

Існують різні типи компонентів, які можна класифікувати за призначенням і за реалізацією. Компоненти за призначенням:

—функціональні (Functional IP);

—інфраструктурні (Infrastructure IP) [x85];

—комунікаційні (Communication IP).

Компоненти за реалізацією:

—soft cores – представлені у вигляді синтезованих моделей на мовах VHDL, Verilog, і абстрактні від технології реалізації;

—hard cores – топології кристала, орієнтовані на конкретну архітектуру.

Функціональні компоненти в сукупності виконують необхідні завдання, пов'язані з цільовим використанням мікросхеми. Розрізняють цифрові і аналогові компоненти [126]. Широко розповсюджені цифрові компоненти:

—математичні операції: інкремент, декремент, додавання, віднімання, множення, ділення, квадратний корінь, зсув, абсолютне значення;

—аудіо / відео компоненти: кодування / декодування зображень, відео, звуку, розпізнавання мови, перетворення форматів;

—сигнальні процесори: аудіо, відео, фільтри, трансформації;

—пам'яті: реєстрові файли, кеш-пам'яті, статичні, динамічні, флеш, ПЗУ, FIFO / LIFO;

—стандартні осередки: буфери, логічні елементи, елементарні арифметичні операції;

—тактові генератори, таймери;

—контролери: арбітри шини, введення / виведення, дискові, клавіатури, пам'яті, мережеві, USB;

—мікроконтролери;

—процесори / співпроцесори;

—шифрування / дешифрування.

Інфраструктурні компоненти забезпечують всі необхідні дії, які не пов’язані з цільовим використанням мікросхеми. До таких дій належить вбудоване самотестування, самодіагностика, відновлення працездатності. Інфраструктурні компоненти є апаратною надмірністю. Але за рахунок цих компонентів можна істотно збільшити вихід придатних виробів при виготовленні і підвищити надійність при експлуатації. Існують методи реалізації інфра-

структурних компонентів на вбудованій ПЛІС [86]. Ці компоненти можуть бути виключені з конфігурації мікросхеми після всіх етапів тестування.

Комунікаційні компоненти застосовуються для організації різних топологій і протоколів мереж на кристалі [87].

Для підтримки трансляції C++ і SystemC описів в синтезований VHDL або Verilog код необхідно розробити структурні моделі системного рівня для різних типів даних і структур. Стандартна бібліотека шаблонів (STL, Standard Template Library) є невід'ємною частиною стандарту мови C++. Продуктивність інженерів буде істотно знижена, якщо вони не будуть використовувати контейнери і алгоритми зі стандартної бібліотеки шаблонів C++ [127].

Контейнер – це структура, яка об'єднує об'єкти одного типу. Стандартна бібліотека реалізує наступні контейнери: одновимірний масив (vector), двозв'язний список (list), черга елементів (queue), черга елементів з двома кінцями (deque), стек елементів (stack), асоціативний масив елементів (map), набір елементів (set), набір булевих елементів (bitset). Вставка і видалення об'єкта – це мінімальний набір операцій для будь-якого контейнера. Контейнери універсальні відносно збережуваного типу даних, який задається під час конструювання контейнера.

Контейнер «вектор» реалізує абстракцію над звичайною моделлю пам'яті, в якій розмірність шини даних є фіксованою (наприклад, 8, 16, 32 або 64 біта), зазвичай кратною 8 бітам. Адресація виробляється тільки за допомогою цілих чисел. За допомогою контейнера «вектор» можна створити одномірні масиви для довільних типів даних.

Далі пропонується аналіз, проектування і верифікація логічної моделі контейнера «вектор», яка аналогічна за поведінкою з реалізацією в стандартній бібліотеці C++.

3.1 Розробка моделі контейнера «вектор» системного рівня

Для контейнера «вектор» доступні операції, наведені в табл. 3.1 (тут і далі використовується синтаксис мови C++).

Таблиця 3.1. Операції з контейнером «вектор»

Операція	**Опис**
reference operator [] (size_type n)	Доступ до елементу масиву за індексом без перевірки виходу за його межі.
reference at (size_type n)	Доступ до елементу масиву за індексом з перевіркою виходу за межі.
reference front ()	Доступ до першого елементу.
reference back ()	Доступ до останнього елемента.
push_back (const T & x)	Додавання елемента x в кінець масиву.
pop_back ()	Видалення останнього елемента в масиві.
size_type size () const	Повертає кількість елементів в масиві.
size_type max_size () const	Повертає максимальну кількість елементів у векторі.
bool empty () const	Повертає «істину», якщо вектор порожній.

На першому етапі потрібно формалізувати інтерфейс SystemC-модуля для контейнера «вектор». На рис. 3.2 показана система, що містить всі необхідні компоненти системного рівня [128].

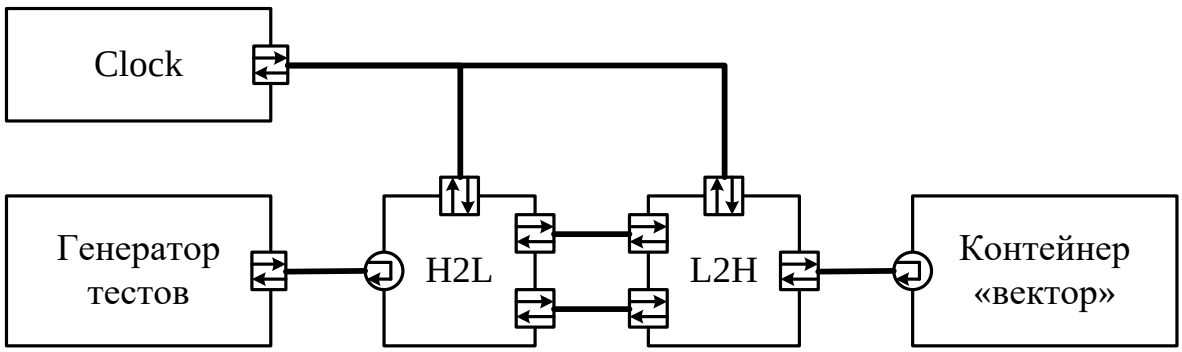


Рис. 3.2. Повний комплект модулів і транзактор для розробки системної моделі контейнера «вектор»

Clock – це генератор тактових імпульсів. Контейнер «вектор» – синхронний пристрій, у якого всі операції здійснюються за переднім фронтом сигналу синхронізації.

Генератор тестів – це високорівневий модуль, який містить генератор тестів для контейнера і аналізатор відповідних реакцій. Цей генератор буде повторно використаний для верифікації моделі на логічному рівні.

H2L (high level to low level) – це транзактор, який перетворює високорівневі виклики функцій контейнера до протоколу на рівні перемикання логічних сигналів.

L2H (low level to high level) – це транзактор, який здійснює перетворення, зворотне H2L – протокол рівня логічних сигналів до високорівневих викликам функцій контейнера «вектор». На рис. 3.3 показаний детальний інтерфейс транзактор L2H. Цей же інтерфейс матиме майбутня апаратна реалізація.

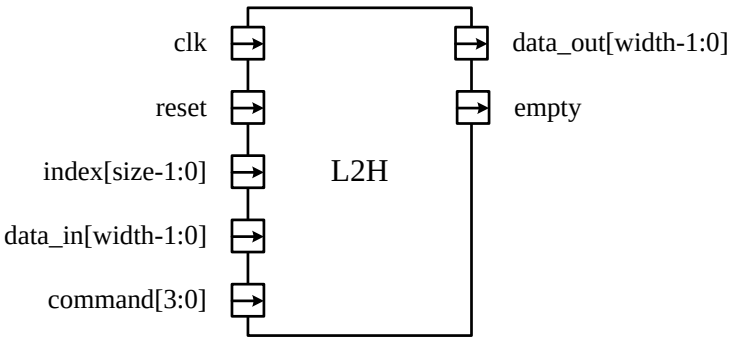


Рис. 3.3. Інтерфейс логічного рівня моделі контейнера «вектор»

У лістингу 3.1 показаний високорівневий інтерфейс контейнера «вектор».

Лістинг 3.1. Високорівнева інтерфейс контейнера «вектор»

```
17.   template <class T>
18.   class vector_if: public sc_interface {
19.   public:
20.   virtual T & operator [] (size_t n) = 0;
21.   virtual T & at (size_t n) = 0;
22.   virtual T & front () = 0;
23.   virtual T & back () = 0;
24.   virtual void push_back (const T & x) = 0;
25.   virtual void pop_back () = 0;
26.   virtual size_t size () const = 0;
27.   virtual size_t max_size () const = 0;
28.   virtual vector () = 0;
29.   virtual vector (size_t n) = 0;
30.   };
```

3.3 Проектування моделі контейнера «вектор»

Логічна модель контейнера «вектор» реалізована за допомогою примітивів блокової пам’яті кристала Xilinx Virtex-5. Один блок пам’яті кристала розрахований на 36 кбіт даних і може бути налаштований або як два незалежних модуля пам’яті по 18 кбіт, або як один модуль ємністю 36 кбіт. Можна користуватися наступними конфігураціями пам'яті: 32К по одному біту, 16К по 2, 8К по 4, 4К по 9, 2К по 18 або 1К по 36 біт.

Вибір кристалів Xilinx Virtex визначається наступними факторами. Ці кристали доступні на ринку в низьких партіях за прийнятні ціни. Тут вони істотно виграють перед кристалами жорсткої логіки, для яких тільки підготовчі витрати на виготовлення маски і пробну серію можуть досягати мільйонів доларів. ПЛІС Virtex має можливість багаторазового перепрограмування, що дозволяє відчувати різні конфігурації проектованого пристрою, а також виправляти логічні дефекти в проекті. Кристали жорсткої логіки такої можливості не мають.

Кристали Xilinx Virtex також мають необхідні ресурси для реалізації контейнера «вектор»: елементи блокової пам’яті.

Блокова пам'ять підтримує синхронні операції запису / читання, два порти доступу повністю симетричні і працюють незалежно один від одного, розділяючи дані в пам’яті. Інтерфейс моделі пам’яті показаний на рис. 3.4.

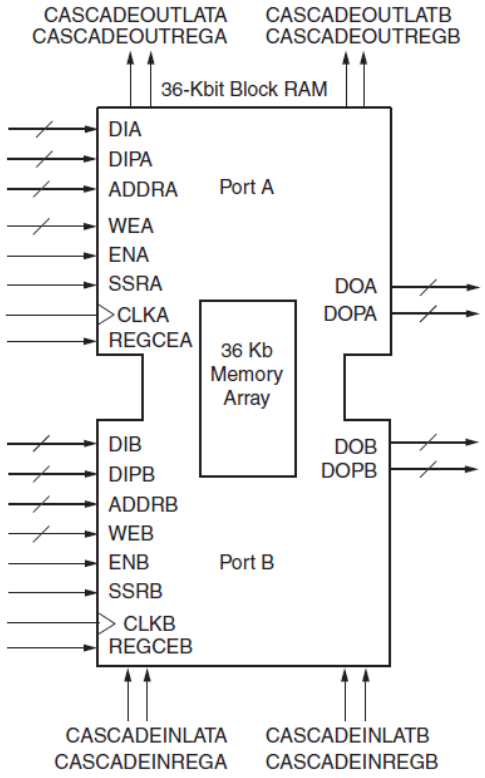


Рис. 3.4. Інтерфейс блоку пам’яті Xilinx Virtex-5

Обмеження логічної моделі:

—контейнер займає всю пам'ять, яку надає базовий елемент блокової пам’яті, використаний в реалізації;

—максимальну кількість елементів вектора визначається розміром одного елемента і обсягом елементів пам’яті, використаним в реалізації. Може бути обчислено за формулою $V = w \times k$, де w – розмір елемента в бітах, k – кількість елементів, V – обсяг займаної пам'яті.

Для успішної реалізації моделі контейнера «вектор» необхідно розробити додаткову логіку, реалізують необхідні операції, для елементів пам'яті (див. рис. 3.4).

Позначимо вектор буквою M, де M (i) – i-й елемент вектора. Змінна IP зберігає індекс першого вільного елемента. Спочатку IP = 0. При додаванні

чергового елемента в кінець вектора, значення IP збільшується на одиницю. Змінна V – записуване або зчитуване значення.

3.3.1 Проектування конструктора. Контейнер «вектор» надає повний набір конструкторів, деструкторів і операцій копіювання. Існують наступні способи створити контейнер:

—vector () створює порожній контейнер;

—vector (size_type n) створює контейнер обсягом n елементів;

—vector (size_type n, const T & t) з n копіями об'єкта t;

—vector (const vector &) – конструктор копіювання.

*3.3.2 Дешифратор команд.*Для реалізації сукупності операцій використовується дешифратор команд, який переводить з двійкового коду інструкції в позиційний код, який активує сигнали контейнера. У табл. 3.2 наведені двійкові коди і відповідні їм команди.

Таблиця 3.2. Коди команд

Код	Команда	Керуючі сигнали
0000	reference operator [] (size_type n), читання	V = M (n)
0001	reference at (size_type n), читання	V = M (n)
0010	reference front ()	V = M (0)
0011	reference back ()	V = M (IP)
0100	push_back (const T & x)	M (IP) = V; IP ++
0101	pop_back ()	$IP = IP - 1$
0110	size_type size () const	
0111	size_type max_size () const	
1000	vector ()	
1001	vector (size_type n)	$IP = n - 1$
1010	vector (size_type n, const T & t)	$\forall M(i) = t, i = 0..n - 1$
1011	reference operator [] (size_type n), запис	M (n) = V
1100	reference at (size_type n), запис	M (n) = V

Інтерфейс логічної моделі показаний на рис. 3.3. Модель має наступні параметри:

— DATA_WIDTH визначає ширину шини даних. Цей параметр залежить від того, для якого типу даних створений контейнер «вектор». Наприклад, якщо контейнер створений для типу *int* мови C++, то ширина шини даних буде дорівнює 32 бітам;

— CMD_WIDTH визначає ширину шини команд. Цей параметр вибирається залежно від максимальної кількості доступних команд, число яких дорівнює $2^{\text{CMD\_WIDHT}}$;

— SIZE визначає ширину шини адреси. Цей параметр впливає на максимальну кількість адресованих комірок контейнера і дорівнює 2^{SIZE}.

Логічна модель контейнера «вектор» має інтерфейсні сигнали:

— глобальний синхроімпульс *clk*;

— скидання *reset*, при високому рівні якого відбувається скидання елементів пам’яті в нулі, що відповідає очищенню контейнера;

— шина адреси index, розмірністю [SIZE-1: 0], яка використовується для вказівки порядкового номера елемента у векторі під час запису або читанні;

— шина даних *data_in*, розмірністю [DATA_WIDTH-1: 0], яка використовується для запису даних в вектор;

— шина команд *command*, розмірністю [CMD_WIDTH-1: 0], яка використовується для управління контейнером;

— шина даних *data_out*, розмірністю [DATA_WIDTH-1: 0], яка використовується для читання даних з вектора;

— сигнал *empty*, високий рівень якого сигналізує про те, що вектор порожній.

Лістинг. 3.2. Інтерфейс логічної моделі IP-core мовою VHDL

```
31.   entity stl_vector is
32.   generic(
33.   DATA_WIDTH: NATURAL: = 32;
34.
35.   - Ширина шини комманди
36.   CMD_WIDTH: NATURAL: = 4;
37.
```

```
38.   - Ширина шини адреси
39.   SIZE: NATURAL: = 12);
40.   port (
41.       - inputs
42.   clk: in STD_LOGIC;
43.   reset: in STD_LOGIC;
44.   index: in STD_LOGIC_VECTOR (SIZE - 1 downto 0);
45.   data_in: in STD_LOGIC_VECTOR (DATA_WIDTH - 1 downto 0);
46.   command: in STD_LOGIC_VECTOR (CMD_WIDTH - 1 downto 0);
47.
48.   - outputs
49.   data_out: out STD_LOGIC_VECTOR (DATA_WIDTH - 1 downto 0);
50.   empty: out STD_LOGIC);
51.
52.   end entity stl_vector;
```

Структурна модель контейнера «вектор» показана на рис. 3.5. Тут дешифратор команд *dc* перетворює двійковий код в керуючі сигнали.

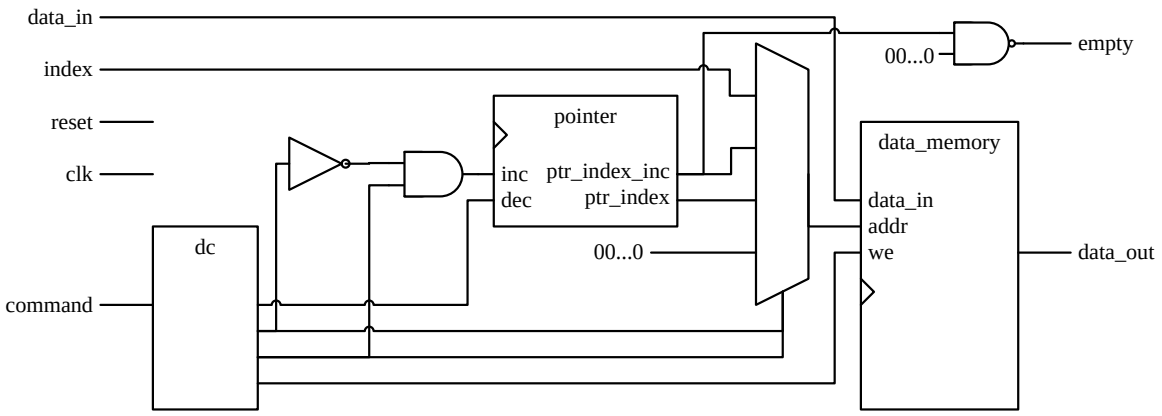


Рис. 3.5. Структурна модель контейнера «вектор»

Блок *pointer* містить два регістри: адреса останнього елемента (для порожнього вектора він дорівнює -1) і адреса першого вільного елемента (для порожнього вектора дорівнює нулю). На вході блоку є три керуючих сигнали: *reset, inc, dec*. При високому рівні сигналу *inc* і передньому фронті синхроимпульса внутрішні регістри збільшуються на одиницю. При високому рівні сигналу *dec* і передньому фронті синхроімпульсу внутрішні регістри зменшуються на одиницю.

Мультиплексор індексу управляється шиною з двох сигналів. У режимі читання / запису осередку за індексом сигнали повинні бути встановлені в 00, в режимі *push_back ()* сигнали повинні бути встановлені в 01, в режимі *back ()*

сигнали повинні бути встановлені в 10, в режимі *front ()* сигнали повинні бути встановлені в 11 .

3.3 Аналіз та проектування контейнера «список»

Перелік – це множина елементів $A = \{a_1, a_2, \dots, a_i, \dots, a_n\}$, кожен з яких може бути представлений трійкою $\langle V, N, P \rangle$, де V – значення елемента списку, $N \in A$ – посилання на наступний елемент, $P \in A$ – посилання на попередній елемент. Для першого елемента (a1) значення P дорівнює нулю, а для останнього елемента (an) значення N дорівнює нулю. Структурна організація контейнера «список» показана на рис. 3.6.

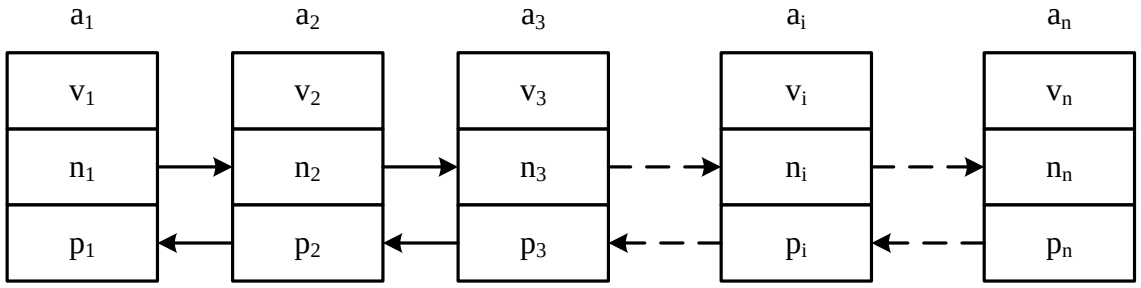


Рис. 3.6. Структурна організація контейнера «список»

Розглянемо типові операції, що виконуються контейнером «вектор» (табл. 3.3). У цій таблиці X – це умова, що збуджує ту чи іншу операцію.

Таблиця 3.3. Типові команди контейнера «список»

X	Команда	Опис команди
x1	begin ()	Посилання на перший елемент списку
x2	end ()	Посилання на останній елемент списку
-	bool empty ()	повертає true, якщо контактів немає
x3	push_front ()	Додає новий елемент в початок
x4	push_back ()	Додає новий елемент в кінець
x5	pop_front ()	Видаляє елемент з початку списку

x6	pop_back ()	Видаляє елемент з кінця списку
x7	insert ()	Додає елемент перед поточним
x8	erase ()	Видаляє поточний елемент
x9	next ()	Посилання на наступний елемент
x10	prev ()	Посилання на попередній елемент

Для успішної реалізації моделі контейнера «список» необхідно розробити додаткову логіку, що реалізує необхідні операції. Складемо перелік регістрів, необхідних для роботи контейнера.

Таблиця 3.4. Список регістрів контейнера «список»

Мнемоніка	Мета регістру
НАЧ	Індексний регістр, містить адресу першого елемента в списку
КОН	Індексний регістр, містить адресу останнього елемента в списку
ТЕК	Індексний регістр, містить адресу поточного елемента в списку
СВБД	Індексний регістр, містить адресу першого вільного елемента
РД1	Регістр даних, який містить один елемент списку ai
РД2	Регістр даних, який містить один елемент списку ai
РД1.ЗНАЧ	Доступ до значення vi елемента списку ai (частковий доступ до регістру РД1)
РД1.СЛЕД	Доступ до значення ni елемента списку ai (частковий доступ до регістру РД1)
РД1.ПРЕД	Доступ до значення pi елемента списку ai (частковий доступ до регістру РД1)
РД2.ЗНАЧ	Доступ до значення vi елемента списку ai (частковий доступ до регістру РД2)
РД2.СЛЕД	Доступ до значення ni елемента списку ai (частковий доступ

	до регістру РД2)
РД2.ПРЕД	Доступ до значення рі елемента списку аі (частковий доступ до регістру РД2)

На підставі типових операцій, перерахованих в табл. 3.3, наведемо список мікрооперацій в табл. 3.5. Мікрооперація – елементарна, неподільна дія цифрового пристрою, наприклад, читання з пам'яті, арифметична операція.

Таблиця 3.5. Мікрооперації контейнера «список»

Y	Мнемоніка	Опис
y1	ЧТ РД1, [НАЧ]	Читання першого елемента в списку за адресою, що міститься в регістрі НАЧ, в регістр даних РД1
y2	ЧТ РД1, [КОН]	Читання останнього елемента в списку за адресою, що міститься в регістрі НАЧ, в регістр даних РД1
y3	ЗП IP1, РД1.СЛЕД	Записати в індексний регістр IP1 значення регістра РД1.СЛЕД
y4	ЧТ РД1, [РД1.СЛЕД]	Читання наступного елемента в списку
y5	ЧТ РД1, [РД1.ПРЕД]	Читання попереднього елемента в списку
y6	УВ СВБД	Збільшити значення індексного регістра СВБД на одиницю
y7	ЗП [СВБД], РД1	Записати вміст регістра РД1 в пам'ять за адресою, що міститься в індексному регістрі СВБД
y8	ЗП КОН, СВБД	Записати значення індексного регістра СВБД в індексний регістр КОН
y9	ЗП НАЧ, СВБД	Записати значення індексного регістра СВБД в індексний регістр НАЧ
y10	ЗП РД1.ПРЕД, ТЕК	Записати значення індексного регістра ТЕК в регістр РД1.ПРЕД
y11	ЗП РД1.СЛЕД,	Записати значення індексного регістра ТЕК в

	ТЕК	регістр РД1.СЛЕД
у12	ЗП РД1.ПРЕД, СВБД	Записати значення індексного регістра СВБД в регістр РД1.ПРЕД
у13	ЗП РД1.СЛЕД, СВБД	Записати значення індексного регістра СВБД в регістр РД1.СЛЕД
у14	ЗП КОН, ТЕК	Записати значення індексного регістра ТЕК в індексний регістр КОН
у15	ЗП НАЧ, ТЕК	Записати значення індексного регістра ТЕК в індексний регістр НАЧ
у16	ЗП РД1.ЗНАЧ	Записати значення з шини даних пристрою в регістр РД1.ЗНАЧ
у17	ЗП РД1.СЛЕД, 0	Обнулити значення регістра РД1.СЛЕД
у18	ЗП РД1.ПРЕД, 0	Обнулити значення регістра РД1.ПРЕД
у19	ЗП [ТЕК], РД1	Записати значення регістра РД1 в пам'ять за адресою з індексного регістра ТЕК
у20	ЗП РД2, РД1	Записати значення регістра РД1 в регістр РД2
у21	ЗП РД1.ПРЕД, РД2.ПРЕД	Записати значення регістра РД2.ПРЕД в регістр РД1.ПРЕД
у22	ЗП РД1.СЛЕД, РД2.СЛЕД	Записати значення регістра РД2.СЛЕД в регістр РД1.СЛЕД
у23	ЗП РД2.СЛЕД, ТЕК	Записати значення індексного регістра ТЕК в регістр РД2.СЛЕД
у24	ЧТ РД1, [РД2.ПРЕД]	Читання елемента з пам'яті за адресою, що міститься в регістрі РД2.ПРЕД, в регістр РД1
у25	ЗП РД2.ПРЕД, ТЕК	Записати значення індексного регістра ТЕК в регістр РД2.ПРЕД
у26	ЧТ РД2.ЗНАЧ	Прочитати значення з шини даних пристрою в регістр РД1.ЗНАЧ
у27	ЗП [СВБД], РД2	Записати значення регістра РД2 в пам'ять за адресою, що міститься в індексному регістрі СВБД

Наприклад, операція *push_back ()* – запис елемента в кінець списку складається з микрооперацій, показаних в лістингу 3.5.

Лістинг 3.5. Мікрооперації для операції push_back ()

```
53.   ЧТ РД1, [КОН]; прочитати останній елемент
54.   ЗП КОН, СВБД; змінити покажчик на останній елемент
55.   ЗП РД1.ПРЕД, ТЕК; змінити покажчик на попередній елемент
56.   ЗП РД1.ЗНАЧ; записати значення елемента зі входу
57.   ЗП [СВБД], РД1; записати елемент за вільномю адресою
58.   УВ СВБД; збільшити покажчик на вільний елемент
```

Операція *pop_back ()* – видалення останнього елемента зі списку складається з мікрооперацій, показаних в лістингу 3.6.

Лістинг 3.6. Микрооперации для операції pop_back ()

```
59.   ЧТ РД1, [КОН]; прочитати останній елемент
60.   ЧТ РД1, [РД1.ПРЕД]; прочитати попередній елемент
61.   ЗП РД1.СЛЕД, 0; змінити покажчик на наступний елемент
62.   ЗП КОН, ТЕК; змінити покажчик на останній елемент
63.   ЗП [ТЕК], РД1; оновити поточний елемент в пам'яті
```

Операція *push_front ()* – запис елемента в початок списку складається з микрооперацій, показаних в лістингу 3.7.

Лістинг 3.7. Микрооперации для операції push_front ()

```
64.   ЧТ РД1, [НАЧ]; прочитати перший елемент
65.   ЗП НАЧ, СВБД; змінити покажчик на перший елемент
66.   ЗП РД1.СЛЕД, ТЕК; змінити покажчик на наступний елемент
67.   ЗП РД1.ЗНАЧ; записати значення елемента зі входу
68.   ЗП [СВБД], РД1; записати елемент за вільною адресою
69.   УВ СВБД; збільшити покажчик на вільний елемент
```

Операція *pop_front ()* – видалення елемента з початку списку складається з мікрооперацій, показаних в лістингу 3.8.

Лістинг 3.8. Микрооперации для операції pop_front ()

```
70.   ЧТ РД1, [НАЧ]; прочитати перший елемент
71.   ЧТ РД1, [РД1.СЛЕД]; прочитати наступний елемент
72.   ЗП РД1.ПРЕД, 0; змінити покажчик на попередній елемент
73.   ЗП НАЧ, ТЕК; змінити покажчик на перший елеметов
74.   ЗП [ТЕК], РД1; оновити поточний елемент в пам'яті
```

Операція *erase ()* – видалення поточного елемента зі списку складається з мікрооперацій, показаних в лістингу 3.9.

Лістинг 3.9. Микрооперации для операції erase ()

```
75.   ЗП РД2, РД1; копіювати поточний елемент
76.   ЧТ РД1, [РД1.СЛЕД]; читати наступний елемент
77.   ЗП РД1.ПРЕД, РД2.ПРЕД; змінити покажчик на попередній елемент
78.   ЗП [ТЕК], РД1; оновити поточний елемент в пам'яті
79.   ЧТ РД1, [РД1.ПРЕД]; прочитати попередній елемент
80.   ЗП РД1.СЛЕД, РД2.СЛЕД; змінити покажчик на наступний елемент
81.   ЗП [ТЕК], РД1; оновити елемент в пам'яті
```

Операція *insert ()* – вставка елемента списку перед поточним складається з микрооперацій, показаних в лістингу 3.10.

Лістинг 3.10. Мікрооперації для операції insert ()

```
82.   ЗП РД2, РД1; копіювати поточний елемент
83.   ЗП РД1.ПРЕД, СВБД; змінити покажчик на попередній елемент
84.   ЗП [ТЕК], РД1; оновити поточний елемент
85.   ЗП РД2.СЛЕД, ТЕК; виправити покажчик на наступний елемент
86.   ЧТ РД1, [РД2.ПРЕД]; прочитати попередній елемент
87.   ЗП РД1.СЛЕД, СВБД; змінити указатель на наступний елемент
88.   ЗП [ТЕК], РД1; оновити поточний елемент
89.   ЗП РД2.ПРЕД, ТЕК; виправити покажчик на пердидущій елемент
90.   ЧТ РД2.ЗНАЧ; прочитати значення
91.   ЗП [СВБД], РД2; записати новий елемент в пам'ять
92.   УВ СВБД; збільшити покажчик на вільний елемент
```

Маючи список всіх мікрооперацій і зовнішніх змінних, можна побудувати змістовну граф-схему алгоритму для контейнера «список» (див. рис. 3.7). У цій ГСА показані тільки умови і вершини з операціями.

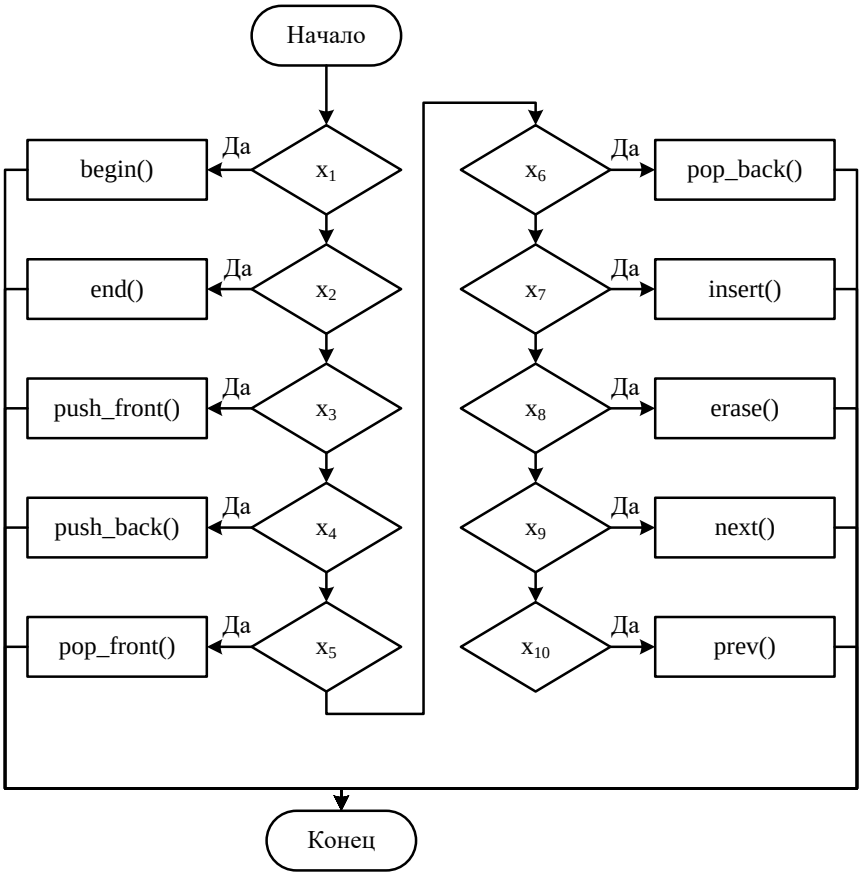


Рис. 3.7. Змістовна ГСА для контейнера «список»

3.4 Верифікація логічних моделей

Для верифікації отриманих моделей були використані тести, що перевіряють 100% режимів роботи пристрою. Вихідні еталони були отримані з програми на C++.

Визначення 3.1. *Верифікація* – процес аналізу системи або компонентів для визначення коректності перетворень вхідного опису на черговій стадії проектування [88].

Визначення 3.2. *Валідація* – процес визначення працездатності системи і її компонентів шляхом перевірки відповідності основним вимогам специфікації після виконання кожної стадії проектування. Валідація може застосовуватися щодо вихідної специфікації і кінцевого продукту проектування.

Визначення 3.3. Сертифікація – письмова гарантія того, що система або її компоненти повністю задовольняють вимогам специфікації і прийнятні для використання за призначенням.

Існує кілька типів тестів: функціональні, навантажувальні, діагностичні, стресові, тести на безпеку. У цьому розділі розглядаються тільки функціональні тести – перевірка правильності функціонування моделі, і навантажувальні тести – виміри продуктивності і часових характеристик моделі.

На рис. 3.8 показана схема процесу верифікації логічної моделі.

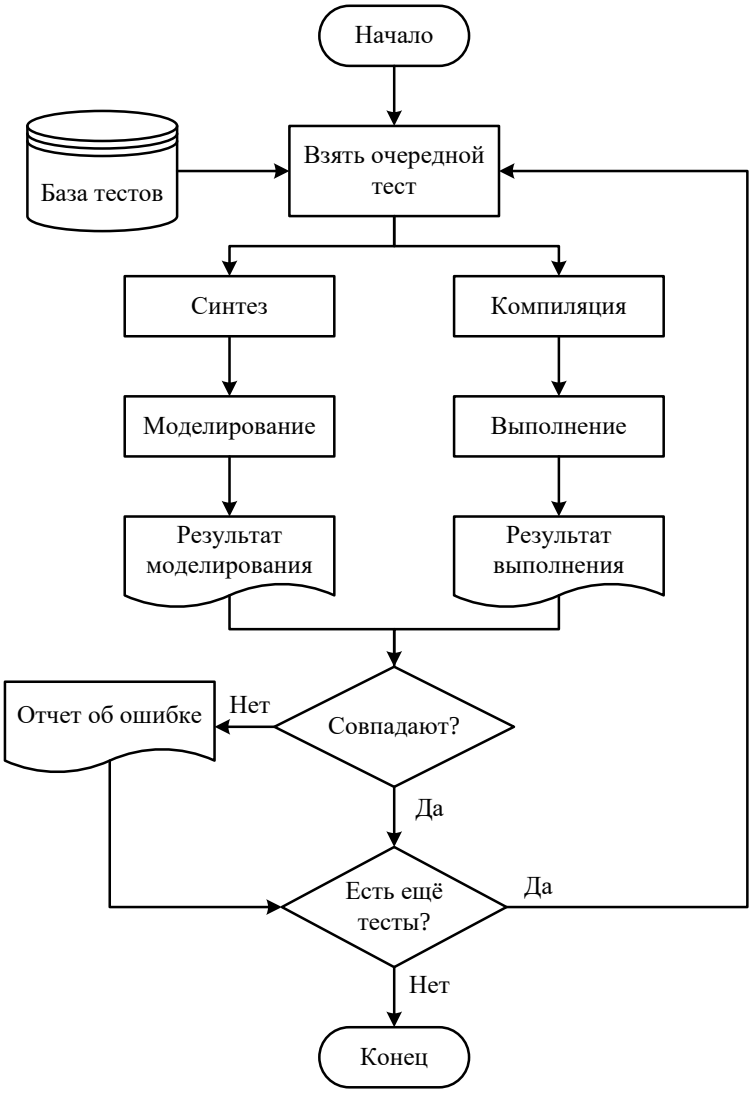


Рис. 3.8. Схема верифікації логічної моделі

Створюється набір тестів на C++, в яких використовуються контейнер «вектор». У цих тестах присутні виклики всіх методів, які підтримує контейнер. На наступному етапі тести компілюються звичайним C++ компілятором для отримання виконуваного файлу. При виконанні цієї програми в файл зберігаються результати роботи контейнера.

З іншого боку, відбувається синтез вихідних текстів. В результаті синтезу виходить модель системного рівня, що містить контейнер «вектор», описаний на MOA VHDL. Далі здійснюється моделювання за допомогою програми Aldec Active-HDL для отримання результатів. Ці результати порівнюються з результатами, отриманими в результаті роботи звичайної програми. Логічна модель складена вірно, якщо результати моделювання і виконання збігаються.

Розглянемо простий приклад тесту (див. лістинг 3.10). У прикладі в контейнер послідовно завантажуються значення 1, 2, 3, 3, 4, 5, потім, послідовно будуть видалені знизу, після кожного видалення зчитується останній елемент у векторі. Таким чином, у висновку повинні відобразитися елементи: 4, 3, 3, 2, 1.

Лістінг 3.10. Приклад використання контейнера «вектор»

```
 93.   int main ()
 94.   {
 95.   std :: vector <int> vec;
 96.   vec.push_back (1);
 97.   vec.push_back (2);
 98.   vec.push_back (3);
 99.   vec.push_back (3);
100.   vec.push_back (4);
101.   vec.push_back (5);
102.   vec.pop_back ();
103.   std :: cout << vec.back ();
104.   vec.pop_back ();
105.   std :: cout << vec.back ();
106.   vec.pop_back ();
107.   std :: cout << vec.back ();
108.   vec.pop_back ();
109.   std :: cout << vec.back ();
110.   return 0;
111.   }
```

Наведемо приклади різних режимів роботи контейнера «вектор». На рис. 3.9 показана ініціалізація контейнера на позначці 0-100 нс. Далі показана робота команди *push_back ()* – послідовно в контейнер завантажуються числа 0, 1, 2, 3, 3, 4, 5.

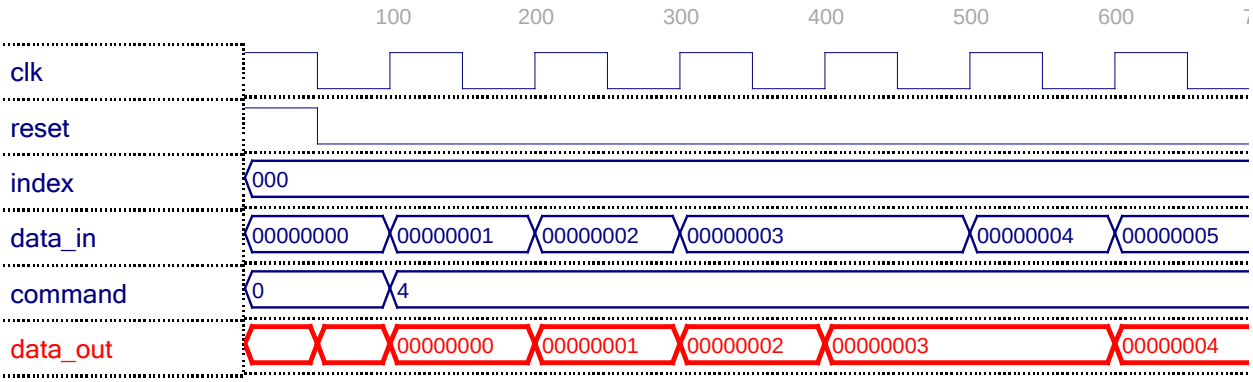


Рис 3.9. Скидання і завантаження контейнера

На рис. 3.10 показана робота команди *back ()* (код команди 3), яка показує значення останнього елемента у векторі, а також команди *pop_back ()* (код команди 5), яка видаляє останній елемент у векторі. На виході можна вважати елементи: 5, 4, 3, 3.

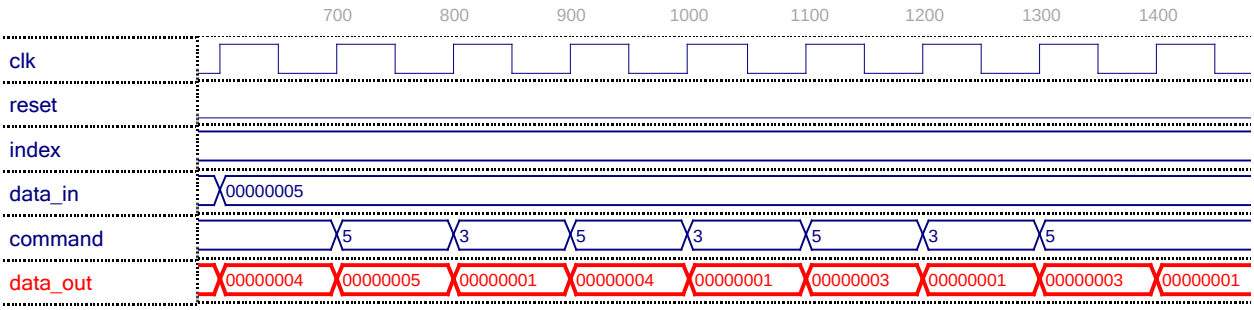


Рис. 3.10. Робота команд *back ()* і *pop_back ()*

На рис. 3.11 показана робота команд запису і читання даних за індексом (коди команд 11 і 0 відповідно). Спочатку відбувається запис значень 10, 11, 12, 13, 14 за адресами 1, 2, 3, 4, 5, а потім читання за адресами 1, 2, 3, 4, 5.

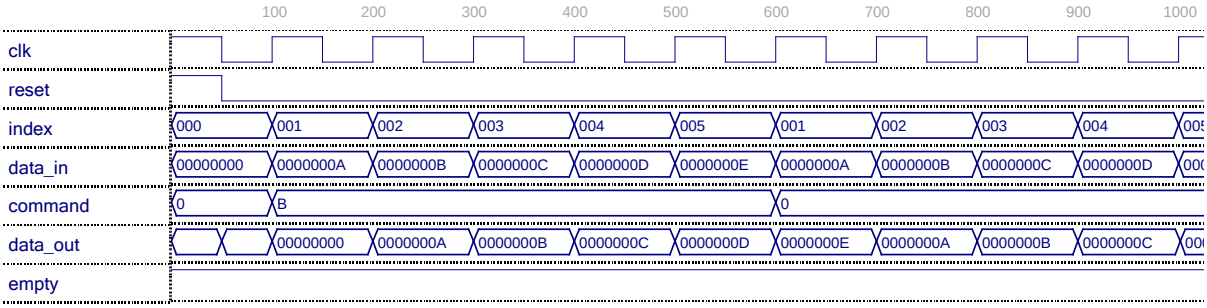


Рис. 3.11. Робота команд запису і читання за індексом

Наступним етапом верифікації є перевірка відповідності функціональної моделі і моделі з часовими затримками. В функціональній моделі відсутня інформація про час поширення сигналу від входів елементів до виходів. Мається на увазі, що сигнал поширюється миттєво або за один дельта-цикл моделювання [89]. Програма логічного синтезу, трасування і розміщення генерує спеціальний файл затримок поширення сигналу для обраної архітектури. Цей файл може бути завантажений в САПР Aldec Active-HDL для побудови часової моделі.

Останнім етапом тестування є перевірка моделі, реалізованої в тому чи іншому кристалі. В роботі обрана архітектура ПЛІС Xlinix – дешева і високопродуктивна мікросхема з можливістю багаторазового перепрограмування. Таким чином, весь цикл перевірки та затвердження логічної моделі показаний на рис. 3.12.

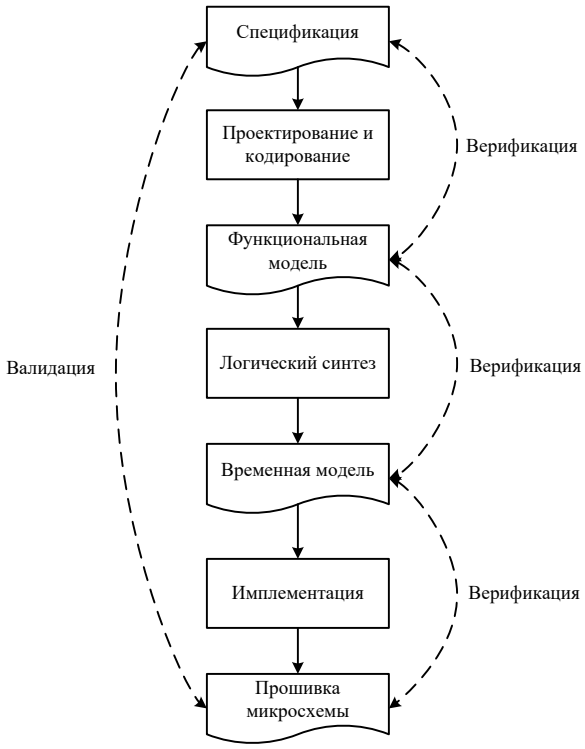


Рис. 3.12. Відповідність етапів перевірки та затвердження

3.5 Висновки

Таким чином, в розділі розглянуті та вирішені питання створення квазіоптимальних синтезованих моделей функціональних модулів в формі векторних і облікових контейнерів і їх подальша імплементація в RTL-код, використовуваних далі при синтезі функціонально-складних IP-core цифрових систем на кристалах.

При цьому вирішені такі завдання:

1. Розроблено і апробовано структури даних для опису функціональних примітивів системного рівня, орієнтованих на використання семантичних і синтаксичних конструкцій мови C++ і SystemC з метою забезпечення паралельного синтезу і верифікації архітектурних рішень.

2. Розроблено і протестовано програмні модулі, що реалізують векторні і спискові моделі опису функціональних примітивів, а також інфраструктура для реалізації методів мультіверсної розробки операційних пристроїв в рамках інтегрованої системи проектування функціональних і архітектурних

рішень SoC на основі використання продуктів верифікації та синтезу компаній Aldec і Xilinx.

3. Проведено верифікація програмно-структурних рішень, що виникають в процесі синтезу функціональних примітивів на основі використання тестових послідовностей, взятих з відкритих бібліотек ISCAS-89, opencores.com.

РОЗДІЛ 4
СИСТЕМА АВТОМАТИЗОВАНОГО ПРОЕКТУВАННЯ

У розділі представлені програмно-апаратна реалізація моделей, методів і структур даних для проектування цифрових систем на кристалах, яке включає процедури створення специфікації, синтезу, тестування, моделювання та верифікації на основі запропонованої інфраструктури, що містить промислові засоби компаній Aldec і Xilinx. Розглянуті також питання тестування програмних продуктів на реальних цифрових проектах створення IP-Core як примітивів для реалізації цифрових систем на кристалах.

Мета – розробка і тестування інфраструктури проектування цифрових систем на кристалах, яка характеризується паралельним виконанням мультиверсного синтезу функціональності, що забезпечує істотне зменшення часу створення проекту в умовах обмеження на апаратні витрати.

Завдання:

1. Розробка методу мультиверсного синтезу керуючих і операційних автоматів в заданій інфраструктурі проектування, орієнтованих на архітектурні рішення в метриці, що мінімізує час виконання функціональності за рахунок розпаралелювання операцій при обмеженні на апаратні витрати.

2. Програмна реалізація моделей і методів мультиверсної розробки операційних пристроїв в рамках інтегрованої системи проектування функціональних і архітектурних рішень SoC на основі використання продуктів верифікації та синтезу компаній Aldec і Xilinx.

3. Тестування і верифікація програмних модулів інфрастурктури проектування цифрових систем на кристалах, а також визначення ефективності запропонованих моделей, методів і структур даних при створенні реальних компонентів цифрових виробів.

4.1 Організація системи автоматизованого проектування

За своєю суттю програма синтезу C++ в VHDL є машиною по трансформації вихідного опису в результуючий. Таких трансформацій відбувається кілька [129]. Вхідна модель представлена на мові C++. Це алгоритмічний опис, який вирішує поставлене перед інженером завдання.

Перший етап перетворення – *синтаксичний аналіз*. На цьому етапі у вхідному описі з потоку символів виділяються лексичні елементи: ключові слова, оператори і лексеми. В результаті цього етапу виходить синтаксична модель вихідного алгоритмічного опису.

Другий етап перетворення – це *трансформації на рівні синтаксичної моделі*. Вони застосовуються для отримання моделей з меншою кількістю станів і займають менше апаратних ресурсів. До таких трансформацій відносяться: обчислення констант, видалення недосяжного коду, вбудовування функцій, операції над циклами (розгортки, згортки, розпаралелювання).

Третій етап перетворень – *побудова граф-схеми алгоритму*. У цій моделі алгоритм представлений вигляді відносин вершин двох типів: операцій (арифметичних, логічних або введення / виводу) і розгалужень.

Четвертий етап перетворень – *побудова автоматної моделі*. Вона представлена у вигляді вершин з операціями і переходів між ними.

П'ятий етап перетворень – це *синтез операційного і керуючого автоматів*. В результаті цього етапу виходить структурна модель, визначена на множині логічних елементів, регістрів, арифметично-логічних пристроїв і елементів пам'яті.

Шостий етап перетворень – це *збереження моделі операційного і керуючого автомата в VHDL-код*. Результатом цього етапу є набір вихідних файлів на мові VHDL, які описують пристрій на рівні регістрових передач. Ця VHDL-модель реалізує вихідний алгоритм, спочатку складений на мові C++.

4.2 Тестування системи

Тестування системи складається з декількох етапів:

— розробка схеми тестування компілятора;

— підготовка тестів;

— підготовка еталонів;

— виконання тестів і аналіз результатів.

Були підготовлені тести для різних частин компілятора: синтаксичний аналізатор; будівник граф-схеми алгоритму; будівник цифрового автомата; будівник VHDL-моделі синтезованого пристрою.

При підготовці тестів до синтаксичного аналізатору враховувалися наступні особливості компілятора:

— робота препроцесора;

— підтримка базових типів мови C++;

— підтримка базових конструкцій мови C++.

Тест для компілятора – це вихідний текст на мові C++. Примітивний тест показаний в лист. 4.1. Наприклад, цей тест перевіряє успішну декларацію і ініціалізацію об’єктів типу *int*, декларацію функції, арифметичну операцію «складання» [130].

Лістинг 4.1. Примітивний тест для компілятора

```
112.   int f1 ()
113.   {
114.       int a = 2, b = 3;
115.       return a + b;
116.   }
```

При виконанні тесту зберігається повернене значення функції в файл для подальшого порівняння з еталоном. Еталони були підготовлені з використанням компілятора Microsoft Visual Studio 2008. Було припущено, що цей компілятор не містить помилок.

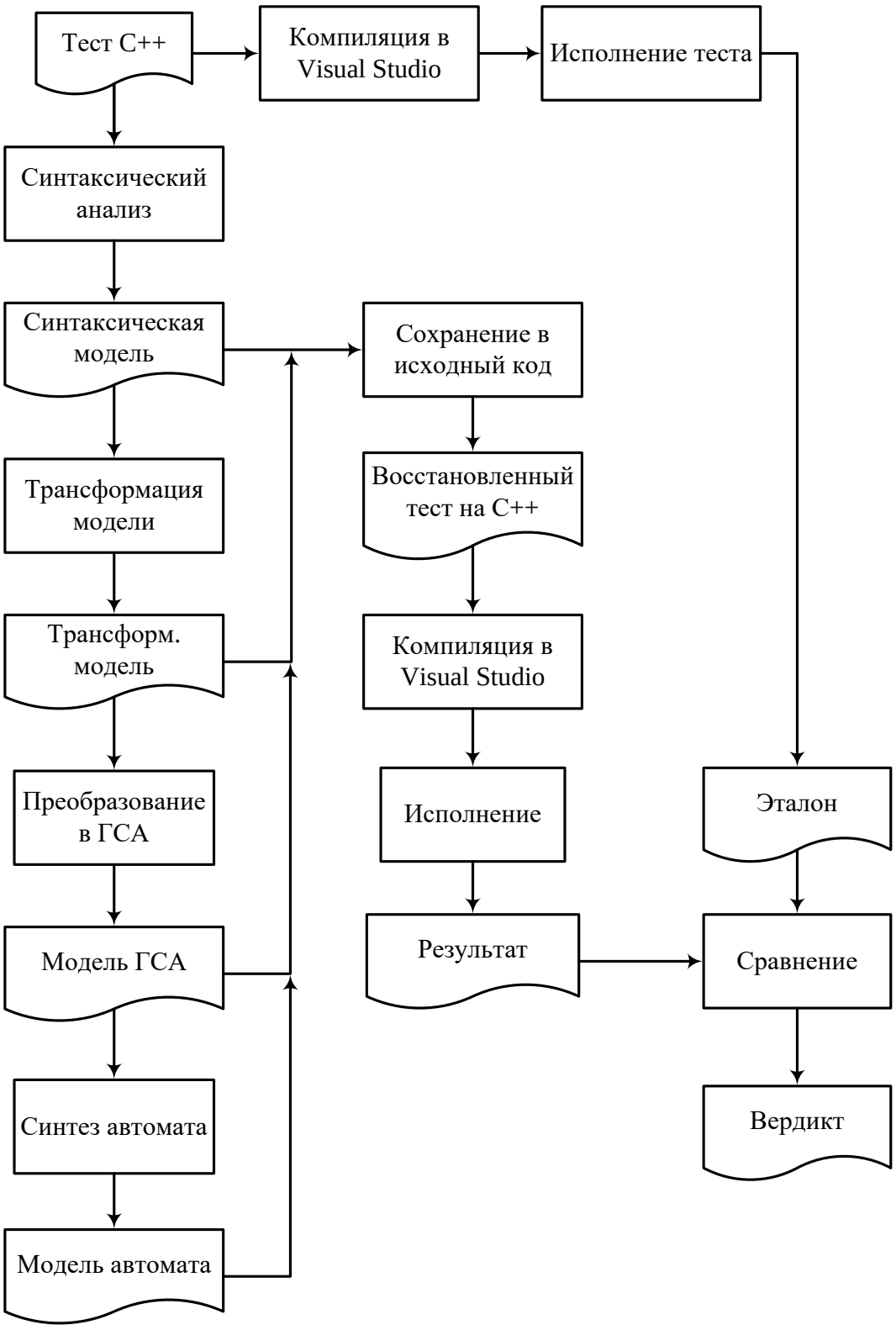


Рис. 4.1. Схема тестування різних внутрішніх моделей компілятора

4.3 Порівняння з аналогами

В американському патенті № 6 226 776 «System for Converting Hardware Designs in High-Level Programming Language to Hardware Implementations» [90] пропонується система автоматизованого проектування, яка конвертує алгоритмічний опис на мові ANSI C в апаратні реалізації на ПЛІС або кристалах жорсткої логіки. У патенті запропоновані численні приклади перетворень алгоритмів з мови C++ в синтезовані підмножини мови Verilog. Розглянемо приклад, показаний в лист. 4.2.

Лістинг 4.2. Вихідний алгоритм на мові C++

```
117.   int sum1 (int n)
118.   {
119.       int i, sum = 0;
120.       for (i = 0; i <n; i ++)
121.           sum + = i;
122.       return sum;
123.   }
124.
125.   int sum2 (int array [], int size)
126.   {
127.       int i, sum = 0;
128.       for (i = 0; i <size; i ++)
129.           sum + = array [i];
130.       return sum;
131.   }
132.
133.   int f1 ()
134.   {
135.       int i;
136.       int array [10];
137.       int size = sizeof (array) / sizeof (* array);
138.       for (i = 0; i <size; i ++)
139.           array [i] = i * 2;
140.       return sum1 (size) + sum2 (array, size);
141.   }
```

Програма високорівневого синтезу, запропонована в цій дисертації, зробила оптимізовану ГСА, як показано на рис. 4.2.

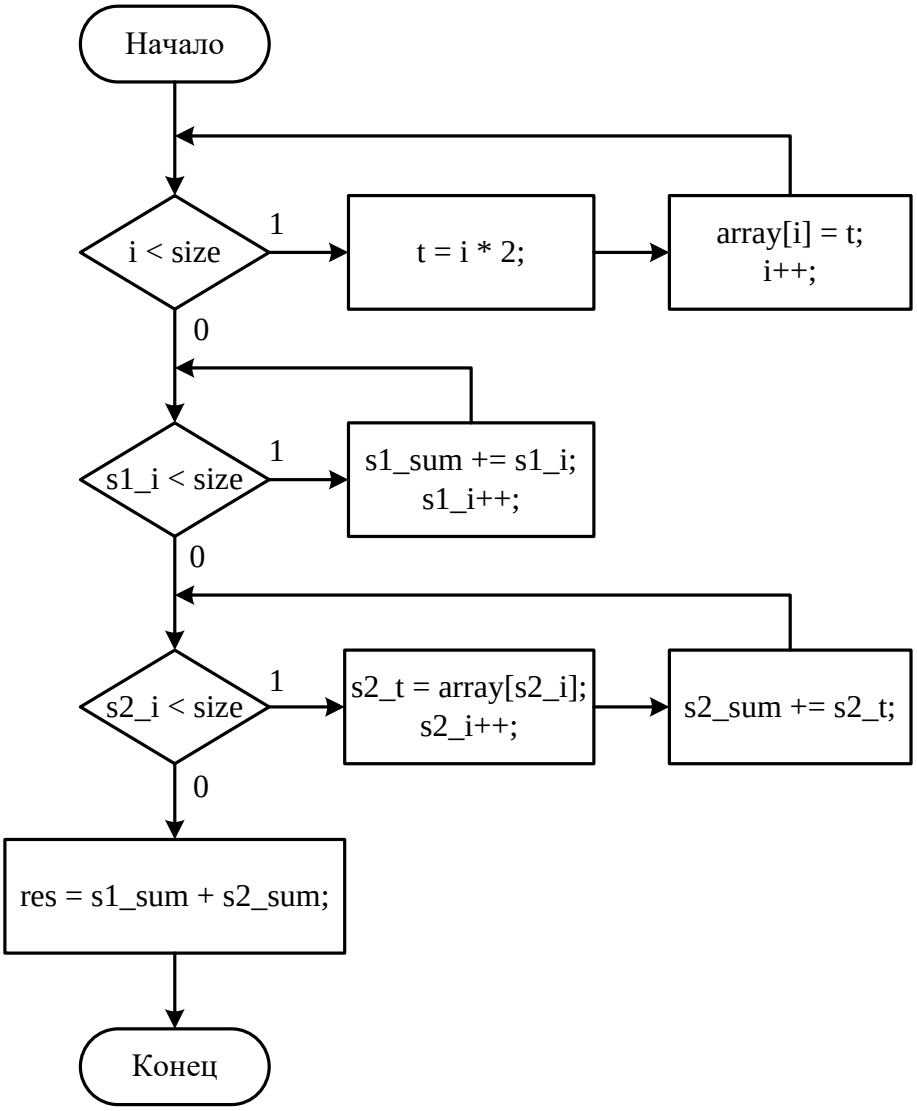


Рис. 4.2. СГСА, отримана в результаті синтезу

При компіляції прикладу були зроблені наступні оптимізації: вбудовування функцій, виявлення паралелізму на рівні микрооперацій. Таким чином, після синтезу здобута модель автомата Мілі з 7-ю станами. У патенті [90] показано, що в результаті конвертації цього прикладу виходить 16 станів автомата.

4.4 Проектування системи на кристалі

Розглянемо систему на кристалі, наведену на рис. 4.3.

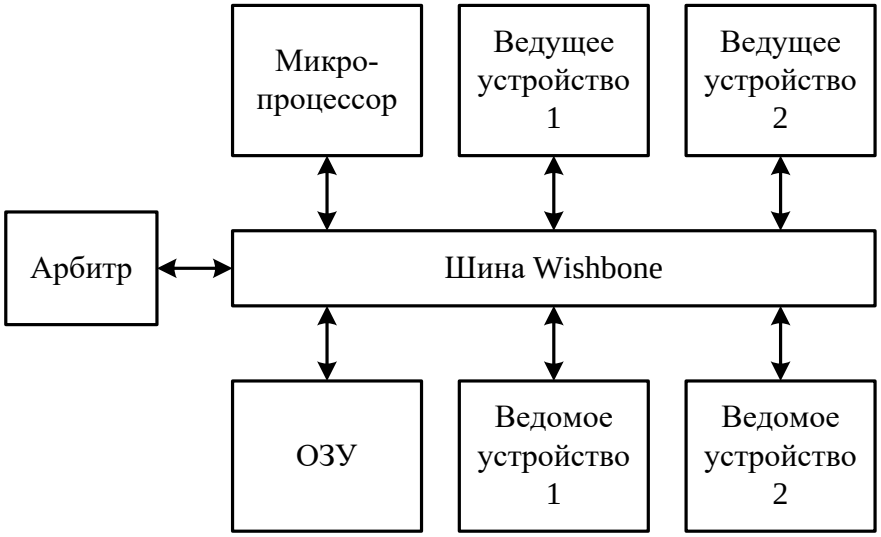


Рис. 4.3. Архітектура тестової системи на кристалі

У даній тестовій системі на кристалі використовується мікропроцесор OpenRISC 1200 [91]. Це 32-розрядний RISC-процесор з відкритим вихідним кодом. Модель процесора поширюється на правах вільної ліцензії, що дозволяє його вивчати, моделювати, розвивати і використовувати в комерційних системах без сплати авторських відрахувань. OpenRISC зроблений на основі гарвардської архітектури, де передбачається роздільне зберігання інструкцій і даних. Мікропроцесор має пёятиступінчастий конвеєр, тим не менше, більшість інструкцій можуть бути виконані за один такт.

У цій системі використовується шина Wishbone.

Для компіляції початкового програмного коду на мові С використовується компілятор *gcc*.

При запуску системи мікропроцесор починає читати інструкції за адресою 256.

4.5 Загальна організація системи проектування

На рис. 4.4 показана структура процесу проектування з використанням простору проектних рішень [131]. На початкових етапах здійснюється введення специфікації проекту на системному рівні з використанням мов C++ і SystemC. Введення моделей здійснюється як в текстовому вигляді, так і в графічному з використанням блокових діаграм і змістовних граф-схем алгоритмів. При визначенні специфікації вказуються функціональні і нефункціональні вимоги. Перші описують закон функціонування моделі, а також вхідні впливу на систему і очікувані відповідні реакції. Серед функціональних вимог можна виділити наступні:

— обмеження на одержувані проектні рішення: вартість реалізації, швидкодія, продуктивність, апаратні витрати, витрати оперативної пам'яті, енергоспоживання;

— доступні архітектури: типи інтегральних схем, елементну базу, а також їх характеристики; типи вбудованих мікропроцесорів.

Вихідні тексти вхідних специфікацій повинні відповідати міжнародним стандартам C++ [92] і SystemC [93].

На наступному етапі здійснюється синтаксичний аналіз вихідної моделі. У разі виявлення помилок, створюється звіт, на підставі якого інженер може виправити помилки і повторити трансляцію моделі. Якщо вихідні тексти моделі не містять помилок, то будується синтаксична модель проектованої системи [132].

Синтаксична модель придатна для широкого класу задач автоматизованого проектування. Серед них: декомпозиція проекту на програмну і апаратну частини, високорівнева оптимізація, логічний синтез і компіляція, побудова перевіряючих тестів.

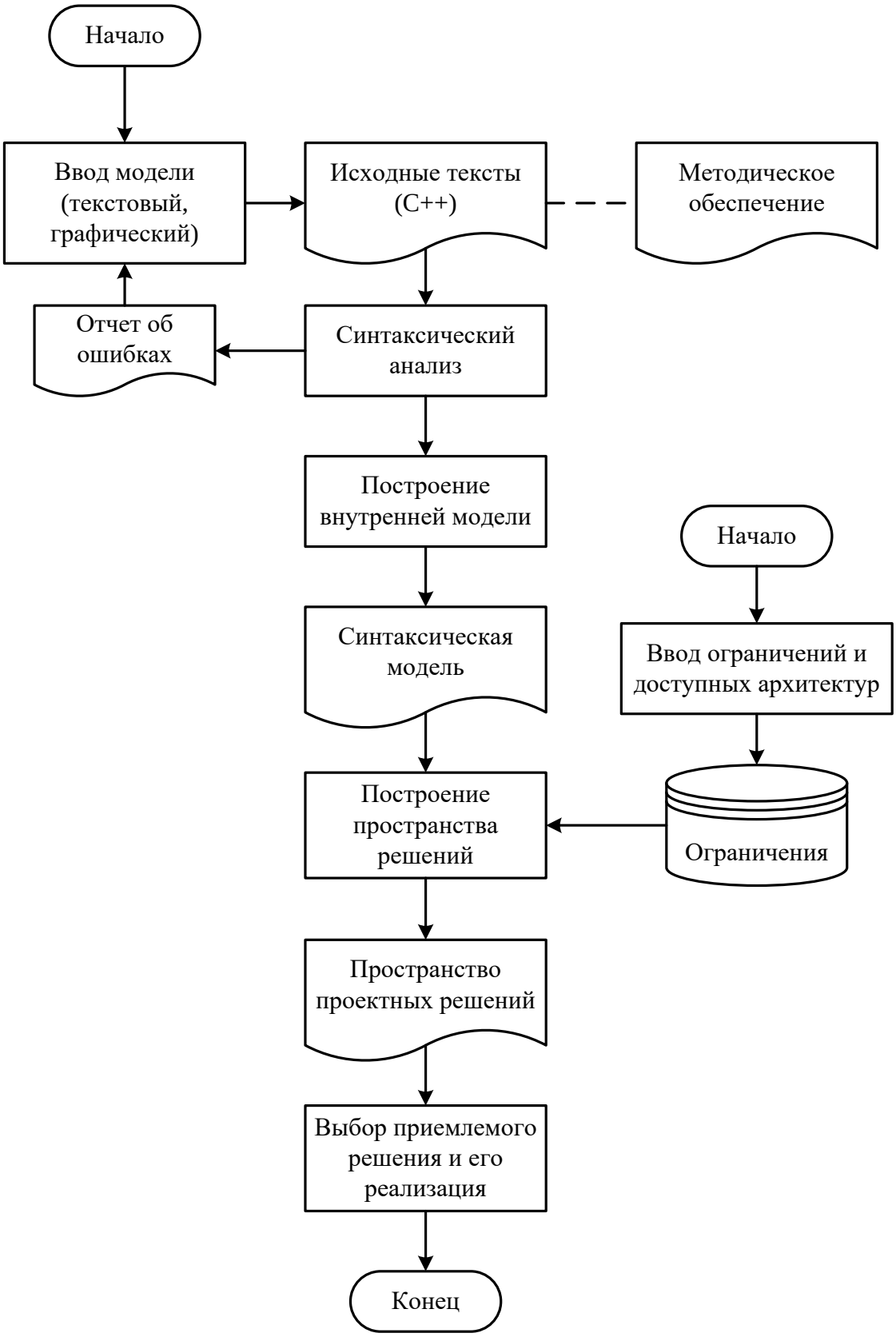


Рис. 4.4. Структура процесу проектування [94] з використанням простору проектних рішень

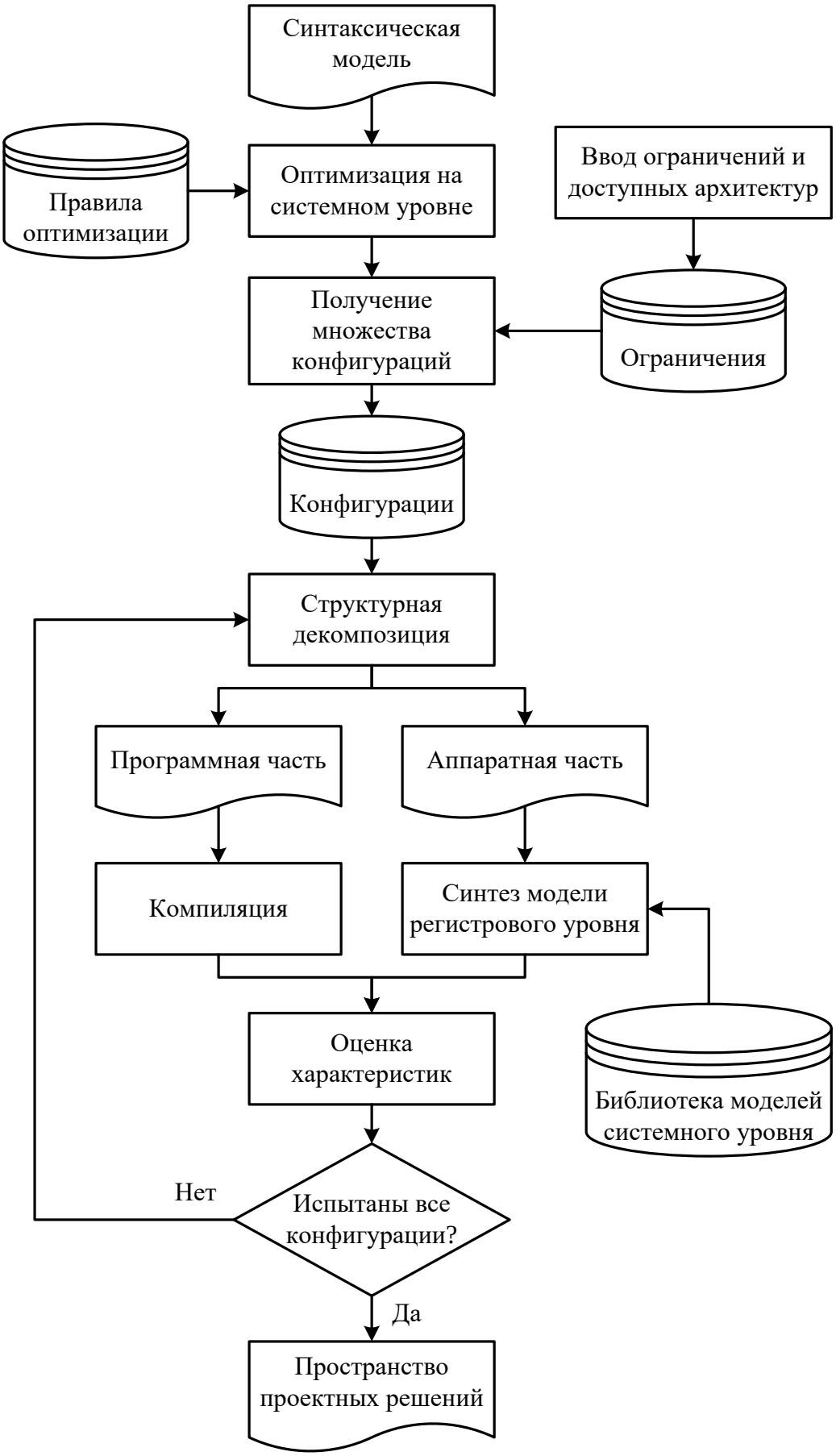


Рис. 4.5. Процес побудови простору проектних рішень

4.6 Структурна декомпозиція проекту

Для успішної реалізації проекту, необхідно вирішити задачу структурної декомпозиції проекту на програмну і апаратну частини. Введемо кілька визначень, які широко використовуються в мові SystemC [133].

Визначення 4.1. Модуль – клас C++, успадкований від класу *sc_module*, оголошеного в бібліотеці SystemC. У мові VHDL поняттю модуля відповідає пара *entity* і *architecture*. Модуль має певний інтерфейс – множину вхідних і вихідних сигналів певного типу, а також множину паралельно взаємодіючих процесів з відповідними списками чутливості. Процес в SystemC аналогічний конструкції *process* в мові VHDL або *always* в мові Verilog.

Визначення 4.2. Ієрархія модулів – множина екземплярів модулів, створених при компіляції SystemC моделі. На цій множині визначені відносини «А є дочірній модуль Б», «А є батьківський модуль Б».

Визначення 4.3. Модуль верхнього рівня – екземпляр модуля, який не встановлено ні в один інший модуль. У разі незв’язаної ієрархії модулів може бути кілька модулів верхнього рівня.

Розглянемо блокову діаграму проекту *sc_main*, показаного на рис. 4.6 (вихідний текст моделі на мові SystemC може бути знайдений в додатку [95]). Блокова діаграма відображає ієрархію модулів, а також відносини модулів між собою. Тут модулі *m_producer* і *m_consumer* є дочірніми до модуля *testbench*, який в свою чергу є дочірнім до модуля *sc_main*.

Проект цифрової системи, описаний на мові SystemC, зручно представити у вигляді кореневого дерева, визначеного на множині ієрархії модулів проекту. Це дерево відображає ставлення «бути встановленим в». Корінь дерева – модуль верхнього рівня, в якому встановлені всі інші екземпляри модулів нижчих рівнів. На рис. 4.7 показано кореневе дерево для проекту *sc_main*.

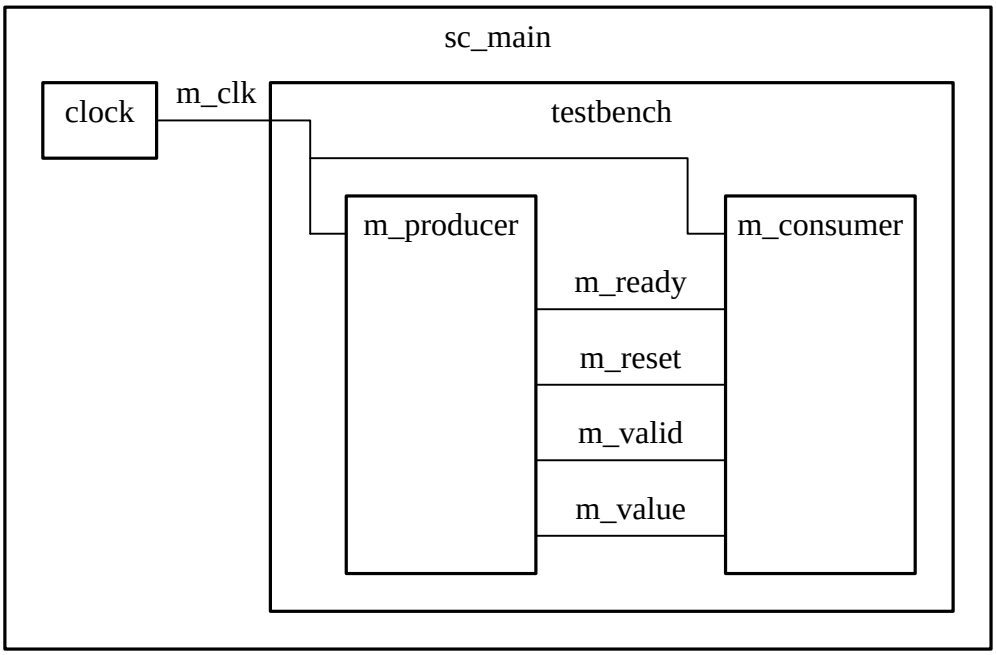


Рис. 4.6. Блокова діаграма проекту *sc_main*

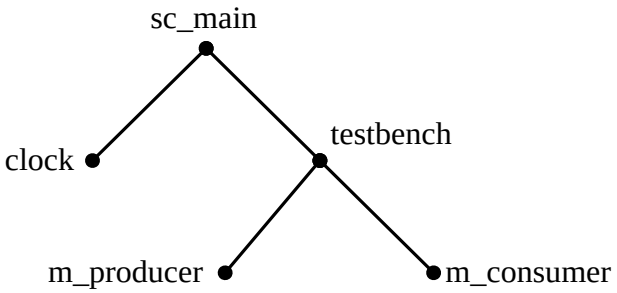


Рис. 4.7. Подання ієрархії модулів у вигляді кореневого дерева

Для реалізації кожного модуля є дві альтернативи: апаратна реалізація і програмна. Кількість конфігурацій за такої умови дорівнює $k = 2^n$, де n – кількість модулів в ієрархії. Ця кількість може бути скорочена за рахунок того, що не всі модулі можуть бути логічно синтезовані в апаратурні структурні моделі. Таким чином, кількість конфігурацій може бути визначено $k =$ но $k = 2^m$, $m \leq n$, де m – кількість модулів, для яких можливо побудувати коректну апаратну модель з SystemC опису. Наприклад, для моделі на рис. 2.4 k = 32.

4.7 Методи структурних і алгоритмічних трансформацій

Існує компроміс між обсягом апаратних ресурсів і часом виконання того чи іншого завдання. Апаратура має величезні можливості в плані виконання функцій з великим рівнем паралелізму. При розпаралелювання функція може бути виконана швидше, але для цього потрібні додаткові апаратурні витрати.

Методи структурної оптимізації полягають в застосуванні різних трансформацій моделей або описів, які призводять до поліпшення часових або апаратних характеристик пристрою.

Методи можуть бути застосовані на різних рівнях абстракції. На алгоритмічному рівні об’єктом оптимізації стає алгоритм роботи пристрою. На мікроархітектурному рівні об’єктом оптимізації виступає структурна модель операційного автомата, а також граф керуючого автомата. Базові відомості про мінімізації умовних і операторних вершин автоматів можуть бути знайдені в роботах С. І. Баранова [96].

Векторизация – така трансформація циклу, при якій циклічні дії виконуються відразу над множиною операндів, а не над одним. У класифікації Флінна такої організації обчислень відповідає модель SIMD – Single Instruction, Multiple Data – одна операція, сукупність операндів.

Цикли – типова гаряча точка в додатку. Навіть швидка операція або функція, виконана мільйон разів, сповільнить роботу системи. Крім корисного навантаження: тіла циклу, є і накладні витрати на організацію циклу: лічильник ітерацій (регістр), операція інкремент / декремент, розгалуження. При векторизації циклу використовується операція «розгортка циклу». Операнди об’єднуються у вектори з певним числом елементів, який називається *ступінь векторизації циклу*. Як правило, операція виконується за один такт над усіма елементами вектора [134].

Розглянемо приклад, показаний в лістингу 4.2. Тут йде послідовне підсумовування елементів двох векторів, результат записується в третій вектор.

Лістинг 4.2. Послідовна обробка елементів масиву в циклі

```
142.   for (int i = 0; i <N; i ++)
143.   c [i] = a [i] + b [i];
```

На рис. 4.8 схематично показано виконання арифметичних операцій в оригінальному циклі (а), і в векторизованих циклах (б, в). У першому випадку за одну ітерацію виконується дія над однією парою операндів, а в другому і третьому випадках – над двома і чотирма парами операндів.

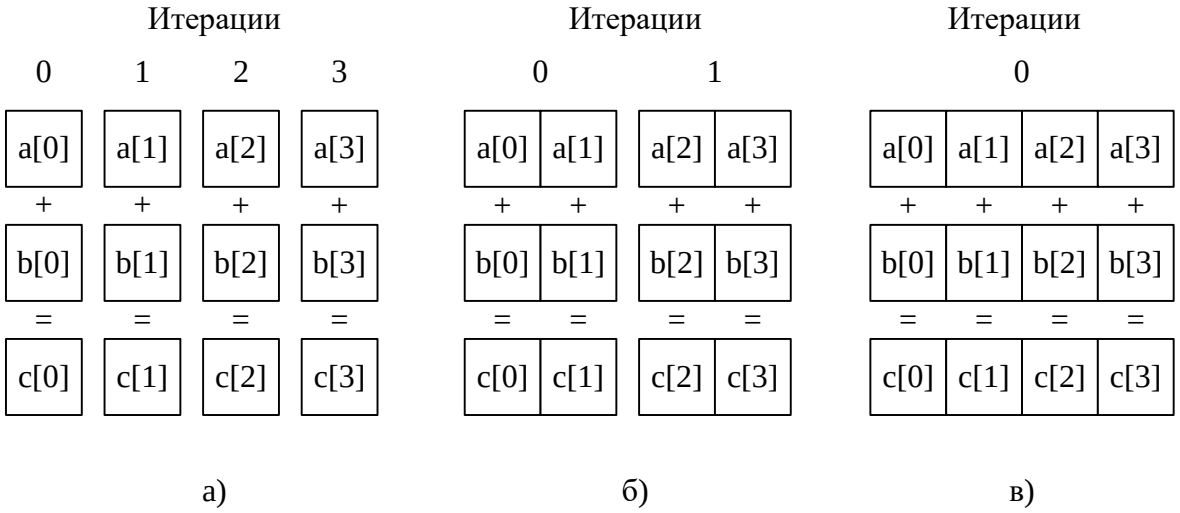


Рис. 4.8. Виконання операцій: а – за чотири ітерації в оригінальному циклі; б – за дві ітерації в циклі зі ступенем векторизації 2; в – за одну ітерацію в циклі зі ступенем векторизації 4

На рис. 4.9 наведена змістовна ГСА для цього прикладу. Таким чином, цикл містить чотири стани автомата. Для виконання програми підсумовування векторів з N елементів потрібно $k = 4 \times N$ тактів роботи пристрою. Апаратні витрати: 4 регістра (a1, b1, c1, i), 2 суматора, функція порівняння «менше».

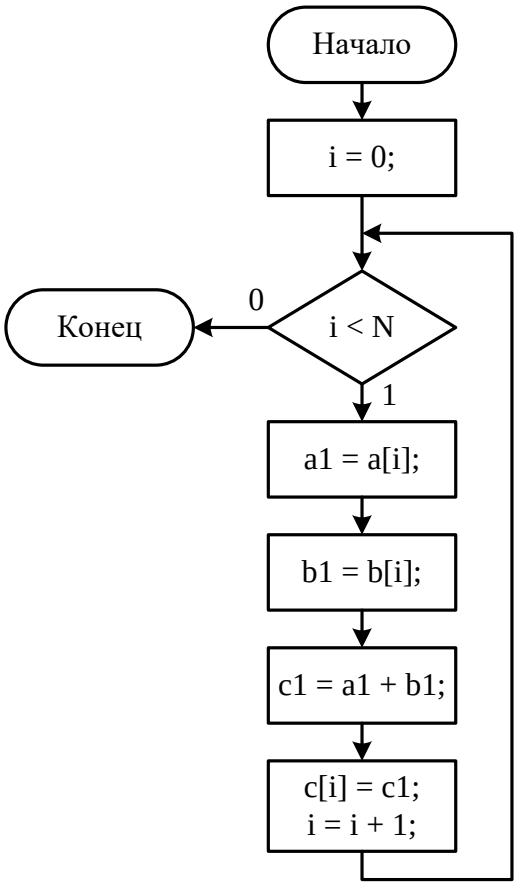


Рис. 4.9. СГСА для прикладу в лістингу 4.2

Розглянемо приклад, показаний в лістингу 4.3. Здійснено трансформація «векторизація циклу». Тут на одному витку циклу виконується дві операції додавання. Важлива умова – N повинно бути кратно ступеню векторизації.

Лістинг 4.3. Векторизований цикл з лістингу 4.1

```
144.   for (int i = 0; i <N; i + = 2) {
145.   c [i]   = A [i] + b [i];
146.   c [i + 1] = a [i + 1] + b [i + 1];
147.   }
```

На рис. 4.10 показана СГСА для прикладу з лістингу 4.3. Цикл включає в себе 7 станів автомата. Для виконання програми підсумовування векторів з N елементів, потрібно $k = \frac{7 \times N}{2}$ тактів роботи пристрою. Апаратні витрати: 7 регістрів (a1, a2, b1, b2, c1, c3, i), 3 суматора, функція порівняння «менше».

У цьому прикладі видно, що в тілі циклу присутні чотири операції читання з пам’яті і дві операції запису в пам’ять, які повинні виконуватися послідовно. Дві операції додавання можуть бути виконані паралельно. Можна підрахувати, що цикл зі ступенем векторизації l, в якому міститься s операцій (які виконуються послідовно) і p операцій (які виконуються паралельно), може бути виконаний за k тактів, що визначається формулою:

$$k = \frac{(s+1) \times N}{l}. \qquad (4.1)$$

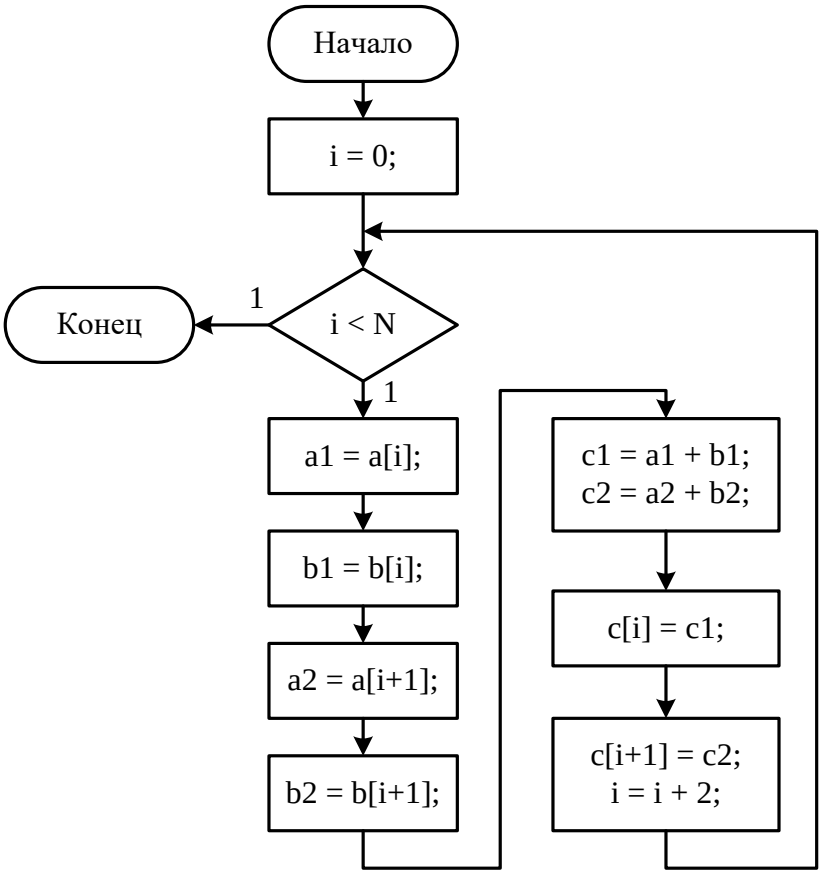


Рис. 4.10. СГСА для прикладу з лістингу 4.3

Розглянемо різні оптимізації на мікроархітектурному рівні. *Злиття (об’єднання) мікрооперацій* – така трансформація ГСА, при якій мікрооперації в різних операційних блоках (див. рис. 4.11, а), об'єднуються в один опе-

раційний блок (див. рис. 4.11, б). Така трансформація можлива, якщо мікрооперації y1 і y2 можуть виконуватися паралельно і їх одночасне виконання призводить до того ж результату, що і послідовне. Допускається об'єднання довільної кількості мікрооперацій на лінійній ділянці ГСА. При такій трансформації скорочується кількість операторних вершин, що призводить до скорочення кількості станів керуючого автомата Мура та скорочення кількості тактів в циклі. Ця трансформація не впливає на апаратні витрати.

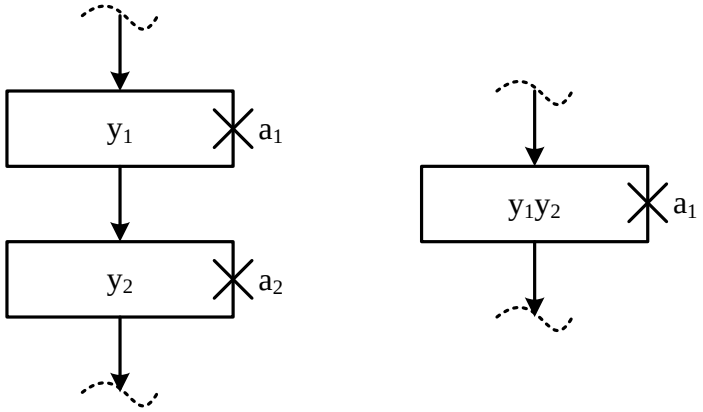
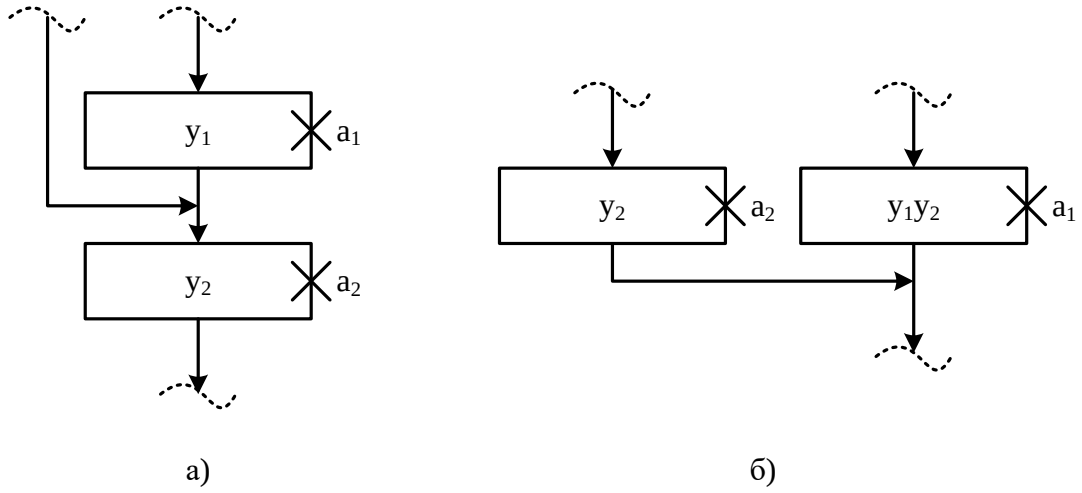


Рис. 4.11. Злиття мікрооперацій: а – вихідний підграф, б – трансформований підграф

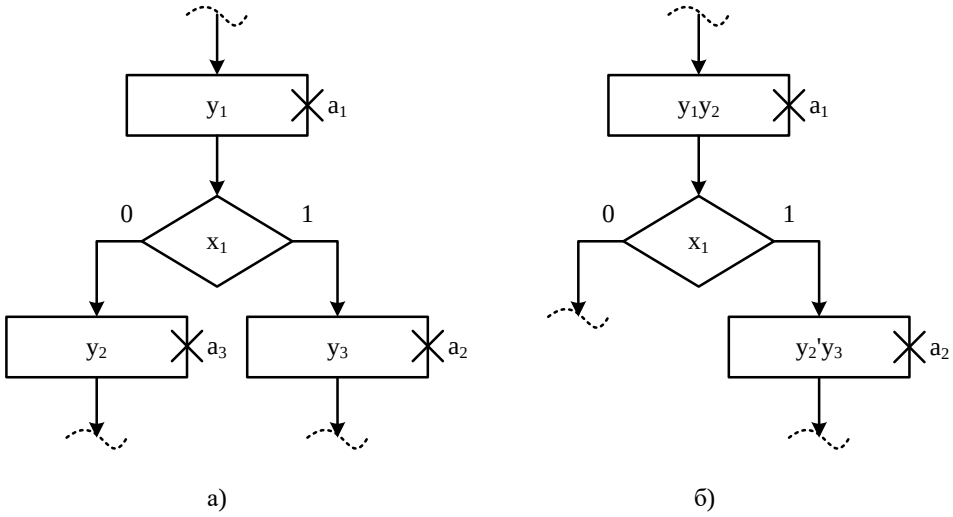
У тому випадку, якщо між двома операційними вершинами є вхідна дуга (див. рис. 4.12, а), то злиття операцій відбувається за такими правилами. Необхідно об’єднати операції y1 і y2 в стан a1 за правилами, зазначеним вище. Мікрооперацію y2 слід внести в ланцюг до збігання дуг (див. рис. 4.12, б). Таке перетворення не призводить до зміни кількості станів в автоматі, і не впливає на апаратурні витрати. Виграш полягає в скороченні кількості тактів в циклі.



а) б)

Рис. 4.12. Злиття мікрооперацій: а – вихідний підграф, б – трансформований підграф

Розглянемо більш складний випадок об'єднання операторів. У вихідному підграфі на рис. 4.13, а, мікрооперації y1 і y2 можуть бути виконані паралельно. У ланцюзі між станами a1 і a2 присутній умовний оператор і розгалуження. У тому випадку, якщо y1 і y2 будуть об’єднані в стані a1, то в операторі альтернативної гілки необхідно помістити мікрооперацію y2’ (читається «ігрек два штрих»), яка має зворотну дію до мікрооперації y2.



а) б)

Рис. 4.13. Складний випадок об’єднання операторів:
а – вихідний підграф; б – трансформований підграф

Обчислення констант – така трансформація операційного автомата, при якій вноситься апаратна надмірність, що обчислює константну функцію від змінної. При такій трансформації скорочується кількість станів автомата, а також довжина циклу в тактах. При цьому зростає час поширення сигналу від регістру до виходів.

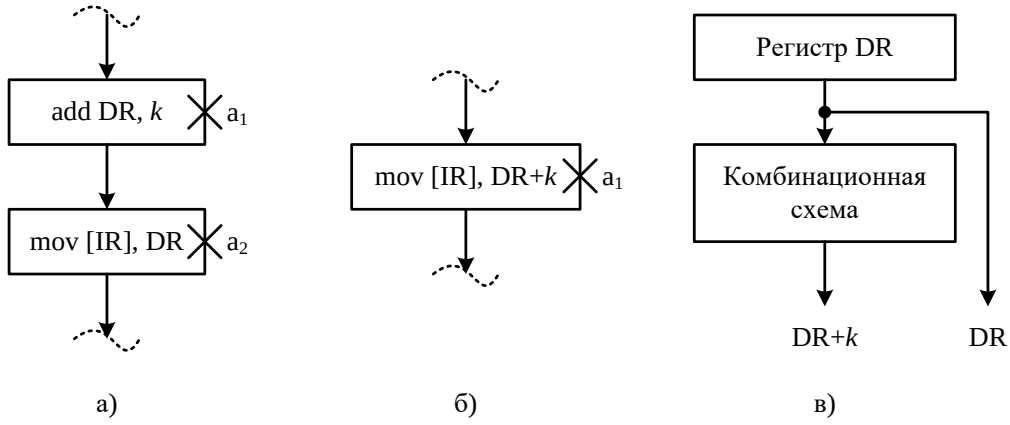


Рис. 4.14. Обчислення констант: а – вихідний підграф; б – трансформований підграф; в – структурна реалізація

На рис. 4.15 показана оптимізована ГСА, в якій об’єднані мікрооперації і обчислено константний вираз. Кількість станів скорочено з 5 до 3. Найкоротший цикл а2 становить один такт, найдовший а2а3 – два такту.

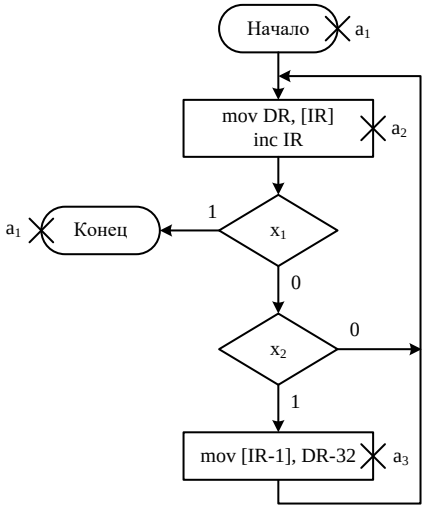


Рис. 4.15. Оптимізована ГСА

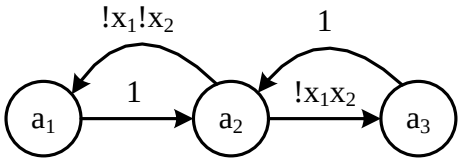


Рис. 4.16. Оптимізований граф переходів

4.8 Перетворення несинтезованих конструкцій

Синтаксис мови C++ надзвичайно різноманітний. Існує велика кількість бібліотек, а також прийомів програмування на C ++, які не можуть бути перетворені в схему на рівні регістрових передач на поточний момент. Наприклад, популярними є функції і класи введення-виведення інформації: scanf, printf, std :: cin, std :: cout, а також операції з файлами, які не мають прямого відображення в логічну схему. На даний момент програми синтезу виявляють такі конструкції на ранніх стадіях компіляції вихідних файлів, а потім або ігнорують ці функції, або повідомляють про присутність несинтезованих конструкцій і зупиняють процес синтезу [135].

У деяких випадках доцільно підготувати заміну такої несинтезованої конструкції, щоб продовжити верифікацію моделі, поки не буде готова синтезованая заміна. На рис. 4.17 показаний графік проекту, в якому верифікація не може бути розпочато до тих пір, поки не будуть підготовлені синтезовані заміни несинтезованих конструкцій.

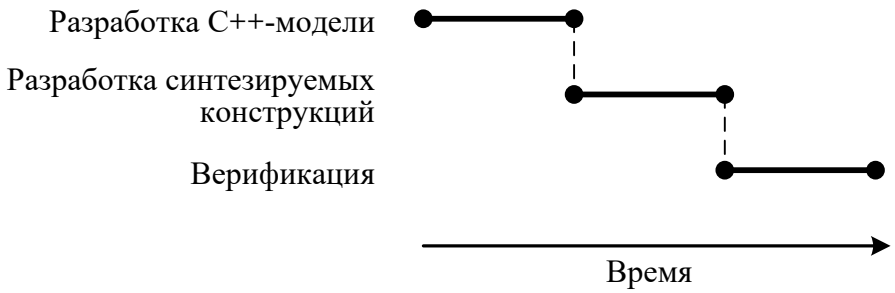


Рис. 4.17. Графік проектування пристрою

Існує рішення цієї проблеми, яке полягає у використанні віртуальних прототипів на мові SystemC. Віртуальний SystemC-прототип – це модель на

мові SystemC, інтерфейс якої строго відповідає проектованого пристрою, а архітектура виконана на високому рівні опису. Зазвичай, SystemC-прототипи не синтезуються, а використовуються для верифікації системи в цілому на ранніх етапах проектування.

На рис. 4.18 показаний графік проекту, в якому несинтезованих конструкції замінені SystemC-прототипами. За рахунок швидкого переходу від C++ до SystemC моделі, верифікація всієї цифрової системи може бути розпочато раніше, ніж буде закінчена розробка синтезованих замін.

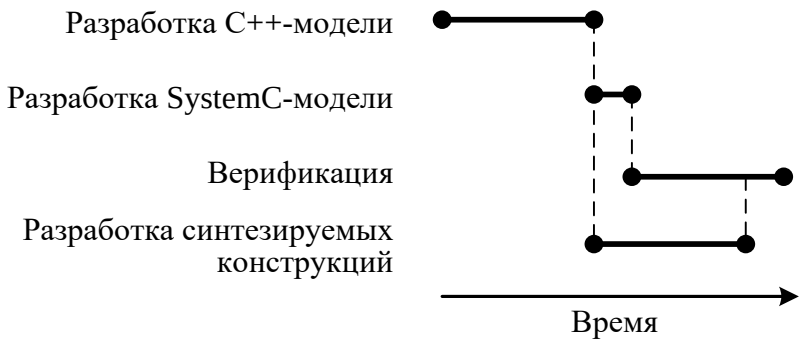


Рис. 4.18. Графік проектування пристрою з використанням SystemC-прототипів

Основні етапи перетворення наступні.

1. Визначити фрагмент коду, який не може бути синтезований.

2. Визначити інформаційні залежності цього фрагмента від суміжного коду.

3. Виділити цей фрагмент коду в окрему C ++ функцію.

4. Розробити SystemC-прототип на основі цієї C ++ функції.

5. Виконати синтез C ++ моделі в структурну VHDL модель.

6. Інтегрувати SystemC модель в структурну VHDL модель.

В результаті перетворення виходить система, як показано на рис. 4.19. На етапі верифікації буде використана SystemC-модель (рис. 4.19,а). Після того, як буде готовий структурний еквівалент, то буде використана VHDL-модель компонента (рис. 4.19,б).

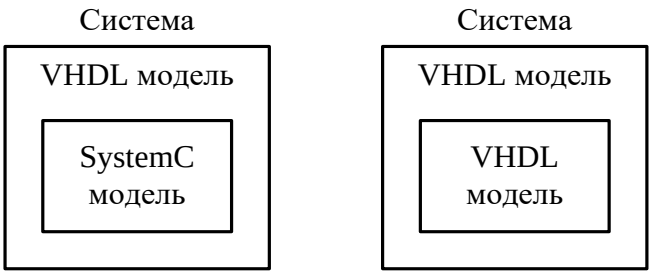


Рис. 4.19. Ієрархія структурної моделі: а – з використанням SystemC-моделі компонента; б – з використанням VHDL-моделі компонента

Розглянемо приклад, показаний в лістингу 4.4. У цьому прикладі створюється об’єкт статичної пам’яті, обсягом 1024 осередки, які ініціалізуються псевдовипадковими значеннями [136]. Припустимо, що функція rand () не має структурного еквівалента на цей момент.

Лістинг 4.4. Приклад програми з несинтезованою конструкцією

```
148.   int main ()
149.   {
150.   const int size = 1024;
151.   int mem [size];
152.   for (Int i = 0; i <size; i ++)
153.   mem [i] = rand ();
154.   return 0;
155.   }
```

Введемо кілька визначень.

Визначення 4.4. *Транзакція* – модельна подія високого рівня, яка включає в себе множину подій нижчого рівня. У транзакції дотримується строга послідовність і часові інтервали між подіями. Зазвичай, транзакція – це цілісна операція, наприклад, читання або запису даних. Синтаксично транзакція виглядає, як виклик функції або методу класу.

Визначення 4.5. *Транзакційна модель* – така модель системи, де всі події між елементами відбуваються на рівні транзакцій.

Визначення 4.6.*Транзактор* – елемент системи, який перетворює транзакцію до сукупності подій моделями низького рівня, наприклад, регістрового або вентильного. Транзактор також виконує зворотне перетворення – з регістрового і вентильного рівнів на рівень транзакцій. Транзактор реалізується за допомогою класу, що має два інтерфейси: високого рівня і рівня регістрових передач або навіть вентильного.

Тут і далі буде використовуватися нотація для моделей системного рівня, як показано на рис. 4.20. Нотація взята з [97]. Інтерфейс системного рівня відповідає поняттю інтерфейсу мови C ++ – сукупність абстрактних методів, які реалізує модуль.

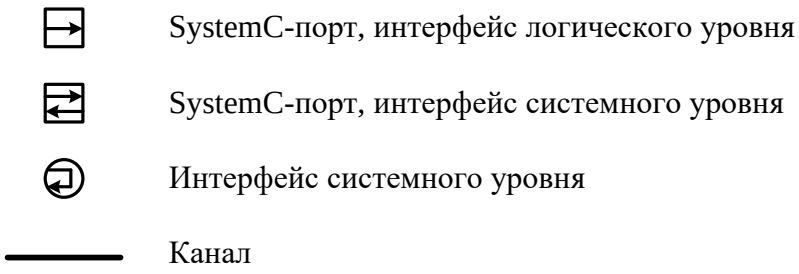


Рис. 4.20. Нотація моделей системного рівня

Зобразимо цю програму у вигляді транзакційної моделі [98] на мові SystemC, як показано на рис. 4.21. Необхідно зробити декомпозицію на два модуля: а – Main, основний, де відбуваються всі обчислення; б – Rand, де буде проводитися обчислення функції rand (). Модуль Main матиме один порт абстрактного типу rand_if (див. лістинг 4.4).

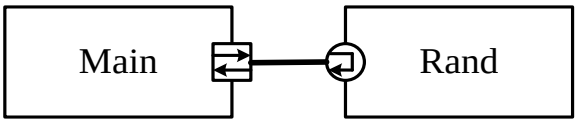


Рис. 4.21. Система з двох модулів

Створення абстрактного інтерфейсу необхідно для того, щоб можна було створювати різні модулі, які успадковують один і той же інтерфейс.

Лістинг 4.5. Абстрактний інтерфейс для модуля rand

```
156.   struct rand_if: public sc_interface {
157.   virtual int rand_f () = 0;
158.   };
```

Модифікований модуль Main показаний в лістингу 4.6. Тут створено клас-обгортка мовою SystemC.

Лістинг 4.6. Модуль Main

```
159.   SC_MODULE (Main) {
160.   sc_port <rand_if> rand_m;
161.   SC_CTOR (Main) {
162.   SC_THREAD (process);
163.   }
164.   void process () {
165.   for (int i = 0; i <size; i ++)
166.   // підстановка замість функції rand ()
167.   mem [i] = rand_m-> rand_f ();
168.   sc_stop ();
169.   }
170.   private:
171.   const int size = 1024;
172.   int mem [size];
173.   };
```

Реалізація модуля Rand показана в лістингу 4.7.

Лістинг 4.7. Реалізація модуля Rand

```
174.   SC_MODULE (Rand), public rand_if {
175.   virtual int rand_f () {
176.   return rand ();
177.   }
178.   };
```

Для успішної інтеграції віртуального прототипу в систему на вентильному рівні, необхідно деталізувати інтерфейси системи. Для цього між модулями Main і Rand необхідно помістити два транзактори: а) H2L (High Level To Low Level), який перетворює високорівневий інтерфейс rand_if до протоколу на рівні логічних сигналів; б) L2H (Low Level to High Level) – транзактор, зворотний H2L. За своєю суттю, Rand – це генератор випадкових чисел, який генерує чергове число за запитом. Для цього в систему введений модуль Clock – генератор синхроімпульсів. Таким чином, при деталізації системи в ній з'явилося поняття часу. Ми отримали часову трансакціонну модель з безчасової транзакційної моделі.

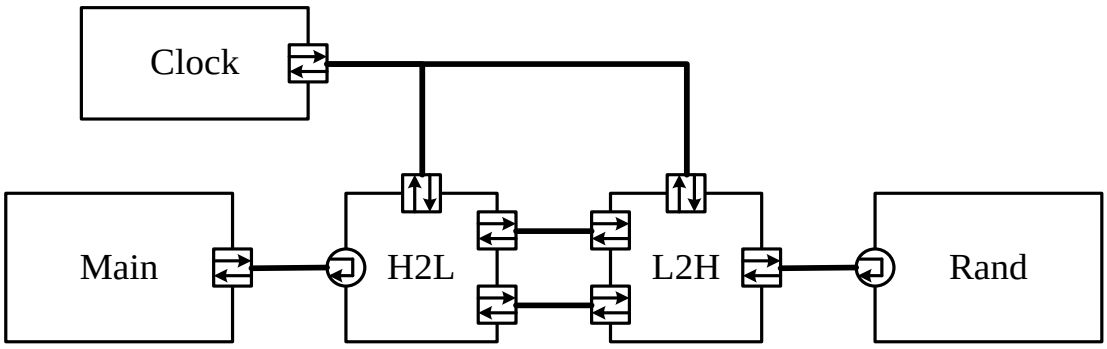


Рис. 4.22. Деталізована система

Ієрархія класів на мові UML показана на рис. 4.23. З діаграми класів видно, що модулі Rand і H2L мають ідентичний інтерфейс rand_if. Це дає можливість вільно підключати будь-який з них до порту модуля Main.

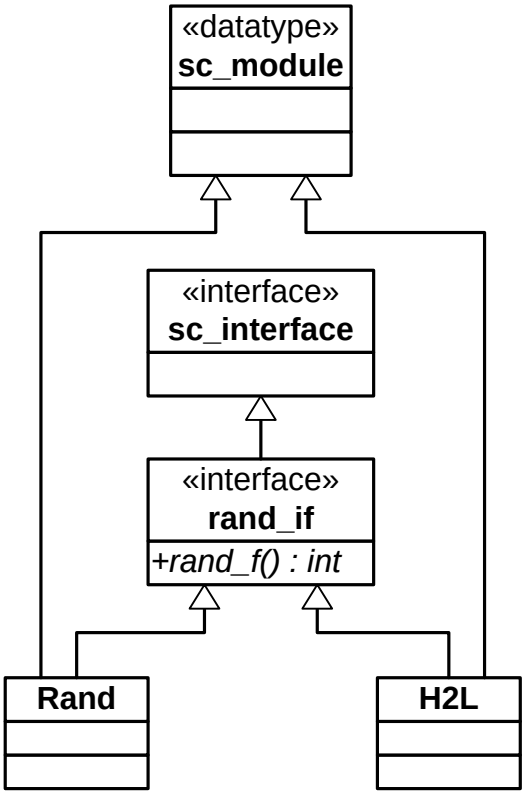


Рис. 4.23. Ієрархія класів: множинне успадкування від класу *sc_module* і інтерфейсу *rand_if*

Побудуємо змістовну граф-схему алгоритму для основного процесу модуля Main (рис. 4.24).

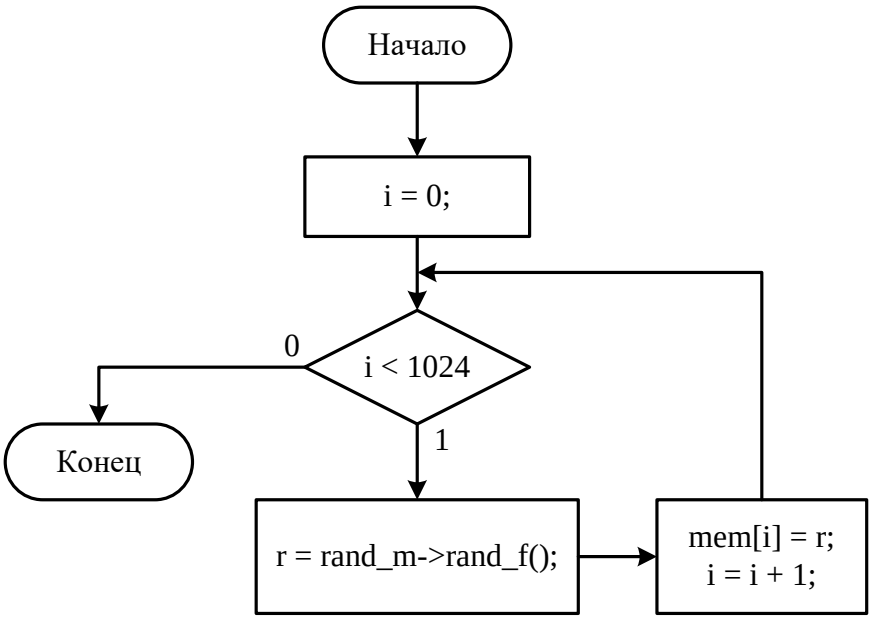


Рис. 4.24. СГСА для процесу process модуля Main

У лістингу 4.8 показана реалізація транзактору [99] L2H. Логіка роботи транзактору наступна: транзактор реагує на передній фронт синхронізації *clk*, і якщо сигнал дозволу *en* встановлений в одиницю, то виставити на вихід *output* значення функції *rand_f ()* модуля, підключеного до порту *rand_m*. Важливо відзначити те, що інтерфейс транзактор (сигнали *clk*, *enable*, *output*) вже мінятися не будуть. Ці ж сигнали будуть використані при установці реального синтезованого модуля Rand, коли такий модуль буде доступний. Те ж саме стосується і протоколу роботи цього транзактору.

Лістинг 4.8. Реалізація транзактору L2H

```
179.   SC_MODULE (L2H) {
180.   sc_port <rand_if> rand_m;
181.   sc_in <bool> clk;
182.   sc_in <en> enable;
183.   sc_out <int> output;
184.   SC_CTOR (L2H) {
185.   SC_METHOD (process);
186.   sensitive << clk.posedge ();
187.   }
188.   void process () {
189.   if (en)
190.   output = rand_m-> rand_f ();
191.   }
192.   };
```

На рис. 4.25 показана часова діаграма роботи транзактору L2H.

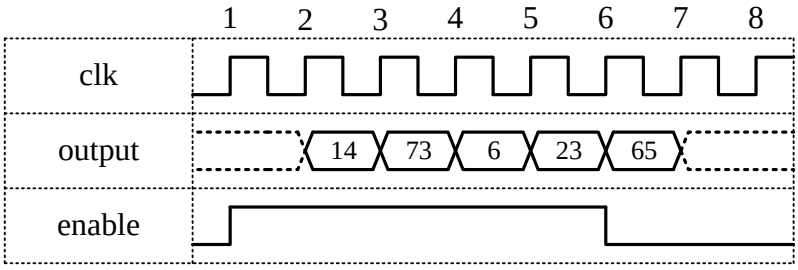


Рис. 4.25. Часова діаграма роботи транзактору H2L

Наступним етапом інтеграції віртуального прототипу є апаратний синтез модуля Main і його злиття з транзактором H2L. Після цього кроку система набуде вигляду, як показано на рис. 4.26. Модуль Main-HW являє собою апаратну реалізацію модуля Main (лістинг 4.6).

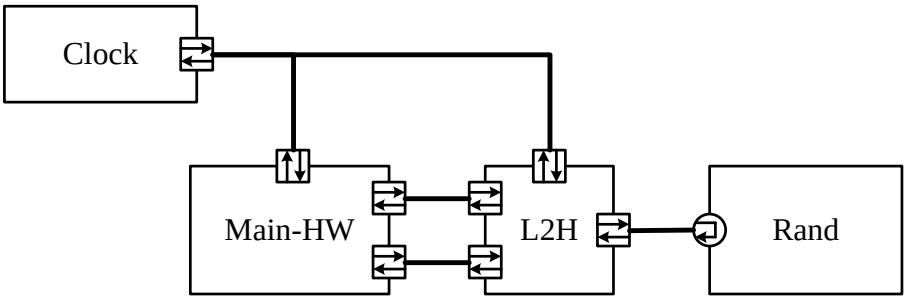


Рис. 4.26. Злиття транзактору H2L і модуля Main в модуль Main-HW

СГСА для цього модуля з умовою підключення до транзактору L2H показана на рис. 4.27. Мається на увазі, що сигнали *en* і *output* – це відповідні сигнали транзактору L2H.

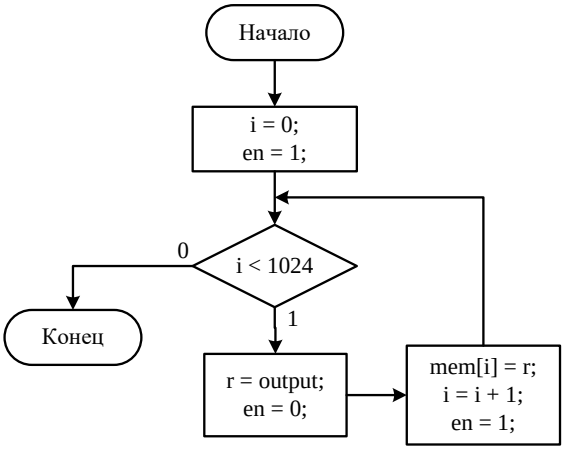


Рис. 4.27. Модифікована СГСА для модуля Main-HW

4.9 Результати практичного синтезу мультиверсного методу проектування

Було проведено порівняння продуктивності розробленої апаратної моделі і програмної реалізації на процесорі експериментальним шляхом. Дані отримані в результаті роботи 10 інженерів над одним проектом. Оцінювалися такі характеристики: часові витрати на проектування, швидкодія, енергоспоживання, площа на кристалі.

Було вибрано три найпоширеніші реалізації: послідовна, паралельна, конвеєрна [100].

Послідовною називається реалізація в якій всі мікрооперації впорядковані за часом. Перевага застосування такої реалізації в тому, що використовується менша площа на кристалі, менше енергоспоживання, але низька продуктивність.

Паралельною називається реалізація, в якій паралельно виконуються дві і більше мікрооперації за один такт роботи. Особливості: високе енергоспоживання, продуктивність і площа на кристалі.

Конвеєрною називається реалізація, в якій за один такт роботи виконується дві і більше мікрооперації на різному етапі виконання. Особливості: висока продуктивність, займає більше місця на кристалі і високе енергоспоживання.

Для адитивної (мультиплікативної) оцінки ефективності методів всі значення були інтегрально оцінені. При інтегральній оцінці найефективніше значення приймалося за одиницю, а інші вираховувались за такою формулою:

$$y_i = \frac{x_i}{x_{max}}, \tag{4.2}$$

де x_{max} – найефективніше значення, x_i – значення, яке було отримано в ході експерименту.

Ця оцінка добре показала, в якій реалізації який метод ефективніший. У послідовній реалізації за всіма критеріями ефективним є автоматичний метод. У паралельній реалізації за площею на кристалі ефективніше працює ручний метод, а за всіма іншими критеріями виграє автоматичний метод. У конвеєрної реалізації ручний метод виграє за енергоспоживанням, в усьому іншому ефективніше автоматичний метод.

Після чого була зроблена адитивна (мультиплікативна) оцінка ефективності кожної окремо реалізації, і загальна адитивна (мультиплікативна) оцінка ефективності методу.

Загальний вигляд адитивної моделі наступний:

$$y_i = \sum_{i=1}^{n} x_i \,, \tag{4.3}$$

де y_i – коефіцієнт кожного інженера для адитивної оцінки, x_i – коефіцієнт за кожним критерієм, n – кількість критеріїв.

Загальний вигляд мультиплікативної моделі має вид:

$$y_i = \prod_{i=1}^{n} x_i, \tag{4.4}$$

де y_i – коефіцієнт кожного інженера для мультиплікативної оцінки, x_i – коефіцієнт за кожним критерієм, n – кількість критеріїв.

На рис. 4.28 – 4.30 представлені результати адитивної оцінки кожної реалізації.

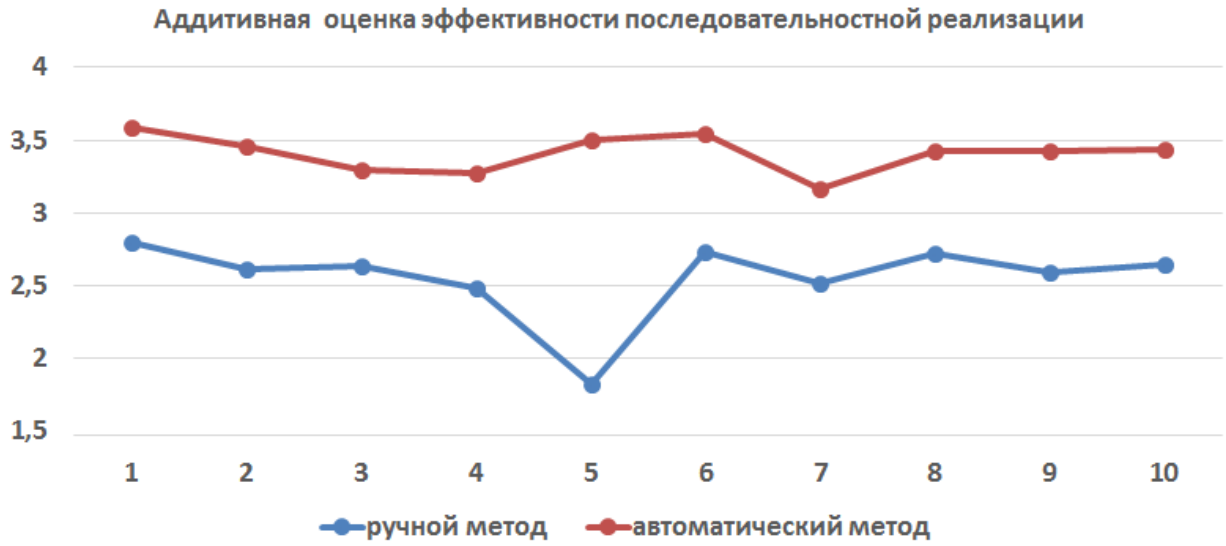


Рис 4.28. Адитивна оцінка ефективності послідовної реалізації

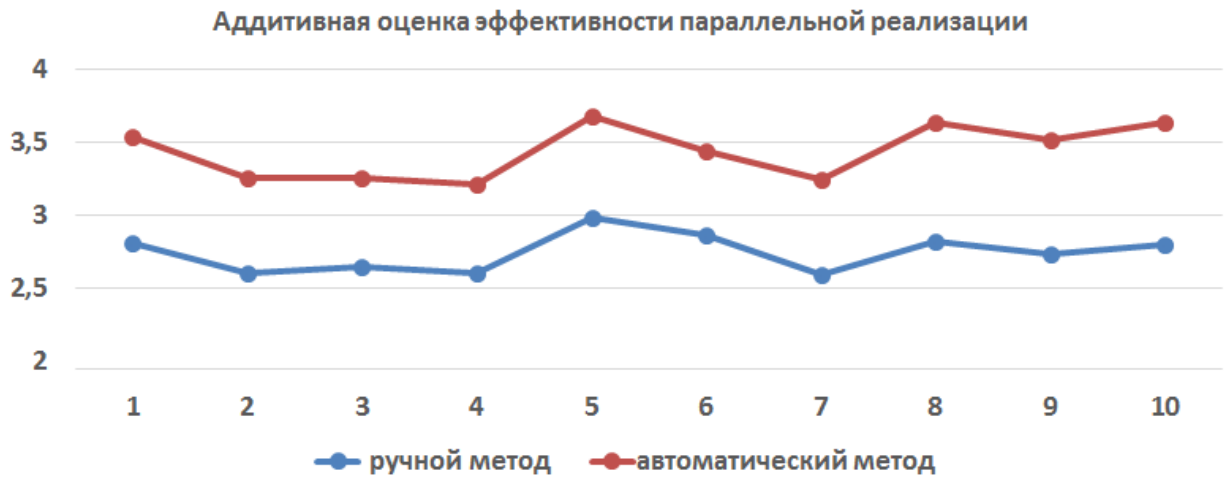


Рис 4.29. Адитивна оцінка ефективності паралельної реалізації

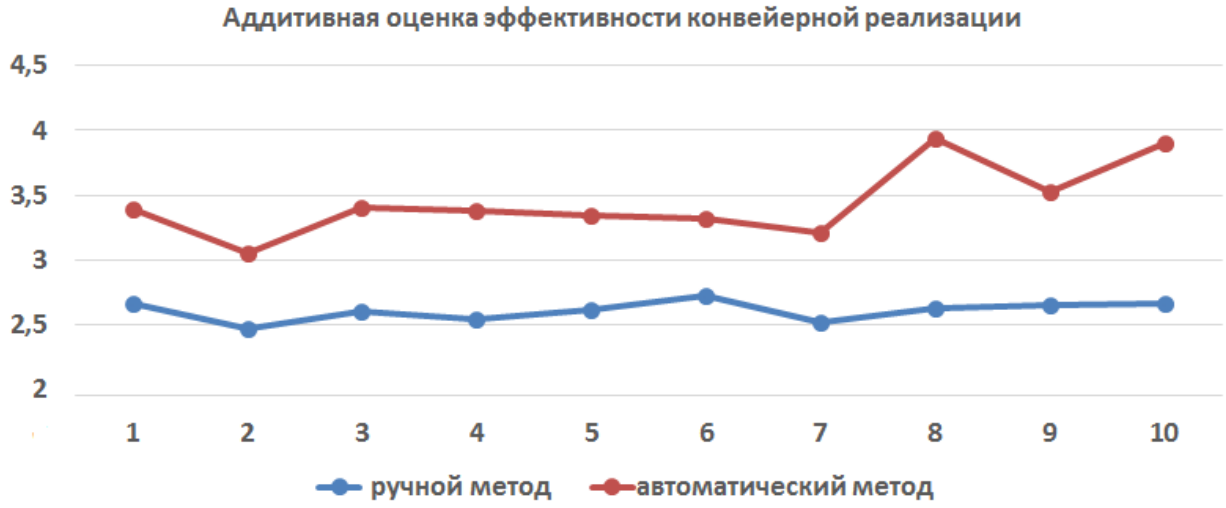


Рис 4.30. Адитивна оцінка ефективності конвеєрної реалізації

На рис.4.31- 4.33 представлені результати мультипликативной оцінки кожної реалізації.

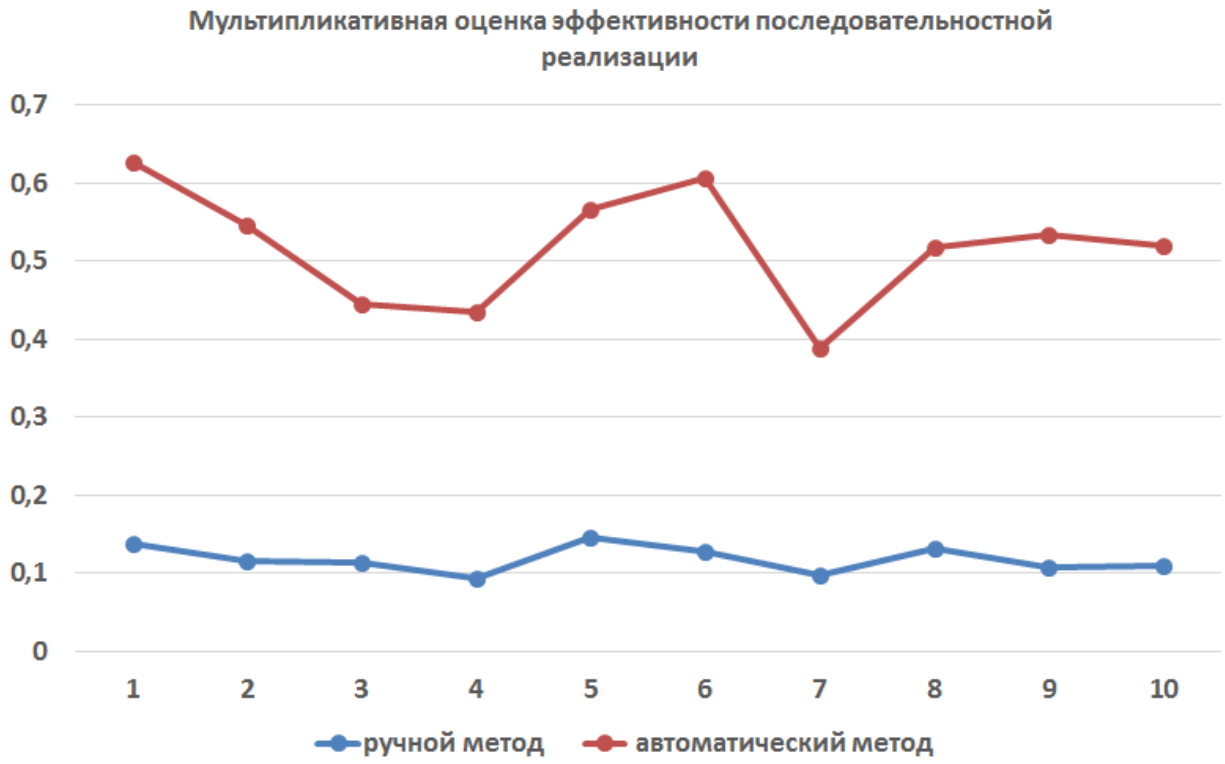


Рис. 4.31. Мультиплікативна оцінка ефективності послідовної реалізації

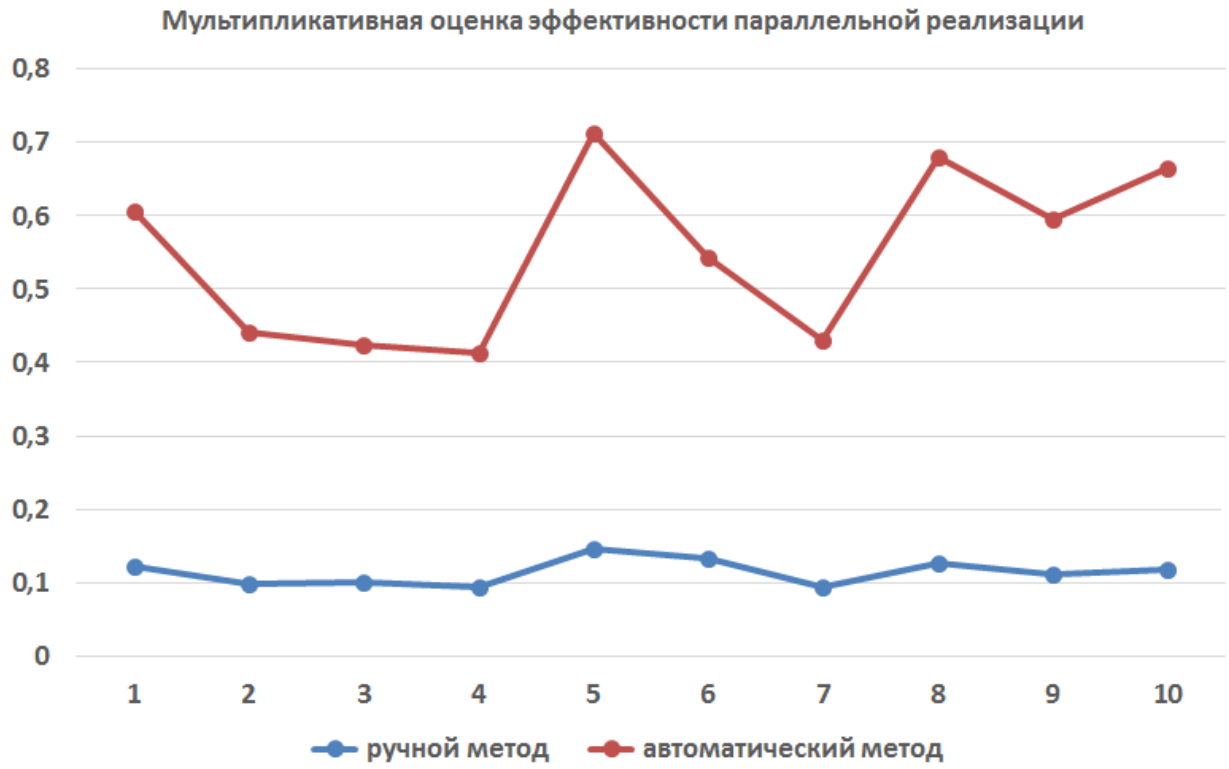


Рис. 4.32. Мультиплікативна оцінка ефективності паралельної реалізації

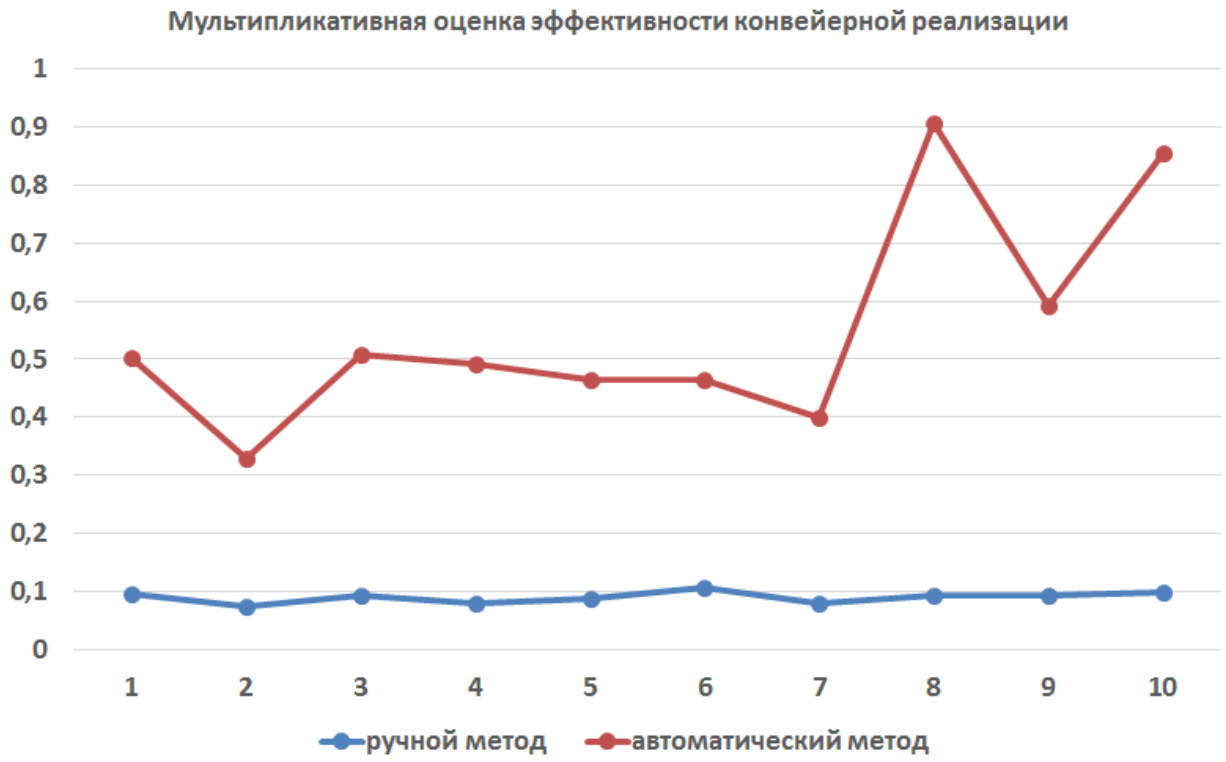


Рис. 4.33. Мультиплікативна оцінка ефективності конвеєрної реалізації

В результаті дослідження отримали, що при адитивній оцінці автоматичний метод краще в 1,301292012 разів (рис. 4.34).

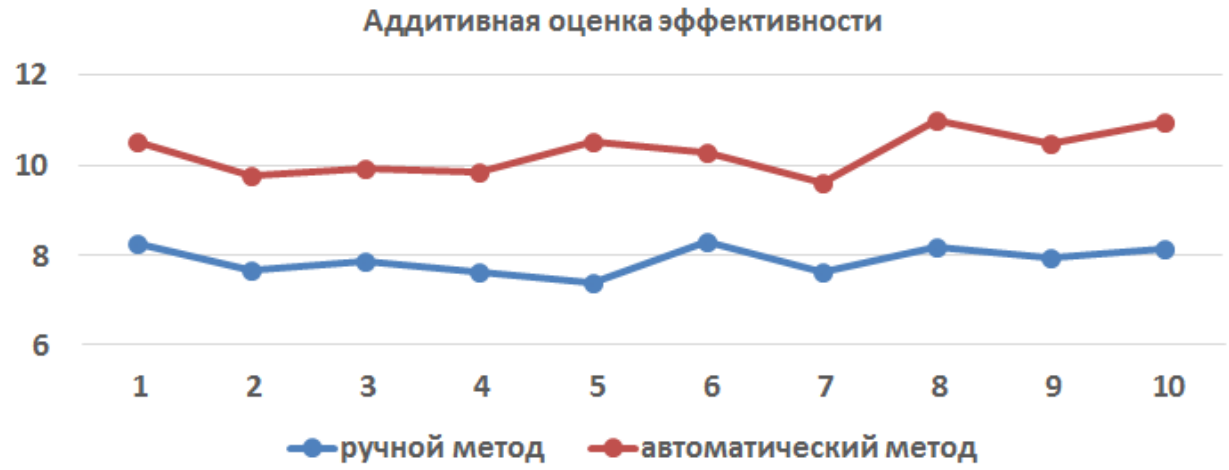


Рис. 4.34. Адитивна оцінка ефективності

При мультипликативній оцінці автоматичний метод краще в 133,2182966 раз. На рис. 4.35 видно, що значення ручного методу практично наближаються до нуля, в той час як в автоматичному методі значення наближаються до 1. Це свідчить про те, що автоматичний метод дійсно ефективніше.

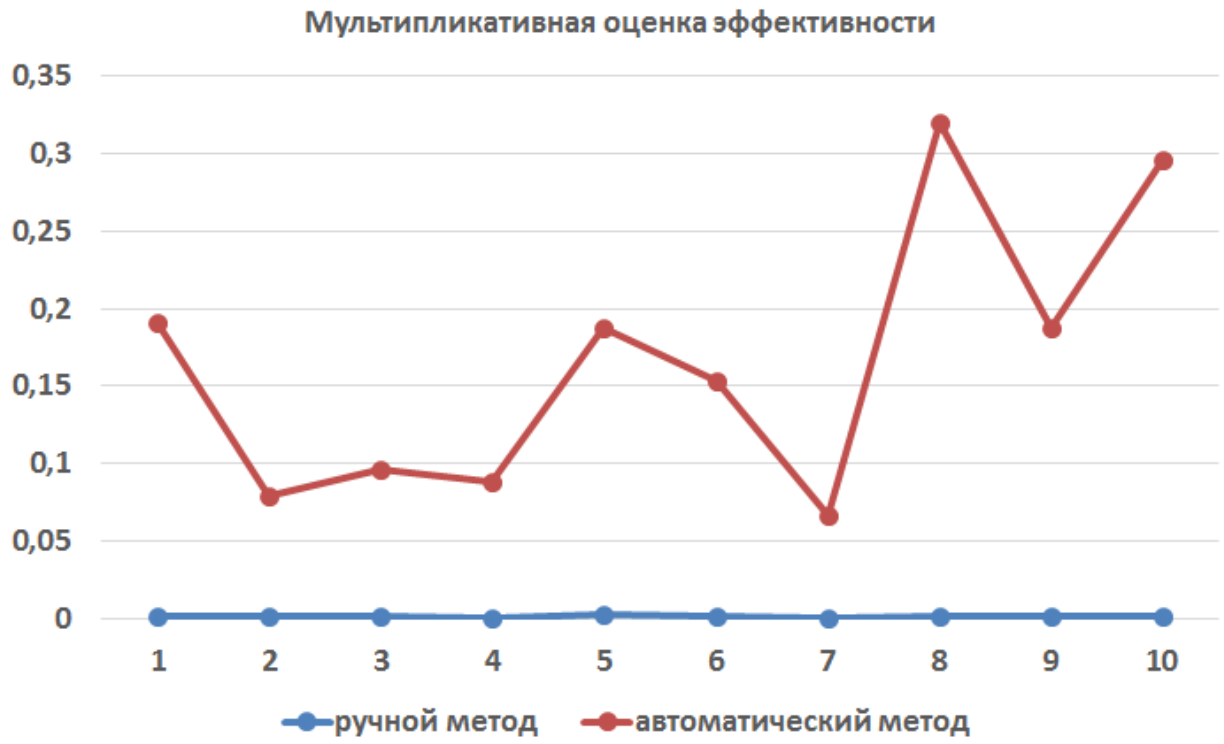


Рис. 4.35 Мультиплікативна оцінка ефективності

На рис. 4.36 наведено архітектуру дисертаційного дослідження, яка містить компоненти, відображені у вигляді моделей, методів і алгоритмів:

1) визначення специфікації системи;

2) парсерінг опису з мови SystemC у мову VHDL;

3) паралельний опис проекту мовами VHDL, Verilog;

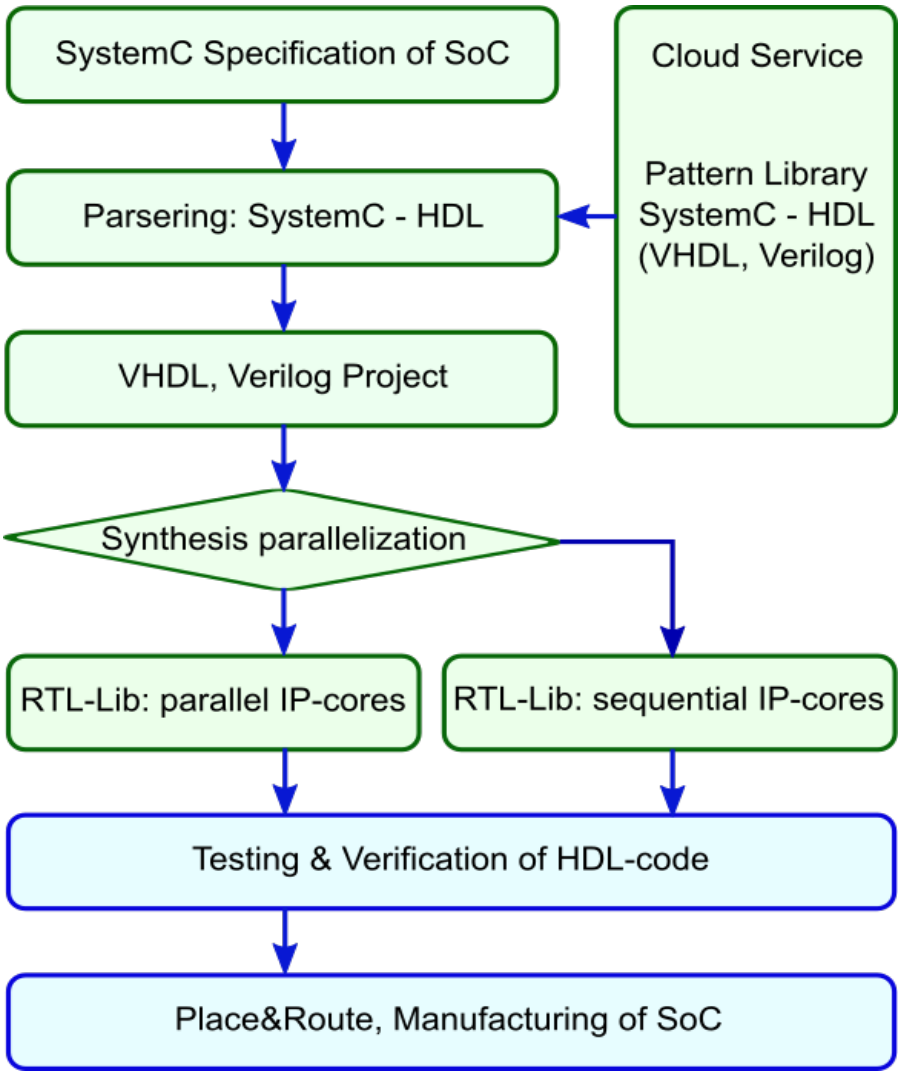
4) диверсифікація проектування цифрової системи на кристалі;

5) тестування та верифікація HDL-коду;

6) розміщення, трасування та виробництво SoC;

7) бібліотеки IP-cores для паралельного та послідовностного проектування;

8) імплементація архітектури та методів у хмарний сервіс.



ь

Рис. 4.36. Архітектура проектування SoC

4.10 Архітектури для System-C-driven Design Computing

Архітектури для design and test computing повинні бути орієнтовані на використання стандартів IEEE 11.49, 1500, 1687, які підтримують внутрішні формати структур даних, специфікації існуючих мов опису апаратури, методів синтезу, тестування, верифікації, аналізу, діагностування, виробництва та відновлення працездатності в режимі реального часу при відмові компонентів. Крім того, необхідно орієнтувати результати наукових досліджень на технології SoC design and test від провідних компаній планети: Synopsys, Cadence, Mentor, Aldec, ARM, які задають комп'ютингову моду і показують вченим актуальні невирішені проблеми [98-100].

Домінуючою архітектурою для TLM Design є замкнута в цикл система, заснована на використанні бібліотек нових та існуючих рішень, що створюють функціонально і timing-коректні рішення (рис. 4.37). В цьому випадку можна використовувати технологію, яка укладається в чотири рівні верифікації проекту цифрового виробу, мета якої – усунути всі можливі помилки на ранніх стадіях синтезу (рис. 4.38). Більш укрупнено схема процесу верифікації може бути представлена графом, який створює причинно-наслідковий зв'язок між стадіями проходження проекту, а також визначає основні відмінності між верифікацією і валідацією (рис. 4.39).

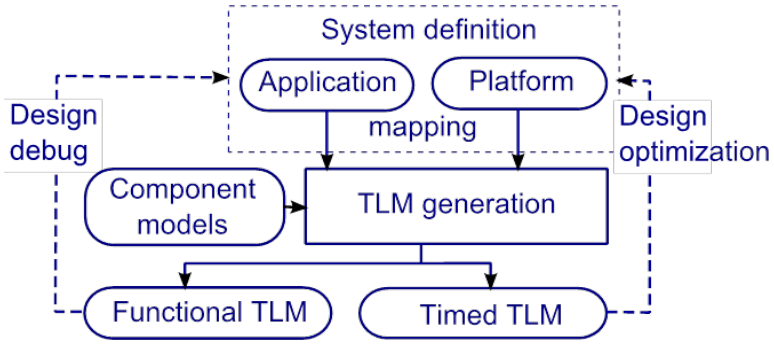


Рис. 4.37. Архітектура TLM Design

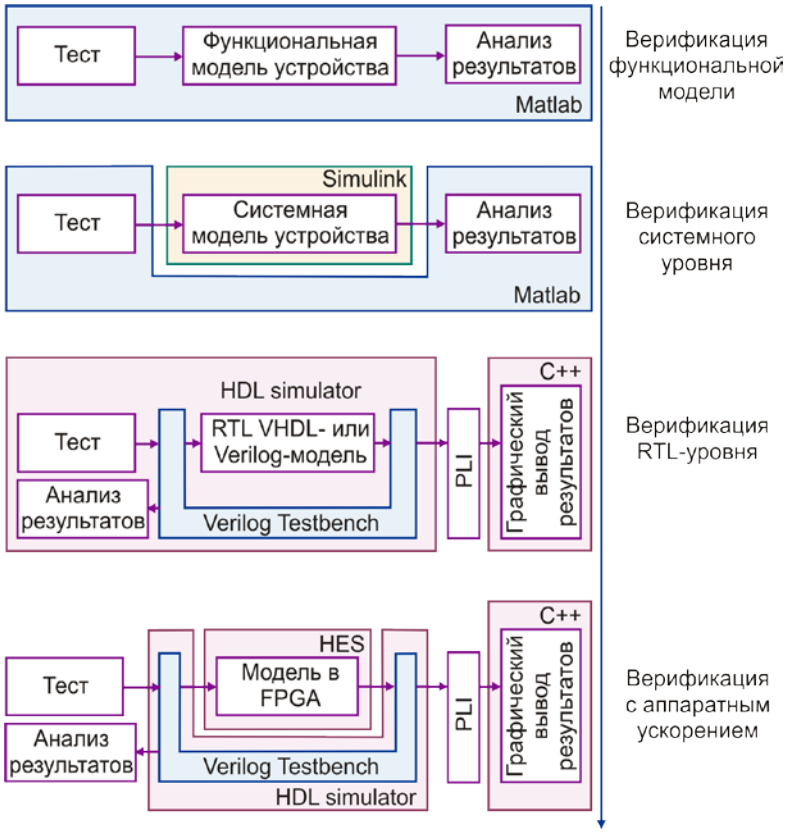


Рис. 4.38. Рівні верифікації TLM Design

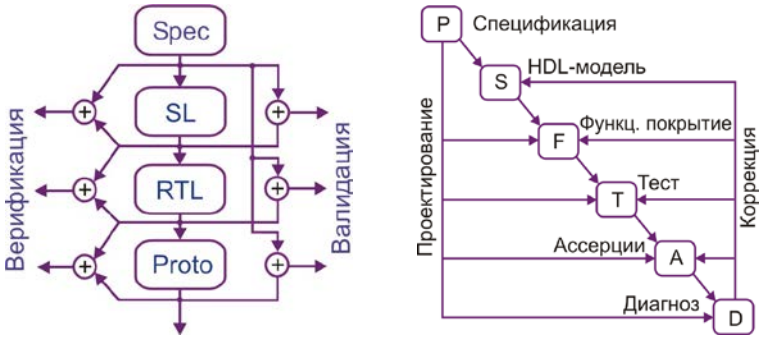


Рис. 4.39. Графи валідації та верифікації TLM Design

Слід зауважити, що технології проектування стають все більш симетричними щодо імплементації в апаратні або програмні рішення (рис. 4.40). Питання полягає лише в тому, який продукт бажає отримати замовник щодо продуктивності, ремонтопридатності і якості виробу. Однак завжди враховується параметр time-to-market, який є домінуючим при визначенні економічної стратегії Design Computing (рис. 4.41).

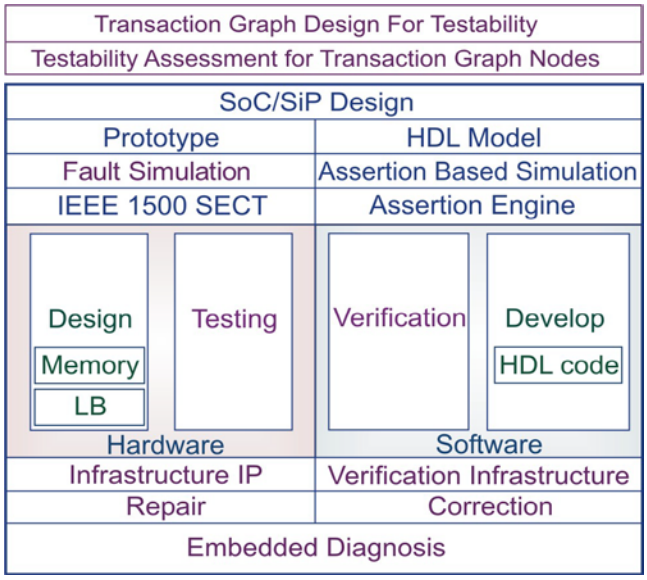


Рис. 4.40. Архітектура для HW / SW TLM Design

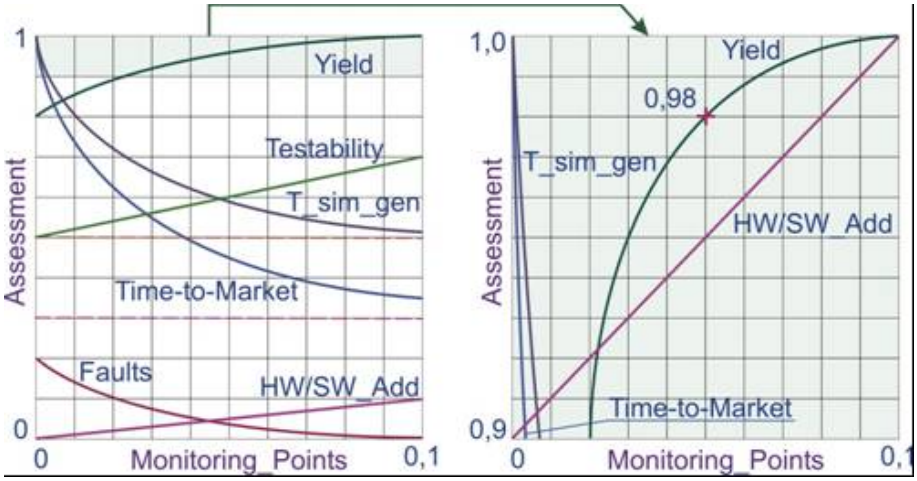


Рис. 4.41. Time-to-market для HW / SW TLM Design

Згідно з існуючими стандартами будь-яка Computing Design архітектура повинна містити компілятори з мов опису апаратури, включаючи System C, System Verilog, C++ (рис.4.42). Це дає можливість диверсифікувати процес створення цифрової системи, а також підвищити якість процесу проектування за рахунок паралельного аналізу і синтезу декількох прототипів і подальшого метричного вибору кращого з них (рис. 4.43).

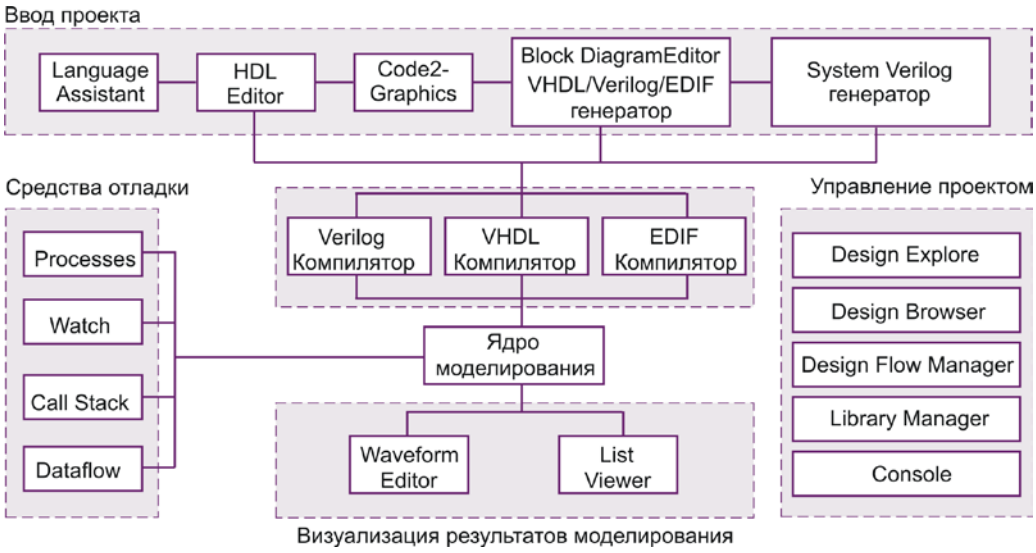


Рис. 4.42. Архітектура компілятора для HW / SW TLM Design

$$Q = Q(\overline{\square((Q \square Q_i) \square Q)}) \square Q_i(\square((Q \square Q_i) \square Q));$$
$$Y = \square((Q \square Q_i) \square Q);$$
$$Q = Q\overline{Y} \square Q_i Y.$$

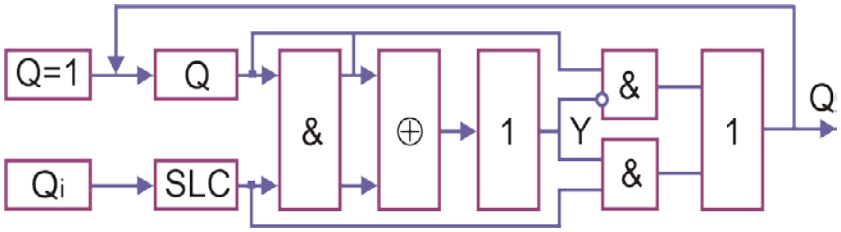


Рис. 4.43. Секвенсор для вибору рішення HW / SW TLM Design

Економіка технологій проектування сьогодні змушує всіх вендорів відмовлятися від пакетів дискет або флеш-пам'яті і приходити тільки до рішень хмарного комп'ютингу, який стає потенційно доступним всім жителям планети (рис. 4.44). Крім того, створення нового мікросервіса компьютінгового проектування забезпечує його просування в галузі SoC design and test, включаючи покупку даного сервісу, якщо він цікавий провідними компаніям.

Таким чином, хмарний сервіс-комп'ютинг (рис. 4.45) для вирішення завдань синтезу та аналізу, тестування і верифікації, як online-доступна і економічно вигідна технологія є найбільш конкурентоспроможною кібер-укладом, який буде домінувати в світі в найближчі 10 років.

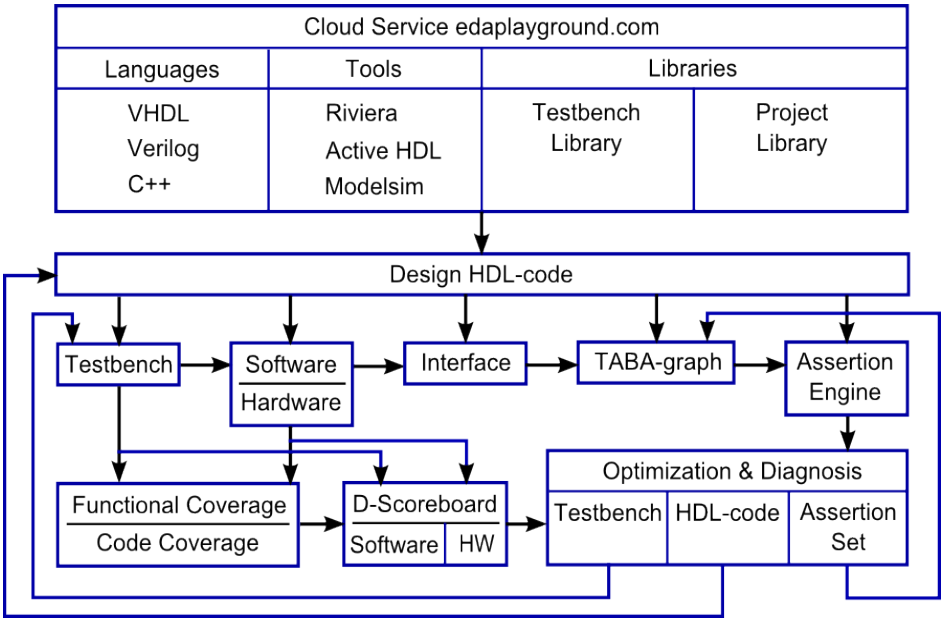


Рис. 4.44. Cloud-driven HW / SW TLM Design

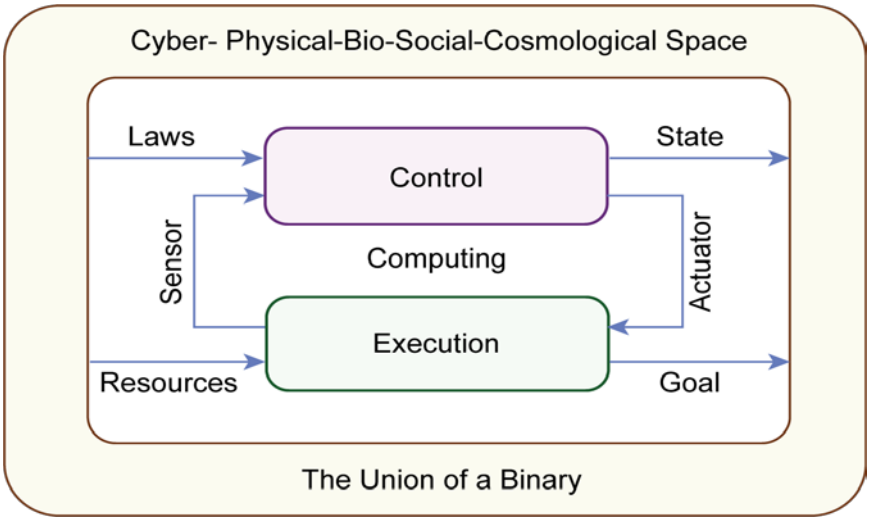


Рис. 4.45. Cloud-driven Computing for HW / SW TLM Design

Завдання вчених в нових незалежних державах – опанувати існуючу кіберкультуру SoC design and test від провідних компаній планети з метою подальшої пропозиції інноваційних технологічних рішень в межах існуючих стандартів за описом проектів засобами мов, включаючи System C, System Verilog. Все, що стосується хмарно-програмної імплементації інноваційних ідей, моделей, методів і алгоритмів, то ця область є доступною для кожного вченого України.

4.11 Висновки до розділу 4

Розроблені програмно-апаратні реалізації моделей, методів і структур даних для проектування цифрових систем на кристалах, які включають процедури створення специфікації, синтезу, тестування, моделювання та верифікації на основі запропонованої інфраструктури, що містять промислові засоби компаній Aldec і Xilinx. Розглянуті питання тестування розроблених програмних продуктів на реальних цифрових проектах створення IP-Core як примітивів для реалізації цифрових систем на кристалах.

При цьому досягнута мета – розробити й подати веріфіковані інфраструктурні модулі проектування цифрових систем на кристалах, які характеризуються паралельним виконанням мультиверсного синтезу функціональності, що забезпечує істотне зменшення часу створення проекту в умовах обмеження на апаратні витрати.

Вирішені наступні завдання:

1. Розроблено і описано метод мультиверсного синтезу керуючих і операційних автоматів в заданій інфраструктурі проектування, орієнтованих на архітектурні рішення в метриці, що мінімізує час виконання функціональності за рахунок розпаралелювання операцій при обмеженні на апаратні витрати.

2. Виконана програмна реалізація моделей і методів мультіверсної розробки операційних пристроїв в рамках інтегрованої системи проектування функціональних і архітектурних рішень SoC на основі використання продуктів верифікації та синтезу компаній Aldec і Xilinx.

3. Здійснено тестування і верифікація програмних модулів інфрастурктури проектування цифрових систем на кристалах, а також визначення ефективності запропонованих моделей, методів і структур даних при створенні реальних компонентів цифрових виробів.

4. Приклади реалізації цифрових блоків, а також коди програмних модулів інфраструктури синтезу і верифікації компонентів цифрових систем на кристалах представлені в додатку В.

ВИСНОВОК

В результаті виконання досліджень за темою дисертаційної роботи були отримані наступні основні результати.

Сутність ринково-орієнтованого науково-технічного дослідження визначена як мультіверсне проектування архітектури цифрового виробу на основі заданої специфікації в середовищі SystemC (C ++) і автоматичному виборі синтезованих функціональних структур з метою істотного зменшення часу створення проекту і підвищення виходу придатної продукції за рахунок паралельного синтезу і верифікації архітектурних рішень системного рівня відповідно до запропонованої метрики.

Досягнута *мета дослідження* – істотне зменшення часу проектування обчислювальних архітектур і підвищення якості цифрових виробів *шляхом* мультіверсного синтезу структури цифрового виробу на основі заданої специфікації в середовищі SystemC (C++) і автоматичному підборі функціональних компонентів *за рахунок* паралельного синтезу і верифікації архітектурних рішень системного рівня відповідно із запропонованою метрикою.

Проведені науково-технологічні дослідження в рамках дисертаційної роботи характеризуються успішним вирішенням актуальної науково-практичної задачі мультиверсного проектування архітектури цифрового виробу на основі заданої специфікації в середовищі SystemC (C++) і автоматичного вибору синтезованих функціональних структур з метою істотного зменшення часу створення проекту і підвищення виходу придатної продукції за рахунок паралельного синтезу та верифікації архітектурних рішень системного рівня відповідно до запропонованої метрики. Основна інноваційна ідея – паралельний автоматичний синтез квазіоптимальної архітектури відповідно до запропонованої специфікації і метрики на основі підбору синтезованих функціональних структур.

Автором одержано такі наукові та практичні результати:

1. **Новий метод** синтезу інтерфейсних структур і протоколів виконання транзакцій RT-рівня на основі аналізу специфікації SoC-архітектури системного рівня, який характеризується використанням двобічної стандартної шини Wishbone обміну даними між функціональними модулями. Здійснено синтез типових структур мовних конструкцій в моделі RT-рівня і їх оцінку з позиції швидкодії та апаратних витрат, які використовуються як бібліотечні описи для вибору оптимальних рішень в процесі створення обчислювальних пристроїв.

2. **Новий метод** синтезу RTL-моделей функціональностей, який характеризується одназначним перетворенням C++ і SystemC-описів цифрових блоків системного рівня в алгоритми і структури даних автоматної моделі Мура, що задана синтезованою підмножиною мовних конструкцій VHDL. Створено інфраструктуру проектування і верифікації компонентів цифрових систем на кристалах з метою перевірки алгоритмів перетворення специфікації з системного рівня на рівень RTL-опису яка дає можливість істотно зменшити час виконання всіх процесів проектування, тестування і верифікації.

3. **Удосконалені структури** даних для опису функціональних примітивів системного рівня, орієнтованих на використання семантичних і синтаксичних конструкцій мови C++ і SystemC, які відрізняються забезпеченням умов для виконання паралельного синтезу і верифікації архітектурних рішень. Розроблено і протестовано програмні модулі, що реалізують векторні і спискові моделі опису функціональних примітивів, а також інфраструктура для реалізації методів мультіверсної розробки операційних пристроїв в рамках інтегрованої системи проектування функціональних і архітектурних рішень SoC на основі використання продуктів верифікації та синтезу компаній Aldec і Xilinx.

4. **Удосконалений метод** мультиверсного синтезу керуючих і операційних автоматів в заданій інфраструктурі проектування, орієнтованих

на архітектурні рішення в метриці, що мінімізує час виконання функціональності за рахунок розпаралелювання операцій при обмеженні на апаратні витрати. Виконана програмна реалізація моделей і методів мультіверсної розробки операційних пристроїв в рамках інтегрованої системи проектування функціональних і архітектурних рішень SoC на основі використання продуктів верифікації та синтезу компаній Aldec і Xilinx.

Практична реалізація полягає в розробці локальних і серверних програм, розміщених на хмарних сервісах Amazon Web Services. Система виконана за мікросервісною архітектурою, яка має властивості масштабованості, надійності, можливість розробляти та оновлювати модулі системи незалежно один від одного. Клієнтський додаток реалізовано на мові C++, він має графічний інтерфейс користувача з підсвічуванням синтаксису. Дані експерименту були отримані в результаті роботи 10 інженерів над одним проектом. Оцінювалися такі характеристики: часові витрати на проектування, швидкодія, енергоспоживання, площа на кристалі. Було вибрано три найпоширеніші реалізації: послідовна, паралельна, конвеєрна. В результаті експерименту отримали, що в разі адитивної оцінки автоматичний метод краще в 1,3 рази. При мультипликативній оцінці автоматичний метод краще в 133,2 рази. В роботі показано, що значення ручного методу практично наближаються до нуля, в той час як в автоматичному методі значення наближаються до 1. Це доводить, що мультиверсний метод дійсно ефективніше.

БІБЛІОГРАФІЯ

1. Wadel, L. 1956. Simulation of Digital Filters on an Electronic Analog Computer. J. ACM 3, 1 (Jan. 1956), 16-21. DOI = http://doi.acm.org/10.1145/320815.320820

2. Leichner, GH 1957. Designing Computer Circuits with a Computer. J. ACM 4, 2 (Apr. 1957), 143-147. DOI = http://doi.acm.org/10.1145/320868.320872

3. Morris, EF and Wohr, TE 1958. Automatic implementation of computer logic. Commun. ACM 1, 5 (May. 1958), 14-20. DOI = http://doi.acm.org/10.1145/368819.368858

4. House, RW and Rado, T. 1963. On a Computer Program for Obtaining Irreducible Representations for Two-Level Multiple Input-Output Logical Systems. J. ACM 10, 1 (Jan. 1963), 48-77. DOI = http://doi.acm.org/10.1145/321150.321154

5. Katz, JH 1963. Optimizing bit-time computer simulation. Commun. ACM 6, 11 (Nov. 1963), 679-685. DOI = http://doi.acm.org/10.1145/368310.368393

6. Breuer, MA 1964. Techniques for the simulation of computer logic. Commun. ACM 7, 7 (Jul. 1964), 443-446. DOI = http://doi.acm.org/10.1145/364520.364577

7. Pistilli, PO 1964. Introduction. In Proceedings of the SHARE Design Automation Workshop DAC '64. ACM, New York, NY, 1.1-1.5. DOI = http://doi.acm.org/10.1145/800265.810737

8. Bolino, AC 1964. Economic and social aspects of automation. In Proceedings of the SHARE Design Automation Workshop DAC '64. ACM, New York, NY, 2.1-2.10. DOI = http://doi.acm.org/10.1145/800265.810738

9. Paul, FG 1964. Design automation effects on the organization. In Proceedings of the SHARE Design Automation Workshop DAC '64. ACM, New York, NY, 3.1-3.14. DOI = http://doi.acm.org/10.1145/800265.810739

10. Spitalny, A., Mann, WS, and Hartstein, G. 1965. Computer aided design of integrated circuits. In Proceedings of the SHARE Design Automation Project

DAC '65. ACM, New York, NY, 10.1-10.33. DOI = http://doi.acm.org/10.1145/800266.810763

11. McClure, RM 1965. A Programming Language for Simulating Digital Systems. J. ACM 12, 1 (Jan. 1965), 14-22. DOI = http://doi.acm.org/10.1145/321250.321252

12. Ulrich, EG 1965. Programming languages for non-numeric processing-2: Time sequenced logical simulation based on circuit delay and selective tracing of active network paths. In Proceedings of the 1965 20th National Conference (Cleveland, Ohio, United States, August 24 - 26, 1965). L. Winner, Ed. ACM, New York, NY, 437-448. DOI = http://doi.acm.org/10.1145/800197.806065

13. Pickrell, WE 1965. A laminate design system. In Proceedings of the SHARE Design Automation Project DAC '65. ACM, New York, NY, 8.1-8.11. DOI = http://doi.acm.org/10.1145/800266.810761

14. Gordon E. Moore, Cramming more Components onto Integrated Circuits, Electronics, 38 (8), April 9, 1965.

15. Mamelak, JS 1966. The Placement of Computer Logic Modules. J. ACM 13, 4 (Oct. 1966), 615-629. DOI = http://doi.acm.org/10.1145/321356.321370

16. Jirauch, D. 1966. Artificial intelligence in automated design. In Proceedings of the SHARE Design Automation Project DAC '66. ACM, New York, NY, 9.1-9.10. DOI = http://doi.acm.org/10.1145/800267.810781

17. Dewey, A. 1983. VHSIC hardware description (VHDL) development program. In Proceedings of the 20th Design Automation Conference (Miami Beach, Florida, United States, June 27 - 29, 1983). Annual ACM IEEE Design Automation Conference. IEEE Press, Piscataway, NJ, 625-628.

18. Gajski's y-chart, IEEE Computer, 1983, December.

19. Lis, JS and Gajski, DD 1989. VHDL synthesis using structured modeling. In Proceedings of the 26th ACM / IEEE Conference on Design Automation (Las Vegas, Nevada, United States, June 25 - 28, 1989). DAC '89. ACM, New York, NY, 606-609. DOI = http://doi.acm.org/10.1145/74382.74486

20. Camposano R. "From Behavior to Structure: High-Level Synthesis," IEEE Design and Test of Computers, vol. 7, no. 5, pp. 8-19, Sep / Oct, 1990.

21. Swan, S. 2006. SystemC transaction level models and RTL verification. In Proceedings of the 43rd Annual Conference on Design Automation (San Francisco, CA, USA, July 24 - 28, 2006). DAC '06. ACM, New York, NY, 90-92. DOI = http://doi.acm.org/10.1145/1146909.1146937

22. A Case Study: Quantitative Evaluation of C-Based High-Level Synthesis Systems, Omar Hammami, Zhoukun Wang, Virginie Fresse, and Dominique HouzetEURASIP Journal on Embedded SystemsVolume 2008 (2008), Article ID 685128, 13 pages

23. ISO / IEC 14882: 2003 Programming Language C ++

24. Stroustrup, Bjarne (2000). The C ++ Programming Language, Special Edition, Addison-Wesley, 46. ISBN 0-201-70073-5.

25. Hormati, A., Kudlur, M., Mahlke, S., Bacon, D., and Rabbah, R. 2008. Optimus: efficient realization of streaming applications on FPGAs. In Proceedings of the 2008 international Conference on Compilers, Architectures and Synthesis For Embedded Systems (Atlanta, GA, USA, October 19 - 24, 2008). CASES '08. ACM, New York, NY, 41-50. DOI = http://doi.acm.org/10.1145/1450095.1450105

26. Diederik Verkest, Johan Cockx, Freddy Potargent, Gjalt de Jong, Hugo de Man, "On the use of C ++ for System-on-Chip Design," wvlsi, pp.42, IEEE Computer Society Workshop on VLSI'99, 1999.

27. Verkest, D., Kunkel, J., and Schirrmeister, F. 2000. System level design using C ++. In Proceedings of the Conference on Design, Automation and Test in Europe (Paris, France, March 27 - 30, 2000). DATE '00. ACM, New York, NY, 74-83. DOI = http://doi.acm.org/10.1145/343647.343709

28. IEEE Std. 1666-2005 IEEE Standard SystemC Language Reference Manual

29. http://www.systemc.org/community/about_systemc/

30. Ghosh, A., Kunkel, J., and Liao, S. 1999. Hardware synthesis from C / C ++. In Proceedings of the Conference on Design, Automation and Test in Europe

(Munich, Germany). DATE '99. ACM, New York, NY, 82. DOI = http://doi.acm.org/10.1145/307418.307529

31. David C. Black, Jack Donovan. SystemC: From the Ground Up. Kluwer Academic Publishers, Boston, 2004. P. 263.

32. Bruschi, F. and Ferrandi, F. 2003. Synthesis of Complex Control Structures from Behavioral SystemC Models. In Proceedings of the Conference on Design, Automation and Test in Europe: Designers 'Forum - Volume 2 (March 03 - 07, 2003). Design, Automation, and Test in Europe. IEEE Computer Society, Washington, DC, 20112.

33. Herber, P., Fellmuth, J., and Glesner, S. 2008. Model checking SystemC designs using timed automata. In Proceedings of the 6th IEEE / ACM / IFIP international Conference on Hardware / Software Codesign and System Synthesis (Atlanta, GA, USA, October 19 - 24, 2008). CODES / ISSS '08. ACM, New York, NY, 131-136. DOI = http://doi.acm.org/10.1145/1450135.1450166

34. Blanc, N. and Kroening, D. 2008. Race analysis for SystemC using model checking. In Proceedings of the 2008 IEEE / ACM international Conference on Computer-Aided Design (San Jose, California, November 10 - 13, 2008). International Conference on Computer Aided Design. IEEE Press, Piscataway, NJ, 356-363.

35. Caldari, M., Conti, M., Coppola, M., Curaba, S., Pieralisi, L., and Turchetti, C. 2003. Transaction-Level Models for AMBA Bus Architecture Using SystemC 2.0. In Proceedings of the Conference on Design, Automation and Test in Europe: Designers 'Forum - Volume 2 (March 03 - 07, 2003). Design, Automation, and Test in Europe. IEEE Computer Society, Washington, DC, 20026.

36. http://www.arm.com/products/solutions/AMBAHomePage.html

37. Grimpe, E. and Oppenheimer, F. 2003. Extending the SystemC synthesis subset by object-oriented features. In Proceedings of the 1st IEEE / ACM / IFIP international Conference on Hardware / Software Codesign and System Synthesis (Newport Beach, CA, USA, October 01 - 03, 2003). CODES + ISSS '03. ACM, New York, NY, 25-30. DOI = http://doi.acm.org/10.1145/944645.944652

38. Grimpe, E .; Oppenheimer, F., "Object-oriented high level synthesis based on SystemC," Electronics, Circuits and Systems, 2001. ICECS 2001. The 8th IEEE International Conference on, vol.1, no., Pp.529-534 vol.1 , 2001URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=957798&isnumber=20704

39. http://www.systemc.org/downloads/drafts_review/

40. http://www.cecs.uci.edu/~specc/reference/

41. De Micheli, G. 1999. Hardware synthesis from C / C ++ models. In Proceedings of the Conference on Design, Automation and Test in Europe (Munich, Germany). DATE '99. ACM, New York, NY, 80. DOI = http://doi.acm.org/10.1145/307418.307527

42.http: //www.cadence.com/products/sd/silicon_compiler/pages/default.aspx

43. http://www.synopsys.com/products/sls/system_studio/system_studio.html

44. http://www.mentor.com/products/esl/high_level_synthesis/catapult_synthesis/

45. Catapult C Synthesis Datasheet, Mentor Graphics, 2008. http://www.mentor.com/products/esl/high_level_synthesis/catapult_synthesis/upload/Catapult_DS_pdf.pdf

46. eXCite Professional Optimization for Challenging Design Requirements, Datasheet, Y Exploration Inc. http://www.yxi.com/Library/excite_pro_datasheet_6_3_04.pdf

47. Forte Design Systems Datasheet, 2008, http://www.forteds.com/products/cynthesizer_datasheet_2008.pdf

48. Haubelt, C., Schlichter, T., Keinert, J., and Meredith, M. 2008. SystemCoDesigner: automatic design space exploration and rapid prototyping from behavioral models. In Proceedings of the 45th Annual Conference on Design Automation (Anaheim, California, June 08 - 13, 2008). DAC '08. ACM, New York, NY, 580-585. DOI = http://doi.acm.org/10.1145/1391469.1391616

49. http://www.systemcrafter.com/

50. Sumit Gupta, Nikil Dutt, Rajesh Gupta, Alex Nicolau, "SPARK: A High-Lev l Synthesis Framework For Applying Parallelizing Compiler Transformations," VLSI Design, International Conference on, vol. 0, no. 0, pp. 461, 16th International Conference on VLSI Design, 2003.

51. http://www.necst.co.jp/product/cwb/english/

52. Gerstlauer, A., Peng, J., Shin, D., Gajski, D., Nakamura, A., Araki, D., and Nishihara, Y. 2008. Specify-explore-refine (SER): from specification to implementation. In Proceedings of the 45th Annual Conference on Design Automation (Anaheim, California, June 08 - 13, 2008). DAC '08. ACM, New York, NY, 586-591. DOI = http://doi.acm.org/10.1145/1391469.1391617

53. Guo, Z., Najjar, W., and Buyukkurt, B. 2008. Efficient hardware code generation for FPGAs. ACM. Trans. Architec. Code Optim. 5, 1, Article 6 (May 2008), 26 pages. DOI = 10.1145 / 1369396.1369402 http://doi.acm.org/10.1145/1369396.1369402

54. Schreiber, R., Aditya, S., Mahlke, S., Kathail, V., Rau, BR, Cronquist, D., and Sivaraman, M. 2002. PICO-NPA: High-Level Synthesis of Nonprogrammable Hardware Accelerators . J. VLSI Signal Process. Syst. 31, 2 (Jun. 2002), 127-142. DOI = http://dx.doi.org/10.1023/A:1015341305426

55. Aditya, S., Rau, BR, and Kathail, V. 1999. Automatic Architectural Synthesis of VLIW and EPIC Processors. InProceedings of the 12th international Symposium on System Synthesis (November 01 - 04, 1999). International Symposium on Systems Synthesis. IEEE Computer Society, Washington, DC, 107.

56. http://cebatech.com/c2r_compiler

57. Grant Martin, Gary Smith, "High-Level Synthesis: Past, Present, and Future," IEEE Design and Test of Computers, vol. 26, no. 4, pp. 18-25, July / Aug. 2009 doi: 10.1109 / MDT.2009.83.

58. Andres Takach. High-Level Synthesis: Status, Trends, and Future Directions // IEEE Design & Test. Year: 2016, Volume: 33, Issue: 3. Pages: 116 - 124,

59. M. Dashtbani, A. Rajabzadeh and M. Asghari, "High Level Synthesis as a service," *2015 5th International Conference on Computer and Knowledge Engineering (ICCKE)*, Mashhad, 2015, pp. 331-336.

60. http://www.mentor.com/products/esl/blog/post/gary-smith-s-esl-2009-market-trends-dea5c96c-e9b7-4a8a-b6ae-6b8bef4abe05

61. Nikil D. Dutt, Daniel D. Gajski, "Design Synthesis and Silicon Compilation," IEEE Design and Test of Computers, vol. 7, no. 6, pp. 8-23, Nov / Dec, 1990.

62. Raj, VK, Pangrle, BM, and Gajski, DD 1984. Microprocessor synthesis. In Proceedings of the 21st Conference on Design Automation (Albuquerque, New Mexico, United States, June 25 - 27, 1984). Annual ACM IEEE Design Automation Conference. IEEE Press, Piscataway, NJ, 676-678.

63. Smith, J. and De Micheli, G. 1998. Automated composition of hardware components. In Proceedings of the 35th Annual Conference on Design Automation (San Francisco, California, United States, June 15 - 19, 1998). DAC '98. ACM, New York, NY, 14-19. DOI = http://doi.acm.org/10.1145/277044.277048

64. Giulio Gorla, "Designing with Intellectual Property," wvlsi, pp.125, IEEE Computer Society Workshop on VLSI'99, 1999.

65. Kalavade, AP 1 995 System-Level Codesign of Mixed Hardware-Software Systems. Doctoral Thesis. UMI Order Number: AAI9621207., University of California, Berkeley.

66. Séméria, L. and Ghosh, A. 2000. Methodology for hardware / software co-verification in C / C ++ (short paper). In Proceedings of the 2000 Conference on Asia South Pacific Design Automation (Yokohama, Japan). ASP-DAC '00. ACM, New York, NY, 405-408. DOI = http://doi.acm.org/10.1145/368434.368712

67. Savoiu, N., Shukla, S., and Gupta, R. 2005. MTP: a Petri net-based framework for the analysis and transformation of SystemC designs. In Proceedings of the 2005 Workshop on Software and Compilers For Embedded Systems (Dallas, Texas, September 29 - October 01, 2005). SCOPES '05, vol. 136. ACM, New York, NY, 99-108. DOI = http://doi.acm.org/10.1145/1140389.1140400

68. Rajesh K. Gupta, Giovanni De Micheli, "Hardware-Software Cosynthesis for Digital Systems," IEEE Design and Test of Computers, vol. 10, no. 3, pp. 29-41, Jul., 1993

69. Dong-hyun Lee, Hoh Peter In, Keun Lee, Sooyong Park, Mike Hinchey, "A Survival Kit: Adaptive Hardware / Software Codesign Life-Cycle Model," Computer, vol. 42, no. 2, pp. 100-102, Feb. 2009 doi: 10.1109 / MC.2009.34

70. Gong, J., Gajski, DD, and Bakshi, S. 1997. Model refinement for hardware-software codesign. ACM Trans. Des. Autom. Electron. Syst. 2, 1 (Jan. 1997), 22-41. DOI = http://doi.acm.org/10.1145/250243.250247

71. Arató, P., Mann, Z. Á., And Orbán, A. 2005. Algorithmic aspects of hardware / software partitioning. ACM Trans. Des. Autom. Electron. Syst. 10, 1 (Jan. 2005), 136-156. DOI = http://doi.acm.org/10.1145/1044111.1044119

72. Klingauf, W. 2005. Systematic Transaction Level Modeling of Embedded Systems with SystemC. In Proceedings of the Conference on Design, Automation and Test in Europe - Volume 1 (March 07 - 11, 2005). Design, Automation, and Test in Europe. IEEE Computer Society, Washington, DC, 566-567. DOI = http://dx.doi.org/10.1109/DATE.2005.293

73. Narayan, S. and Gajski, DD 1995. Interfacing incompatible protocols using interface process generation. In Proceedings of the 32nd ACM / IEEE Conference on Design Automation (San Francisco, California, United States, June 12 - 16, 1995). DAC '95. ACM, New York, NY, 468-473. DOI = http://doi.acm.org/10.1145/217474.217572

74. Chung, K., Gupta, RK, and Liu, CL 1996. An algorithm for synthesis of system-level interface circuits. In Proceedings of the 1996 IEEE / ACM international Conference on Computer-Aided Design (San Jose, California, United States, November 10 - 14, 1996). International Conference on Computer Aided Design. IEEE Computer Society, Washington, DC, 442-447.

75. Borriello, G., Lavagno, L., and Ortega, RB 1998. Interface synthesis: a vertical slice from digital logic to software components. In Proceedings of the 1 998 IEEE / ACM international Conference on Computer-Aided Design (San Jose,

California, United States, November 08 - 12, 1998). ICCAD '98. ACM, New York, NY, 693-695. DOI = http://doi.acm.org/10.1145/288548.289119

76. Hines, K. and Borriello, G. 1997. Dynamic communication models in embedded system co-simulation. In Proceedings of the 34th Annual Conference on Design Automation (Anaheim, California, United States, June 09 - 13, 1997). DAC '97. ACM, New York, NY, 395-400. DOI = http://doi.acm.org/10.1145/266021.266178

77. Madsen, J. and Hald, B. 1995. An approach to interface synthesis. In Proceedings of the 8th international Symposium on System Synthesis (Cannes, France, September 13 - 15, 1995). ISSS '95. ACM, New York, NY, 16-21. DOI = http://doi.acm.org/10.1145/224486.224490

78. Yun, C., Kang, D., Bae, Y., Cho, H., and Jhang, K. 2008. Automatic interface synthesis based on the classification of interface protocols of IPs. In Proceedings of the 2008 Conference on Asia and South Pacific Design Automation (Seoul, Korea, January 21 - 24, 2008). with EDA Technofair Design Automation Conference Asia and South Pacific. IEEE Computer Society Press, Los Alamitos, CA, 589-594.

79. Smith, J. and De Micheli, G. 1998. Automated composition of hardware components. In Proceedings of the 35th Annual Conference on Design Automation (San Francisco, California, United States, June 15 - 19, 1998). DAC '98. ACM, New York, NY, 14-19. DOI = http://doi.acm.org/10.1145/277044.277048

80. Gupta, S., Dutt, N., Gupta, R., and Nicolau, A. 2004. Loop Shifting and Compaction for the High-Level Synthesis of Designs with Complex Control Flow. In Proceedings of the Conference on Design, Automation and Test in Europe - Volume 1 (February 16 - 20, 2004). Design, Automation, and Test in Europe. IEEE Computer Society, Washington, DC, 10114.

81. The Impact of Loop Unrolling on Controller Delay in High Level Synthesis [p. 391] S. Kurra, NK Singh and PR Panda Date 2008.

82. Sirowy, S., Stitt, G., and Vahid, F. 2008. C is for circuits: capturing FPGA circuits as sequential code for portability. InProceedings of the 16th interna-

tional ACM / SIGDA Symposium on Field Programmable Gate Arrays (Monterey, California, USA, February 24 - 26, 2008). FPGA '08. ACM, New York, NY, 117-126. DOI = http://doi.acm.org/10.1145/1344671.1344689

83. WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores Revision: B.3, Released: September 7, 2002 http://www.opencores.org/downloads/wbspec_b3.pdf

84. Daniel Åkerlund, Implementation of a 2x2 NoC with Wishbone interface. Master Thesis IMIT / LECS 2005-74. Laboratory of Electronics and Computer Systems, Royal Institute of Technology (KTH), 14th November 2005. P. 70.

85. Zorian, Y. 2002. Embedding infrastructure IP for SOC yield improvement. In Proceedings of the 39th Conference on Design Automation (New Orleans, Louisiana, USA, June 10 - 14, 2002). DAC '02. ACM, New York, NY, 709-712. DOI = http://doi.acm.org/10.1145/513918.514098

86. Abramovici, M., Stroud, C., and Emmert, M. 2002. Using embedded FPGAs for SoC yield improvement. In Proceedings of the 39th Conference on Design Automation (New Orleans, Louisiana, USA, June 10 - 14, 2002). DAC '02. ACM, New York, NY, 713-724. DOI = http://doi.acm.org/10.1145/513918.514099

87. Ogras, UY and Marculescu, R. 2008. Analysis and optimization of prediction-based flow control in networks-on-chip. ACM Trans. Des. Autom. Electron. Syst. 13, 1 (Jan. 2008), 1-28. DOI = http://doi.acm.org/10.1145/1297666.1297677

88. Егоров, Александр. Модели и методы совместной верификации проектируемых цифровых систем на кристаллах. Диссертация на соискание кандидата технических наук. Харьков : ХНУРЭ, 2006.

89. Бибило П. Н. Основы языка VHDL. Второе издание. Москва: Солон-Р, 2002. — 224 с.

90. Пат. 6 226 776 США, G 06 F 17/50. System for Converting Hardware Designs in High-Level Programming Language to Hardware Implementations / Yuri V. Panchul, Donald A. Soderman, Denis R. Coleman; заявитель и патенто-

обладатель Synetry Corporation (США); заявл. 16.09.1997, опубл. 01.05.2001. http://www.google.com/patents/about?id=BM4GAAAAEBAJ&dq=6226776

91. OR1200 OpenRISC Processor. http://opencores.org/ wiki1 / index.php? title = OR1K_CPU_Cores & oldid = 1404

92. Язык программирования C++. Международный стандарт ISO/IEC 14882. Второе издание, 15 октября 2003 г. American National Standards Institute, New York.

93. Jayaram Bhasker. A SystemC Primer. 2002

94. Лисяк В. В., Лисяк Н. К. Обзор европейских производителей программного обеспечения САПР РЭА // Известия ЮФУ. Технические науки. 2008. №9 С.81-86.

95. https://github.com/systemc/systemc-2.2.0

96. Баранов С.И. Синтез микропрограммных автоматов (граф-схемы и автоматы). Ленинград: Издательство 'Энергия'. Ленинградское отделение, 1979).

97. SystemC: From the Ground Up, Second Edition. Black, DC, Donovan, J., Bunton, B., Keist, A. Springer, 2010. Pages 279.

98. Хаханов В.І. Проектирование и тестирование цифровых систем на кристаллах / В.И. Хаханов, Е.И. Литвинова, О.А. Гузь. – Харьков: ХНУРЭ. – 2009. – 484 с.

99. Инфраструктура диагностирования программно-аппаратных систем / В. И. Хаханов, С. В. Чумаченко, О. А. Гузь, Е. И. Литвинова // Радіоелектроніка. Інформатика. Управління. – 2012. – № 1 (26). – С. 134–140.

100. Квантовые модели диагностирования цифровых систем / Багдади Аммар Авни Аббас, В. И. Хаханов, Е. И. Литвинова и др. // Радиоэлектроника и информатика. – № 2. – С. 35–43.

101. Хаханов В.И. Технология моделирования и синтеза тестов для сложных цифровых систем / В.И. Хаханов, К.В. Колесников, А.Н. Парфентий, И.В. Хаханова, В.И. Обризан, О.В. Мельникова // Радиоэлектроника и информатика. – 2003. – №1. – С. 70-78. (Входить до міжнародних наукомет-

ричних баз Index Copernicus, Google Scholar, OECSP, OAJI, Scholar Steer, SIS, Cyberleninka, CiteFactor, TIU Hannover, I2OR).

102. Hahanov V. Advanced Software Tools For Fault Simulation And Test Generation / V. Hahanov, A. Egorov, O. Melnikova, V. Obrizan, E. Kamenuka, O. Krapchunova, O. Guz // Радиоэлектроника и информатика. – 2003. – №3. – С. 77-81. (Входить до міжнародних наукометричних баз Index Copernicus, Google Scholar, OECSP, OAJI, Scholar Steer, SIS, Cyberleninka, CiteFactor, TIU Hannover, I2OR).

103. Obrizan V. A Method of High-Level Synthesis and Verification with SystemC Language / V. Obrizan // Радиоэлектроника и информатика. – 2010. – №4. – С. 47-50. (Входить до міжнародних наукометричних баз Index Copernicus, Google Scholar, OECSP, OAJI, Scholar Steer, SIS, Cyberleninka, CiteFactor, TIU Hannover, I2OR).

104. Obrizan V. Matrix-Model for Diagnosing SoC HDL-Code / V. Obrizan, I. Yemelyanov, V. Hahanov, E. Litvinova // Radioelektroniks and informatics. – 2013. – №1. – P.12-19. (Входить до міжнародних наукометричних баз Index Copernicus, Google Scholar, OECSP, OAJI, Scholar Steer, SIS, Cyberleninka, CiteFactor, TIU Hannover, I2OR).

105. Обризан В.И. Метрика для анализа Big Data / В.И. Обризан, А.С. Мищенко, В.И. Хаханов, Tamer Bani Amer // Радиоэлектроника и информатика. – 2014. – № 2. – С. 26-29. (Входить до міжнародних наукометричних баз Index Copernicus, Google Scholar, OECSP, OAJI, Scholar Steer, SIS, Cyberleninka, CiteFactor, TIU Hannover, I2OR).

106. Обризан В.И. Киберфизические системы как технологии киберуправления (аналитический обзор) / В.И. Обризан, А.С. Мищенко, В.И. Хаханов, И.В. Филиппенко // Радиоэлектроника и информатика. – 2014. – № 1. – С. 39-45. (Входить до міжнародних наукометричних баз Index Copernicus, Google Scholar, OECSP, OAJI, Scholar Steer, SIS, Cyberleninka, CiteFactor, TIU Hannover, I2OR).

107. Обризан В.И. Инфраструктура проектирования SOC для метода мультиверсного синтеза / В.И. Обризан // Радиоэлектроника и информатика. – 2016. – №2. – С. 48-60. (Входить до міжнародних наукометричних баз Index Copernicus, Google Scholar, OECSP, OAJI, Scholar Steer, SIS, Cyberleninka, CiteFactor, TIU Hannover, I2OR).

108. Hahanov V. High Performance Fault Simulation For Digital Systems / V. Hahanov, G. Krivoulya, I. Hahanova, O. Melnikova, V. Obrizan // International Scientific Journal of Computing. – 2003. – Vol. 2, Issue 2. – P. 114-121. (Входить до міжнародних наукометричних баз Index Copernicus, Norwegian Social Science Data Services (NSD), Google Scholar, Vernadsky National Library of Ukraine).

109. Hahanov V. Algebra-logical diagnosis model for SoC F-IP / V. Hahanov, V. Obrizan, E. Litvinova, Man K.L. // WSEAS Transactions on Circuits and Systems. – 2008. – No 7. – P. 708-717. (Входить до міжнародних наукометричних баз Scopus, Elsevier, Google Scholar).

110. Hahanov V. Embedded method of SoC diagnosis / V. Hahanov, E. Litvinova, V. Obrizan, W. Gharibi // Elektronika ir Elektrotechnika. – 2008. – No8. – P. 3-8. (Входить до міжнародних наукометричних баз Scopus, Thomson Reuters (ISI), Web of Knowledge Citation Databases, Science Citation Index Expanded (SCIE); Journal Citation Reports (JCR); INSPEC; VINITI; EBSCO Publishing).

111. Хаханов В.И. Обзор международного рынка электронных технологий / В.И. Хаханов, В.И. Обризан, О.В. Мельникова // Вестник НТУ «ХПИ». – Серия: Информатика и моделирование. – 2004. – Вып. 46. – С.111-115.

112. Хаханов В.И. Assert-метод верификации цифровых систем на основе стандарта IEEE 1500 SECT / В.И. Хаханов, В.В. Елисеев, В.И. Обризан // АСУ и приборы автоматики. – 2005. – Вып. 132. – С. 93–105. (Входить до міжнародних наукометричних баз Google Scholar, Cyberleninka).

113. Хаханов В.И. Иерархическое тестирование программно-технических комплексов / В.И. Хаханов, В.В. Елисеев, В.И. Обризан // АСУ и приборы автоматики. – Харьков, 2006. – Вып. 134. – С. 93–102. (Входить до міжнародних наукометричних баз Google Scholar, Cyberleninka).

114. Михтонюк С.В. Архитектурная модель масштабируемой системы асинхронной обработки больших объемов данных / С.В. Михтонюк, Р.С. Хван, В.И. Обризан // АСУ и приборы автоматики. – 2008. – Вып. 142. – С. 13-17. (Входить до міжнародних наукометричних баз Google Scholar, Cyberleninka).

115. Хаханов В.И. MQT-автомат для анализа больших данных / В.И. Хаханов, В.И. Обризан, С.А. Зайченко, И.В. Хаханов // АСУ и приборы автоматики. – 2014. – Вып. 168. – С. 64-72. (Входить до міжнародних наукометричних баз Google Scholar, Cyberleninka).

116. Hahanov V.I. Sigetest – fault simulation and test generation for digital designs / V.I. Hahanov, D.M. Gorbunov, Y.V. Miroshnychenko, O.V. Melnikova, V.I. Obrizan, E.A. Kamenuka // Современные технологии проектирования систем на микросхемах программируемой логики. – 23 сентября 2003. – Харьков. – С. 50-53.

117. Hahanov V. High performance Fault Simulation for Digital Systems / V. Hahanov, O. Melnikova, V. Obrizan, I. Hahanova // Proc. of the Euromicro Symposium on Digital Systems Design. – Turkey. Belek-Antalya. – 2003. – P. 15-16. (Входить до міжнародних наукометричних баз Scopus, IEEE Xplore).

118. Мирошниченко Я.В. Система моделирования неисправностей для дискретных устройств / Я.В. Мирошниченко, О.В. Мельникова, В.И. Обризан / Материалы 7-го молодежного форума «Радиоэлектроника и молодежь в XXI веке». – Украина, Харьков: ХНУРЭ. – 2003. – С. 463.

119. Hahanov V.I. New Features of Deductive Fault Simulation / V.I. Hahanov, V.I. Obrizan, A.V. Kiyaszhenko, I.A. Pobezhenko // Proc. of the 2nd East-West Design and Test Workshop 2004. – September 23-26, 2004. – Alushta. – 2004. – P. 274-280.

120. Шабанов-Кушнаренко Ю.П. Параллелизм мозгоподобных вычислений / Ю.П. Шабанов-Кушнаренко, В.И. Обризан // Материалы 5-го Международного научно-практического семинара «Высокопроизводи-тельные параллельные вычисления на кластерных системах». – 22-25 ноября 2005. – Харьков. – С. 196- 203.

121. Hahanov V.I. High-performance deductive fault simulation method / V.I. Hahanov, I. Hahanova, V.I. Obrizan // Proc. of the 10th IEEE European Test Symposium. – May 22-25, 2005. – Estonia. – Tallinn, 2005. – P. 91-96. (Входить до міжнародних наукометричних баз Scopus, IEEE Xplore).

122. Hahanov V. Hierarchical Testing of Complex Digital Systems / V. Hahanov, V. Obrizan, V. Yeliseev, W. Ghribi // Proc. of the International Conference “Modern Problems of Radio Engineering, Telecommunications and Computer Science (TCSET'2006)”. – February 28 – March 4, 2006. – Slavske, Lviv, Ukraine. 2006. – P. 426-429. (Входить до міжнародних наукометричних баз Scopus, IEEE Xplore).

123. Hahanov V. Verification of digital system by a new asserting mechanism based on IEEE 1500 SECT standard / V. Hahanov, V. Obrizan, I. Hahanova, E. Fomina // Proc. of the International Conference MIXDES 2006. – June 22-24, 2006. – Gdunia, Poland. – 2006. – P. 544-548. (Входить до міжнародних наукометричних баз Scopus, IEEE Xplore).

124. Хаханов В.И. SIGETEST – система моделирования тестов проверки неисправностей цифровых устройств / В.И. Хаханов, В.И. Обризан, Я.В. Мирошниченко, О.В. Мельникова // Каталог аннотаций и разработок по материалам первого украинско-китайского форума «Наука – производство». – Харьков: ХНУРЭ, 2007. – С. 25–26.

125. Hahanov V. Hardware Simulation and Verification Technologies / V. Hahanov, A. Hahanova, V. Obrizan, W. Ghribi // Proc. of the IEEE East-West Design and Test Symposium. – September 7-10, 2007. – Yerevan, Armenia. – Yerevan, 2007. – P. 739-744.

126. Hahanov V. Technologies for hardware simulation and verification / V. Hahanov, A. Hahanova, V. Obrizan, K. Zaharov // Proc. of the International Conference Modern Problems of Radio Engineering, Telecommunications and Computer Science. – TCSET'2008. – February 19-23, 2008. – Slavske, Lviv, Ukraine. – 2008. – P. 560-564. (Входить до міжнародних наукометричних баз Scopus, IEEE Xplore).

127. Hahanov V. Testing challenges of SoC hardware-software components / V. Hahanov, V. Obrizan, S. Miroshnichenko, A. Gorobets // Proc. of the IEEE East-West Design and Test International Symposium. – October 9-12, 2008. – Lviv, Ukraine. – 2008. – P. 149-154. (Входить до міжнародних наукометричних баз Scopus, IEEE Xplore).

128. Obrizan V. A method for automatic generation of an RTL-interface from a C++ description / V. Obrizan // Proc. of the East-West Design & Test Symposium (EWDTS) 2010. – 17-20 Sept. 2010. – St. Petersburg, Russia. – P.186-189. (Входить до міжнародних наукометричних баз Scopus, IEEE Xplore).

129. Hahanova I.V. Transaction level model of embedded processor for vector-logical analysis / I.V. Hahanova, V. Obrizan, A. Adamov, D. Shcherbin // Proc. of the East-West Design & Test Symposium. – 27-30 Sept. 2013. – Rostov-on-Don, Russia. – 4 p. (Входить до міжнародних наукометричних баз Scopus, IEEE Xplore).

130. Hahanova Yu. Metric for Analyzing Big Data / Yu. Hahanova, I. Yemelyanov, V. Obrizan, D. Krulevska, M. Skorobogatiy, A. Hahanova // Матеріали XIII Міжнародної науково-технічної конференции CADSM 2015 «Досвід розробки та застосування приладнати-технологічних САПР в мікроелектроніці ». 24-27 лютого 2015. Львів – Поляна. – С.81-83. (Входить до міжнародних наукометричних баз Scopus, IEEE Xplore).

131. Обризан В.И. Мультиарендные облачные сервисы / В.И. Обризан, Ю.В. Ломова // Материалы XIX Международного молодежного форума «Радиоэлектроника и молодежь в XXI веке». – Украина, Харьков: ХНУРЭ. – 20-22 апреля 2015. – Ч. 5. – С.34-35.

132. Obrizan V. Multiversion parallel synthesis of digital structures based on SystemC specification / V.Obrizan; T. Soklakova // Proc. of the IEEE East-West Design & Test Symposium (EWDTS). – 2016 – Yerevan, Armenia. – 6p. (Входить до міжнародних наукометричних баз Scopus, IEEE Xplore).

133. Гайдук С.М. Сферический мультипроцессор PRUS для решения булевых уравнений / С.М. Гайдук, В.И. Хаханов, В.И. Обризан, Е.А. Каменюка // Радиоэлектроника и информатика. – Харьков, 2004. – № 4 (29). – С. 107–116. (Входить до міжнародних наукометричних баз Index Copernicus, Google Scholar, OECSP, OAJI, Scholar Steer, SIS, Cyberleninka, CiteFactor, TIU Hannover, I2OR).

134. Hyduke S. PRUS – Spherical Multiprocessor for Computation of Boolean equations / S. Hyduke, V.I. Hahanov, V.I. Obrizan, Wade Ghribi // Proceedings of the 8th International Conference CADSM 2005. – Ukraine. – Lviv, 2005. – P. 41-48.

135. Hyduke S.M. PRUS – Processor Network for Digital Circuit Implementation / S.M. Hyduke, V.I. Hahanov, V.I. Obrizan, O. Guz // Proc. of the 8th Euromicro Conference on Digital System Design. – August 30 – September 3, 2005. – Porto, Portugal. – 2005. – P. 239-242. (Входить до міжнародних наукометричних баз Scopus, IEEE Xplore).

136. Шевченко А.А. Математические модели описания потенциально опасных объектов газотранспортной промышлености / А.А. Шевченко, О.А. Довгошея, В.И. Обризан, Д.Н. Красильников // Материалы 2-го Международного радиоэлектронного форума «Прикладная радиоэлектроника». – 19-23 сентября 2005. – Украина. – С. 272- 276.

ДОДАТОК А

Документи, що підтверджують впровадження

«ЗАТВЕРДЖУЮ»
Перший проректор ХНУРЕ
_____ Ключник І.І.
"09" 03 2016 р.

С П Р А В К А

про впровадження в навчальний процес ХНУРЕ результатів дисертаційної роботи Обрізана Володимира Ігоровича «Мультиверсний паралельний синтез цифрових структур на основі SystemC специфікації», представленої на здобуття наукового ступеня кандидата технічних наук за спеціальністю 05.13.05 – комп'ютерні системи та компоненти

Комісія у складі: зав. каф. АПОТ проф. Чумаченко С.В., доц. каф. АПОТ Шкиля О.С., проф. каф. АПОТ Литвинова Є.І. розглянула матеріали дисертаційної роботи Обрізана В.І., які використовуються в навчальному процесі кафедри АПОТ ХНУРЕ у 2015/2016 навчальному році, і прийшла до наступного висновку.

Розроблений у дисертаційній роботі мультиверсний метод проектування у HDL-моделях цифрових пристроїв дозволяє суттєво скороти час на синтез HDL-моделей, особливо в умовах системної специфікації. Зазначений метод використовується в системах верифікації HDL-моделей при автоматизованому проектуванні цифрових пристроїв.

У навчальному процесі кафедри АПОТ ХНУРЕ результати дисертаційної роботи Обрізана В.І. використовуються у таких навчальних дисциплінах:

1. У навчальній дисципліні «Логічне моделювання» для бакалаврів напрямку «Комп'ютерна інженерія» у лекційному матеріалі по темі «Верифікація моделей цифрових пристроїв» та у лабораторних роботах по темі «Побудова мовних моделей цифрових пристроїв».
2. У навчальній дисципліні «Проектування та тестування цифрових систем на ПЛІС» для бакалаврів напрямку «Комп'ютерна інженерія» у лекційному матеріалі та практичних заняттях по темі «Сучасні методи проектування цифрових систем на ПЛІС».

12.05.2016

Зав. каф. АПОТ проф. Чумаченко С.В.,
Доц. каф. АПОТ Шкиль О.С.,
Проф. каф. АПОТ Литвинова Є.І.

ДОДАТОК В
ВИХІДНИЙ КОД С++-КОМПІЛЯТОРА

```
#include "gsabuilder.h"
#include "begin.h"
#include "gsa.h"
#include "gsahelper.h"
#include "gsaobject.h"
#include "constant.h"
#include "decision.h"
#include "function.h"
#include "group.h"
#include "join.h"
#include "operation.h"
#include "variable.h"
#include "terminal.h"
#include "uop.h"
#include "truejoin.h"
#include "falsejoin.h"


////////////////////////////////////

#include <boost/config.hpp>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/depth_first_search.hpp>
#include <boost/graph/reverse_graph.hpp>
#include <boost/property_map/property_map.hpp>
#include <assert.h>
#include <iostream>
#include <utility>
#include <stdlib.h>

////////////////////////////////////////////////////////////////////////////


GSA::GSABuilder::GSABuilder()
{
current = 0;
functionId = 0;
generalStack = new std::vector<std::vector<GSAObject *> *>;
```

```
retVar = 0;
begin = 0;
terminal = 0;

PushStack();
}

//////////////////////////////////////////////////////////////////////////

GSA::GSABuilder::~GSABuilder()
{
PopStack();

if (current)
{
delete current;
current = 0;
}
}

//////////////////////////////////////////////////////////////////////////

void GSA::GSABuilder::PushStack()
{
// Allocate new object stack and push it to the general stack
generalStack->push_back(new std::vector<GSAObject *>);

// Set current stack to the newly created stack
stack = generalStack->back();
}

//////////////////////////////////////////////////////////////////////////

std::vector<GSA::GSAObject *> *GSA::GSABuilder::PopStack()
{
// Popped stack can not be empty. All objects must be processed before.
assert(!stack->size());

std::vector<GSAObject *> *st = generalStack->back();
```

```
// Delete old unused stack
delete st;

generalStack->pop_back();

// Set current stack to the last one on the general stack
stack = generalStack->back();

return stack;
}


//////////////////////////////////////////////////////////////////////


GSA::GSA *GSA::GSABuilder::GetGSA()
{
return current;
}


//////////////////////////////////////////////////////////////////////


void GSA::GSABuilder::Start()
{
current = new GSA();
graph = current->GetGraph();
vertexMap = current->GetVertexMap();
idMap = current->GetIdMap();
}


//////////////////////////////////////////////////////////////////////


void GSA::GSABuilder::End()
{
}


//////////////////////////////////////////////////////////////////////


void GSA::GSABuilder::Compound()
```

```
{
// 1. Allocate new objects stack
PushStack();
}


////////////////////////////////////////////////////////////////////////


void GSA::GSABuilder::CompoundEnd()
{
// Build new group vertex, depending on the stack's objects
Group *group = CreateGroup();

// Append end join
stack->push_back(group->GetEndJoin());

// Create edge between begin vertex and the first vertex
std::vector<GSAObject *>::iterator it = stack->begin();
if (it != stack->end())
{
Vertex *v = dynamic_cast<Vertex *>(*it);
AddEdge(group->GetBeginJoin(), v);
}

// Iterate through all objects on the stack (except the last one)
for(; it != stack->end() - 1; it++)
{
Vertex *v  = dynamic_cast<Vertex *>(*it);
Vertex *v2 = dynamic_cast<Vertex *>(*(it + 1));
AddEdge(v, v2);
}

// Expand all group vertices
for (std::vector<GSAObject *>::iterator yt = stack->begin(); yt != stack->end(); yt++)
{
Vertex *v = dynamic_cast<Vertex *>(*yt);
if (v->GetVertexType() == GSA_VT_GROUP)
{
Group *g = dynamic_cast<Group *>(v);
ExpandGroup(g, group);
}
}

// Remove processed objects from the stack
stack->erase(stack->begin(), stack->end());
```

```
// Pop object stack
PopStack();

// Push itself to the stack
stack->push_back(group);
}


////////////////////////////////////////////////////////////////////////////


void GSA::GSABuilder::Function(const char *id)
{
functionId = id;
}


////////////////////////////////////////////////////////////////////////////


void GSA::GSABuilder::FunctionEnd()
{
// TODO:
// 1. Add processing of function interface.
// Proposed sequence of objects on stack:
// Arg1, Arg2, ..., ArgN, return type, function id, compound object


// Pop group vertex
GSAObject *obj = stack->back();
stack->pop_back();

ObjectType ot = obj->GetObjectType();

Group *group = dynamic_cast<Group *>(obj);

if (group)
{
AddEdge(GetBeginVertex(), group->GetBeginJoin());
AddEdge(group->GetReturnJoin(), GetTerminalVertex());
}
else
// TODO: report fatal error: non-group vertex on the stack
assert(0);
```

```
OptimizeJoins();

clear_vertex(group->GetTag(), *graph);
//remove_vertex(group->GetTag(), *graph);
//gsa->RemoveVertex(group);
}


////////////////////////////////////////////////////////////////////////////


void GSA::GSABuilder::DeclaratorEnd(bool hasInit)
{
if (hasInit)
{
// If the declarator has an initializer, then we will create operation
GSAObject *init = stack->back();
stack->pop_back();

Variable *var = static_cast<Variable *>(stack->back());
stack->pop_back();

//stack->push_back(CreateOperation(new uOp(var, init)));
}
else
{
// Declared object has no initializer, has no meaning for GSA
// Throw away the identifier
Variable *var = static_cast<Variable *>(stack->back());
delete var;
stack->pop_back();
}
}


////////////////////////////////////////////////////////////////////////////


void GSA::GSABuilder::Literal(const char * strLiteral, SCCSyntax::ConstantType cType)
{
// 1. Decode ConstantType
ConstantType ct;
switch (cType)
{
case SCCSyntax::CONST_INT:ct = GSA_CT_INTEGER;break;
case SCCSyntax::CONST_FLOAT:ct = GSA_CT_FLOAT;break;
```

```
case SCCSyntax::CONST_CHAR:ct = GSA_CT_CHAR;break;
case SCCSyntax::CONST_STRING:ct = GSA_CT_STRING;break;
default:
ct = GSA_CT_STRING;
break;
 }

// 2. Push the constant to the object stack
stack->push_back(new Constant(ct, strLiteral));
}


//////////////////////////////////////////////////////////////////////////


void GSA::GSABuilder::BinaryEnd(SCCSyntax::BinaryOperator b)
{
// Get two objects from stack
GSAObject *arg2 = stack->back();
stack->pop_back();

GSAObject *arg1 = stack->back();
stack->pop_back();

// Decode binary operator
OperatorType op;

switch(b)
{
case SCCSyntax::OP_MUL:op = GSA_OP_MUL;break;
case SCCSyntax::OP_DIV:op = GSA_OP_DIV;break;
case SCCSyntax::OP_REM:op = GSA_OP_REM;break;
case SCCSyntax::OP_ADD:op = GSA_OP_ADD;break;
case SCCSyntax::OP_SUB:op = GSA_OP_SUB;break;
case SCCSyntax::OP_LSH:op = GSA_OP_LSH;break;
case SCCSyntax::OP_RSH:op = GSA_OP_RSH;break;
case SCCSyntax::OP_LESS:op = GSA_OP_LESS;break;
case SCCSyntax::OP_GREAT:op = GSA_OP_GREAT;break;
case SCCSyntax::OP_LESS_EQ:op = GSA_OP_LESS_EQ;break;
case SCCSyntax::OP_GREAT_EQ:op = GSA_OP_GREAT_EQ;break;
case SCCSyntax::OP_EQ:op = GSA_OP_EQ;break;
case SCCSyntax::OP_NEQ:op = GSA_OP_NEQ;break;
case SCCSyntax::OP_AND:op = GSA_OP_AND;break;
case SCCSyntax::OP_OR:op = GSA_OP_OR;break;
case SCCSyntax::OP_XOR:op = GSA_OP_XOR;break;
case SCCSyntax::OP_LOGICAND:op = GSA_OP_LOGICAND;break;
```

```
case SCCSyntax::OP_LOGICOR:op = GSA_OP_LOGICOR;break;
case SCCSyntax::OP_COMMA:op = GSA_OP_COMMA;break;
default:
assert(0);
}

// Create new objects and push it to the stack
//stack->push_back(new ::GSA::Function(arg1, arg2, op));
}


//////////////////////////////////////////////////////////////////////////


void GSA::GSABuilder::ReturnStmnt(bool hasExpr)
{
if (hasExpr)
{
// Get the expression
GSAObject *obj = stack->back();
stack->pop_back();


Group *group = CreateGroup();

// 2. Push uop
//value = f1;
Operation *op;// = CreateOperation(new uOp(GetReturnVariable(), obj));
AddEdge(group->GetBeginJoin(), op);
AddEdge(op, group->GetReturnJoin());
stack->push_back(group);
}
else
{
// From begin join right to return join
Group *group = CreateGroup();
AddEdge(group->GetBeginJoin(), group->GetReturnJoin());
stack->push_back(group);
}
}


//////////////////////////////////////////////////////////////////////////


void GSA::GSABuilder::PlusPlus()
```

```
{
// TODO: Process ++ operator
assert(0);
}


//////////////////////////////////////////////////////////////////////////


void GSA::GSABuilder::Assignment()
{
// Get right side of the assignment
GSAObject *o = stack->back();
stack->pop_back();

// Get left side of the assignment
Variable *var = static_cast<Variable *>(stack->back());
stack->pop_back();

//stack->push_back(CreateOperation(new uOp(var, o)));
}


//////////////////////////////////////////////////////////////////////////


void GSA::GSABuilder::Identifier(const char *id)
{
stack->push_back(new Variable(id));
}


//////////////////////////////////////////////////////////////////////////


void GSA::GSABuilder::OnEmptyObject()
{
stack->push_back(0);
}


//////////////////////////////////////////////////////////////////////////


GSA::Variable * GSA::GSABuilder::GetReturnVariable()
{
```

```
if (!retVar)
retVar = new Variable("/val/");

return retVar;
}

////////////////////////////////////////////////////////////////////////

GSA::Terminal * GSA::GSABuilder::GetTerminalVertex()
{
if (!terminal)
terminal = CreateTerminal();

return terminal;
}

////////////////////////////////////////////////////////////////////////

GSA::Begin * GSA::GSABuilder::GetBeginVertex()
{
if (!begin)
begin = CreateBegin();

return begin;
}

////////////////////////////////////////////////////////////////////////

void GSA::GSABuilder::IfStmnt(bool hasElse)
{
// Pop else branch object if present
GSAObject *elseObj = 0;
if (hasElse)
{
elseObj = stack->back();
stack->pop_back();
}

// Pop true branch object
GSAObject *trueObj = stack->back();
```

```
stack->pop_back();

// Pop condition object
GSAObject *condObj = stack->back();
stack->pop_back();

// Create new group vertex
// We will store all elements of if-then-else in this group vertex
Group *group = CreateGroup();

// Create new decision block
Decision *decision = CreateDecision();

const char *decId = decision->GetConditionId();

// Append calculation of the conditional variable and the decision vertex
//Operation *condEval = CreateOperation(new uOp(new Variable(decId), condObj));

//AddEdge(group->GetBeginJoin(), condEval);
//AddEdge(condEval, decision);

AddEdge(group->GetBeginJoin(), decision);

// Process true-branch
Vertex *trueVertex = dynamic_cast<Vertex *>(trueObj);
if (trueVertex)
{
AddEdge(decision->GetTrueJoin(), trueVertex);
AddEdge(trueVertex, group->GetEndJoin());

if (trueVertex->GetVertexType() == GSA_VT_GROUP)
{
Group *g = dynamic_cast<Group *>(trueVertex);
ExpandGroup(g, group);
}
}
else
{
// TODO: report fatal error: non vertex on the stack
assert(0);
}

// Process false-branch
if (elseObj)
{
// Else branch is present, we can cast it to the Vertex class
```

```
Vertex *falseVertex = dynamic_cast<Vertex *>(elseObj);
if (falseVertex)
{
// Cast succeeded
AddEdge(decision->GetFalseJoin(), falseVertex);
AddEdge(falseVertex, group->GetEndJoin());

if (falseVertex->GetVertexType() == GSA_VT_GROUP)
{
Group *g = dynamic_cast<Group *>(falseVertex);
ExpandGroup(g, group);
}
}
else
{
// TODO: report fatal error: non vertex on the stack
assert(0);
}
}
else
{
// Apply end join to the decision block if false-branch is not present
AddEdge(decision->GetFalseJoin(), group->GetEndJoin());
}

// Push newly created group vertex
stack->push_back(group);
}


////////////////////////////////////////////////////////////////////////////


void GSA::GSABuilder::WhileEnd()
{
// Get body & condition from the stack
GSAObject *bodyObj = stack->back();
stack->pop_back();
GSAObject *condObj = stack->back();
stack->pop_back();

// Create new group vertex
Group *group = CreateGroup();

// Create new decision vertex
Decision *decision = CreateDecision();
```

```
const char *decId = decision->GetConditionId();

// Create new operation vertex to handle condition evaluation
//Operation *conditionEval = CreateOperation(new uOp(new Variable(decId), condObj));
//AddEdge(group->GetBeginJoin(), conditionEval);
//AddEdge(conditionEval, decision);
//AddEdge(decision->GetFalseJoin(), group->GetEndJoin());

AddEdge(group->GetBeginJoin(), decision);
AddEdge(decision->GetFalseJoin(), group->GetEndJoin());

// Cast body object to the Vertex class
Vertex *body = dynamic_cast<Vertex *>(bodyObj);
if (body)
{
AddEdge(decision->GetTrueJoin(), body);
AddEdge(body, decision);

if (body->GetVertexType() == GSA_VT_GROUP)
{
Group *g = dynamic_cast<Group *>(body);
ExpandGroup(g, group);
}
}
else
{
// TODO: report fatal error: non-vertex on the stack
assert(0);
}

// Push newly created compound block
stack->push_back(group);
}


////////////////////////////////////////////////////////////////////////////////


void GSA::GSABuilder::ForEnd()
{
// Get body, expression, condition, initializer
GSAObject *bodyObj = stack->back();
stack->pop_back();

GSAObject *exprObj = stack->back();
stack->pop_back();
```

```
GSAObject *condObj = stack->back();
stack->pop_back();

GSAObject *initObject = stack->back();
stack->pop_back();

// Create new group vertex
Group *group = CreateGroup();

// Pointer to the lates added vertex
Vertex *curr = group->GetBeginJoin();

// Init object is present
if (initObject)
{
// Cast to operation
Operation *initOp = dynamic_cast<Operation *>(initObject);
if (initOp)
{
AddEdge(curr, initOp);
curr = initOp;
}
else
// TODO: report error
assert(0);
}

// Pointer to the condition evaluation (alias)
Vertex *condEvalAlias = 0;
Decision *decisionAlias = 0;

// Condition is present
if (condObj)
{
// Create new decision vertex
Decision *decision = CreateDecision();
decisionAlias = decision;
const char *decId = decision->GetConditionId();

// Create new operation vertex to handle condition evaluation
Operation *conditionEval;// = CreateOperation(new uOp(new Variable(decId), condObj));
condEvalAlias = conditionEval;

AddEdge(curr, conditionEval);
AddEdge(conditionEval, decision);
```

```
AddEdge(decision->GetFalseJoin(), group->GetEndJoin());
}

// Body is present
if (bodyObj)
{
// Cast body object to the Vertex class
Vertex *body = dynamic_cast<Vertex *>(bodyObj);
if (body)
{
// If decision is present, then join with true branch
if (decisionAlias)
AddEdge(decisionAlias->GetTrueJoin(), body);
else
AddEdge(curr, body);

curr = body;

if (body->GetVertexType() == GSA_VT_GROUP)
{
Group *g = dynamic_cast<Group *>(body);
ExpandGroup(g, group);
}
}
else
// TODO: report error
assert(0);
}

// Expression is present
if (exprObj)
{
// Cast expression object ot the Operation class
Operation *exprOp = dynamic_cast<Operation *>(exprObj);
if (exprOp)
{
AddEdge(curr, exprOp);
curr = exprOp;
}
else
// TODO: report error
assert(0);
}

// Close the loop with decision
AddEdge(curr, condEvalAlias);
```

```
stack->push_back(group);
}

////////////////////////////////////////////////////////////////////////

void GSA::GSABuilder::OnSizeOf(unsigned int _size)
{
    if (_size > 0)
    {
        char buffer[33];
        _itoa(_size, buffer, 10 );
        // 2. Push the constant to the object stack
        stack->push_back(new Constant(GSA_CT_INTEGER, buffer));
    }
}

////////////////////////////////////////////////////////////////////////

GSA::Operation * GSA::GSABuilder::CreateOperation(uOp *u)
{
Operation *op = Operation::CreateOperation(u);
AddVertex(op);

return op;
}

////////////////////////////////////////////////////////////////////////

GSA::Terminal * GSA::GSABuilder::CreateTerminal()
{
Terminal *term = Terminal::CreateTerminal();
AddVertex(term);

return term;
}

////////////////////////////////////////////////////////////////////////
```

```
GSA::Begin * GSA::GSABuilder::CreateBegin()
{
Begin *b = Begin::CreateBegin();
AddVertex(b);

return b;
}


//////////////////////////////////////////////////////////////////////


GSA::Join * GSA::GSABuilder::CreateJoin()
{
Join *j = Join::CreateJoin();
AddVertex(j);

return j;
}


//////////////////////////////////////////////////////////////////////


GSA::Group * GSA::GSABuilder::CreateGroup()
{
Join *b = CreateJoin();
Join *e = CreateJoin();
Join *r = CreateJoin();

Group *g = Group::CreateGroup(b, e, r);
AddVertex(g);

return g;
}


//////////////////////////////////////////////////////////////////////


GSA::Decision * GSA::GSABuilder::CreateDecision()
{
TrueJoin  *tj = CreateTrueJoin();
FalseJoin *fj = CreateFalseJoin();
```

```
Decision *d = Decision::CreateDecision(tj, fj);
AddVertex(d);

AddEdge(d, tj);
AddEdge(d, fj);


return d;
}


//////////////////////////////////////////////////////////////////////////


void GSA::GSABuilder::AddEdge(Vertex *src, Vertex *dest)
{
// It is unable to add something after terminal vertex
if(src->GetVertexType() != GSA_VT_TERMINAL)
{
VertexDescriptor srcTag = src->GetTag();
VertexDescriptor destTag = dest->GetTag();
add_edge(srcTag, destTag, *graph);
}
}


//////////////////////////////////////////////////////////////////////////


void GSA::GSABuilder::AddVertex(Vertex *v)
{
//VertexDescriptor tag = add_vertex(v, *graph);

VertexDescriptor tag = add_vertex(*graph);
put(boost::vertex_index, *graph, tag, v->GetVertexID());
v->SetTag(tag);

vertexMap->insert(std::make_pair(tag, v));
idMap->insert(std::make_pair(tag, v->GetVertexID()));
}


//////////////////////////////////////////////////////////////////////////


GSA::TrueJoin * GSA::GSABuilder::CreateTrueJoin()
```

```
{
TrueJoin *tj = TrueJoin::CreateTrueJoin();
AddVertex(tj);

return tj;
}


//////////////////////////////////////////////////////////////////////////


GSA::FalseJoin * GSA::GSABuilder::CreateFalseJoin()
{
FalseJoin *fj = FalseJoin::CreateFalseJoin();
AddVertex(fj);

return fj;
}


//////////////////////////////////////////////////////////////////////////


void GSA::GSABuilder::ExpandGroup(Group *group, Group *parent)
{
typedef boost::graph_traits<GSAGraph>::in_edge_iterator in_edge_iterator;
typedef boost::graph_traits<GSAGraph>::out_edge_iterator out_edge_iterator;
typedef boost::graph_traits<GSAGraph>::vertex_descriptor v_d;
typedef boost::graph_traits<GSAGraph>::edge_descriptor e_d;

std::pair<in_edge_iterator, in_edge_iterator> p_in;
std::pair<out_edge_iterator, out_edge_iterator> p_out;

// Get list of incoming edges
// Bind all incoming vertices with group begin join
for (p_in = boost::in_edges(group->GetTag(), *graph); p_in.first != p_in.second; p_in.first++)
{
e_d e = *p_in.first;
v_d v = boost::source(e, *graph);

// Add edge (source vertex, begin join vertex)
boost::add_edge(v, group->GetBeginJoin()->GetTag(), *graph);
}

// Get list of outgoing edges
// Bind group end join with all outgoing vertices
```

```
for (p_out = boost::out_edges(group->GetTag(), *graph); p_out.first != p_out.second; p_out. first++)
{
e_d e = *p_out.first;
v_d v = boost::target(e, *graph);
boost::add_edge(group->GetEndJoin()->GetTag(), v, *graph);
}

// Bind return edges
AddEdge(group->GetReturnJoin(), parent->GetReturnJoin());

// destroy group vertex
gsa->RemoveVertex(group);
//boost::clear_vertex(group->GetTag(), *graph);
//boost::remove_vertex(group->GetTag(), *graph);
}


//////////////////////////////////////////////////////////////////////////////


void GSA::GSABuilder::OptimizeJoins()
{
return;
std::pair<VertexIter, VertexIter> vp;
for (vp = boost::vertices(*graph); vp.first != vp.second; )
{
//Vertex *v = (*graph)[*vp.first];
VertexDescriptor vd = *vp.first;

Vertex *v = (*vertexMap)[vd];
++vp.first;
switch (v->GetVertexType())
{
case GSA_VT_JOIN:
{
// Get outgoing vertex (only the first one, two outgoing edges only in decision)
std::pair<OutEdgeIter, OutEdgeIter> p_out;
p_out = boost::out_edges(v->GetTag(), *graph);
if (p_out.first != p_out.second)
{
EdgeDescriptor e = *p_out.first;
VertexDescriptor outVertex = boost::target(e, *graph);

// Get list of ingoing vertices
std::pair<InEdgeIter, InEdgeIter> p_in;
for (p_in = boost::in_edges(v->GetTag(), *graph); p_in.first != p_in.second; p_in.first++)
```

```
{
EdgeDescriptor inEdge   = *p_in.first;
VertexDescriptor inVertex = boost::source(inEdge, *graph);

// Create transit edge
boost::add_edge(inVertex, outVertex, *graph);
}
}

// Clean and remove join vertex
gsa->RemoveVertex(v);
//boost::clear_vertex(v->GetTag(), *graph);
//boost::remove_vertex(v->GetTag(), *graph);
}
default:
continue;
}
}
}


////////////////////////////////////////////////////////////////////////


void GSA::GSABuilder::Optimize()
{
std::list<MyVertexList> *loops = new std::list<MyVertexList>;
FindLoops(begin->GetTag(), *graph, *loops);

// Debug output
for(std::list<MyVertexList>::iterator it = loops->begin(); it != loops->end(); it++)
{
MyVertexList list = *it;
std::cout << list.size();
}

}


////////////////////////////////////////////////////////////////////////


void GSA::GSABuilder::FindLoops(VertexDescriptor entry, const GSAGraph &g, std::list<MyVertexList>
&loops)
{
boost::function_requires<boost::BidirectionalGraphConcept<GSAGraph> >();
```

```
std::vector<EdgeDescriptor> back_edges;
std::vector<boost::default_color_type> color_map(boost::num_vertices(g));

std::pair<VertexIter, VertexIter> vp;
unsigned int i = 0;
for (vp = boost::vertices(*graph); vp.first != vp.second; )
{
VertexDescriptor vd = *vp.first;
put(boost::vertex_index, *graph, vd, i);

++vp.first;
++i;
}

// Find all back-edges in the graph
depth_first_visit(g, entry,
make_back_edge_recorder(std::back_inserter(back_edges)),
make_iterator_property_map(color_map.begin(), get(boost::vertex_index, g)));

// Find out all vertices of each loop
for (std::vector<EdgeDescriptor>::size_type i = 0; i < back_edges.size(); ++i)
{
loops.push_back(std::list<MyVertexList>::value_type());
ComputeLoopExtent(back_edges[i], g, loops.back());
}

return;
}


//////////////////////////////////////////////////////////////////////////////


void GSA::GSABuilder::ComputeLoopExtent(EdgeDescriptor back_edge, const GSAGraph &g, MyVertexList
&loop_set)
{
boost::function_requires<boost::BidirectionalGraphConcept<GSAGraph> >();
typedef boost::color_traits<boost::default_color_type> Color;

VertexDescriptor loop_head, loop_tail;
loop_tail = source(back_edge, g);
loop_head = target(back_edge, g);
Vertex *targetVertex = (*vertexMap)[loop_head];

// ¬˚˜ËÒÎËÚ¸ „□‡ÌË˚ ˆËÍÎ‡: ‰ÓÒÚËÊËÏ˚ ËÁ „ÓÎÓ‚˚
std::vector<boost::default_color_type> reachable_from_head(num_vertices(g), Color::white());
```

```
        depth_first_visit(g, loop_head, boost::default_dfs_visitor(),
make_iterator_property_map(reachable_from_head.begin(), get(boost::vertex_index, g)));

        // ¬˚˜ËÒÎËÚ¸ „□‡ÌËˆ ˆËÍÍ‡: ‰ÓÒÚËÊËÏ˚ Í ı,ÓÒÚÛ
        std::vector<boost::default_color_type> reachable_to_tail(num_vertices(g));
        boost::reverse_graph<GSAGraph> reverse_g(g);
        depth_first_visit(reverse_g, loop_tail, boost::default_dfs_visitor(),
make_iterator_property_map(reachable_to_tail.begin(), get(boost::vertex_index, g)));

        // ¬˚˜ËÒÎËÚ¸ „□‡ÌËˆ ˆËÍÍ‡: ÔÂ□ÂÒÂ˜ÂÌËÂ Ì‡·Ó□Ó, ‰ÓÒÚËÊËÏÓÒÚË
        VertexIter vi, vi_end;
        for(boost::tie(vi, vi_end) = vertices(g); vi != vi_end; ++vi)
        if(reachable_from_head[get(boost::vertex_index, *graph, *vi)] != Color::white()
        && reachable_to_tail[get(boost::vertex_index, *graph, *vi)] != Color::white())
        loop_set.push_back(*vi);
        }


        ////////////////////////////////////////////////////////////////////////////
```