Алехин В.А.

Наименование дисциплины:
Проектирование систем на кристалле

Для магистрантов
По направлению подготовки 09.04.01

Москва, 2018

Оглавление

Список обозначений

BPM	Business Process Modeler - модуль построения моделей бизнес-процессов в форме диаграмм потоков данных
BPwin	CASE-средство , реализующее в качестве методологии IDEF0
CAD	Computer Aided Design – автоматизированное проектирование
CAE	Computer Aided Engineering– поддержка инженерных расчетов
CALS	Continuous Acquisition and Life cycle Support - непрерывная информационная поддержка поставок и жизненного цикла
CAM	Computer Aided Manufacturing - компьютерная поддержка изготовления
CAN	
CAPP	Computer Aided Process Planning - средства планирования технологических процессов
CASE	Computer-Aided Software/System Engineering – разработка программного обеспечения/программных систем с использованием компьютерной поддержки
CMMI	
COM	Component Object Model – компонентная модель объектов
CORBA	Common Object Request Broker Architecture – общая архитектура с посредником обработки запросов объектов
DCOM	Distributed COM – распределенная COM
DFD	data flow diagrams — диаграммы потоков данных
ECAD	Electronic CAD- системы автоматизированного проектирования (САПР) для радиоэлектроники
EDA	Electronic Design Automation – автоматизированное проектирование электроники
EDM	engineering data management – управление инженерными данными
ERD	entity-relationship diagrams
ERwin	CASE-средство, которое в качестве методологии использует IDEF1X
ERX	Entity-Relationship eXpert - модуль концептуального моделирования данных
ER-модель	entity-relationship model, модель «сущность — связь»
ESD	Entity Structure Diagram - диаграммы структур сущностей
ICAM	Integrated Computer-Aided Manufacturing - интегрированная компьютеризация производства
IDEF	Integrated DEFinition

IDEF0	метод функционального моделирования
IDEF1	метод моделирования информационных потоков
IDEF1X	
IDEF1X	метод моделирования данных и проектирования реляционных баз данных
IDEF2	метод динамического моделирования систем
IDEF3	метод получения описания функционирования системы и моделирования как причинно-следственных связей
IDEF4	метод объектно-ориентированного проектирования.
IDEF5	метод получения онтологического описанияи исследования сложных систем
ISO	International Organization for Standardization-международная организация по стандартизации
IT	Информационные технологии
MSF	Microsoft Solutions Framework — методология разработки программного обеспечения
OLE I	Object Linking and Embedding – связывание и внедрение объектов
OSTD	(ObjectStateTransitionDescription)– «внешнее описание»
PDM	Product Document Management – система управления данными об изделии
PFD	Process Flow Description - «внутреннее описание» и описания переходов из одного состояния в другое
PIM	Product information management – управление информацией об изделии
PLM	Product Lifecycle Management – термин используется для обозначения процесса управления полным циклом изделия
RAD	Rapid Application Development – быстрая разработка приложений
RDM	Relational Data Modeler - модуль реляционного моделирования
RUP	Rational Unified Process - рациональный унифицированны процесс.
SADT	Structured Analisys and Design Technique - технология структурного анализа и проектирования
SWEBOK	Software Engineering Body of Knowledge - Сфера знаний в области разработки программного обеспечения
TDM	technical data management - управление техническими данными
TIM	technical information management - управление технической информацией
UML	Unified Modeling Language - унифицированный язык моделирования

WRM	Workgroup Repository Manager - менеджер репозитория рабочей группы
АПК	Аппаратно-программный комплекс
АС	Автоматизированная система
ИПЭ	Информационная поддержка эксплуатации
ПАК	Программно-аппаратный комплекс
ПО	Программное обеспечение
ПС	Программные средства

Глава 1. Современные системы на кристалле (СнК) – очередной этап развития микроэлектроники

1.1. Основные тенденции развития электроники.

В 60-е годы XX века известный менеджер и теоретик микроэлектроники Гордон Мур сформулировал тенденцию в развитии технологии микросхем в виде эмпирического правила, получившего название «закон Мура». Согласно этому правилу минимальный размер элементов микросхем уменьшается в $\sqrt{2}$ раз, а число элементов на кристалле увеличивается в 2 раза через каждые 2,5 года. И вот уже 40 лет «закон Мура» выполняется лишь с небольшими отклонениями.

До настоящего времени полупроводниковая промышленность при подготовке планов развития ориентируется на «закон Мура». Согласно «дорожной карте» технологическое оборудование, технологические процессы, физические структуры элементов микросхем разрабатываются для создания нового производства, обеспечивающего уменьшение размеров элементов в $\sqrt{2}$ раз и увеличение числа элементов вдвое. Технологические маршруты классифицируются по поколениям и унифицируются в одном поколении. «Дорожная карта» позволяет технологам и конструкторам всего мира сосредоточить усилия на создании единственного, полностью совместимого комплекта оборудования и технологического маршрута нового поколения.

Двадцать пять лет назад в микросхемах использовались только биполярные транзисторы. Ограничения на мощность электронной аппаратуры привели к появлению КМОП приборов. Сейчас актуально стоит вопрос: чем можно будет заменить КМОП-транзисторы в постоянной гонке за быстродействие и миниатюризацию электронной аппаратуры.

Для МОП-транзистора физический предел длины затвора лежит в области 10нм, а технологический – в области 15нм. Размеры элементов менее 60 нм получают с использованием электронно-лучевой литографии. Этот размер считается пределом оптической литографии. В лабораторных условиях получены МОП-транзисторы и устройства на их основе с длиной затвора до 8нм, т.е. уже получены микросхемы на пределе физических ограничений.

Электроника разделится на ряд технологически независимых направлений. Уже сейчас формируются следующие направления:

- функциональная электроника, включающая микромеханику, оптоэлектронику, акустоэлектронику, магнитоэлектронику и т.д.;

- традиционная схемотехническая электроника на основе широкозонных полупроводников, позволяющая использовать приборы с размерами 2÷4нм;

- квантовая электроника, использующая в основе вычислений квантовые взаимодействия между атомами. Уже создан прототип квантового компьютера.

Микроэлектроника переживает очередной этап бурного развития, и во многом это связано с быстрой разработкой и внедрением новой элементной базы. Благодаря микропроцессорам с развитой периферией и гибкими возможностями – так называемым «системам на кристалле» – в цифровой аппаратуре произошла настоящая революция.

Рынок предъявляет всё новые требования к электронным продуктам, как по функциональным, так и по техническим параметрам. Чтобы победить в острой конкурентной борьбе, современные электронные устройства должны иметь:

- уникальный набор функций;
- развитый пользовательский интерфейс;

- высокую производительность базовой платформы, позволяющую модернизировать устройство;
- встраиваемую операционную систему;
- низкое энергопотребление;
- встраиваемые цветные ЖК-дисплеи с высоким разрешением;
- возможность подключения к сети Ethernet, в том числе и с использованием PoE
- возможность хранения больших объёмов данных в энергонезависимой памяти и на внешних носителях;
- полный набор стандартизованных проводных и беспроводных интерфейсов.

Реализовать эти требования в одном электронном устройстве можно с помощью систем на кристалле (см. рисунок). Сегодня на рынке представлено большое разнообразие готовых кристаллов, а также решений, интегрируемых на ПЛИС, от ведущих мировых производителей:

Intel, Freescale, Texas Instruments, Marvell, Analog Devices, Altera, Atmel, NXP, Xilinx, Cirrus Logic, RDC, Cypress, Sharp, NetSilicon и др.

Проектирование устройств на базе таких кристаллов открывает новые возможности.

Сегодняшняя тенденция развития заключается в применении встраиваемых систем на основе System-on-Chip (СнК) в разработке электронных систем. Такие СнК - решения обычно состоят из встроенного процессора (процессоров), встроенных памяти, аппаратных ускорителей (или IP-ядер), высокоскоростных коммуникационных интерфейсов и реконфигурируемой логики. Вследствие этого, разработки этих электронных систем становятся все более сложными, поскольку они предъявляют более жесткие требования к более низкой стоимости, более высокой производительности, качеству продукции, безопасности и скорости продаж). Кроме того, в соответствии с законом Мура при дальнейшем развитии

возможностей цифрового оборудования, существует потребность, чтобы увеличенное количество функциональных возможностей содержались в более ограниченном пространстве прибора.

Традиционно во всех проектах предусматривалось создание сложной интегральной схемой, состоящей из 500000 вентилей. По существу, сложные схемы состояли из основной логики и некоторых жестких макросов, таких как встроенная память. С быстрым прогрессом в технологии обработки полупроводников плотность вентилей на кристалле увеличилась в соответствии с тем, что предсказывал закон Мура. Это помогло реализовать более сложные конструкции на одной и той же ИС.

Что такое система на кристалле

За последние несколько лет, с появлением таких передовых технологий, как услуги мобильной связи, предоставляющие интернет через мобильные телефоны возрастает потребность в том, чтобы разместить традиционные микропроцессоры, память и периферийные устройства все на одной микросхеме. Это было отмечено как начало эпохи СнК. Обычно СнК содержит один или более программируемых процессоров, встроенную памяти, таймеры, контроллеры прерываний, шины, специально разработанные сложные аппаратные модули и встроенное программное обеспечение.

С технологической точки зрения технологии СнК рассматривают как направление на повышение степени интеграции микросхем для электронных систем с целью уменьшения стоимости производства и габаритов конечных изделий. СнК интересны прежде всего в каче-стве технологии, позволяющей добиться реализации прикладной функциональности программно-аппаратным способом с гибко устанавливаемой границей (программы и аппаратуры) или вообще без нее. Поэтому используют следующее определение:

система на кристалле – это вычислительная система, архитектура которой разработана целевым образом для решения прикладной задачи (или класса задач) и реализована в виде комплекса функционально специализированных аппаратных и программных компонент на базе конфигурируемой микроэлектронной платформы.

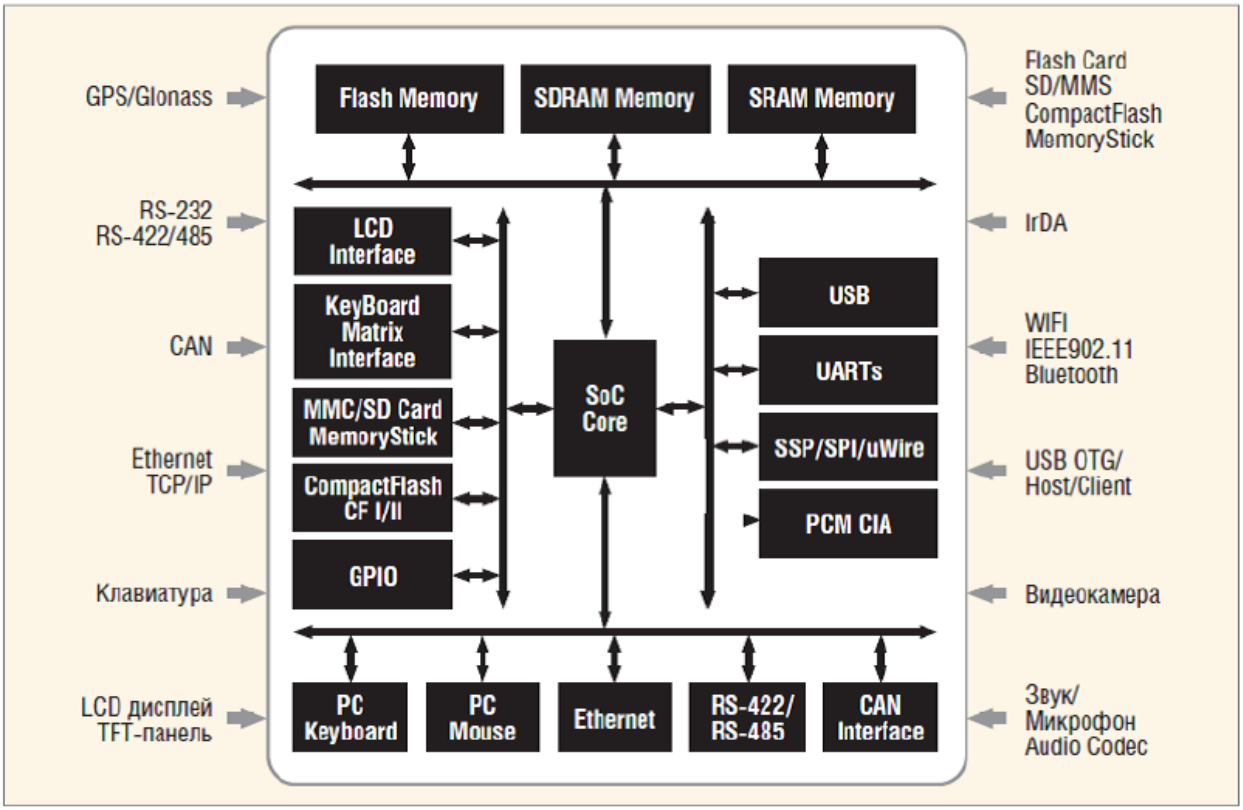


Рис. 1.1. Структура системы на кристалле (СнК).

Система на кристалле (СнК), как правило, содержит высокопроизводительное процессорное ядро и большой набор периферии, которую мы привыкли видеть в персональных компьютерах и современных мобильных устройствах.

Типовая встраиваемая система, построенная на базе СнК, содержит следующие интерфейсы и контроллеры:

- системную шину и контроллеры шин LPC/ISA, PCI, PCMCIA
- контроллеры управления NOR/ NAND Flash, SDRAM, SRAM, DDR
- контроллер Ethernet

- последовательные интерфейсы UART, SPI/SSP/uWire, RS-232, RS-
- 422/485, CAN
- беспроводные интерфейсы WiFi/ IEEE802.11, ZigBee, Bluetooth, IrDA
- интерфейсы Flash-карт памяти SD/MMC, CompactFlash и Memory-Stick
- контроллер ЖКИ (STN/TFT/OLED)
- контроллер матричной клавиатуры
- модули беспроводной передачи данных GSM/GPRS, CDMA и др.
- модули приёма сигналов спутниковых навигационных систем GPS и Glonass
- аппаратную поддержку вычислений с плавающей точкой,
- шифрования, DRM и т.п.
- интерфейсы для звуковых и видеосигналов.

Классификация систем на кристалле

Большое число производителей и «неограниченный» выбор кристаллов СнК вносят некоторую неразбериху при выборе элементной базы для реализации того или иного устройства.

Существует ряд признаков, по которым можно классифицировать системы на кристалле:

- по процессорному ядру: ARM, MIPS, PowerPC, x86 и др.;
- по производительности ядра и частоте системной шины;
- по набору интерфейсов;
- по стоимости кристалла и его минимальной обвязки;
- по позиционированию кристалла производителем.

Кроме того, инженеру-разработчику необходимо учитывать доступность СнК, планируемый цикл производства кристалла, назначение и

эксплуатационные характеристики будущего изделия, технологии печатных плат и пайки, полноту документации и технической поддержки и т.п. В такой ситуации даже грамотный специалист может принять не оптимальное решение. Предлагаемая классификация сформирована на основе выполненных проектов (с учётом доступности микросхем) и призвана помочь российским разработчикам готовых изделий.

1.1.1. Классификация СнК по применению

Стандартные решения могут быть реализованы кристаллами СнК из следующих классификационных групп:

для бюджетных применений;

для устройств удалённого управления;

для терминальных устройств;

двухъядерные (dual core), предназначенные для обработки данных для специализированных вычислителей на основе ПЛИС.

1.1.2. СнК для бюджетных применений

СнК этой группы могут быть использованы в бортовой электронике, системах управления и контроля доступа, системах оповещения, промышленных контроллерах, устройствах вывода звуковой информации.

Кристаллы могут применяться в проектах, не требующих большого объёма программного кода или использования операционных систем.

Кристалл из этой группы (см. табл. 1) характеризуется низкой стоимостью (до 10 долл. США), несложной схемотехникой конечного продукта (не требует применения внешней памяти, имеет невысокие частоты обмена по шинам), возможностью реализации устройства на двухслойных печатных платах (ПП), простотой монтажа, отладки и тестирования. Данные СнК произошли от микроконтроллеров и унаследовали их периферию (GPIO, UART, I2C, SPI, АЦП/ЦАП, ШИМ), но получили более производительное ядро ARM7 и специфичные для СнК интерфейсы (USB, ЖКИ, Ethernet).

1.1.3. СнК для устройств удалённого управления

Кристаллы этой группы (см. табл. 2) целесообразно применять в изделиях, реализующих удалённое управление – с использованием Ethernet или беспроводных интерфейсов – в устройствах сбора данных, серверах контроллерного оборудования, сетевом оборудовании (точки доступа, шлюзы, маршрутизаторы). Высокопроизводительное ядро позволяет использовать операционные системы с поддержкой файловых систем, стека протокола TCP/IP, FTP-сервера и web-сервера. Стоимость кристаллов составляет 10…20 долл.; схемотехника – средней сложности (требуется подключение микросхем памяти, реализация физического уровня интерфейсов), возможна реализация устройства на ПП с 4 – 6 слоями.

1.1.4. СнК для терминальных устройств

Кристаллы этой группы (см. табл. 3) подходят для электронных устройств со встроенными ЖК-матрицами большого разрешения. Конечным продуктом могут быть планшетные и панельные компьютеры, измерительные приборы, бортовые компьютеры с экранами высокого разрешения, медицинские мониторы и терминалы, информационные киоски и панели. Микросхемы этой группы стоят довольно дорого (20…30 долл.), но оправдывают своё применение за счёт высокой степени интеграции современных интерфейсов. Поэтому разрабатываемая схемотехника имеет среднюю сложность: печатную плату можно выполнить в 6 – 8 слоев, и в большинстве случаев потребуется монтаж корпусов типа BGA.

Таблица 1. СнК для бюджетных применений

Производитель	NXP	Atmel	CirrusLogic
Класс	LPC21xx, LPC22xx, LPC24xx	AT91SAM7x	EPM7309, EPM7311, EPM7312
Ядро	ARM7TDMI	ARM7TDMI	ARM7TDMI
Частота, МГц	60...72	55	74
Интерфейсы	ЖКИ, SD/MMC, USB Host Device, OTG, Ethernet 10/100	USB, Ethernet 10/100	ЖКИ, контроллер клавиатуры и тактильного дисплея, цифровой звуковой интерфейс, порт кодека мультимедиа
Периферия	ADC, DAC, UART, SPI/SSP, I2S/I²C, CAN	UART, SPI, SSC, TWI, CAN	IrDA, UART, SSI

Таблица 2. СнК для устройств удалённого управления

Производитель	RDC	Atmel	CirrusLogic
Типономинал	R8610	AT91RM9200, AT91SAM9x	EPM9301, EPM9302
Ядро	Ядро x86	ARM920, ARM926	ARM920
Частота, МГц	150	180...240	166...200
Интерфейсы	2 × Ethernet MAC, USB 2.0, UARTs, LPC, PCI	Ethernet MAC, USB 2.0, UARTs, SPI, SSP, TWI, MCI	Ethernet MAC 10/100, USB 2.0, IrDA, АЦП, SPI, I2S

1.1.5. Двухъядерные СнК для обработки данных

Микросхемы этой группы (см. табл. 4) идеально подойдут для применения в устройствах, где требуется параллельная обработка данных или сбор информации с её одновременным выводом. Устройства, в которых могут потребоваться такие возможности, бывают абсолютно разными: от мультимедиа до измерительной техники. К примеру, в измерительных приборах часто необходимо выполнять свёртку с использованием ядра ЦПОС и одновременный вывод информации на ЖКИ, а также обработку сигналов клавиатуры. С такой задачей успешно справится процессор OMAP5912, оснащённый ядрами DSP и ARM с общей системной шиной.

Микросхемы этой группы подойдут для различных мобильных устройств, т.к. обладают низким энергопотреблением. Стоимость и другие характеристики СнК этой группы схожи с характеристиками предыдущей группы, однако существенным отличием является работа с двухъядерной архитектурой при написании и отладке встраиваемого программного обеспечения.

1.1.6. СнК для специализированных вычислителей на основе ПЛИС

Безусловно, среди разработчиков популярны гибкие решения на базе ПЛИС (см. табл. 5), но их применение оправдано при реализации алгоритмов параллельной обработки данных, скоростных алгоритмов обработки потоков, совокупности уникальных или специфических интерфейсов, интеграции различных ядер и алгоритмов цифровой обработки сигналов в одном устройстве. Применение ПЛИС усложняет процесс разработки и повышает стоимость конечного продукта. Большим преимуществом ПЛИС является реконфигурируемость, что позволяет даже небольшой проектной фирме иметь 1 – 2 платформы и строить на их основе разнообразные изделия.

Таблица 3. СнК для терминальных устройств

Производитель	Freescale	Atmel	CirrusLogic
Типономинал	MC9328MXL	AT91SAM9261, AT91SAM9263	EPM9312(без акс.), EPM9307, EPM9315
Ядро	ARM9	ARM9	ARM9
Частота, МГц	200...350	200	200
Интерфейсы	USB Host Device	Ethernet 10/100, USB Host Device	Ethernet 10/100, USB Host Device
Периферия	ЖКИ 16/18 бит, кодер/декодер, MPEG-4, H.263, 2D-ускоритель	ЖКИ 2000×2000 пикселей, 2D-ускоритель	ЖКИ 1024×768 пикселов, 2D-ускоритель

Таблица 4. СнК для обработки данных с двухъядерной архитектурой

Производитель	Texas Instruments	Analog Devices
Типономинал	OMAP5912	BlackFin BF561
Ядро	ARM926 + TMS320C55x	Dual BlackFin
Частота, МГц	192	600
Интерфейсы	USB 1.1 Host, Client, OTG, UART, SPI, IrDA, I²C, контроллер SDRAM	UART, IrDA, SPI, SPORT, контроллер SDRAM
Периферия	Интерфейс камеры, интерфейс матричной клавиатуры, HDQ/1-Wire, интерфейс, MMC/SD, контроллер ЖКИ	

Таблица 5. Гибкие СнК на базе ПЛИС (CSoC и SoPC)

Производитель	Altera	Xilinx
Семейство	Stratix, Cyclone	Spartan, Virtex
Ядро	Nios, Nios2, ARM, 8051 core, PIC	MicroBlaze, PicoBlaze, ARM, ядро I8051, PIC, интегрированный PowerPC
IP-ядра для цифровой обработки сигналов	DSP	Фильтрация, модуляция/демодуляция, шифрование/дешифрование, корреляция, генерирование сигналов, синхронизация
IP-ядра для работы с видеоданными и изображениями	Обработка изображений и потокового видео	Передача видео по Ethernet (100/1000 Мбит/с), 2D- и медианный фильтры, кодер/декодер JPEG, JPEG2000, MPEG

1.4. Услуги контрактной разработки при проектировании СнК

Алехин В.А. Инструментальные средства систем автоматизированного программирования для проектирования систем на кристалле. РТУ – МИРЭА, 2019.

При всех преимуществах СнК не стоит забывать, что процесс разработки продукта на такой элементной базе весьма трудоёмкий и требует наличия не только грамотных специалистов, но и большой ответственности.

Проекты с использованием СнК не могут быть выполнены «на коленке» одним-двумя разработчиками, т.к. необходимо пройти трудоёмкие этапы:

- выбрать архитектуру;
- подобрать элементную базу с учётом её стоимости, доступности и совместимости
- разработать принципиальную схему с большим числом связей;
- трассировать печатную плату с высокой плотностью монтажа;
- верифицировать схему и трассировку ПП;
- разместить производство опытных образцов в надёжной фирме;
- осуществить первичный «подъём» (bring-up) платы;
- разработать тестовое ПО и при необходимости дополнительно оборудование, а также монитор ПО;
- подготовить полный пакет поддержки аппаратуры (BSP), который включает в себя: первичный загрузчик с подпрограммами тестирования периферии и памяти, операционную систему с драйверами интерфейсов и устройств, установленных на плате, системные утилиты, автоматические скрипты и пакеты сборки ПО разработать прикладное и пользовательское ПО, а также интерфейсы пользователя;
- провести интеграционное тестирование продукта;
- подготовить конструкторскую документацию и инструкции по установке, прошивке, тестированию и программированию устройства;
- выпустить установочную партию изделий и подготовить продукт к серийному выпуску (отдельный трудоёмкий этап работы).

Не все фирмы могут обеспечить эффективное выполнение перечисленных этапов. Тем не менее, разработка изделия требует решения всего комплекса задач с участием менеджеров, маркетологов, системотехников, программистов, дизайнеров, конструкторов, инженеров и других экспертов. В этой ситуации использование услуг контрактной разработки имеет большое значение, т.к. компания, занимающаяся применением СнК, выполнит разработку изделия быстрее и качественнее, чем собственная команда инженеров, не специализирующихся в данной области.

В таком проекте контрактный разработчик принимает на себя комплекс обязательств по исполнению всех этапов проекта (электроника, схемы, платы, корпуса, интерфейсы, программное обеспечение), а также изготовление конечного устройства. Разработчик обеспечивает своевременное исполнение этапов и информирует заказчика о ходе их выполнения. В конечном счёте, именно он несёт полную ответственность за работоспособность конечного устройства, что избавляет заказчика от рисков, связанных с потерей времени и средств при невыполнении проекта собственной группой разработчиков.

Эффективность контрактной разработки объясняется привлечением высококвалифицированных экспертов с узкой специализацией, прозрачностью бюджета и сроков разработки, применением решений, опробованных и отработанных в других проектах и отраслях промышленности.

Во многих случаях производителю выгоднее передать разработку изделия контрактному разработчику, а свои усилия сосредоточить на исследовании рынка и продвижении продукции. Поэтому грамотное использование услуг контрактной разработки способно обеспечить новый качественный уровень изделий и сократить их время выхода на рынок.

Представленная в статье классификация основана на реальном опыте применения указанных систем-на-кристалле в успешных продуктах, которые были разработаны на заказ компанией Promwad.

1.2. Примеры реализации встроенных систем на СнК

Встраиваемые (или встроенные) системы и сети (embedded systems & networks) можно определить как специализированные (заказные) микропроцессорные системы, непосредственно взаимодействующие с объектом контроля или управления и объединённые с ним конструктивно.

Активно растет доля ВсС со сложной внутренней организацией, которая проявляется в таких особенностях, как многопроцессорная гетерогенная архитектура, распределённый характер вычислений, широкий диапазон потенциально доступных разработчику вычислительных ресурсов.

Большинство сегодняшних ВсС составляют распределённые информационно-управляющие системы (РИУС), в которых доля технических решений, характерных для иных классов вычислительных систем (ВС), не является доминирующей. Это позволяет сделать вывод об актуальности поиска и развития всего многообразия технических решений в области ВсС (а не только ограниченного их числа в рамках ряда канонических аппаратно-программных платформ), а также методов и средств их проектирования.

Под аппаратной платформой понимают группу совместимых микропроцессорных систем (компьютеров), которые могут выполнять одинаковые программы и служат как основа или база для создания близких по назначению систем.

Платформа – это набор правил и руководств по архитектуре аппаратного и программного обеспечения, вместе с набором блоков, которые входят в архитектуру.

Платформы встроенных систем покрывают огромное число устройств от домашних беспроводных роутеров до сложных навигационных и

мультимедиа систем легковых автомобилей и промышленных управляющих систем роботизированной сборки таких автомобилей. Характеристики платформ образуют множество различных требований к их интерфейсам и производительности. При разработке встроенной системы наиболее вероятным является выбор между различными СнК-устройствами (системы на кристалле), включающими только те периферийные компоненты, которые необходимые конкретному приложению. На рис. 2.1 приведена характерная платформа на основе СнК.

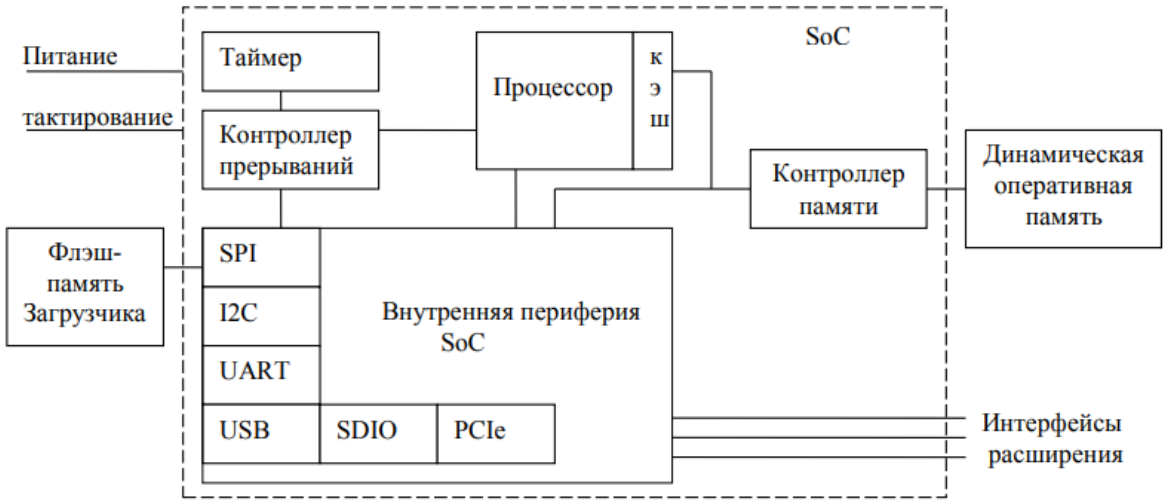


Рис. 1.1. Пример платформы на основе SoC

Рис. 1.2. Пример платформы ВсС на кристалле

Основной тенденцией современной электроники, в том числе, микроэлектроники, является постоянное улучшение массогабаритных характеристик изделий, повышение их надёжности, быстродействия. Это характерно и для современных интегральных микросхем (ИМС), как полупроводниковых, так и гибридных, что достигается за счёт уменьшения геометрических размеров элементов ИМС. Последнее позволяет существенно увеличить степень интеграции элементов на кристалле или плате и значительно расширить функциональные возможности ИМС. Особенно важное значение это имеет для микросхем микропроцессоров (МП).

Благодаря уменьшению размеров транзисторов удаётся значительно расширить возможности микросхем МП и функциональных блоков на их основе, таких как система на кристалле (СнК) и система в корпусе (СвК). Система на кристалле (СнК) это СБИС, интегрирующая на кристалле различные функциональные блоки, которые образуют законченное изделие для автономного применения в электронной аппаратуре.

Система на кристалле может включать как цифровые, так и аналоговые блоки. Основным цифровым блоком обычно является процессор, выполняющий программную обработку цифровых данных. Специализированные блоки обработки обеспечивают аппаратное выполнение функций, специфических для данной системы. Это могут быть, например, блоки цифровой обработки сигналов (DSP), аналоговые схемы, преобразователи потоков данных и другие устройства. Различные типы модулей памяти (SRAM, DRAM, ROM, EEPROM, Flash) могут входить в состав СнК или подключаться к ней, как внешние блоки. Таймеры, АЦП и ЦАП, широтно-импульсные модуляторы и другие цифровые устройства могут интегрироваться в состав СнК в качестве периферийных устройств.

Интерфейс с внешними устройствами обеспечивается с помощью параллельных и последовательных портов, различных шинных и коммуникационных контроллеров и других интерфейсных блоков, в том числе аналоговых (усилителей, преобразователей). Состав блоков, интегрируемых в конкретной СнК, варьируется в зависимости от ее функционального назначения. Организация связей между блоками системы также может быть различной: возможно использование различных стандартизованных шин или специализированных локальных интерфейсов.

Сложность проектирования СнК и невозможность обеспечения в ряде случаев требуемого уровня параметров аналоговых блоков привела к появлению альтернативного типа СБИС- «систем в корпусе» (СвК), которые содержат нескольких кристаллов внутри одного корпуса. Кристаллы

располагают на одном уровне или один над другим, дополняют пассивными или иными необходимыми компонентами и формируют интегрированные модули в одном корпусе, осуществляющие полноценное функционирование конечного электронного устройства. Тем самым достигается высокая миниатюризация изделия: уменьшаются вес, габариты, повышается надёжность. Ведущие мировые фирмы, работающие в космической и оборонной отраслях, около 20 лет используют в своих изделиях «системы в корпусе» (СвК) (System in Package). СвК – это объединение нескольких различных кристаллов, в том числе сформированных на основе кремния на изоляторе (КНИ) и кремния на сапфире (КНС), модулей памяти, цифровой логики, пассивных компонентов, фильтров, антенн, в одном стандартном или специально спроектированном корпусе. При разработке СвК главное внимание уделяют не увеличению количества применяемых транзисторов, а числу различных функций, которые можно интегрировать в одном устройстве на основе апробированных ранее технологических решений максимально надёжным и дешёвым способом.

Применение СвК позволяет обеспечить:

- значительное увеличение выполняемых функций в единице объема и веса;

- снижение энергопотребления;

- создание уникальных аналого-цифровых систем;

- резкое сокращение себестоимости проектов и сроков реализации за счет сокращения квалификационных испытаний.

Системы на кристаллах и системы в корпусе являются в настоящее время основой электронной компонентной базы для систем ВПК, в том числе ракет различного назначения. Именно они во многом определяют сегодня возможности ВПК. Кроме этого СнК и СвК являются неотъемлемой компонентой в таких приоритетных направлениях, как космическое

приборостроение, атомная энергетика и высокоточные интеллектуальные системы вооружения.

В последние годы во все больших количествах стали необходимыми следующие виды военной техники, насыщенные электроникой:

- высокоточные боеприпасы с навигацией, связью, радиолокацией, видением и опознаванием целей;

- малые и сверхмалые беспилотные летательные аппараты (БЛА) с навигацией, связью, опознаванием "свой– чужой", многоспектральным видением;

- крылатые ракеты с навигацией, связью, опознаванием "свой–чужой", многоспектральным видением и опознаванием целей;

- переносные зенитные ракетные комплексы (ПЗРК), в том числе противо-БЛА, с опознаванием "свой – чужой", многоспектральным видением и опознаванием целей;

-малые и сверхмалые космические аппараты со связью, навигацией, радиолокацией и многоспектральным видением;

- системы защиты стационарных и подвижных объектов со связью, опознаванием "свой–чужой", многоспектральным видением, опознаванием и управлением защитой от угроз;

- экипировка бойца со связью, навигацией, опознаванием "свой - чужой", инфра - и радиовидением.

Происходящий в последние 20–25 лет переход систем радиолокации, средств связи, радиоэлектронной борьбы и радиоразведки к пассивным и активным фазированным антенным решеткам (ФАР и АФАР) привел к тому, что требующееся для каждого антенного комплекса количество электронных компонентов увеличивается кратно числу каналов антенной решетки. Особенно значителен этот рост для радиолокаторов сантиметровых диапазонов длин волн. Так, например, самолетная АФАР имеет 2–3 тыс. каналов, АФАР стрельбовых локаторов наземных комплексов ПВО/ВКО – от

10 до 40 тыс. каналов. В ближайшие годы суммарная потребность в электронных компонентах для антенных комплексов различного назначения может достигнуть сотен тысяч – миллионов комплектов ЭКБ ежегодно, а к 2020 году для выполнения Госпрограммы вооружения потребуется уже несколько миллионов комплектов ЭКБ в год.

Таким образом, ежегодная потребность в ЭКБ для комплектации РЭА новейших видов ВВСТ может быть оценена в миллионы – десятки миллионов штук.

1.4.2. Примеры СнК зарубежных фирм

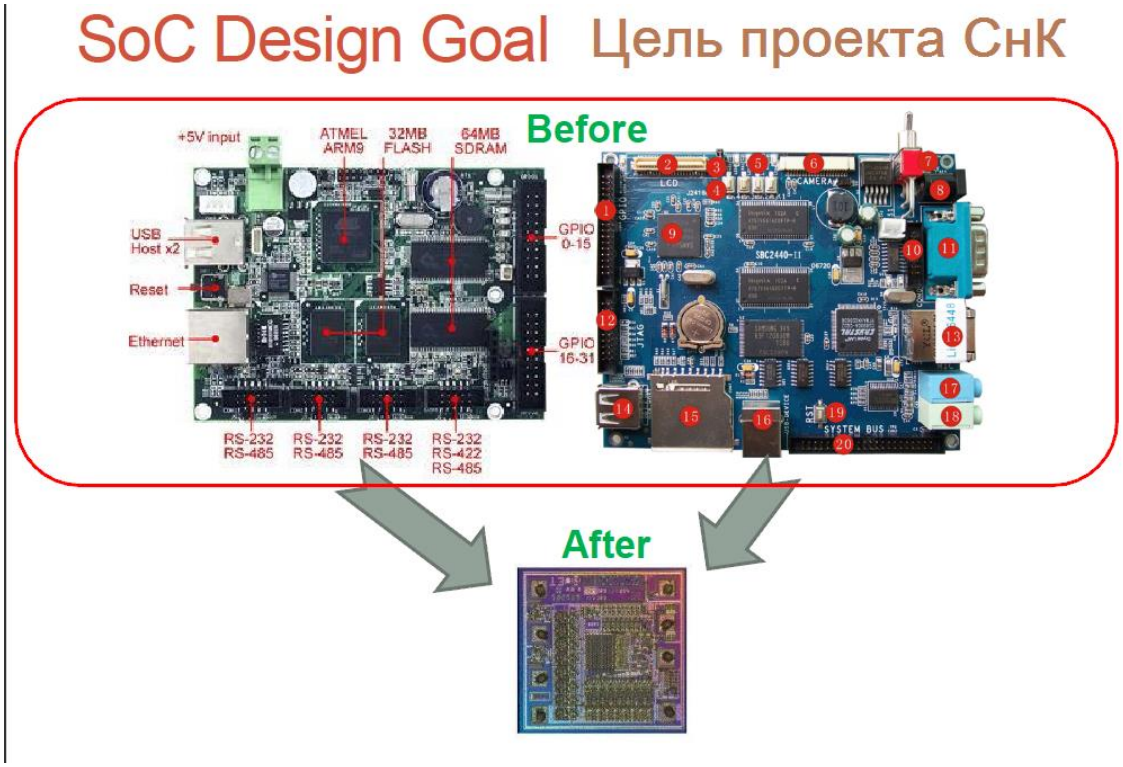


Рис. 1.3. Цели проектирования СнК

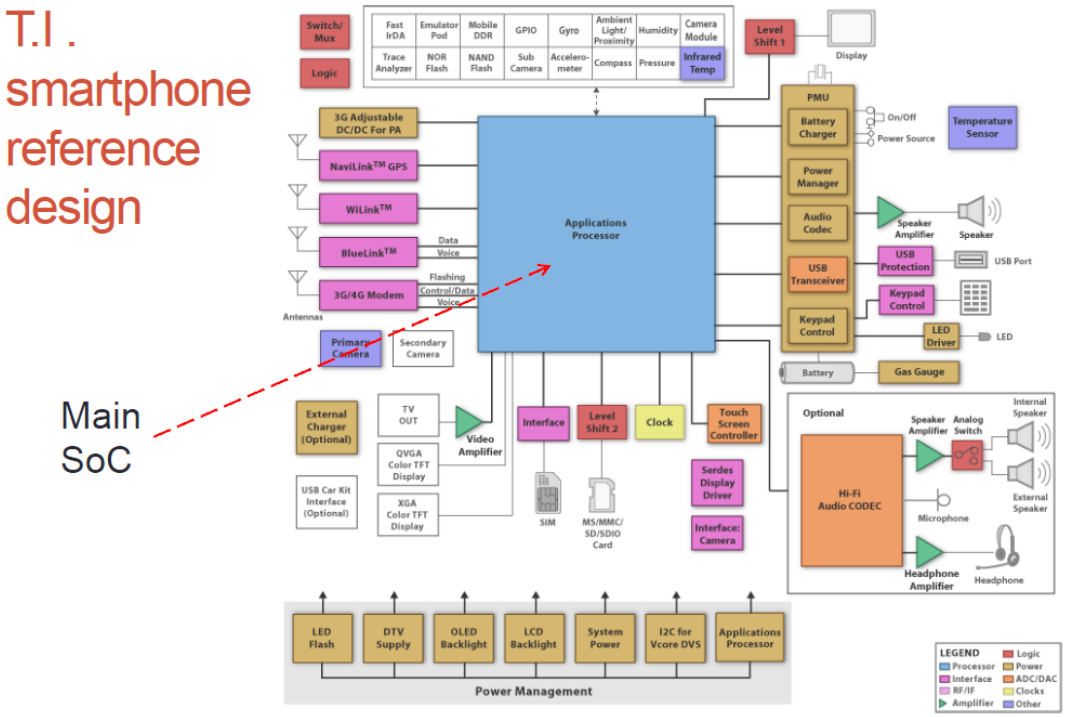


Рис. 1.4. СнК для сматрфона

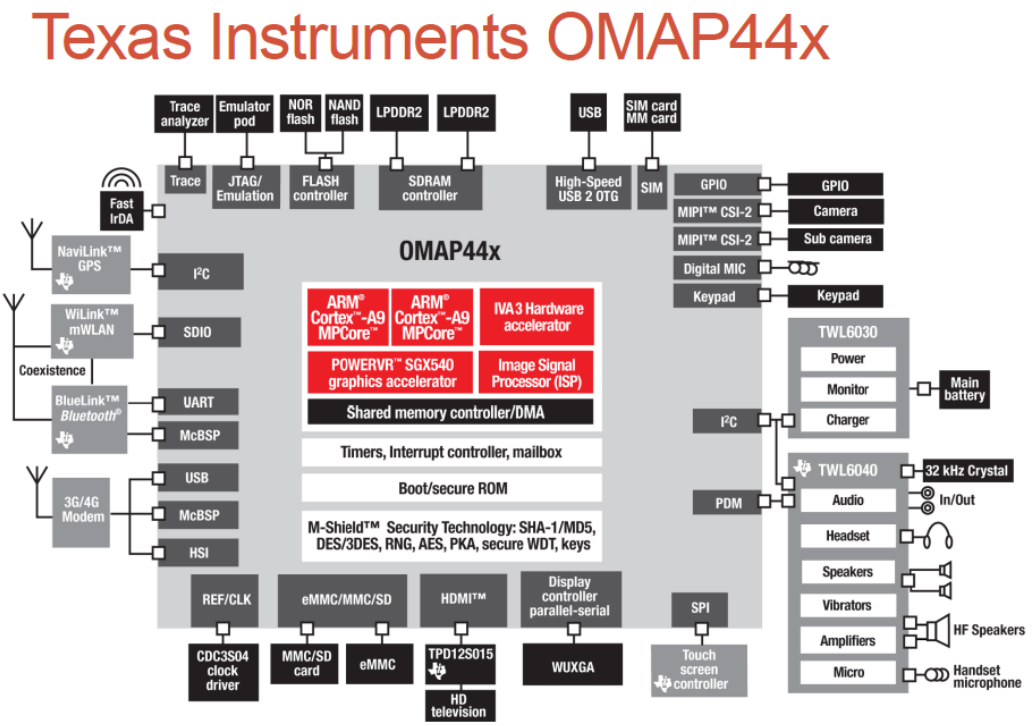


Рис. 1.5. Структура СнК компании Texas Instruments

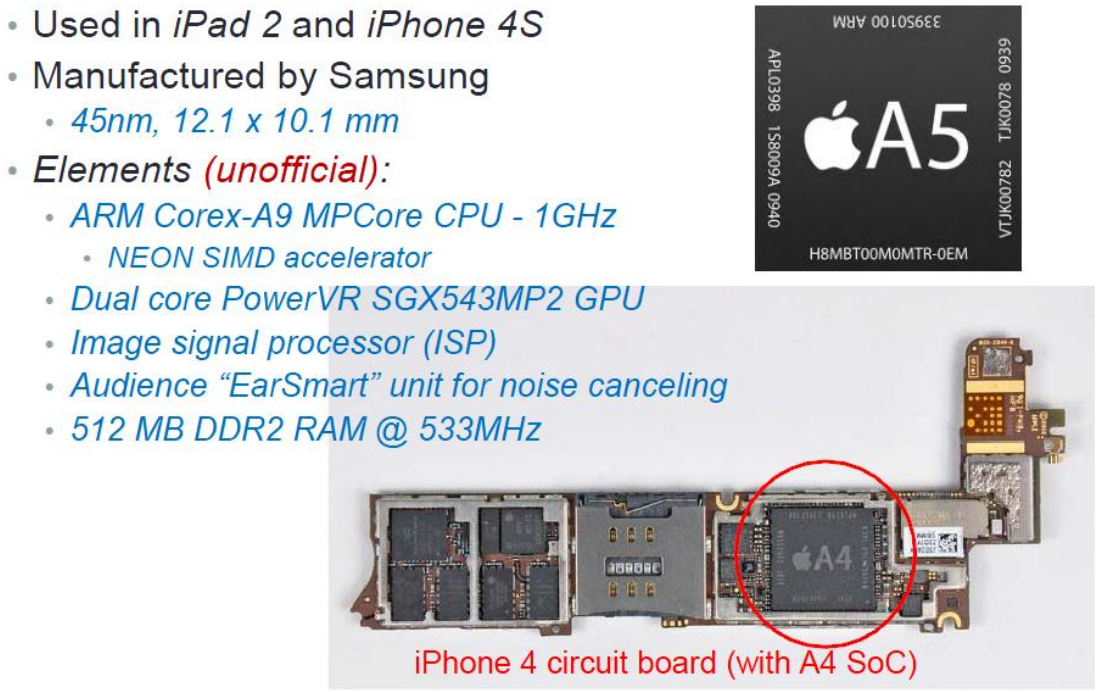


Рис. 1.6. Плата iPhone с СнК

Глава 2. Особенности проектирования СнК

2.1. Методы и средства проектирования СнК

Проектирование систем на кристалле является универсальной и многоплановой дисциплиной, объединяющей в себе методы проектирования законченных аппаратно-программных комплексов, встраиваемых систем на основе стандартных процессоров и процессорных ядер, разработки встроенного программного обеспечения, программируемых (ПЛИС), полузаказных и заказных интегральных схем.

В общем случае, система на кристалле может включать в себя различные типы блоков: программируемые процессорные ядра, блоки ASIC, блоки программируемой логики, памяти, периферийных устройств, аналоговые компоненты и различные интерфейсные схемы. Не обязательно все такие блоки должны быть физически реализованы на одном кристалле: процессоры, блоки памяти, ПЛИС или ASIC могут использоваться как

отдельные компоненты. В рамках излагаемой методологии ставится задача спецификации, верификации и оптимизации системы в целом, а реализация отдельных устройств в контексте проектирования рассматривается как проектирование её составных частей.

Данный подход наряду с системами применим и к проектированию отдельных ПЛИС, в которых интегрируется всё большая и большая функциональность, включая процессоры, память, блоки цифровой обработки сигналов, DSP, высокоскоростные входы/выходы и ряд других сложных IP-блоков. Так, разработчики, использующие ПЛИС APEX компании Altera, могут выбирать между ядром процессора ARM и интегрированием собственного процессорного ядра Altera NIOS. Таким же образом разработчики, использующие ПЛИС Xilinx Virtex-II Pro, могут интегрировать ядро процессора PowerPC фирмы IBM или собственное процессорное ядро Xilinx MicroBlaze. С другой стороны, производители полузаказных интегральных схем начинают встраивать блоки программируемой логики в ASIC, и разница между этими двумя подходами начинает размываться.

2.2. Высокоуровневое проектирование СнК

Ранее проектирование цифровых устройств начиналось с размещения транзисторов для реализации нужной функции. Очевидно, что при таком ручном методе проектирования гибкость заключается в выборе размеров транзисторов и способе проведения проводных соединений, однако при этом достигается оптимальная реализация заданной функции.

По мере усложнения проектов возникла необходимость в еще более высоком уровне абстракции, при котором количество элементов уменьшилось бы, по сравнению с числом вентилей. **Основное внимание на этом уровне абстракции сосредоточено на передаче данных между регистрами, логическими узлами и шинами. Поэтому он и называется уровнем регистровых передач (RTL).**

С течением времени проекты еще более усложнились, и возникла необходимость в переходе на более высокий, по сравнению с регистровыми передачами, уровень абстракции. *В настоящее время этот уровень, называемый* ***уровнем электронной системы (Electronic System Level – ESL****), или системным уровнем, является наивысшим. На системном уровне разработчик заботится только о функционировании разрабатываемой системы и описывает алгоритм, который должен быть реализован. Алгоритм описывается с помощью процедурного языка, подобного языку программирования С.* Описание системы на этом уровне не содержит синхросигналов или временных задержек вентильного уровня.

Средства проектирования системного уровня включают средства ввода, моделирования и, конечно, программы генерации аппаратной части.

Генерация аппаратуры из описания системного уровня может выполняться одним из двух возможных способов.

Первый способ аналогичен применяемому на других уровнях абстрагирования и заключается в трансляции описания системного уровня на низший уровень абстракции, то есть на уровень регистровых передач.

Альтернативным способом является путь, когда процедурное описание системного уровня может компилироваться для выполнения на заданном процессоре. Этот метод становится возможным только на системном уровне, потому что описание системного уровня является процедурным, и для этого используется язык описания программной части системы, подобный языку С.

Именно последний вышеупомянутый метод генерации аппаратуры из описания системного уровня и должен стать методом проектирования встраиваемых систем. Традиционную методику, то есть трансляцию описания на системном уровне в описание на уровне регистровых передач, часто называют С-синтезом, или синтезом системного уровня. На рис. 2.1

приведены обсуждаемые здесь уровни абстракции описания цифровой системы.

Рис. 2.1. Уровни абстракции цифровой системы

Анализ состояния и перспектив развития проектирования на различных уровнях абстракции (рис. 2.2), если в 1990 году реализация проекта (начиная с логического уровня) занимала 90% во всём объёме проектных работ, то в 2000 году эта доля сократилась до 55% и к 2010 году проектирование на архитектурном и функциональном уровнях будет составлять 70% в общем объёме работ, и только 30% придётся на конкретную реализацию проекта в выбранном элементном (библиотечном) базисе.

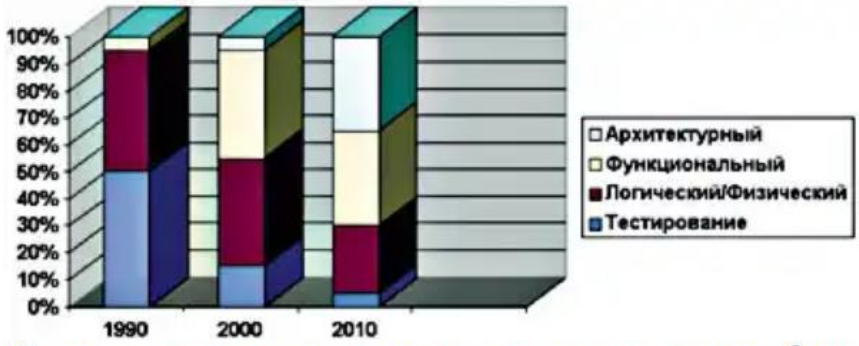


Рис. 2.1. Тенденции проектирования на различных уровнях абстракции

Рис. 2.2. Тенденции проектирования на различных уровнях абстракции

Проектирование систем на кристалле, состоящих из нескольких миллионов вентилей, является непростой задачей, и средства проектирования интегральных схем и ПЛИС, непрерывно усложняясь, эволюционируют в сторону системного уровня проектирования. В конечном счёте, чтобы использовать множество IP-блоков и объёмы интегральных схем в несколько миллионов вентилей, нужны соответствующие средства проектирования, позволяющие использовать все эти возможности в разработках.

2.3. Общий маршрут проектирования СнК

Общий маршрут проектирования систем на кристалле показан на рис. 2.3. и состоит из следующих основных этапов:

концептуальное проектирование системы; основной задачей данного этапа является исследование проектируемой системы и получение её исполняемых спецификаций на языке высокого уровня (стандартно на C/C++, SystemC);

проектирование, то есть трансформация исполняемой спецификации проекта на уровень регистровых передач (получение спецификаций на языках Verilog/VHDL, SystemC) и далее на вентильный уровень;

верификация проекта, то есть проверка проекта и проектных решений на соответствие исходной спецификации и другим требованиям в процессе проектирования и детализации;

физическое проектирование, начиная от выбора технологического и библиотечного базиса и заканчивая получением финального описания проекта в формате GDSII (Gerber files) .

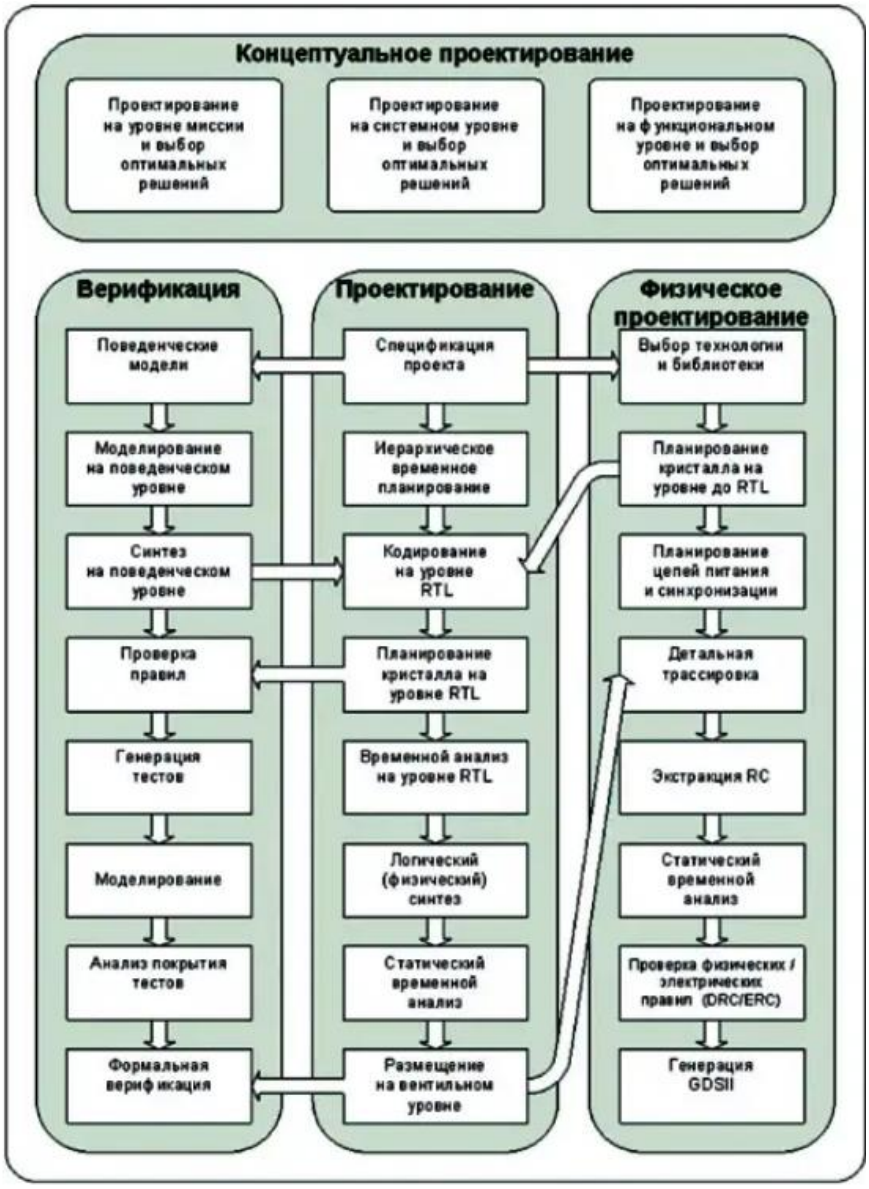


Рис. 2.3. Общий маршрут проектирования СнК

Это традиционный маршрут проектирования СнК **.** Проектирование СнК исторически является развитием технологий и средств разработки специализированных ИМС (ASIC) и ПЛИС. Обобщенная схема традиционного маршрута представлена на рис. 2.4. Процесс является последовательным, с выделением техно-логических этапов – уровней, и итеративным, т.е. на каждом этапе можно сделать откат назад для

корректировки проекта (обратные связи на рисунке не показаны). Проектирование программного обеспечения выполняется обособленно от разработки аппаратных средств, после получения виртуальных или физических прототипов аппаратуры. На всех уровнях используется компонентный подход. Компоненты – функциональные, схемотехнические, топологические и программные блоки – организуются в библиотеки, пригодные для повторного использования.

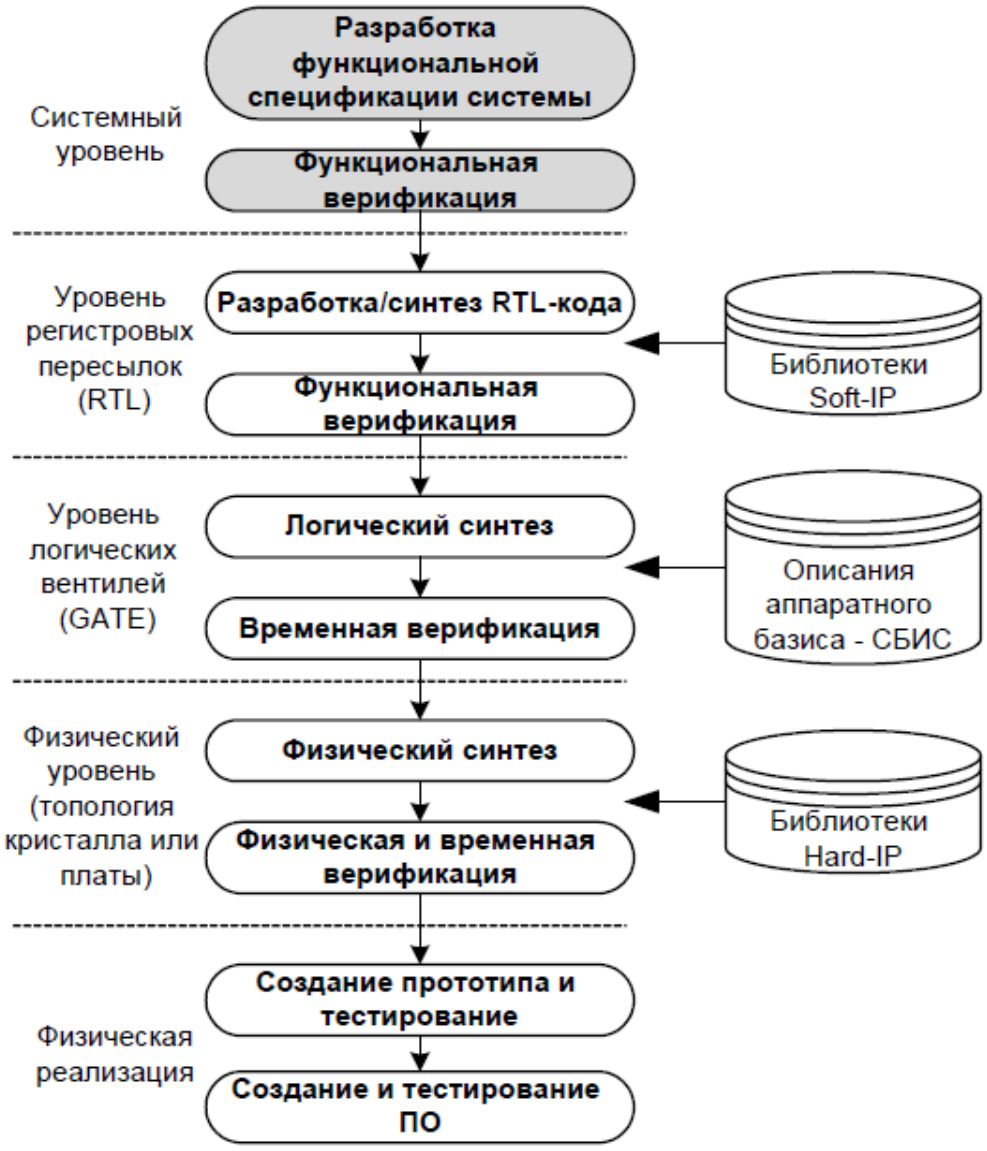


Рис. 2.4. Обобщенная схема традиционного маршрута

При проектировании систем на кристалле концептуальный уровень является критическим для оценки общих характеристик системы. На этом

уровне создаётся общая исполняемая спецификация проектируемой системы, позволяющая исследовать и оценить различные варианты её построения и выбрать оптимальное решение, которое будет реализовано в дальнейшем. Здесь решаются следующие задачи:

- создание функциональной модели системы, то есть описание системы с точки зрения тех алгоритмов и функций, которые она должна выполнять, без привязки к способам их реализации;
- моделирование системы в её операционной среде (на уровне "миссии") с реальными данными и сигналами (аудио- и видеоинформацией, радиоканалами, расположением и движением объектов и др.)
- определение архитектуры системы с точки зрения необходимых ресурсов и их организации для программно-аппаратной реализации функциональной модели.

Таким образом, имея исполняемую спецификацию системы, поведенческие модели и общую архитектуру, проектирование, верификация и топологическая реализация системы далее ведутся параллельно.

2.4. Концептуальный уровень проектирования

Основной целью в процессе спецификации проекта является определение и спецификация основных функций системы и создание исполняемой системной модели. С использованием моделирования, эта системная модель используется для верификации корректности работы системы с функциональной точки зрения в операционной среде, в которой она должна работать, а также для определения необходимых аппаратных ресурсов для работы и архитектуры системы. Общий маршрут проектирования на данном этапе приведён на рис. 2.5.

Рис. 2.5. Маршрут проектирования на концептуальном уровне

На этапе общей спецификации проекта определяется операционная среда, в которой должна работать система, основные сценарии работы, общие функциональные характеристики и протоколы. Так, здесь могут моделироваться различные электронные приводы, системы управления, зоны покрытия, взаимное расположение и движение объектов, например, для распределённых систем типа "базовые станции - мобильные терминалы" и другое.

Далее создаётся функциональная спецификация системы, целью которой является определение и моделирование функционирования системы с точки зрения выполняемых алгоритмов. Здесь может быть задано и

промоделировано поведение всей системы в целом или её отдельных блоков. Как правило, на этом уровне функции системы моделируются с реальными данными и сигналами. Например, можно описать цифровой приёмник-передатчик и протестировать его с использованием реалистичной модели радиотракта.

На этапе исследования проекта моделируемые функции трансформируются и разделяются для выполнения на ряде платформ, или архитектур, которые содержат различные наборы компонентов, такие как программируемые процессоры, память, ASIC, ПЛИС или блоки системы на кристалле. Используя различные виды оценок, целью здесь является нахождение оптимальной архитектуры, которая отвечает заданным критериям, таким как работа в реальном времени, производительность, стоимость, потребляемая мощность. Программные функции оцениваются с точки зрения размера кода и наихудшего времени выполнения, измеряемого количеством тактов процессора, а аппаратные функции - в количестве эквивалентных вентилей.

Наконец, производится уточнение спецификации системы, где создаётся более детальное описание системной архитектуры, которая передаётся на проектирование. Такое описание на системном уровне может содержать некоторые детали последующей реализации, но функциональная часть состоит из поведенческих моделей на языках C/C++/SystemC. Далее уже используется совместное программно-аппаратное проектирование с применением моделей конкретных процессоров и шин (функциональных моделей), блоков, описанных на языках проектирования аппаратуры VHDL/Verilog и так далее.

2.5. Архитектурно-ориентированное проектирование СнК

Основные недостатки традиционных маршрутов разработки СнК касаются системного уровня проектирования. Предлагается перенести акцент проектирования именно на этот этап – на проработку архитектуры

СнК – путем расширения и детализации решаемых на системном уровне задач. Одновременно для всесторонней оценки результата предлагается распараллелить потоки реализации, верификации и прототипирования синтезированной архитектуры СнК (рис. 2.6). Такой архитектурно-ориентированный подход к созданию СнК позволяет создавать эффективное специализированное решение для конкретной прикладной задачи в рамках нескольких четко определенных шагов: формирование и анализ архитектурного шаблона на системном уровне, тонкая настройка и взаимное согласование аппаратных и программных компонент в рамках этапов реализации, верификации, прототипирования, адаптация под требования иных существенных аспектов проекта.

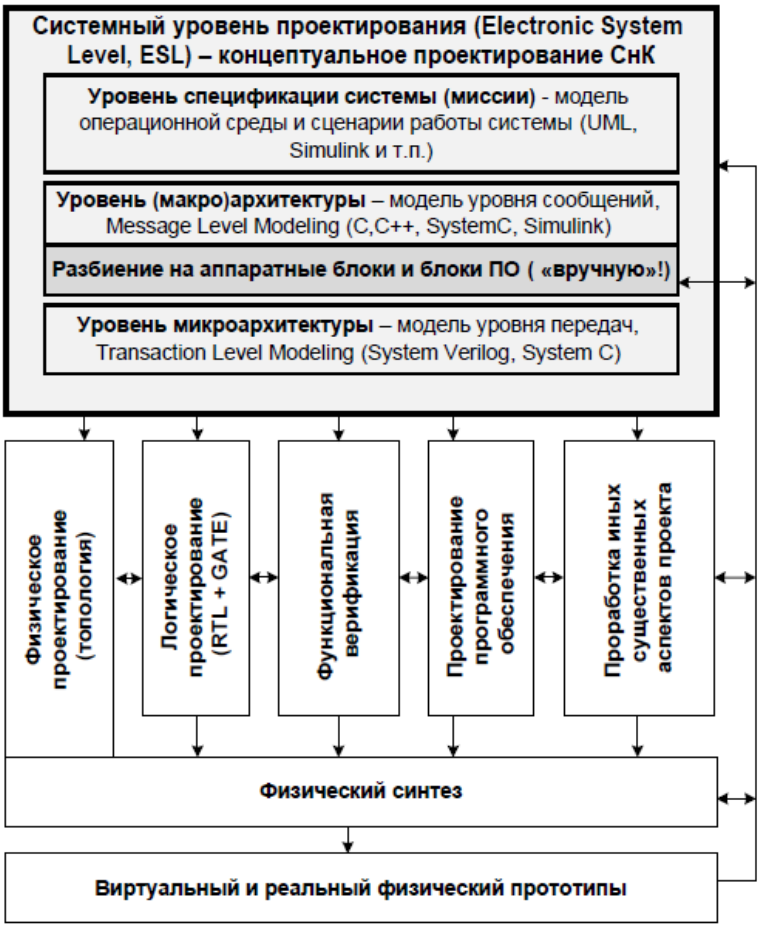


Рис. 2.6. Распараллеливание потоков в синтезированной архитектуре

Перечислим ключевые особенности архитектурно-ориентированного проектирования СнК.

1) Ведущая роль системного уровня. Он существенно усложнен и по структуре (выделено несколько подуровней), и по наполнению. Учитывая влияние аппаратной составляющей на весь процесс и на результат, данный этап проектирования получил название Electronic System Level (ESL). С другой стороны, на этом этапе определяется архитектура системы в целом, поэтому иногда его именуют общесис-темным уровнем – Entire System Level, сохраняя ту же аббревиатуру ESL.

2) Параллельное, связанное исполнение задач (ранее – последовательных этапов) проектирования. Должна обеспечиваться совместная верификация программных и аппаратных компонент системы и, одновременно, разнообразных требований к системе.

3) Среди иных требований с самых ранних этапов реализации СнК учитываются особенности микроэлектронной аппаратуры (СБИС), например, особенности и требования технологии производства современных микросхем, так называемого «глубокого субмикрона».

2.6. Проектирование и функциональная верификация

Функциональная верификация занимает всё более важное место в общем маршруте проектирования. Если раньше под проектированием понималась разработка проекта на уровне регистровых передач (и далее переход на вентильный уровень средствами логического синтеза), а верификация проводилась средствами логического моделирования, то сейчас верификация начинается на поведенческом уровне на стадии разработки общей спецификации проекта.

Основными требованиями, предъявляемыми к составу средств функционального проектирования и верификации, являются:

- анализ архитектуры, производительности и других системных параметров проектируемых систем;

- проектирование аппаратно-программных систем, то есть возможность совместной разработки и верификации аппаратуры и встроенного программного обеспечения;
- проектирование систем с использованием процессорных блоков, то есть использование моделей процессоров при разработке аппаратуры и программного обеспечения;
- единая среда проектирования, от системного уровня до уровня регистровых передач и вентильного уровня с поддержкой языков C, C++, SystemC уровней 1.0 и 2.0 и языков описания аппаратуры Verilog и VHDL;
- наличие библиотек и высокоуровневых конструкций для функциональных блоков и коммуникационных каналов, включая таблицы связности;
- средства управления данными и документирования проектов.

Методология проектирования в системе проектирования VisualElite компании Summit Design (www.sd.com) предоставляет системным архитекторам, разработчикам и программистам единую среду проектирования для спецификации, верификации и анализа архитектуры и функционирования от системного уровня до уровня регистровых передач (рис. 2.7).

Типичная система на кристалле состоит из интерфейса внешней шины, возможно, встроенного процессора, памяти "на кристалле" (или широкополосного интерфейса к внешней памяти), ряда функциональных модулей и шины "на кристалле" (On-chip Bus, OCB), которая их соединяет. И первая решаемая задача - это анализ архитектуры системы и её производительности. По мере получения исполняемых спецификаций аппаратуры и программного обеспечения, проверить их взаимодействие и архитектуру системы можно в Virtual-CPU, которая может быть сконфигурирована для любого процессора или шины.

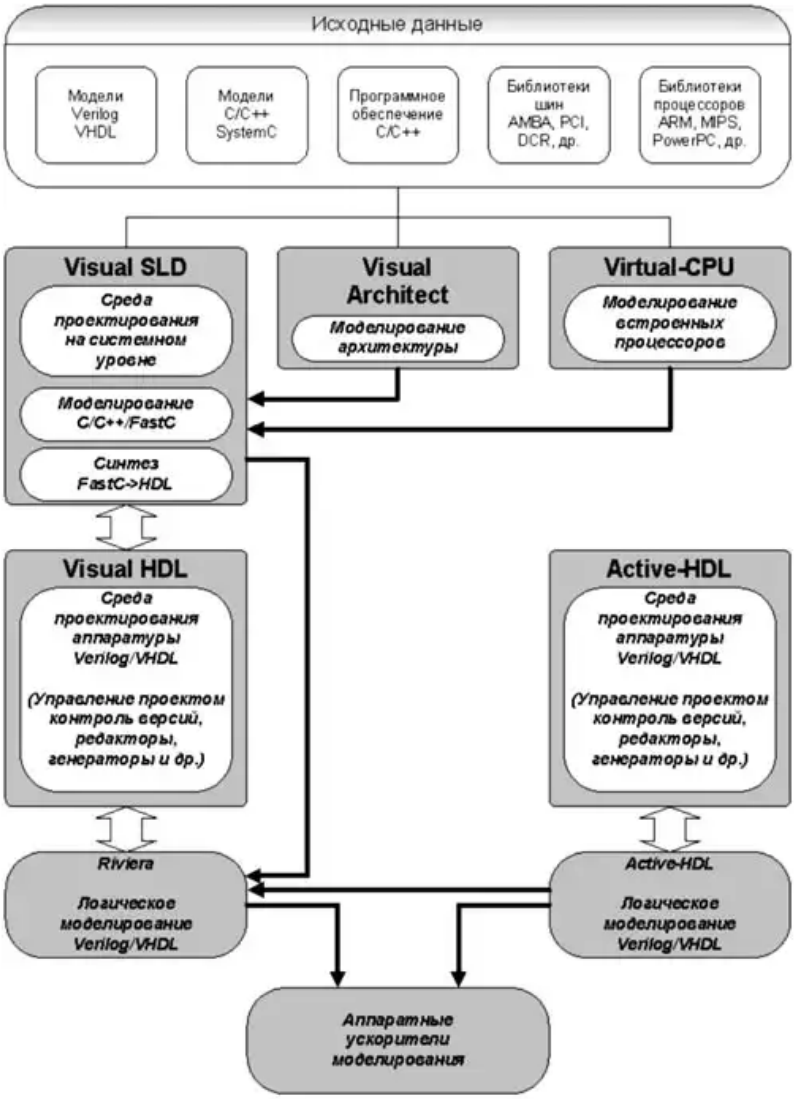


Рис. 2.7. Маршрут проектирования в системе VisualElite

После того, как необходимые итерации спецификации архитектуры проделаны и отверифицированы, реализация аппаратуры и программного обеспечения может продолжаться параллельно, и на этом этапе необходимо проектирование аппаратно-программных интерфейсов для создания и управления системными регистрами, которые устанавливают аппаратно-программный интерфейс и логически разделены между периферийными аппаратными устройствами в системе и прикладным программным обеспечением, исполняемым во встроенных ядрах.

На высших уровнях представления используются языки C/C++ или SystemC. Для моделирования кода C/C++ используется встроенное ядро

моделирования, которое осуществляет планирование и исполнение моделирования в соответствии со структурой и поведенческими функциями проекта вместе с программными объектами, например, такими как операционные системы. Когда модули C/C++/SystemC постепенно детализируются с переходом на нижние уровни абстракции, разработчик может заместить модуль С представлением HDL (Verilog и VHDL) или синтезировать его, если данный блок представлен на уровне С в соответствии с определёнными правилами.

Редактор блок-диаграмм является основным редактором, который поддерживает блоки, написанные на C/C++, SystemC и блоки HDL. Он предоставляет возможности построения иерархии и параллельных процессов, вместе с другими аппаратными особенностями RTL, такими как узлы, пины, компоненты и блоки. Блоки могут описывать алгоритмы на уровне транзакций, тактовую семантику или некоторое детальное событийное поведение. Для определения семантики таких параллельных блоков предоставляется компактный набор конструкций типа операторов чувствительности, временных функций и набор предопределённых функций чтения/записи, блокированных или не блокированных, для различных структур данных и протоколов.

Между блоками поток данных контролируется посредством коммуникационных каналов. Каждый канал реализует определённый протокол, такой как очередь, защищённый канал связи или аппаратный сигнал, поддерживаемые библиотеками классов C++. Однако, пользователи могут определять и добавлять свои собственные коммуникационные каналы, используя специальный пакетный механизм, который обеспечивают как С, так и C++ пакеты.

Когда модули C/C++/SystemC постепенно детализируются с переходом на нижние уровни абстракции, Visual Elite без дополнительной настройки ассоциирует коммуникационные каналы блока С с сигналами HDL,

обеспечивая, таким образом, единую среду проектирования и верификации от системного уровня до уровня реализации.

2.7. Архитектурное планирование кристалла

Архитектурное планирование требуется для всех типов кристаллов, как для систем на кристалле, которые строятся из IP-блоков, так и больших (несколько миллионов вентий) ASIC.

При проектировании ASIC, когда нет заданной физической иерархии, архитектурное планирование применяется для проектов, слишком больших для типичных процедур проектирования (синтеза, размещения и трассировки). Целью архитектурного планирования в данном маршруте является разбиение кристалла на блоки, которые затем могут быть спроектированы независимо друг от друга, как если бы они являлись отдельными кристаллами. Для каждого такого блока генерируется набор граничных требований, включая его конфигурацию, расположение портов и временные ограничения. Затем производится ассемблирование проекта, аналогично системе на кристалле.

Задача архитектурного планирования состоит из двух составных частей: с одной стороны, это собственно планирование, связанное с разбиением проекта на блоки, с другой стороны, это иерархическая интеграция проекта на основе блоков.

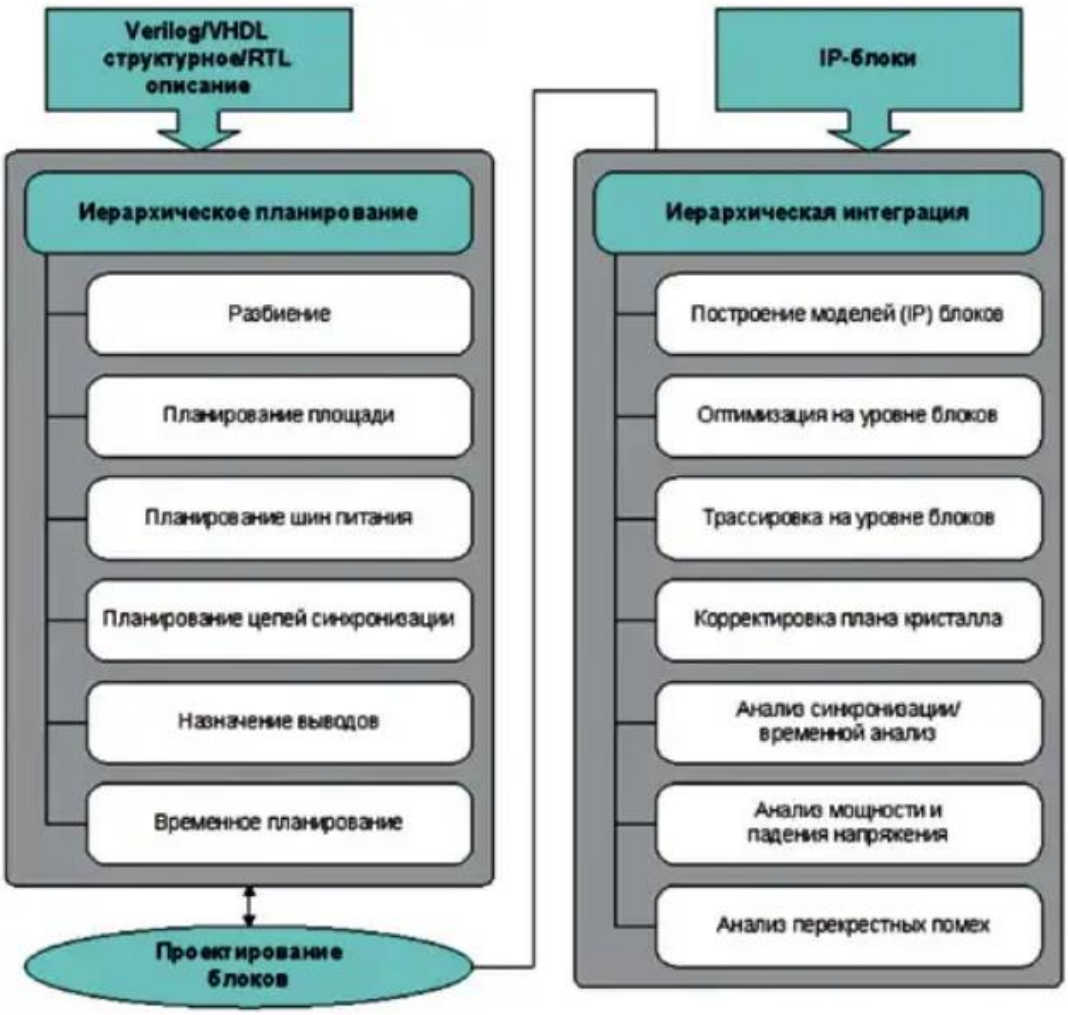


Рис. 2.8. Иерархическое планирование и интеграция

2.8. Логический синтез и проектирование физического прототипа

Целью данного этапа проектирования является создание из исходного описания всего проекта или его отдельных блоков на уровне регистровых передач на языках Verilog и/или VHDL списка цепей в базисе библиотечных элементов производителя и физического прототипа проекта/блоков.

Описание проекта на уровне регистровых передач на языках Verilog или VHDL в целом технологически независимо, хотя до разработки RTL-кода необходимо принимать во внимание последующую реализацию, если речь идёт о проектировании системы на кристалле. Поэтому логический синтез является ключевым и универсальным инструментом при проектировании цифровых систем и их реализации в виде ПЛИС, физических прототипов как на основе ПЛИС (макетов), так и виртуальных

прототипов кристаллов. Здесь прототипы играют разную роль - в то время, как макеты на основе ПЛИС используются для функциональной верификации, виртуальный физический прототип используется для определения всех параметров топологической реализации кристалла.

2.9. Проектирование физического виртуального прототипа

Физический виртуальный прототип является представлением проекта системы на кристалле, ASIC или отдельного блока, которое доступно до финальной топологии и содержит достаточную физическую информацию, чтобы точно предопределить все параметры, такие как временные характеристики, занимаемую площадь, потребляемую мощность и другие.

Физический прототип должен быть достаточно точным, чтобы он мог быть передан на проектирование топологии в полной уверенности, что конечная реализация будет отвечать всем проектным требованиям.

Основной задачей проектирования физического прототипа является решение всех проблем, которые могут возникнуть при проектировании топологии перед тем, как оно будет начато, и передача его на финальную реализацию.

Проектирование физического прототипа может использоваться совместно с архитектурным планированием для проверки проектных требований и ограничений. При этом будут определены любые проектные требования, которые невозможно реализовать, и информация будет передана назад для внесения изменений в план кристалла. Затем будет сделана логическая оптимизация, глобальные размещение и трассировка, разводка шин питания и цепей синхронизации и анализ проекта.

Таким образом, результатом будет являться гораздо более точное представление блоков, чем было доступно ранее. Эта информация передаётся назад планировщику, так что архитектурный план кристалла будет инкрементно обновляться, и соответственно уточняться требования к другим блокам.

2.10. Проектирование физической топологии полузаказных схем

Этап проектирования физической топологии ИС заключается в получении описания топологии в формате GDSII для передачи его на производство из исходного списка цепей.

Традиционный подход к задаче физического проектирования топологии ИС состоит из целого ряда последовательных операций (состав которых определяется используемой технологией, в зависимости от тех физических эффектов, которые необходимо учитывать при проектировании) со множеством итераций, поскольку по результатам проектирования на каждом этапе необходимо бывает вносить изменения и проводить перепроектирование на предыдущих стадиях, и, кроме того, требует детальных знаний технологии и опыта проектирования.

2.11. Маршрут проектирования компании Cadence

В настоящее время Cadence предоставляет широкий набор программных средств для проектирования современных электронных устройств от интегральных схем и ПЛИС до систем на кристалле (СНК) и Интернета вещей.

На рис. 2.9 показаны этапы проектирования систем на кристалле, включающие:

- Системное проектирование;
- Аппаратное проектирование;
- Проектирование топологии интегральной схемы (ИС);
- Проектирование корпуса ИС;
- Проектирование печатной платы;
- Разработка программных средств;
- Отладка и тестирование системы.

Рис. 2.9. Этапы проектирования СНК

На рис. 2.10 показан маршрут проектирования сложной электронной системы. Сначала проводится системное проектирование на языках C++ и SystemC с использованием библиотек, стандартов, сложных заказных готовых IP (Intellegence Properties) блоков. Затем проводят одновременно аппаратное и программное проектирование цифровых и смешанных аналогово-цифровых и заказных блоков с использованием языков Verilog, VHDL, AMS, выполняют логический синтез FPGA (field-programmable gate array - программируемая логическая интегральная схема (ПЛИС)) и ASIC (application specific integrated circuit – интегральная схема специального назначения). Проводят физическое прототипирование, создают библиотеку производителя, проектируют топологию, выполняют верификацию топологии с возвратом для уточнения на системное прототипирование, эмуляцию и системное программирование. Такой замкнутый цикл может повторяться многократно, пока не будут достигнуты надежные требуемые параметры устройства. Только после этого проект передают в производство, корпусирование и разработку печатной платы.

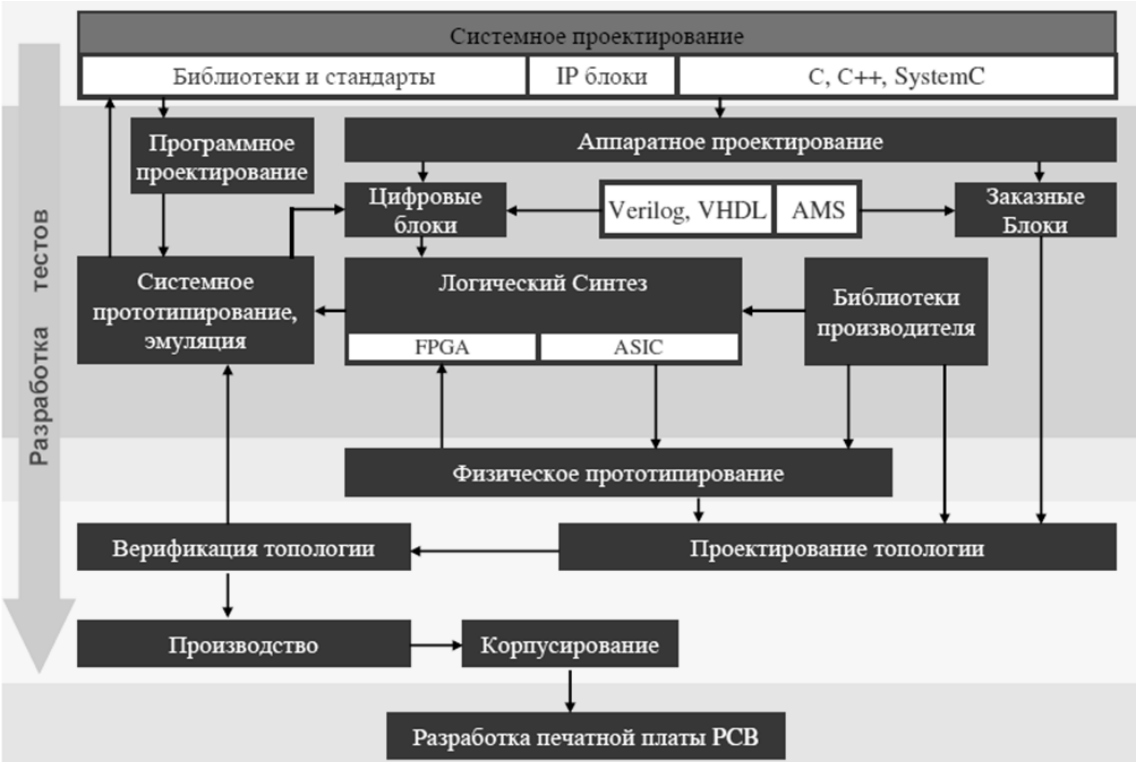
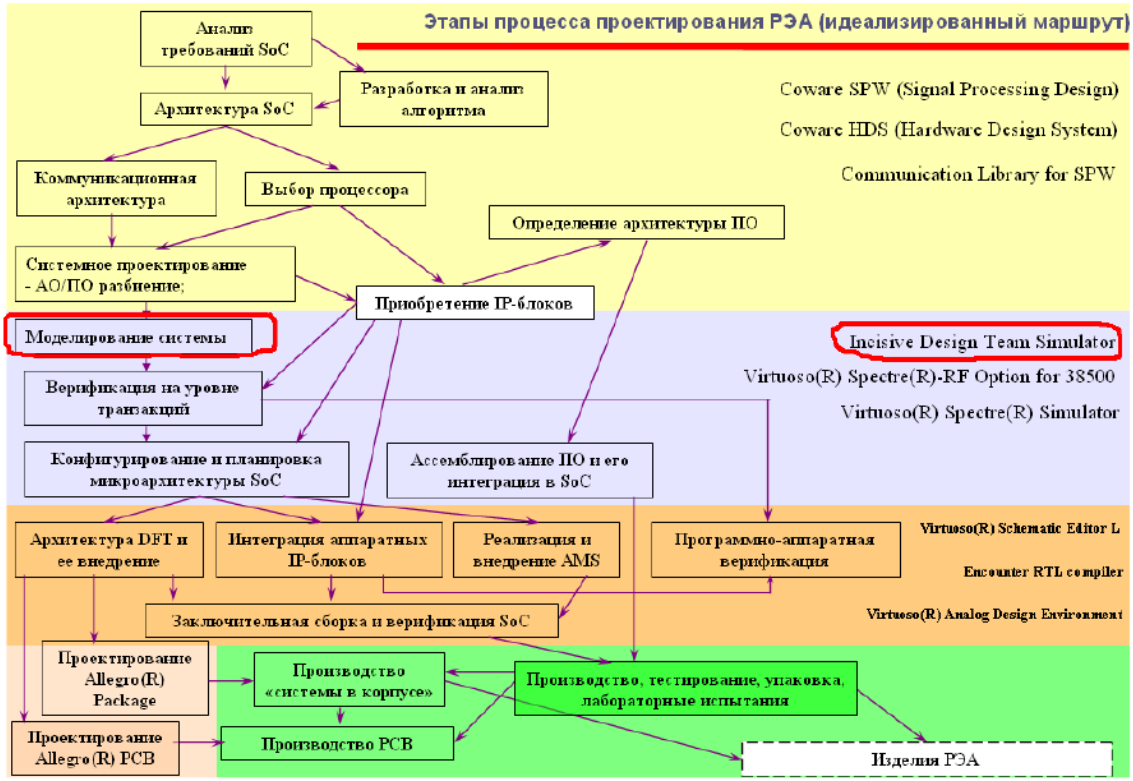


Рис. 2.10. Маршрут проектирования Cadence

Чтобы научиться этому, надо начать с простых электронных устройств и изучить OrCAD 17.2.



Design СнК LabWRK

Глава 3. Программное обеспечение для проектирования систем на кристалле

3.1. Инструментальные программы для систем автоматизированного проектирования СнК

Наибольшее развитие получили САПР проектирования и производства интегральных микросхем класса «Система на кристалле». Стоит отметить, что САПР данного типа для системного уровня проектирования могут использоваться для проектирования аппаратуры любой вычислительной системы. Специализация системы происходит на уровне синтеза решений в конкретный базис с использованием библиотеки компонентов либо малой (транзисторов, логических вентилей, ячеек ПЛИС), либо высокой степени гранулярности (процессоров, периферийных контроллеров, вычислительных платформ и т.п.).

Также, доступны САПР для проектирования:

- печатных плат и многокристальных модулей (PCB & MCM);
- механических соединений, корпусов, составных частей бортовых систем автомобилей, оптических систем и пр. (Computer Aided Engineering, CAE);
- и многие другие.

Распределение САПР по областям показано на Рис. 3.1.

Главными игроками рынка САПР являются компании Cadence, Synopsys, Mentor Graphics. Они предоставляют интегрированные решения для всех этапов проектирования вычислительной техники, начиная от системного уровня и заканчивая подготовкой производства и выпуском печатных платы, аналоговой и цифровой электроники в виде интегральных микросхем и др.

Доля рынка, закрепленная за САПР различных компаний, представлена на Рис. 3.2.

В список других компаний входят такие компании, как Agnisys, Altium, Altera, Xilinx, National Instruments и др.

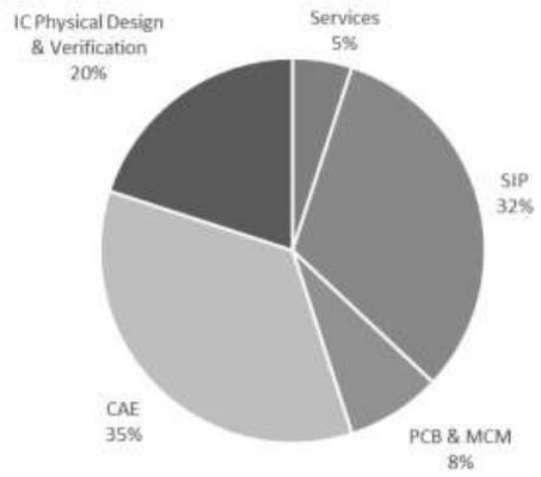


Рис. 17. Распределение САПР по областям (данные EDA Consortium, Market Statistics Service, 2015 г.).

Рис. 3.1. Распределение САПР по областям

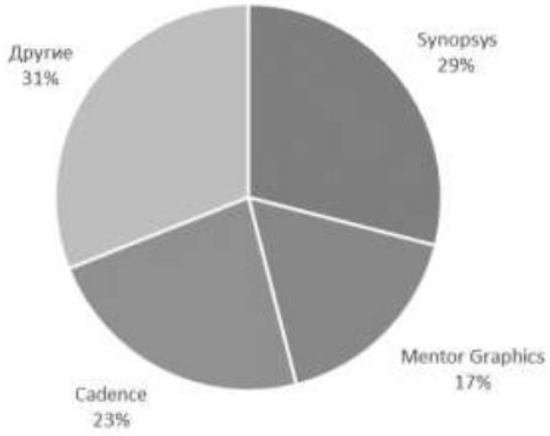


Рис. 18. Доля рынка, закрепленная за САПР различных компаний (по данным на четвертый квартал 2015 г.).

Рис..3.2. Доля рынка, закрепленная за САПР различных компаний

В настоящее время коммерческие САПР поддерживают интегрированные маршруты проектирования, как интегральных микросхем, систем на ПЛИС, так и печатных плат, а также многокристальных модулей. При этом предоставляются средства комплексного решения задач

управления проектом, разработки новых технических решений, анализа и контроля качества получаемых результатов.

В соответствии с представленными на рынке САПР можно выделить следующие стадии получения целевой вычислительной системы:

• уровень системного проектирования;

• уровень технической реализации.

Структура и взаимосвязь результатов разработки вычислительной системы представлены на Рис. 3.3.

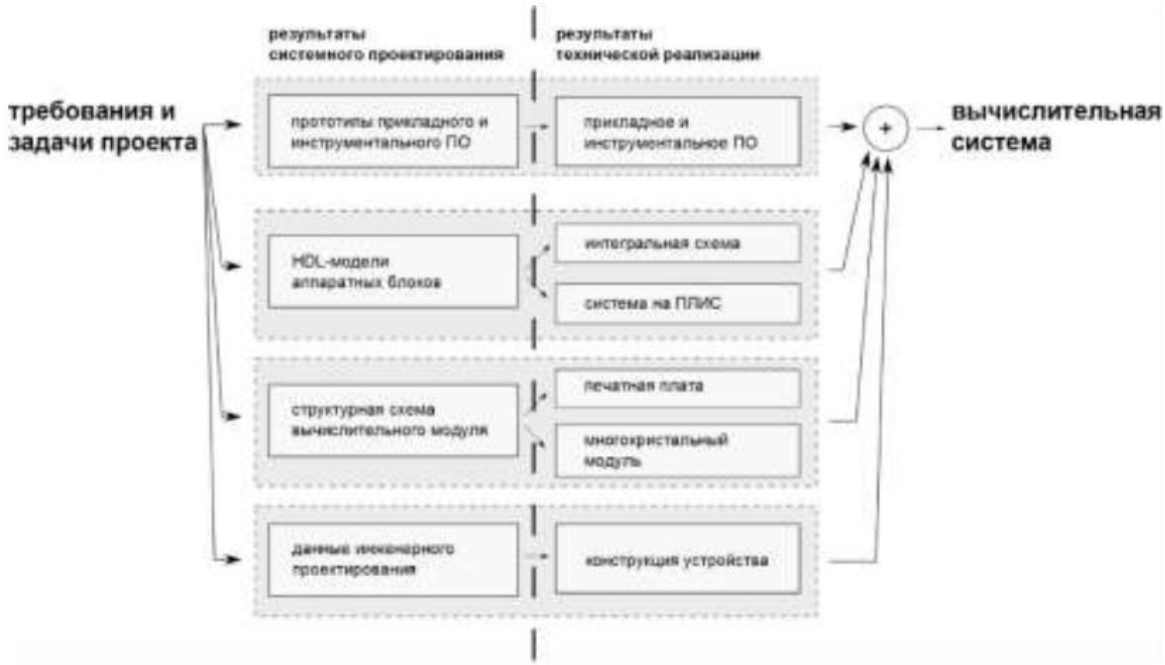


Рис. 3.3. Структура результатов проектирования вычислительной системы средствами САПР

На уровне системного проектирования создается проект системы, который представляется в виде набора файлов, описывающих конкретные, формализованные методы (пути) удовлетворения требований. При этом формат файлов выбирается такой, чтобы содержимое файлов возможно было однозначно интерпретировать на уровне технической реализации. В ходе системного проектирования производится разделение системы на вычислительные модули и определение требования к технологии их реализации/производства в виде печатных плат или многокристальных

модулей, производится уточнение требований к инженерному проектированию (к корпусу, жгутам, креплениям и т.п.).

Под технической реализацией понимается подготовка и непосредственно выпуск составных частей системы в виде интегральных микросхем, систем на ПЛИС, печатных плат и многокристальных модулей.

При производстве цифровых интегральных микросхем или реализации частей системы на ПЛИС результаты системного проектирования представляются в виде HDL-моделей системы и её функциональных узлов. Под HDL-моделями понимаются набор описаний поведения аппаратных блоков с точностью до такта системного сигнала синхронизации.

При производстве печатных плат и многокристальных модулей проект системы может представляться как в абстрактном виде – структурно-функциональной схемы, так и в более детальном – принципиальной схемы будущего устройства (вычислительного модуля).

Существующие технологии позволяют проводить параллельную разработку программного обеспечения (ПО) и аппаратной платформы системы. Это технологии Ко-дизайна.

В идеальном случае прототипы инструментального и прикладного ПО, созданные на этапе системного проектирования, с небольшими доработками возможно использовать на выпущенном экземпляре системы, то есть её технической реализации.

Рассмотрим основные технологии, используемые на каждой стадии маршрута проектирования и поддерживаемые средствами современных САПР.

3.2. САПР системного уровня и языки описания проектов

Основной задачей этапа системного проектирования является создание проекта архитектуры будущей системы и анализ её системных характеристик: производительности, энергопотребления, габаритов, надежности и т.п. Под созданием проекта архитектуры понимается

определение номенклатуры функциональны узлов, их назначения, алгоритмов поведения и связей с другими подсистемами.

На данном этапе применяется технология проектирования системы с использованием виртуальных компонент. Виртуальный компонент – это исполняемая модель функционального узла системы с определенной функциональностью. Поведение данного компонента детализируется в процессе проектирования. Так проектирование архитектуры начинается с использованием виртуальных компонент, в которых не учитывается параметр времени, и заканчивается созданием потактовых моделей соответствующих узлов.

Сегодня у каждого производителя есть свои средства для работы с виртуальными компонентами. Cadence предоставляет соответствующие средства в рамках Virtual System Platform, Synopsys – Virtulazer, CoMET, METeor, Mentor Graphics – Vista.

Данные средства позволяют создавать сами виртуальные компоненты, объединять их в систему, создавать тестовые окружения, проводить моделирование и измерять характеристики работы системы. *Виртуальные компоненты, как правило, описываются на языке SystemC или SystemVerilog с использованием библиотеки TLM и являются транзакционными моделями соответствующих функциональных узлов системы.* Примерами виртуальных компонент являются процессоры, контроллеры периферийных устройств, графические ускорители и др.

Существует три основных способа получения необходимых виртуальных компонент. Они могут быть:

- созданы вручную «с нуля»,
- взяты в качестве готовых блоков из библиотеки производителя
- получены средствами САПР из высокоуровневого описания.

Входными данными для разработки виртуальных компонентов вручную «с нуля», так и с использованием средств САПР является набор

декларативных описаний архитектуры системы и её составных частей. В качестве декларативных описаний могут выступать:

• не стандартизированное описание архитектуры, синтаксис и семантика которого определены в рамках конкретной команды разработчиков;

• описание архитектуры с использованием UML-подобных языков, например, SysML;

• описание архитектуры с использованием ADL (ArchitectureDescription Language) и PDL (Processor Description Language) языков, например, таких как nML, LISA, ArchC и др.;

• математическое описание алгоритмов обработки данных.

Процессы создания виртуальных компонентов из ADL/PDL описания поддержаны современными средствами САПР.

Активные работы по созданию таких средств проводит фирма Synopsys. Она уже выпустила на рынок САПР Processor Designer, где основным языком описания архитектуры является язык LISA, а также недавно создала ещё ряд САПР для проектирования многопроцессорных систем ASIP Designer, MP Designer, в которых используется язык nML.

Получение HDL-моделей и соответствующего инструментального ПО для программирования, отладки и тестирования создаваемых блоков при таком подходе может быть выполнено также с помощью средств САПР.

САПР Processor Designer, ASIP Designer, MP Designer позволяют синтезировать из ADL/PDL описания не только виртуальные компоненты, но и создавать HDL-модели на языках Verilog и VHDL, а также полный набор инструментального ПО для них: компилятор, загрузчик, отладчик, инструменты для тестирования.

Получение HDL-моделей и соответствующего инструментального ПО для программирования, отладки и тестирования создаваемых блоков при таком подходе может быть выполнено также с помощью средств САПР.

Еще одним способом автоматизации проектирования HDL-моделей является использованием средств визуального проектирования отдельных составных частей функциональных узлов, например, устройств управления, контроллеров периферийных интерфейсов. Mentor Graphics для этих целей создала отдельные инструменты: HDL Designer, HDL Author, HDL Visual Elite. В их задачи входят преобразование структурного описания системы в виде системы взаимосвязанных блоков в HDL- описания соответствующей структуры, а также синтез HDL-моделей дискретных алгоритмов управления из их описания в виде формальных моделей конечных автоматов.

Прототипы ПО разрабатываются с использованием виртуальных компонентов. Прикладное ПО запускается непосредственно на соответствующих виртуальных моделях процессоров. При разработке алгоритмов обработки данных могут использоваться средства различных математических пакетов типа Matlab, Simulink, Maple, Mathematica.

Подобные инструменты есть также и у Synopsys. Они включены в САПР System Studio и имеют средства интеграции с проектами Matlab и Simulink.

Под статическим или формальным анализом понимается анализ структурного описания проекта на предмет ошибок.

Под динамическим анализом понимается проверка требований проекта в процессе выполнения исполняемых моделей функциональных блоков в симуляторе с помощью технологий компьютерного моделирования. В процессе моделирования проверяется поведение устройства при различных вариантах входных воздействий.

Поддерживается одновременное моделирование виртуальных компонентов, написанных на языках SystemC и SystemVerilog, а также более низкоуровневых HDL- моделей, написанных на языках Verilog HDL, VHDL (рис. 3.4).

Рис. 22. Использование средств компьютерного моделирования для визуализации поведения разрабатываемой системы.

Рис. 3.4. Средства компьютерного моделирования для визуализации поведения системы

Средства визуализации/анализа проекта на транзакционном уровне поддерживаются в рамках библиотеки TLM для языков SystemC и SystemVerilog.

Для задач динамического анализа Cadence разработала Incisive Enterprise Simulator. Фирма Synopsys реализует соответствующие средства в среде моделирования VCS. Mentor Graphics для данных задач предлагает использовать QuestaSim либо свой более старый, но более стабильный и заслуживший признание множества пользователей продукт ModelSim.

Повышение сложности проектируемых СБИС, жесткие требования к срокам их проектирования (сокращение времени выхода изделия на рынок) поставили перед разработчиками новые проблемы. В сложившихся условиях самостоятельное проектирование разработчиком СнК всех СФ-блоков, входящих в ее состав, не всегда целесообразно. Поэтому в последние годы широкое распространение получила практика разработки отдельных СФ-блоков для их последующего представления на рынок средств проектирования СнК. СФ-блоки, предназначенные для использования в разнообразных проектах, стали называть IP (Intellectual Property) модулями,

тем самым подчеркивается, что эта продукция является предметом интеллектуальной собственности.

СФ-блоки, используемые при проектировании СнК, имеют две основные формы представления:

– в виде топологических фрагментов, которые могут быть непосредственно реализованы в физической структуре кристалла;

— аппаратно реализованные (hard) СФ-блоки;

– в виде моделей на языке описания аппаратуры (Verilog, VHDL), которые средствами САПР могут быть преобразованы в топологические фрагменты для реализации на кристалле;

— синтезируемые (soft) СФ-блоки.

Таким образом, разработчик может либо непосредственно «вмонтировать» в структуру проектируемой СБИС топологически готовый СФ-блок, либо использовать имеющуюся модель СФ-блока и выполнить его схемотехническое и топологическое проектирование в составе реализуемой СБИС СнК.

3.3. Особености проектирования программируемых СнК

Использование программируемых систем на кристалле (ПСнК) — это сравнительно новый подход при разработке встраиваемых приложений. Такие устройства позволяют создать оптимальную конфигурацию, уменьшить количество компонентов и мощность, потребляемую системой, а также сократить время разработки проекта. Спектр решений для встраиваемых систем весьма широк: от ASIC до микроконтроллеров (МК). Проекты на базе ASIC чрезвычайно дороги и требуют длительного времени разработки, но возможности реализуемых функций практически не имеют ограничений. Проекты на МК могут быть очень быстро созданы — их разработка занимает месяцы или даже недели, но реализуемые функции ограничены возможностями кристалла, которые определяет производитель МК.

Однако несмотря на то, что эти два подхода различаются с точки зрения технической реализации, у них есть много схожего: оба они, главным образом, используют процессорные ядра ARM, включают стандартные коммуникационные интерфейсы, содержат большое количество аналоговых функциональных блоков на кристалле, а также поддерживают различные режимы малого энергопотребления.

В спектр встраиваемых решений входят и программируемые платформы, обеспечивающие необходимую гибкость реализации функций, которые могут быть интегрированы в устройстве. Примерами такого класса устройств являются FPGA и CPLD, которые характеризуются широкими функциональными возможностями и существенной емкостью. Однако эти устройства не реализуют всех требований к программируемым платформам, т.к. ориентированы лишь на цифровые функции и решения. Следовательно, требуется программируемая платформа, которая обеспечивает гибкость при создании как цифровых, так и аналоговых функций на кристалле и в то же время не требует от разработчика, чтобы он был экспертом в этих областях. Такая платформа должна включать стандартный процессор, а также поддерживаться широко доступными инструментами разработки и необходимой экосистемой.

Идеальная система должна точно отвечать требованиям приложения, обеспечивая весь необходимый набор периферии и интерфейсов на кристалле, высокий уровень производительности и отсутствие бесполезных функций. Чтобы достичь такой гибкости, требуется платформа, которая позволяет создать определенную конфигурацию с использованием аналоговых блоков и программируемой цифровой логики на кристалле. При этом не обязательно, чтобы разработчики были специалистами в HDL-программировании или аналоговом проектировании.

Микроконтроллеры поддерживаются инструментами разработки и имеют в своем составе аналоговые блоки, но в них совершенно отсутствует

возможность конфигурирования. FPGA имеют конфигурируемую логику и достаточно удобное программное обеспечение, но их недостаток заключается в отсутствии возможности реализации аналоговых функций и сравнительно высоком энергопотреблении.

Платформой, которая способна полностью заполнить на рынке нишу между ASIC и МК, является программируемая система на кристалле. Разработки в этой области ведет ряд мировых производителей, которые специализируются на устройствах, интегрирующих аналоговые и цифровые блоки на одном кристалле. Компания Cypress Semiconductor относится к числу пионеров в области создания ПСнК с аппаратными ядрами и использует для обозначения таких систем термин РСнК (Programmable System on Chip).

Под ПСнК понимается микросхема с интегрированным процессором, памятью, логикой и периферией. При этом окончательная конфигурация программируется пользователем под конкретную задачу. ПСнК можно разделить на однородные и блочные системы. В однородных ПСнК одни и те же области кристалла при программировании могут быть использованы для реализации разных функций. При этом разработчик сам размещает на кристалле необходимые блоки, которые называются программными ядрами. При проектировании таких систем можно использовать IP-блоки. Однородные ПСнК отличаются большой гибкостью и универсальностью применения, хотя приобретение IP-блоков требует значительных затрат.

В блочных ПСнК используются аппаратные ядра, т.е. области кристалла, выделенные под строго определенные функции и выполненные по технологии ASIC. Такой подход снижает универсальность, но уменьшает площадь кристалла и значительно повышает производительность системы в целом.

ПСнК компании Cypress Semiconductor в основе своей архитектуры имеют встроенные аппаратные ядра (в том числе ARM-процессор и память), а также программируемые аналоговые и цифровые блоки.

Программируемые системы на кристалле имеют три основных преимущества:

– Интеграция. Способность интегрировать на одном кристалле дискретные компоненты, снизить таким образом стоимость необходимых элементов и уменьшить затраты на производство (стоимость печатной платы), а также сократить мощность, рассеиваемую системой, за счет уменьшения количества используемых компонентов.

– Программируемость аналоговых функций. Возможность интеграции аналоговых компонентов, таких как усилители, фильтры, АЦП, преобразователи сигналов и др.

– Гибкость. Традиционное преимущество программируемых устройств — постоянная возможность внесения изменений в систему, параллельная работа над проектом, разработка прототипа и подготовка производства — ускоряет вывод продукта на рынок.

3.3.1. ARM-архитектура —стандарт для встраиваемых систем

В течение многих лет ARM-архитектура была де-факто стандартом на рынке встраиваемых систем, подобно тому как Intel доминировала на рынке ПК. После появления несколько лет назад семейства процессорных ядер Cortex-M сейчас трудно найти микроконтроллер, не использующий ARM-процессор.

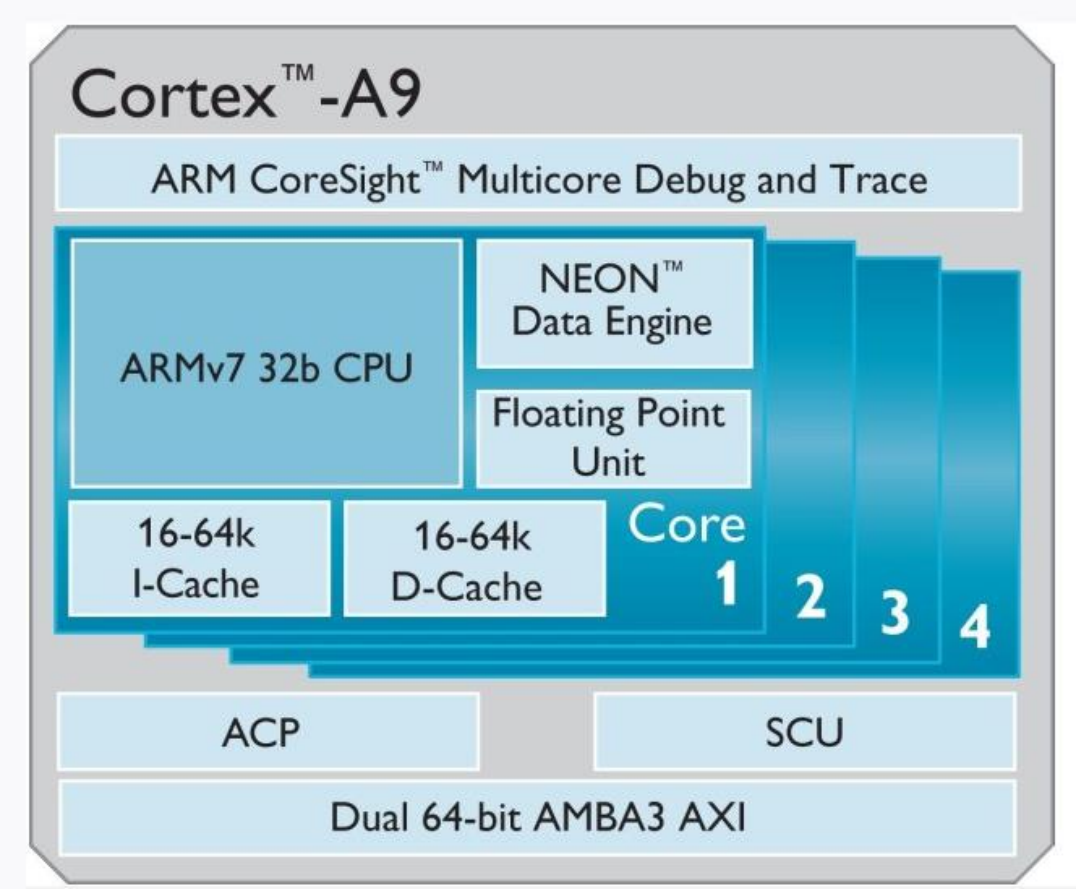


Схема строения однокристальной системы с четырьмя ядрами ARM Cortex-A9

Архитектура ARM (от англ. Advanced RISC Machine — усовершенствованная RISC-машина; иногда — Acorn RISC Machine) — семейство лицензируемых 32-битных и 64-битных микропроцессорных ядер разработки компании ARM Limited.

Среди лицензиатов - AMD, Apple, Analog Devices, Atmel, Xilinx, Cirrus Logic[en], Intel (до 27 июня 2006 года), Marvell, NXP, STMicroelectronics, Samsung, LG, MediaTek, Qualcomm, Sony, Texas Instruments, Nvidia, Freescale, Миландр, ЭЛВИС, HiSilicon.

Значимые семейства процессоров: ARM7, ARM9, ARM11 и Cortex.

Эти процессоры имеют низкое энергопотребление, поэтому находят широкое применение во встраиваемых системах и преобладают на рынке мобильных устройств, для которых данный фактор немаловажен.

В основном процессоры семейства завоевали сегмент массовых мобильных продуктов (сотовые телефоны, карманные компьютеры) и встраиваемых систем

средней и высокой производительности (от сетевых маршрутизаторов и точек доступа до телевизоров). Отдельные компании заявляют о разработках эффективных серверов на базе кластеров ARM процессоров, но пока это только экспериментальные проекты с 32-битной архитектурой.

Архитектура развивалась с течением времени и, начиная с ARMv7, были определены 3 профиля:

A (application) — для устройств, требующих высокой производительности (смартфоны, планшеты);

R (real time) — для приложений, работающих в реальном времени;

M (microcontroller) — для микроконтроллеров и недорогих встраиваемых устройств.[3]

Профили могут поддерживать меньшее количество команд (команды определенного типа).

Режимы

Процессор может находиться в одном из следующих операционных режимов:

User mode — обычный режим выполнения программ. В этом режиме выполняется большинство программ.

Fast Interrupt (FIQ) — режим быстрого прерывания (меньшее время срабатывания).

Interrupt (IRQ) — основной режим прерывания.

System mode — защищённый режим для использования операционной системой.

Abort mode — режим, в который процессор переходит при возникновении ошибки доступа к памяти (доступ к данным или к инструкции на этапе prefetch конвейера).

Supervisor mode — привилегированный пользовательский режим.

Undefined mode — режим, в который процессор входит при попытке выполнить неизвестную ему инструкцию.

Переключение режима процессора происходит при возникновении соответствующего исключения или же модификацией регистра статуса.

Набор команд

Чтобы сохранить устройство чистым, простым и быстрым, оригинальное изготовление ARM было исполнено без микрокода, как и более простой 8-разрядный процессор 6502, используемый в
предыдущих микрокомпьютерах от Acorn Computers.

Функции RISC

Архитектура ARM обладает следующими особенностями RISC:

Архитектура загрузки/хранения

Нет поддержки нелинейного (не выровненного по словам) доступа к памяти (теперь поддерживается в процессорах ARMv6, за некоторыми исключениями, и полностью в ARMv7)

Равномерный 16x32-битный регистровый файл

Фиксированная длина команд (32 бита) для упрощения декодирования за счет снижения плотности кода. Позднее режим Thumb повысил плотность кода.

Одноцикловое исполнение

Справедливость этого подтверждается той поддержкой, которую получает ARM-технология в области встраиваемых систем.

Поставщики IP-блоков для СнК рассматривают в качестве наиболее приоритетной архитектуры для своих новых продуктов шину AMBA от ARM. Компании, разрабатывающие ОС реального времени (ОСРВ), в первую очередь включают поддержку ARM-процессоров в своих системах. Таким образом, любая платформа, которая рассчитывает на успех на рынке программируемых устройств, должна использовать процессорное ядро ARM. Причина этого заключается в том, что для исключения потерь времени при проектировании СнК разработчикам необходимо предложить хорошо известную им платформу, включая архитектуру центрального процессора,

компиляторы, интегрированную среду разработки, отладчики, ОСРВ и комплект микропрограммного обеспечения.

При переносе программного обеспечения на другую платформу могут возникнуть большие трудности. За исключением действительно бюджетных систем, которые могут работать на базе проверенных временем 8-разрядных процессоров, подобных 8051, любая программируемая платформа, не поддерживающая ARM-процессор, быстро уходит на второй план, обслуживая лишь некоторые сегменты рынка, в которых ARM не является доминирующей технологией.

3.3.2. Реализация аналоговых функций в программируемых устройствах

Существует много доступных платформ, в которые интегрированы различные аналоговые функции, но проблемой является реализация в СнК аналоговых схем более низкого уровня, которые традиционно не входили в состав кристалла. Причем, введение аналоговых функций в кристалл должно полностью освободить разработчика от проблем, связанных с проектированием аналоговых блоков, и представлять собой процесс, подобный реализации любого другого IP-блока в системе.

При создании цифровых функций следует ввести необходимый функциональный блок в проект, развести соответствующие линии ввода/вывода, проверить синхронизацию.

При аналоговом проектировании поведение даже простых схем зависит от конкретной конфигурации, разводки кристалла и топологии внешней платы, что в целом довольно трудно учитывать. Например, блоки переключаемых конденсаторов могут быть сконфигурированы самыми разными способами: в виде усилителя с программируемым коэффициентом усиления, трансимпедансного усилителя, аналогового фильтра и даже смесителя частоты. Функционирование этих блоков зависит от их конфигурации и частоты переключения конденсаторов. Возможность реализации таких блоков на одном кристалле весьма привлекательна, однако

сложности проектирования, связанные с учетом всех нюансов их работы, и конфигурирование многочисленных регистров не вызывают восторга у разработчиков. Решить эту проблему позволяет соответствующее программное обеспечение. Включение аналоговых функций в устройство — это лишь одна задача. Но без удобного инструмента разработки, который упрощает процесс конфигурирования, реализация множества аналоговых функций в составе кристалла существенно увеличивает время проектирования устройства. Таким образом, программный инструментарий, обеспечивающий высокий уровень абстрактного представления и задающий параметры проекта, освобождает разработчика от дополнительных забот.

Другими словами, подобно тому как разработчику не требуется вникать в то, как работает устаревший АЦП, ему не нужно изучать конфигурацию регистров и битовые поля интегрированных устройств, чтобы просто ввести в проект необходимый блок АЦП. Разработчик должен иметь возможность сконфигурировать АЦП на базе таких его свойств и характеристик как доступное разрешение, максимальная частота выборки, диапазон напряжений и т.д. Следующим шагом после выбора АЦП для проекта должна быть его адаптация к требованиям приложения путем задания желаемого значения параметра, например, диапазона входного напряжения.

Примером инструмента разработки, который выполняет конфигурирование аналоговых блоков для программируемой платформы, является ПСнК Creator компании Cypress Semiconductor. ПСнК Creator поддерживает устройства с архитектурой РСнК 3 и ПСнК 5 с ARM-процессором с помощью интерфейса ввода описания схемы, который позволяет пользователю конфигурировать выбранные компоненты с помощью редакторов параметров.

Предварительно созданные аналоговые (и цифровые) компоненты представлены в каталоге, где имеется доступ к примерам проектов и

технической документации. Когда компонент вводится в проект, инструментальное средство генерирует прикладные программные интерфейсы (API) для приложений, которые обеспечивают взаимодействие с ним без необходимости расшифровки наборов регистров и синхронизации с АЦП.

3.3.3. Интегрирование аналоговой и цифровой частей проекта с помощью программного средства

При том, что решение проблемы программирования аналоговых функций с помощью ввода описания схем аналоговых компонентов является эффективным, оно не является законченным. Разработчикам необходим инструмент, который поддерживает и цифровую часть проекта и, что может быть более важно, программное приложение.

Ввод описания схемы с помощью программного средства разработки далеко не нов в цифровом проектировании, и платформы, поддерживающие интеграцию цифровой и аналоговой частей проекта в одном устройстве, становятся все более популярными. Однако разработчикам неудобно использовать множество инструментов для работы над одним проектом. Инженеры предпочли бы вводить цифровые и аналоговые элементы проекта в одном редакторе и создавать, отлаживать и тестировать такой проект в одной среде разработки.

Разработчики МК выполняют проектирование системы в интегрированной среде разработки (**Integrated Development Environment — IDE**), которая объединяет редактирование исходного кода, управление проектом, а также инструменты компилирования и отладки в единой интегрированной системе. То же самое, по сути, необходимо и для проектирования ПСнК, в которых интегрируются аналоговые и цифровые блоки.

После построения схемы система разработки автоматически генерирует API для компонентов. С помощью API разработчик может,

например, поменять время ожидания счетчика, заблокировать прерывание, выключить тактовый сигнал и выполнить другие действия без использования справочного руководства или внесения изменений в код программы.

Генерация API на самом деле является лишь расширением программного обеспечения для абстрактного представления системы на базе аппаратно-устанавливаемых параметров. Например, для того чтобы установить тактовый сигнал, разработчик должен ввести его в проект и установить желаемую частоту. Инструмент разработки решает, каким образом получить требуемое значение частоты в пределах заданных допусков, имея в распоряжении доступные источники тактовой частоты как на кристалле, так и вне его.

Инструмент, который генерирует API для системных ресурсов, таких как тактовые сигналы, прерывания, DMA и линии ввода/вывода, позволяет существенно сэкономить время проектирования и делает разработку системы на базе программируемого устройства даже проще, чем для систем на базе МК и ASIC. В сочетании с популярными для встраиваемых систем процессорами с ARM-ядрами интегрированный инструмент ввода схем представляет собой привлекательную альтернативу для всего спектра встраиваемых проектов.

Технология программируемых устройств постоянно совершенствуется благодаря поддержке мощных процессорных ядер с ARM-архитектурой и возможности интеграции многих аналоговых функций в кристалле, что позволяет уменьшить количество компонентов в системе. Все более очевидно, что именно программные средства, поддерживающие программируемые платформы, определяют, будет ли успешной данная платформа на рынке или нет.

В последние годы разработаны и широко применяются языки для моделирования смешенных аналогово-цифровых систем (SystemC-AMS, Verilog-AMS, VHDL-AMS и др.).

3.4. Использование сложнофункциональных блоков

Системы на кристалле (СНК) это методология разработки заказных микросхем на основе уже готовых сложнофункциональных блоков (СФ-блоков). Основой методологии является совместимость СФ-блоков в системе по принципу “включи и работай” (plug-and-play в англоязычной литературе).

Для реализации этого принципа СФ-блоки должны разрабатываться как автономные устройства со своими системами питания, синхронизации и интерфейсами.

Первая составляющая методологии систем на кристалле – это единые требования к СФ-блокам. Обязательными являются требования технологической совместимости, наличие детальных спецификаций и моделей высокого уровня.

Вторая составляющая – это конкретные базовые решения, обеспечивающие выполнение требований совместимости СФ-блоков.

СФ-блоки должны включать и элементы инфраструктуры системы (интерфейсы, системы питания и синхронизации, встроенные средства контроля). Инфраструктурные блоки не должны занимать большую площадь кристалла и использовать много внешних компонентов.

Третья составляющая – это универсальные правила, обеспечивающие объединение СФ-блоков в систему с наименьшим взаимным влиянием.

Маршрут проектирования СНК существенно сокращается и упрощается по сравнению с маршрутом полностью заказных микросхем. Методология проектирования СНК приближается к методологии разработки систем на печатных платах. Основной этап проектирования – это системный. Именно на этом этапе определяются все основные характеристики разрабатываемого микроэлектронного устройства. Этапы функционального проектирования и верификации объединяются и упрощаются. Моделирование схемы на транзисторном и вентильном уровнях вообще может не проводиться. Используются только модели высокого уровня.

Возможно и исключение этапа макетирования СНК, если все используемые СФ-блоки аттестованы и адекватно описаны на языках высокого уровня (VHDL, VHDL-AMS и др.). Физическое проектирование также существенно упрощается, т.к. число используемых СФ-блоков и сигнальных связей между ними сравнительно невелико. По существу СНК являются полузаказными микросхемами и основные затраты приходятся на создание системы проектирования и распространения СФ-блоков. Основная выгода состоит в том, что каждый

СФ-блок используется во многих изделиях. Кроме этого, в несколько раз сокращается время разработки конечных продуктов. Методология проектирования систем на кристалле предписывает выполнение проекта по двум направлениям.

Направление “сверху – вниз” включает:

составление общей спецификации на СНК;

разработку системной модели;

подготовку требуемой номенклатуры СФ-блоков;

функциональное моделирование СНК;

физическое проектирование;

верификацию модели.

Направление “снизу вверх” включает:

подготовку спецификаций на требуемые СФ-блоки;

отбор готовых блоков;

приобретение или разработку недостающих блоков;

разработку и верификацию моделей высокого уровня для используемых СФ-блоков.

По уровню затрат на разработку и подготовку производства СНК занимают промежуточное место между универсальными микросхемами и ПЛИС. Промежуточных уровней можно выделить несколько. Самый затратный уровень – это комплектование проекта имеющимися СФ-блоками

и разработка недостающих. При этом требуется полный цикл физического проектирования кристалла. В структурных СНК на базовом кристалле уже размещены СФ-блоки. Функциональная схема формируется из заданного набора СФ- блоков путем создания системы металлизированных соединений. Если структура СФ-блоков на базовом кристалле повторяет структуру ячеек ПЛИС, то проект можно полностью отладить на макете с ПЛИС, а затем перенести на базовый кристалл. Такие СНК называют “жесткие копии ПЛИС” (FPGA Hard Copy). Выигрыш достигается за счет исключения системы программирования соединений. Площадь кристалла при этом сокращается до 10 раз, соответственно повышается быстродействие и снижается потребляемая мощность. Самый дешевый способ разработки - это конфигурируемые СНК. По сути – это уже структурные ПЛИС. Разработчик программирует и функции СФ-блоков и связи между ними. Отличие от регулярных ПЛИС состоит в том, что СФ-блоки специализированы и достаточно разнообразны. Специализация блоков позволяет в несколько раз сократить площадь кристалла по сравнению с регулярными ПЛИС.

САПР для проектирования СнК:

HDL Designer - Создание проекта на языках Verilog, VHDL, System Verilog

ModelSim - Система моделирования и среда верификации цифровых систем.

Precision - Средство логического синтеза высокопроизводительных ПЛИС типа PLD и FPGA, оптимизированное с точки зрения простоты использования и высокого качества результатов.

Vista - Расширенный набор средств отладки проектов на языке SystemC, включающий также мощные механизмы просмотра и отладки RTL и C/C++-ориентированных описаний.

Pyxis- Программный для работы с цифровыми, аналоговыми и смешанными проектами.

ADMS - комплексная система моделирования многоязыковых проектов описанных на таких языках как VHDL, Verilog, SystemVerilog, SPICE и т.д.

Eldo - Система аналогового SPICE-моделирования

Calibre - Программный комплекс для верификации топологии

Основные дисциплины, читаемые магистрам в «УНЦ проектирования Mentor Graphics» по направлению «Проектирование систем на кристалле»:

Проектирование сложных систем

Проектирование микроэлектронных устройств

Функциональное проектирование и верификация систем на кристалле

Проектирование аналогово-цифровых блоков и систем с использованием ADMS

Физическая и формальная верификация проекта средствами Calibre

Основы топологического проектирования с использованием Pyxis

Логический и физический синтез ПЛИС типа PLD/FPGA

Основы конструирования и технологии электронных средств с использованием Pyxis

Перспективная база электронных средств

Глава 4. Методология проектирования с использованием языка SystemC

4.1. Недостатки методологии традиционных САПР при проектировании СнК

Как отмечалось выше проектирование конструкции системы на кристалле (СнК) - (СнК) имеет отличительные особенности:

• Несколько доменов проектирования: аппаратное, программное, аналоговое;

• Множество исходных компонентов: DSP, ASIC, IP-ядра;

• Жесткие ограничения: работа в реальном времени, с низким энергопотреблением.

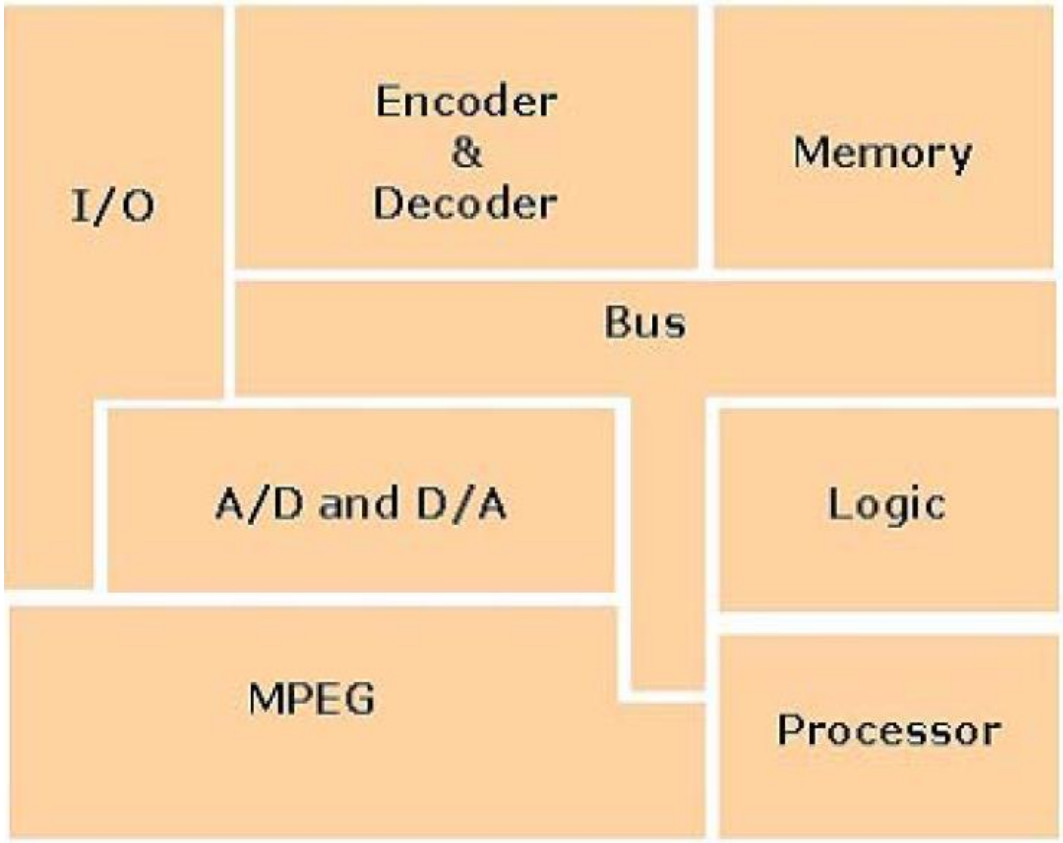


Рис. 4.1. Типичная конструкция СнК

Существенные ресурсы каждого СнК составляют ядра интеллектуальной собственности. Ядро IP – это сложный модуль, выполняющий определенную задачу и созданный для повторного

использования. Эти IP -ядра являются строительными блоками для СнК и занимают очень большой процент площади СнК.

Одна из ключевых задач проекта СнК – разбиение системной функциональности по дихотомии на аппаратное обеспечение (HW) и программное обеспечение (SW) или совместно HW/SW. Функциональность, однажды отнесенная к программному обеспечению, теперь для лучшей производительности может быть реализована на аппаратном уровне, а компоненты аппаратного обеспечения должны интегрироваться с программными API (**Application Programming Interface**) более высокого уровня. В текущей методологии САПР делается априорное разделение и таким образом создаются отдельные аппаратные и программные спецификации. Изменения в разделении HW/SW требуют обширной реорганизации, которая обычно заканчивается неоптимальными проектами. Кроме того, при внедрении встроенных процессоров в ПЛИС, дизайнеры цифровых устройств знакомятся с новой областью САПР, которая включает в себя одновременную разработка как аппаратного, так и программного обеспечения (программа выполненняется на встроенном процессоре).

На рис. 4.2 условно показано соотношение аппаратной и программной части в СнК. Существенными ресурсами каждого СнК являются ядрами интеллектуальной собственности. Ядро IP – это сложный модуль, выполняющий определенную задачу и созданный для повторного использования. Эти IP - ядра являются строительными блоками для СнК и занимают очень большой процент площади СнК.

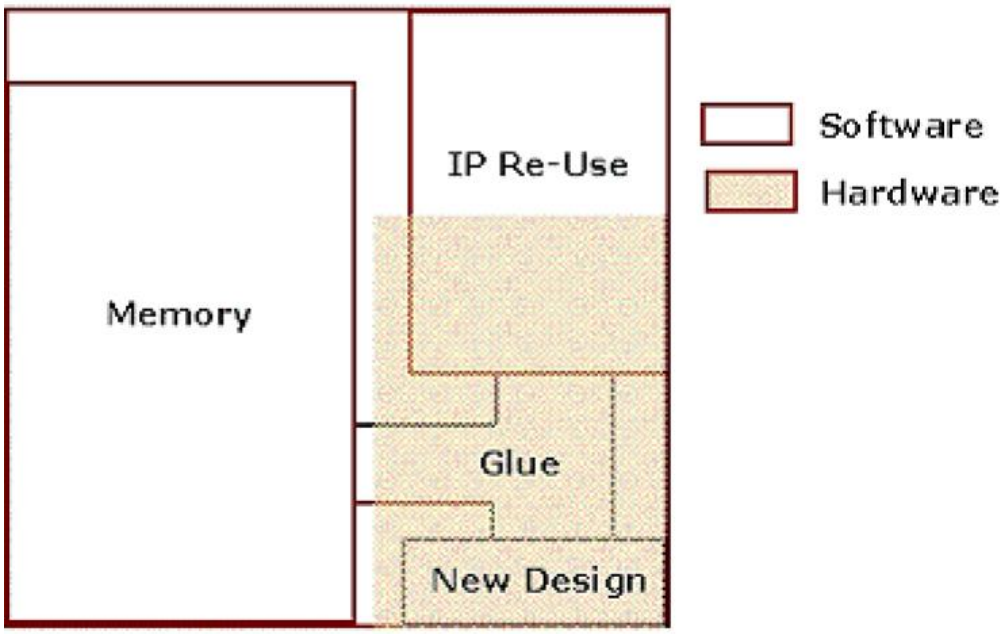


Рис. 4.2. Соотношение аппаратной и программной части в СнК

Еще одним критическим недостатком текущей методологии САПР является то, что она является RTL (register transfer level)-ориентированной, что, в связи с увеличением сложности схемы время моделирования возрастает и постепенно становится неприемлемым. Понятно, что скорость моделирования полной системы является решающим фактором при разработке сложной цифровой системы, такой как СнК. Эта проблема проверки еще более усугубляется, когда количество тестовых векторов, необходимых для проверки, возрастает в 100 раз каждые шесть лет, что в 10 раз превышает количество вентилей на чипе, как указанное законом Мура. Кроме того, полная верификация (проверка техническим условиям) и валидация (проверка соответствия поставленным функциональным целям) системы часто невозможна до тех пор, пока полностью не построен рабочий прототип. Это особенно актуально для распределённой и гетерогенной среды, такой как сетевая встроенная система.

Очевидно, что для сокращения времени разработки и затрат компьютерные инструменты проектирования (САПР) теперь должны включать функции, которые облегчают проектирование пространственное и архитектурное, а также дают более высокий уровень абстракции. Среди

решений вышеупомянутых вопросы проектирования, которые сегодня активно обсуждаются, есть проектирование на уровне абстракции электронной системы (ESL) и применение стандартного языка разработки SystemC.

Этот подход потребует совместного проектирования и совместного моделирование аппаратно-программного обеспечения (HW/SW) , он облегчает исследования в области проектирования и дает высокая скорость моделирования. Предложены методологии совместного моделирования, основанные на SystemC, с симулятором набора инструкций (ISS) в качестве модели процессора в общем исходном файле.

Перечислим основные факторы, которые привели к созданию новой методологии проектирования, основанной на SystemC:

• сложность СнКs продолжает увеличиваться;

• использование моделирования на уровне вентилей или даже RTL для характеристики и изучения полного СнК для типичных сценариев прецедентов не представляется возможным;

• создание моделей на этих уровнях занимает довольно много времени;

• полные модели обычно готовы на поздней стадии разработки системы и внести в них изменения затруднительно или невозможно;

• симуляция идет слишком медленно.

Для решения проблемы требуется:

• модель с более высоким уровнем абстракции на основе SystemC;

• меньше архитектурных деталей в начальном в потоке проектирования;

• более высокая скорость моделирования

• возможность моделирования более сложных систем.

4.2. Цели SystemC

Одной из основных задач SystemC 2.0 является то, чтобы моделировать на системном уровне. Это моделирование систем над уровнем RTL

(register-transfer level) абстракции, в том числе систем, которые могут быть реализованы в программном или аппаратном обеспечении или некоторой комбинации этих составляющих. Модели RTL имеют целью выполнение системы с использованием регистров и комбинационной логики. Одна из проблем в создании языка проектирования на уровне системы является то, что существует широкий спектр дизайнерских моделей вычисления, дизайна уровней абстракции и дизайна методологии, которые используются в конструкции системы. Для решение этой проблемы в SystemC 2.X, небольшой, но очень важной задачи, системное моделирование была добавлено к языку. На вершине этой языкового фундамента мы можем добавить более конкретные модели вычислений, проектирования библиотек, методических указаний по моделированию и методологии дизайна, которые необходимы для проектирования системы. Небольшой, универсальный фундамент моделирования в SystemC 2.0 называется "ядро языка" и является центральным компонентом стандарта SystemC 2.0.

Другие компоненты стандарта SystemC 2.0 включают в себя элементарные модели библиотеки, построенные на ядре языка (например, таймеры, FIFOs, сигналы и т.д.), которые широко применяются. Следует признать, что много различных моделей вычислений и проектных методик могут быть использованы в сочетании с SystemC. По этой причине расчетные библиотеки и модели, необходимые для поддержания этих специфических методик проектирования, целесообразно отделять от ядра стандартного языка SystemC 2.0.

SystemC 2.0 - это язык моделирования, основанный на C ++. Он расширяет возможности C ++, позволяя моделировать описания аппаратных средств, добавляет моделирование аппаратных описаний, библиотеку классов функций, типов данных и другие языковые конструкции на C ++. Эта библиотека классов содержит мощные новые механизмы, которые

позволяют проектным группам моделировать и проверять конструкции, выраженные в истинных системных уровнях абстракции, уточнить их, чтобы отразить варианты реализации и, наконец, связать модель системы с реализацией и проверкой аппаратного обеспечения.

SystemC является надстройкой C/C++ и содержит специальные библиотеки для моделирования HW.

От C/C++ SystemC включает в себя следующие функции:

• Классы и объекты;

• Инкапсуляция - данные и поведение;

• Перегрузка операторов - новые типы и поведение;

• Усиление печати - дополнительная защита;

• Наследование - повторное использование декларации;

• Шаблоны - шаблоны построения.

Преимущества процесса проектирования на основе C / C ++ состоят в следующем:

• Производительность;

• Технические характеристики между архитектурой и исполнением является изменяемыми;

• Высокоскоростное и высокоуровневое моделирование и прототипирование;

• Уточнение, отсутствие перевода на аппаратное обеспечение (отсутствие «семантического разрыва»);

• Уровень системного уровня;

• Завтрашние разработчики систем будут разрабатывать больше программного обеспечения и меньше аппаратного обеспечения;

• Совместная разработка, совместное моделирование, совместная проверка, совместная отладка;

• Аспект повторного использования;

• Оптимальная поддержка повторного использования объектно-ориентированными методами;

• Эффективное повторное использование testbench;

• Особенно широко распространен и широко используется C / C ++!

Недостатки процесса проектирования на основе C / C ++.

• C / C ++ не был создан для проектирования аппаратного обеспечения!

• C / C ++ не поддерживает:

• Аппаратная коммуникация

• Сигналы, протоколы

• Понятие времени

• Тактирование, операции с чередованием по времени

• Параллельность

• Аппаратное обеспечение является одновременно параллельным, работает параллельно;

• Реактивность;

• Аппаратура по своей сути реактивна, реагирует на воздействия, взаимодействует со своей средой (требует обработки исключений);

• Типы аппаратных данных;

• Тип бита, тип вектора бит, многозначные логические типы, целые типы со знаком и без знака, типы с фиксированной точкой.

SystemC - это единый язык для определения, совместного моделирования, уточнения системы аппаратных и программных компонентов вплоть до уровня передачи регистров для синтеза. SystemC предоставляет ядро для моделирования исполняемой системы, написанное в SystemC. Он может быть использован для описания циклических точных моделей на системном уровне, разработки алгоритмов программного обеспечения и аппаратной архитектуры.

SystemC отвечает следующим требованиям:

• Разрешает совместное проектирование аппаратного и программного обеспечения и совместную проверку;

• Быстрое моделирование для проверки и оптимизации;

• Плавный переход на аппаратное и программное обеспечение;

• Поддержка повторного использования проекта и архитектуры;

• Тактирование.

Проектирование систем на кристалле (СнК) необходимо выполнять на четырех уровнях абстрактного описания:

Архитектура;

Поведенческий уровень;

RTL – уровень;

Уровень вентилей.

Скорость итераций увеличивается при проектировании на системном уровне.

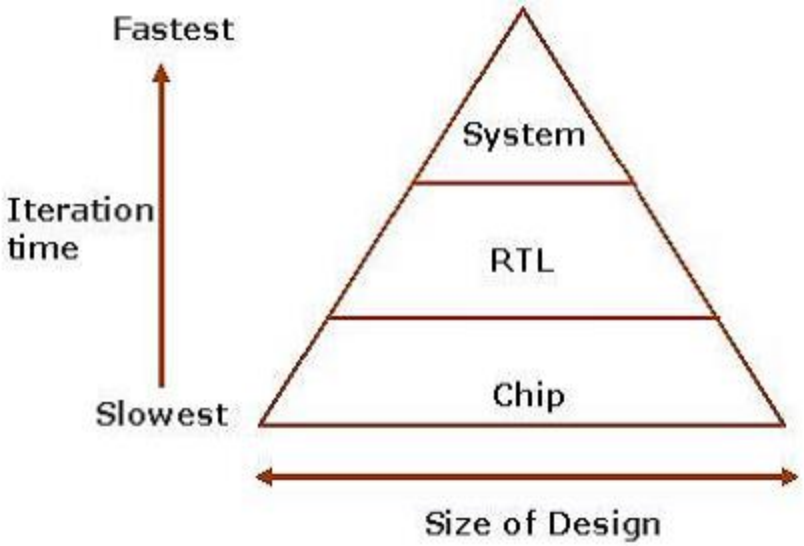


Рис. 4.3. Изучение конструкций на разных уровнях абстракции

К языку программирования на системном уровне предъявляются следующие требования:

1. Спецификация и дизайн на различных уровнях абстракция

2. Создание исполняемых спецификаций конструкции.

3. Создание исполняемых платформ моделей, представляющих возможную реализацию архитектуры.

4. Быстрое моделирование для того, чтобы исследовать дизайн-пространство.

5. Конструкции для разделения функциональности от коммуникаций.

Сравнение применимости различных языков проектирования для решения задач разного уровня показано на рис. 4.4. Как видно, SystemC имеет наиболее широкие возможности моделирования систем на разных уровнях описания.

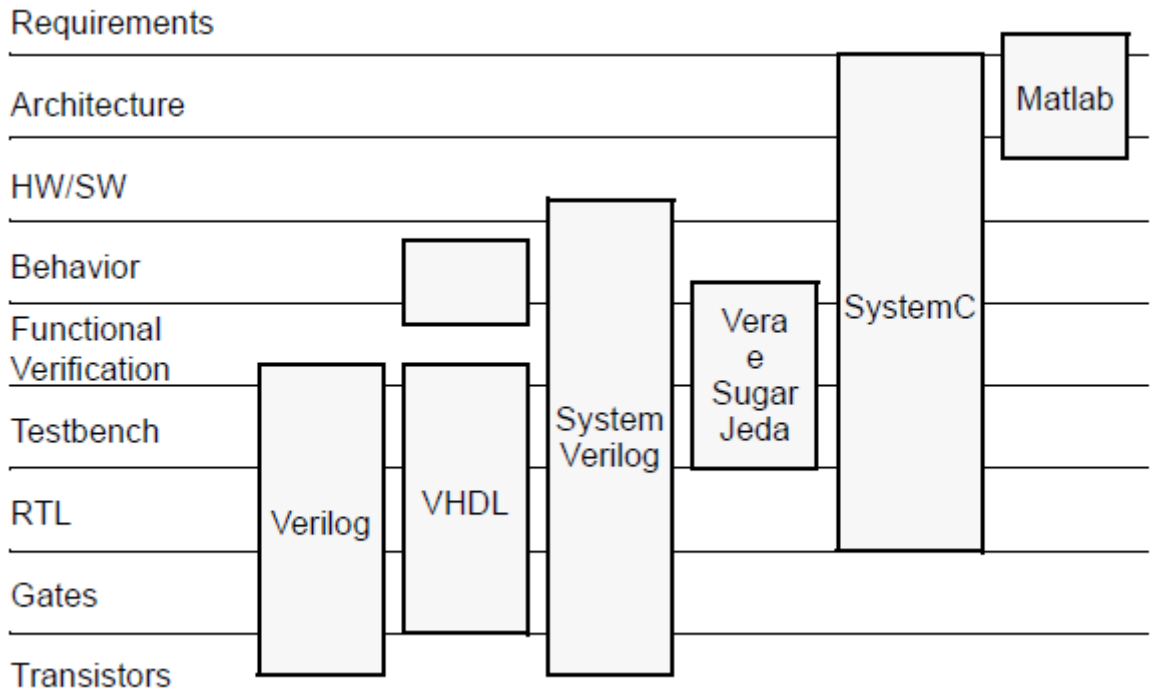


Рис. 4.4. Сравнение применимости различных языков проектирования

С быстро возрастающей сложностью конструкции и ростом стоимости ошибки или отказа, разработчики системы в большинстве областей продукции требуется похожий подход сверху вниз подход, но с улучшенной

методологией. Возникшая методология основана на моделировании на уровне транзакций (*Transaction – level model* -TLM) и используется в SystemC.

Моделирование на уровне транзакций является новой концепцией без точного определения. Рабочая группа Open SystemC Initiative (OSCI) в настоящее время дала определение набора терминологии для TLM и в конечном итоге развивает TLM стандарты. На самом деле, когда инженеры говорят о TLM, они, вероятно, говорить об одном или нескольких из четырех различных стилей моделирования.

Использование TLM позволяет испытывать модели на ранних этапах процесса разработки системы. TLM является концепцией, которая не зависит от языка. Тем не менее, для реализации и результативности TLM модели, полезно иметь такой язык, как SystemC, который имеет функции поддержки независимого уточнения функциональных возможностей и коммуникаций, что имеет решающее значение для эффективного развития TLM.

4.3. Уровни моделирования в SystemC

В SystemC используют семь уровней моделирования:

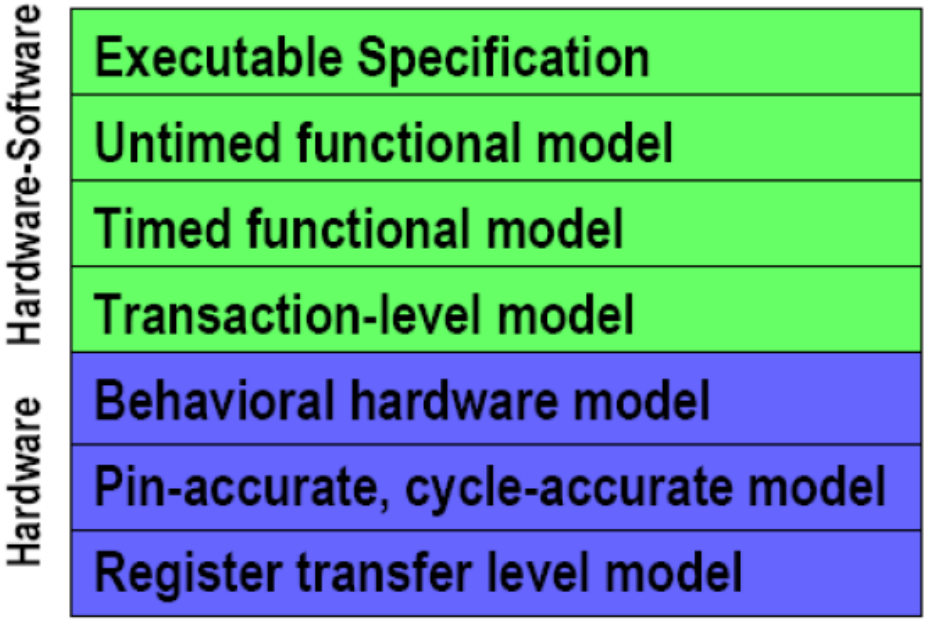


Рис. 4.5. Уровни моделирования в SystemC

1. ***Executable specification*** (исполняемая спецификация) – это модель, которая прямо передает спецификацию проекта в SystemC. Модель предусматривает функционирование конструкции таким образом, который независит от возможного выполнения. Если присутствуют временные заадержки, они учитываются в исполняемой модели.

2. ***Untimed functional model*** подобна предыдущей, но временные задержки не присутствуют в модели. Обычно коммуникации моделируются с использованием FIFO с блокировкой записи и чтения, так что данные надежно передаются между моделями.

3. ***Timed functional model*** – по-прежнему прямая передача данных, задержки добавлены к процессам и отражают особенности функционирования модели. Применяют на ранних стадиях компромиссного анализа hardware-software.

4. Transaction-level model

Связи между модулями смоделированы с использованием функций вызовов. В такой модели коммуникации обычно моделируются путем, который точен в терминах функционирования и часто в терминах времени, но коммуникации не моделируются структурно точно. Например, в TLM для СнК платформы мы можем моделировать различные типы транзакций, которые поддерживаются на шине чипа (короткие транзакции чтение/запись), но мы не можем моделировать реальные проводники шины или пины, которые соединяют модуль с шиной.

Когда используют термин ***platform transaction-level model***, показывают, что модель использует TLM стиль с целью моделировать инфраструктурные связи в платформе СнК и что модули вне такой конструкции структурно соответствуют блокам вне целевой реализации. Этот метод используют для уточнения эффектов в модели, и он является более качественным, точным и эффективным путем моделирования взаимосвязи HW и SW на самом раннем этапе проектирования.

5. Behavioral hardware model – это модель, имеющая контактную и функциональную точность на границах, тактовая точность на границах не учитывается. Эта модель может использоваться как входная для средств поведенческого синтеза.

6. ***Pin-accurate, cycle-accurate hardware model*** – это модель обеспечивает контактную и тактовую точность на границах в добавление к функциональной точности. Для модели не требуется внутренняя структура, которая влияет на целевую реализацию.

7. ***Register - transfer level model*** имеет на своих границах контактную и тактовую точность. В добавление внутренняя структура RTL модели точно отражает регистры и комбинационную логику целевой реализации.

4.4. Краткая история создания и развития SystemC

SystemC является результатом эволюции многих концепций в исследовательских и коммерческих сообществах EDA (Electronic Design Automation). Многие исследовательские группы и EDA Компании внесли свой вклад в этот язык.

SystemC начиналась как очень ограниченный циклический симулятор и «слегка отличный» от языка RTL. Язык эволюционировал и развивался до истинного язык проектирования системы, который включает как программные, так и аппаратные концепции. Хотя SystemC специально не поддерживает аналоговое оборудование или механические компоненты, нет никаких причин, почему эти аспекты системы не могут быть смоделированы с помощью SystemC-конструкций или методами co-simulation.

Некоторые из организаций, которые внесли значительный вклад в изучение языка разработки очень рано поняли, что любой новый язык дизайна должен быть открытыми для сообщества и не быть проприетарными (частными или собственными).

В результате, Open SystemC Initiative (OSCI) была создана в 1999 году. OSCI была сформирована для чтобы:

Развивать и стандартизировать язык;

Облегчить общение между пользователями и поставщиками языка;

Улучшить адаптацию;

Обеспечение механизмы для разработки с программ открытым исходным кодом и их поддержание.

SystemC состоит из языковых и потенциально-методологических библиотек. Создана библиотека верификации SystemC. Авторы рассматривают SystemC Verification (SCV) как наиболее значимую из этих библиотек. Эта библиотека добавляет поддержку современного языка проверки высокого уровня. Такие концепции, как ограниченная рандомизация, самоанализ и запись транзакций. Первый выпуск библиотеки SCV произошел в декабре 2003 после более чем года бета-тестирования.

Несмотря на то, что цифровые схемы являются специальным типом аналоговых схем и методов для анализа и понимания аналоговых и смешанных сигналов (AMS) до настоящего времени использовался слишком сложный и отнимающий много времени аппарат для проектировщика аналоговой системы. Расширение SystemC-AMS для C ++ - важнейшая отправная точка в решении этото вопроса и основывается на существующих версиях SystemC. Для удовлетворения особых требований аналоговых систем и цифрового оборудования с программным обеспечением требуется взаимодействие с их физической аналоговой средой. Например, когда цифровое аппаратное / программное обеспечение взаимодействует с радиочастотными системами, датчиками и исполнительными механизмами и силовой электроники, анализ должен не только решать проблемы, характерные для чисто аналоговых и чисто цифровых систем, а также моделировать их взаимодействие в режиме реального времени. Этим специальным требованиям отвечает SystemC-AMS.

Для изучения книг по SystemC , что читатель владеет практическими знаниями C ++ и минимален знание аппаратного обеспечения. Для навыков

C ++ не требуется, чтобы, читатель является «мастером». Надо, чтобы вы хорошо знаете синтаксис, объектно-ориентированные функции и методы использования C ++.

Как правило, уровень знаний C ++ универсален для современных или недавних выпускников с дипломом об информатике или инженерным специальностям четырехлетнего университета. Краткие сведения о языке C ++ и объектно_ориентированном программировании приведены в этой книге.

Чтобы полностью понять примеры, читателю потребуется минимальное понимание цифровой электроники.

4.5. Методология проектирования SystemC

Чтобы понять методологию проектирования SystemC и ее преимущества, важно сначала понять методологию, отличную от SystemC.

В методологии non-SystemC (рис. 1.6) разработчики системы должны написать исполняемые спецификации в C или C ++, а затем проверить и отладить этот проект. Обнаружив, что эта конструкция удовлетворяет всем техническим требованиям, она передается в группы разработки RTL. Затем группа проекта RTL перезаписывает проект в RTL, чтобы синтезировать его для вентилей. В такой методологии функциональное RTL-описание иногда зависит от выполняемых спецификаций и, следовательно, становится склонным к ошибке. Кроме того, возникает реальная проблема, если на уровне RTL обнаруживается, что что-то в концептуальной модели не может быть реализовано, так как нет общей среды проектирования между разработкой системы и ее реализацией.

В методологии SystemC разработчику системы нужно только написать модель SystemC. Разработчик может итеративно уточнять исполняемые спецификации вплоть до уровня передачи регистров, который все еще находится в SystemC до синтеза. Testbench можно использовать повторно, чтобы гарантировать, что итерационный процесс не имеет ошибок. Если во

время реализации RTL обнаружено, что что-то концептуально неправильно, гораздо проще его переписать.

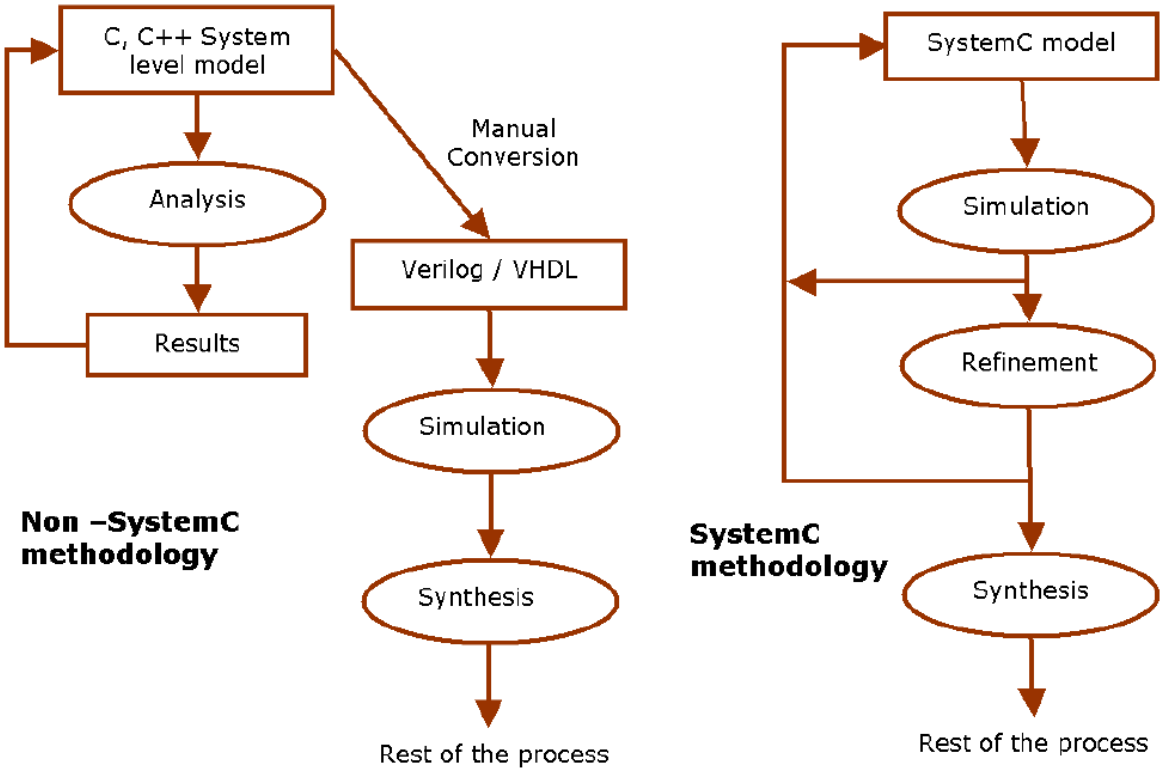


Рис. 4.6. Сравнение методологии проектирования

SystemC использует следующие типы моделей:

• Архитектурная модель системы;

• Модель производительности системы;

• Модель уровня транзакций (TLM);

• Функциональная модель;

• Модель передачи на уровне регистров (RTL).

Некоторые проекты SystemC начинаются как функциональные модели, в то время как большая часть кода SystemC делается на уровень TLM. Почти всегда TLM выступает в качестве исполняемой платформы, которая является достаточно точной для запуска программного обеспечения.

Главной причиной использования SystemC является значительное увеличение производительности симуляции на уровне TLM по сравнению с исполняемыми платформами, смоделированными на уровне RT с

использованием Verilog или VHDL. Модели SystemC TL достаточно быстры, чтобы служить платформой для разработки программного обеспечения, что позволяет раннюю разработку программного обеспечения и совместное моделирование аппаратного и программного обеспечения. Обе TL и функциональные модели достаточно быстры для архитектурного моделирования и анализа на системном уровне.

Типичные числа для модели System on Chip составляют от ~ 1K циклов в секунду до более высоких ~ 300-400K циклов в секунду. Этот диапазоне зависимосит от того, как процессор моделируется в системе. Если симулятор набора инструкций (ISS), то производительность обычно составляет от 1 до 10 тыс. циклов. Если процессор моделируется как прямое подключение к системной шине, то производительность переходит в диапазон 100K и выше.

Вторая причина использования SystemC - функциональная проверка. Одна и та же исполняемая платформа, которая используется для разработки программного обеспечения, часто используется и для проверки всей системы. Эта проверка происходит на раннем этапе проекта, и TLM становится прекрасной проверкой для всей системы.

Поскольку SystemC - это C ++, у него есть ряд неотъемлемых свойств, таких как классы, шаблоны и наследование, которое поддается проверке. Эти возможности дополняются SystemC Verification Library, что делает SystemC мощным языком проверки, а также языком моделирования.

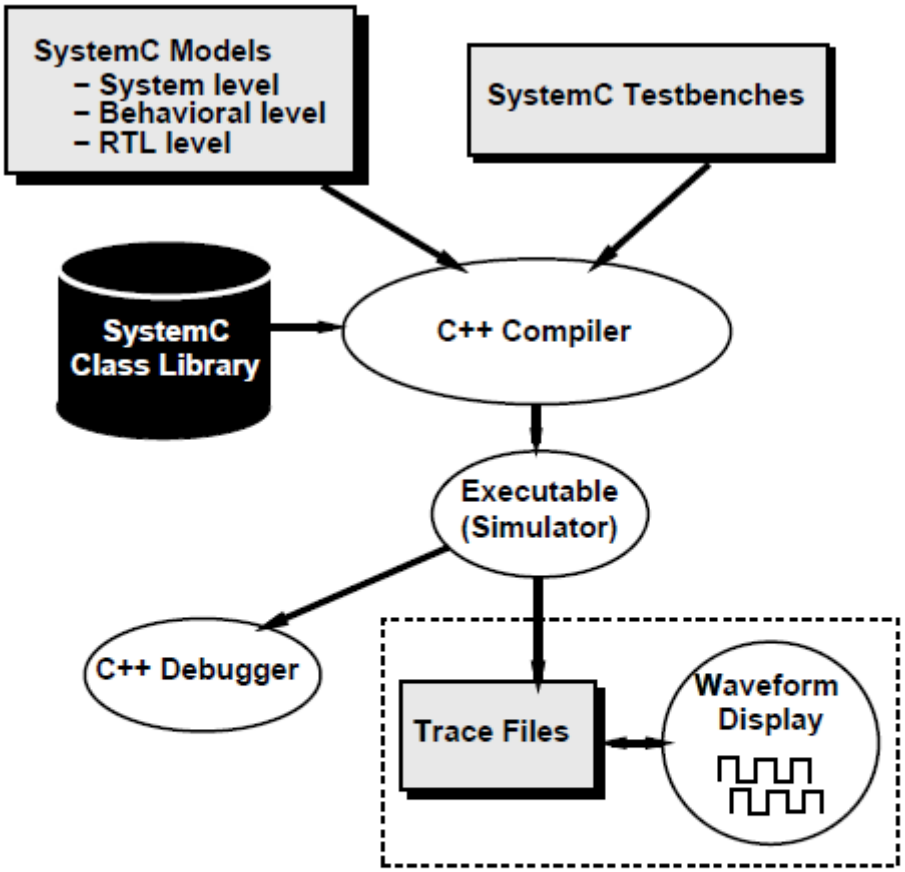


Figure 1: Simulation methodology. SystemC models and test benches are processed by a standard C++ compiler. The generated executable serves as the simulator, and can also be debugged using a standard C++ debugger. History of selected signals can be dumped into a trace file for waveform display. (The figure is adapted from an illustration in [5])

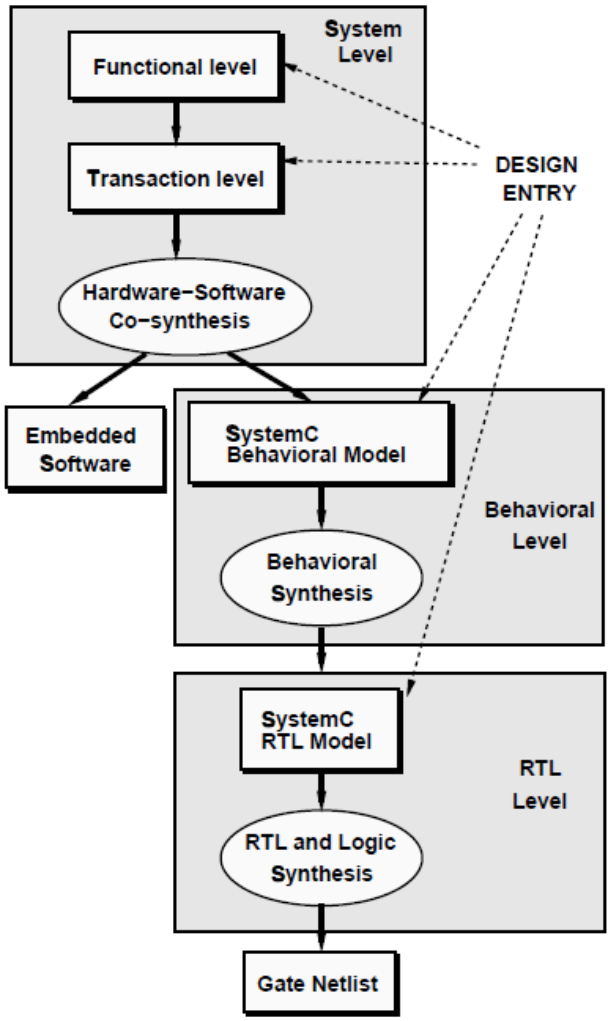


Figure 2: Synthesis/Implementation flow. Design entry could be at the System, Behavioral, or RTL levels. The same test bench could be used to validate models at different levels of abstraction

4.6. Ключевые характеристики SystemC:

• Параллельность – синхронные и асинхронные процессы;

• Понятие времени - несколько тактовых генераторов с произвольным фазовым соотношением;

• Типы данных - битовые векторы, целые числа с произвольной точностью;

• Типы данных фиксированной точки произвольной точности;

• Связь - сигналы, каналы;

• Расширенные протоколы связи;

• Реактивность - просмотр событий;

• Поддержка отладки - вывод осциллограмм;

• Поддержка моделирования;

• Поддержка множества уровней абстракции и итеративной обработки;

• Поддержка создания функциональной модели.

4.7. Потоки SystemC и их использование

На рис. 4.7. показан типичный системный поток SystemC. TLM часто служит как исполняемая платформа для программного обеспечения и как ценная контрольную модель для проверки. Путь к вентилям от TLM либо прямой с использованием поведенческого синтеза, либо TLM служит для ручного перевода либо в SystemC на уровень RTL-модели, либо в HDL (Verilog или VHDL) RTL-модель с последующим синтезом.

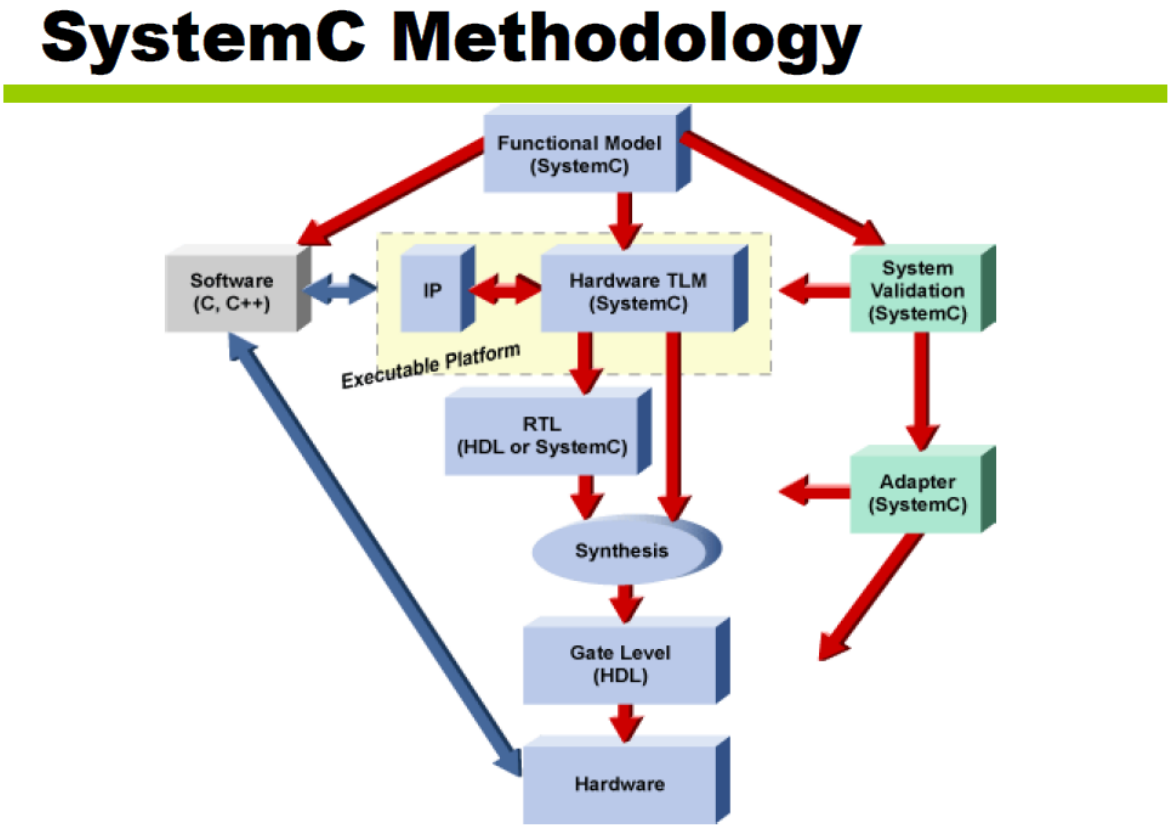


Рис. 4.7. Поток системного проектирования SystemC

Верификация выполняется на уровне TLM. Поскольку TLM переводится на уровни RTL или вентилей (gates), та же самая проверка может быть использована с помощью адаптеров (иногда называемых трансакторами).

Рис. 4.8 посвящен потоку аппаратного проектирования. Из модели системы или TLM можно выбрать путь, используя синтез поведения для вентилей или через синтез RTL. Испытательный стенд может быть уточнен с функционального уровня (UTF / TF), но это не требуется. С помощью использования адаптеров, тестирование с функционального стенда может быть выполнено на RTL и уровне вентильной модели. Модели RTL могут быть в SystemC, Verilog или VHDL. Модели уровня вентилей обычно записываются в Verilog или VHDL. Совместное моделирование требуется, если на стенде SystemC используется сканер Verilog или проектное представление VHDL.

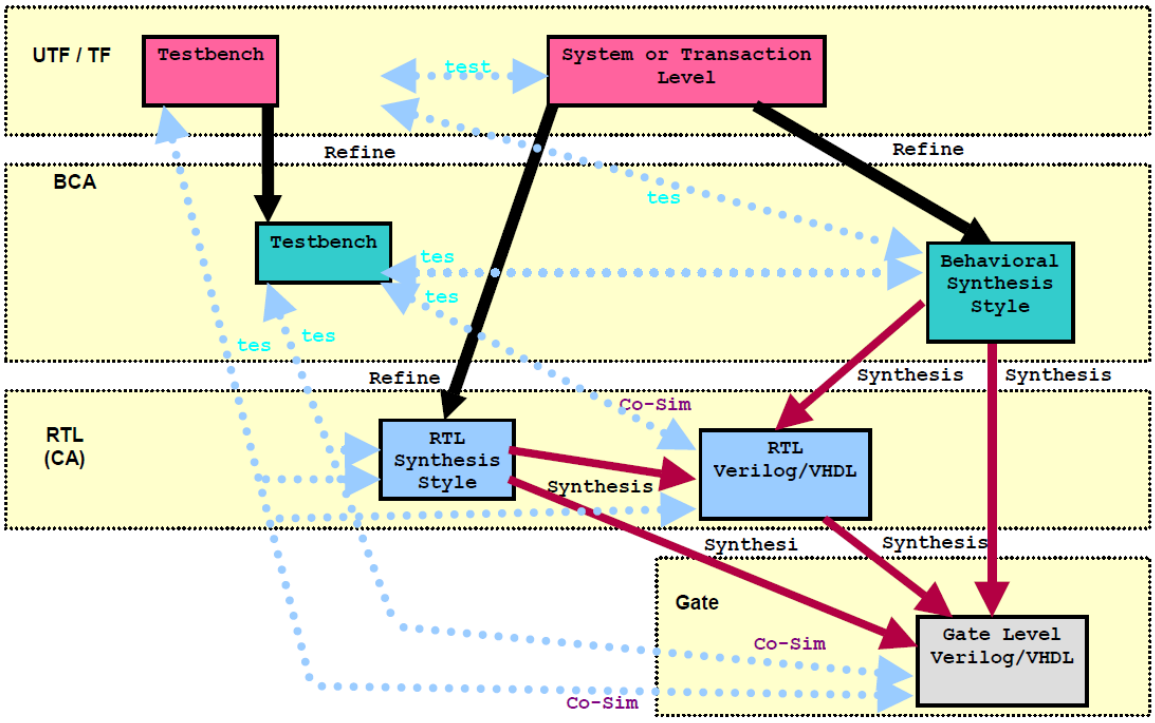


Figure 2 – SystemC Hardware Design Flow

4.8. Поток аппаратного проектирования SystemC

4.8. Модели вычислений

Наиболее известные модели вычислений в SystemC включают:

Статический многоскоростной поток – включает три фазы: чтение входных данных, выполнение вычисление внутри процесса, вывод всех данных, количество всех данных является известным и неизменным;

Динамический многоскоростной поток;

Сетевой процесс Кана, эффективная модель вычислений для создания алгоритмических моделей в применении к сишгальным процессорам, в которых процессы выполняются конкурентно, и каналы FIFO имеют определенную длину.

Дискретные события, которые используют:

RTL (register transfer level) моделирование HW-связано с цифровой аппаратурой, синхронизированной во времени;

Сетевое моделирование (стохастические или ожидающие модели);

TLM (transaction-level model) - платформа моделирования SoC , основанная на трансакиях, мы моделируем коммуникации между модулями, используя функции вызова, которые предсьавляют транзакции, типично поддерживаемые целевой платформой. Каналы sc_signal ипользуются вместо данных, эти сигналы обменяются между различными процессами, путем чтения и записи общих переменных данных.

TLM проекты более краткие и быстрые, чем соответствующие RTL проекты.

Перечислим основные уровни абстракции и модели использования SystemC.

Сюда входят:

– функциональное моделирование системных алгоритмов;

– моделирование системных архитектур на уровне транзакций;

– моделирование на уровне RTL и привязка SystemC к маршрутам реализации;

– недавние добавления к SystemC на тему верификации: библиотека SCV верификации SystemC.

SystemC является языком, идеально подходящим для моделирования встроенных систем, и именно тем языком, с помощью которого можно не только описывать сами модели, но и разрабатывать тестовое окружение для этих моделей и использовать его для тестирования реальных плат.

Глава 5. Основы языка SystemC-2.3.1

5.1. SystemC – надстройка к языку C++

SystemC обращается к моделированию программного и аппаратного обеспечения, используя C ++. Диаграмма (рис. 5.1) иллюстрирует основные компоненты SystemC, которые базируются на стандартном языке C++. Поскольку C ++ уже решает большинство задач программного обеспечения, не удивительно, что SystemC фокусируется прежде всего на проблемах, связанных как с программным обеспечением, так и с аппаратной реализацией. Основной областью применения SystemC является разработка электронных систем, но SystemC применяется к неэлектронным системам.

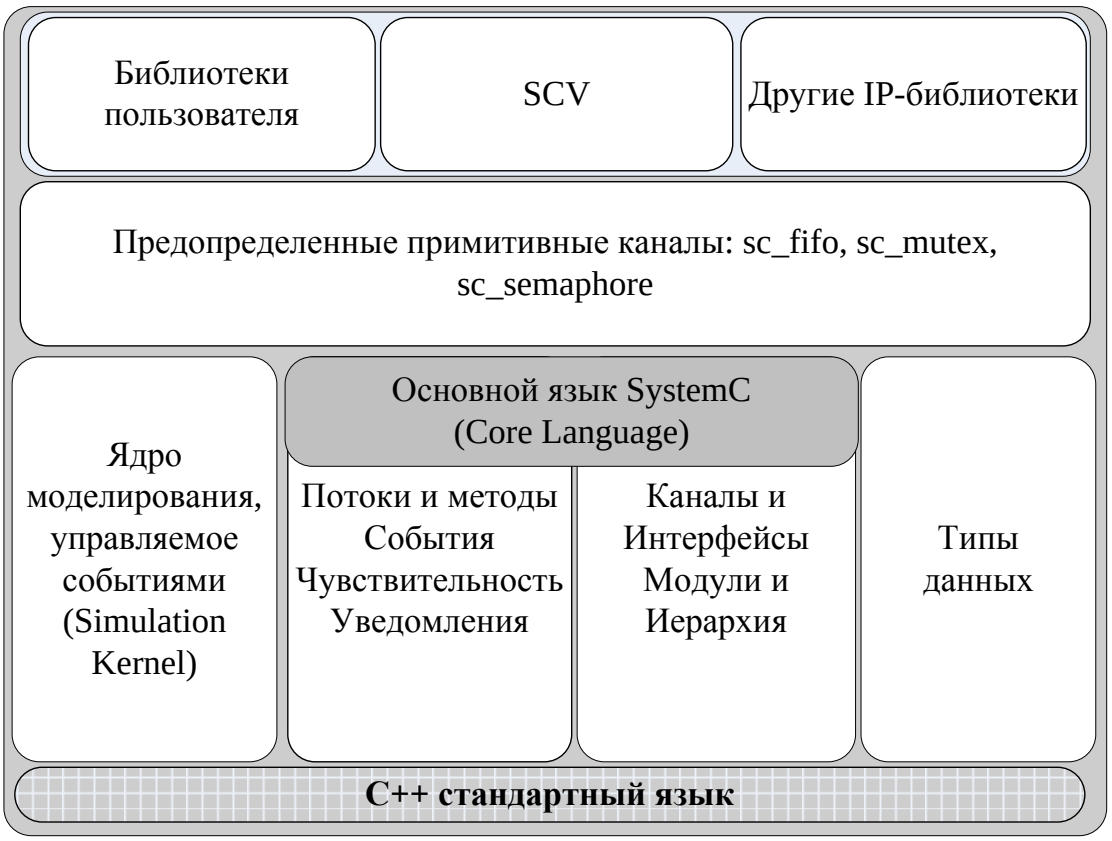


Рис.5.1. Архитектура и главные компоненты языка SystemC

Проект на языке SystemC состоит из набора файлов .cpp и .h. Для того чтобы полноценно работать, не требуется какая-либо среда разработки, поддерживающая данный язык. Достаточным условием является наличие

компилятора C++, библиотеки SystemC и среды разработки для языка C++. Мы будем работать в среде Eclipse CDT и Visual Studio. В качестве компилятора g++, как правило, будет использован Cygwin.

В заключительных главах книги описан порядок установки и настройки сред проектирования, компиляции библиотек SystemC, использования дополнительных программ для отображения результатов моделирования.

5.2. Ядро моделирования (Kernel)

Надстройка к языку C++ (рис. 5.1) включает в себя основные компоненты языка, с которыми мы будем постепенно знакомиться.

Начнем с ядра моделирования Simulation Kernel.

Имитатор SystemC имеет 3 главные фазы работы: разработка, выполнение и постобработка. Выполнение всех операторов до конструкции sc_start() есть фаза разработки. На этом этапе происходит инициализация структур данных и подготовка к следующей фазе выполнения. Фаза выполнения передаёт управление ядру (Kernel) моделирования SystemC, которое управляет работой всех процессов и создаёт иллюзию параллельности их выполнения. Постобработка связана с удалением всех созданных структур данных, освобождением памяти и завершением этапа моделирования.

Работа ядра моделирования и главная программа

Принцип работы ядра моделирования SystemC схож с языками VHDL и Verilog. Если обратиться к Verilog и VHDL, проходит некоторое время между инициализацией кода и началом моделирования. В SystemC, также как и в C/C++, есть строго определённая точка входа в программу. В случае SystemC – sc_main().

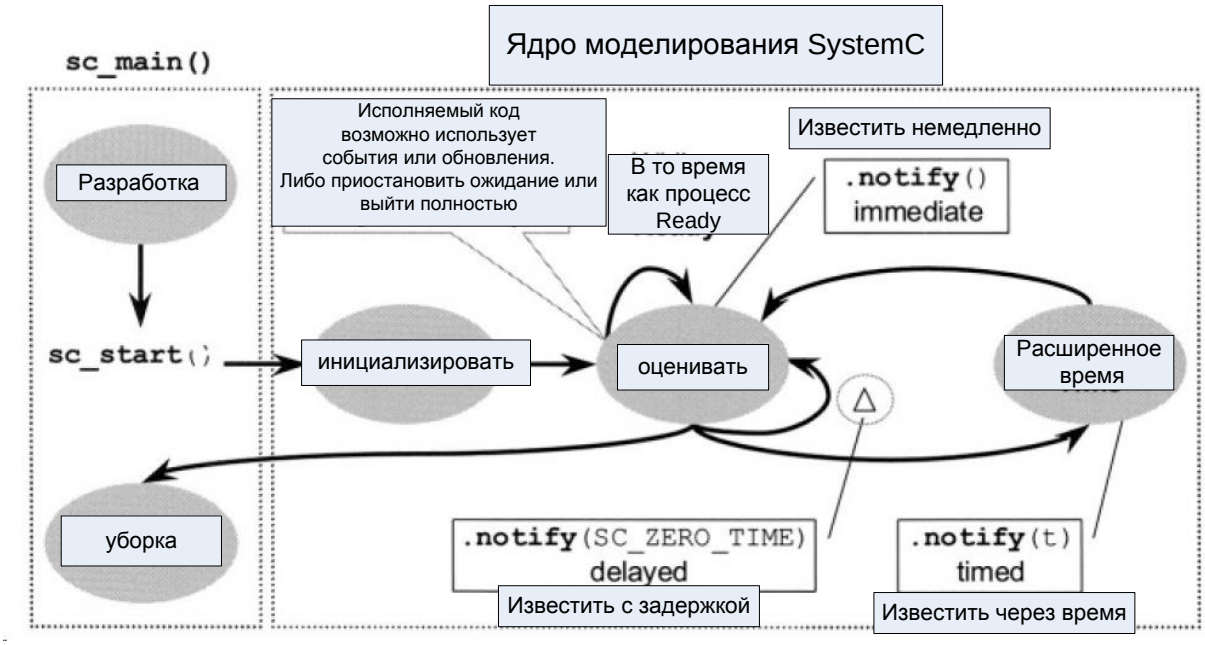


Рис. 5.2. Работа ядра моделирования SystemC

Исполнение команд перед вызовом функции в sc_start () известно как этап разработки. Эта фаза характеризуется инициализации структур данных, установлением соединений, а также подготовкой ко второму этап - выполнению. Контроль фазы выполнения передан ядру моделирования SystemC, которое дирижирует выполнением процессов, чтобы создать иллюзию параллелизм.

Иллюстрации на рис. 5.2 должна выглядеть очень хорошо знакомой тем, кто изучали ядра моделирования Verilog и VHDL. После команды sc_start () все процессы моделирования (за минусом несколько исключений) вызывается случайным образом во время инициализации. После инициализации запускается процесс моделирования, когда происходит событие, к которому процесс чувствителен. Несколько процессов моделирования могут начаться в один и тот же момент времени в симуляторе. Из-за этого случая, все процессы моделирования оцениваются и затем их выходы обновляются. Оценки с последующим обновлением называются *дельта-цикл*. Если нет никаких дополнительных процессов моделирования, которые требуется оценить в настоящее время (в результате обновления), то время моделирования продвигается вперед. При этом, если

не требуется запускать дополнительные процессы моделирования, моделирование заканчивается.

Этот краткий обзор работы ядра моделирования предназначен для того, чтобы дать понимание остальной части книги. Позже эта схема будет использоваться снова, чтобы объяснить важные тонкости. Очень важно понять как функционирует ядро, чтобы полностью понять язык SystemC. SystemC LRM (Справочное руководство) определяет поведение ядра SystemC моделирования, и является окончательным источником.

5.3. Состав ядра языка SystemC (Core Language)

Ядро языка SystemC (Core Language), как показано на рис. 5.1, кроме ядра моделирования включает в себя следующие компоненты: модули, порты, процессы, события, интерфейсы, каналы. Эти компоненты оперируют с различными типами данных, как правило, аналогичными языку C++.

Основные понятия ядра языка SystemC классифицируются по следующей терминологии:

Основные термины

Method	Метод C ++, т.е. функция-член класса.
Module	Конструктивный субъект, который может содержать процессы, порты, каналы, и другие модули. Модули позволяют выразить структурную иерархию.
Interface	Интерфейс предоставляет набор объявлений методов, но не дает никаких реализации метода и нет полей данных.
Channel	Канал реализует один или несколько интерфейсов, а также служит в качестве контейнера для функциональных возможностей связи.
Port	Порт представляет собой объект, через который

	модуль может получить доступ к его каналу интерфейс. Но модули могут также получить доступ к интерфейсу канала напрямую.
Primitive Channel	Примитивный канал является атомарным, то есть, он не содержит процессы или модули, и он не может непосредственно получить доступ к другим каналам.
Hierarchical Channel	Иерархический канал представляет собой модуль, то есть, он может содержать процессы и другие модули, и он может непосредственно получить доступ к другим каналам.
Event	Событие. Процесс может приостановливаться событием или быть чувствительным к одному или нескольким событиям. События позволяют возобновить и активизировать процессы.
Sensitivity	Чувствительность процесса определяет, когда этот процесс будет возобновлен или активирован. Процесс может быть чувствительным к набору событий. Всякий раз, когда одно из соответствующих событий инициируется, процесс возобновляется или активируется.
Static Sensitivity	Чувствительность процесса объявляется статически, т.е. заявляется во время разработки и не может быть изменена после начала моделирования. Так называемый список чувствительности используется для определения статического набора событий.
Dynamic Sensitivity	Чувствительность процесса может быть изменен во время моделирования.
IMC	Метод вызова интерфейса
RPC	Удаленный вызов процедур

Терминология процессов

Процессы играют центральную роль в SystemC. Они описывают функциональные возможности системы и позволяют выразить параллелизм в системе. Процессы содержатся в модулях и они имеют доступ к интерфейсам внешнего канала через порты модуля. Существуют различные типы процессов и различные способы активации процессов. Перед погружением в детали, объясним термины, связанные с процессами.

Thread	Поток. SystemC имеет свой собственный поток исполнения, но он не является упреждающим.
Automatically activated	Некоторые методы модуля (процессы) активируются автоматически, когда происходят события, к которым процессы чувствительны.
Explicitly activated	Некоторые методы модуля, которые должны быть вызваны явно другим кодом, чтобы активироваться.
wait()	Метод, который приостанавливает выполнение потока. Аргументы, передаваемые wait() определяют, когда выполнение потока возобновляется.
Ok to call wait()	SC_METHODs и код, который они называют не может вызвать wait(), так как они не имеют свой собственный поток исполнения. SC_THREADs и код, который они называют можно назвать ожидание ().
SC_THREAD	Модуль-способ, который имеет свой собственный поток исполнения, и который может вызвать код, который вызывает ожидание (). SC_THREADs автоматически активируются. Также известный как процесс потока.

SC_METHOD	Модуль - метод, который не имеет свой собственный поток исполнения, и который не может вызвать код, который вызывает wait(). SC_METHODs автоматически активируется. Также известный как процесс метода.
SC_CTHREAD	Модуль - метод, который имеет свой собственный поток исполнения, и который в его списке чувствительности имеет только положительное или отрицательное событие по фронту тактового импульса. Он может вызвать код wait() с ограниченным списком аргументов. SC_CTHREADs активируется автоматически. Также известный как процесс с тактовой частотой.

Следующей более высокой надстройкой являются понятия элементарных каналов.

На вершине архитектуры языка SystemC добавлены более специфические модели вычислений, библиотеки проектирования, руководства по моделированию, методология проектирования, которые полезны при проектировании.

Нижний слой подчеркивает то, что SystemC создан полностью на языке C++ . Это означает, что любая программа, написанная на SystemC, может быть откомпилирована компилятором C++, чтобы получить исполняемую программу.

5.4. Инициализация процесса

В SystemC планировщик будет выполнять все потоковые процессы и все процессы методов во время фазы инициализации моделирования. Чтобы предотвратить действия планировщика от выполнения процесса потока или процесса метода во время инициализация фазы моделирования, вы можете

использовать функцию dont_initialize (). Функция применяется в конце объявления процесса. Например,

```
SC_MODULE( my_module )
{
// port(s)
sc_in_clk clk;
// process(es)
void proc_a();
void proc_b();

// constructor
SC_CTOR( my_module )
{
SC_THREAD( proc_a );
sensitive << clk.pos();
dont_initialize(); // don’t initialize proc_a
SC_METHOD( proc_b );
sensitive << clk.neg();
dont_initialize(); // don’t initialize proc_b
}
};
```

5.5. Модель времени в SystemC

Так как время симуляции дискретно, существует единица разрешения по времени (time resolution), минимальный квант времени, который должен быть задан. По умолчанию время разрешения составляет 10^{-12} с (одну пикосекунду) и пользователь имеет опцию установки времени разрешения sc_set_time_resolution().

В библиотеке SystemC существует тип данных sc_time, для того чтобы измерять время в процессе моделирования. У времени есть две составляющие: числовое значение и размерность. Языком поддерживается измерение времени в секундах, миллисекундах, микросекундах, наносекундах, пикосекундах, фемтосекундах.

Соответствующие спецификаторы времени следующие:

```
SC_SEC // секунды
SC_MS // миллисекунды
SC_US // микросекунды
SC_NS // наносекунды
SC_PS // пикосекунды
SC_FS // фемтосекунды
```

Объявление переменных времени выглядит следующим образом:

```
sc_time имя_переменной (числовой значение, спецификатор);
sc_time t_Period(10, SC_NS);
```

Это означает, что объект `t_Period` представляет 10 нс.

Временные объекты:

```
sc_set_time_resolution(10,SC_PS);
sc_time t2(3.1416, SC_NS);
```

означают, что будет установлено временное разрешение 10 пс, а время t2 будет установлено 3.1416 нс.

Над переменными типа sc_time SystemC позволяет производить операции сложения, вычитания, масштабирования и др. В библиотеке определена константа `SC_ZERO_TIME`, которая соответствует времени 0.

5.6. Модули SC_MODULE

Основной единицей проектирования является модуль SC_MODULE.

Сложные системы состоят из множества независимо функционирующих компонентов. Эти компоненты могут представлять собой аппаратные средства, программное обеспечение или любой объект.

Компоненты могут быть большими или маленькими. Компоненты часто содержат иерархии более мелких компонентов. Самые маленькие компоненты представляют поведение и состояние. В SystemC мы используем концепцию, известную как SC_MODULE для представления компонентов.

ОПРЕДЕЛЕНИЕ: Модуль – это конструктивный субъект, который может содержать процессы, порты, каналы, и другие модули. Модули позволяют выразить структурную иерархию.

Модуль SystemC является самым маленьким контейнером функциональности с состоянием, поведением и структурой для иерархического подключения.

Модуль SystemC соответствет определению класса C++. Как правило, макрос SC_MODULE используется для объявления класса:

```
#include <systemc.h>
SC_MODULE(Adder) {
//MODULE_BODY
//ports, process, internal data, etc.
SC_CTOR(Adder){
//bode of constructor:
//process declaration, sensitivities, etc.
}
};
```

SC_MODULE – это простой макрос C++ и на языке C++ его можно определить так:

```
#define SC_MODULE (module_name)
struct module_name: public sc_module
```

В рамках этого производного класса модуля, разнообразные элементы составляют MODULE BODY::

Порты;

Отдельные интерфейсов каналов;

Отдельные данные;

Отдельные модули или подмодули;

Конструктор;

Деструктор;

Процессы;

Вспомогательные функции.

Из них необходимым является только конструктор. Однако, чтобы иметь любое полезное решение, вы должны иметь либо процесс или sub-design. Сначала мы посмотрим на конструктор, а затем на простой процесс. Эта последовательность позволяет нам разрбраться с базовым примером минимального дизайна.

Модуль в SystemC – это базовый элемент, включающий в себя процессы (processes) и другие модули.

МОДУЛЬ является старшим в иерархии элементов SystemC.

Наличие МОДУЛЯ позволяет строить SystemC-модели в соответствии с имеющимися у разработчиков аппаратуры представлениями (документацией) об архитектуре и функционировании будущего изделия. Обычно каждый отдельно взятый МОДУЛЬ представляет в модели функционально законченный узел разрабатываемого изделия.

МОДУЛИ объявляются с ключевым словом SC_MODULE.

На рис. 5.3 изображён модуль, который включает в себя несколько *процессов*.

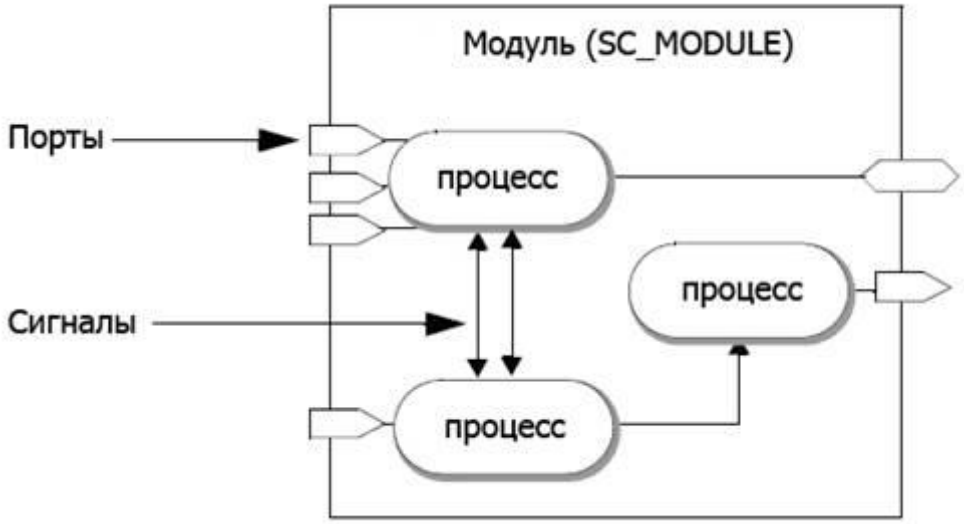


Рис. 5.3. Модуль, содержащий несколько процессов

```
Структура модуля
// Header file
SC_MODULE(module_name) {
Port declarations
Local channel declarations
Variable declarations
Process declarations
Other method declarations
Module instantiations
SC_CTOR(module_name){
Process registration
Static sensitivity list
Module variable initialization
Module instance / channel binding
}
} ;
```

5.6.1. Порты модулей

Порты модуля передают данные к процессам модуля и из него во внешний мир, как в Verilog и VHDL. Вы объявляете направление порта как внутри, так и снаружи. Вы также объявляете тип данных порта как любой тип данных C ++, тип данных SystemC или пользовательский тип. Типы портов:

In: входные порты

Out: выходные порты

Inout: двунаправленные порты

Режимы порта `sc_in, sc_out и sc_inout` предопределены библиотекой классов SystemC.

Синтаксис:

Переменная типа sc_direction;

Далее:

`Port_direction`: один из `sc_in, sc_out, sc_inout`

`Type`: тип данных;

`Variable`: Имя действительной переменной

```
#include "systemc.h"

SC_MODULE (first_counter) {
  sc_in_clk     clock ;       /* Clock input of the
design*/
  sc_in<bool>   reset ;       /* active high, synchronous
Reset input*/
  sc_in<bool>   enable;       /* Active high enable signal
for counter*/
  sc_out<sc_uint<4> > counter_out; /* 4 bit vector output
of the counter*/
```

```
  // Rest of body
}
```

5.6.2. Сигналы модуля

Порты используются для связи вне модуля. Для связи внутри модуля SystemC мы используем сигналы. Сигналы подобны проводам в Verilog. Сигналы могут быть любых допустимых типов данных.

Сигнал также используется для соединения двух портов модулей в родительском модуле. Допустим, у нас есть два дочерних модуля A и B, эти два модуля используются в родительском модуле C, тогда провода используются для соединения портов модуля A и B друг с другом.

Синтаксис:

Переменная типа `sc_signal`;

Пример:

`sc_signal`: зарезервированное слово

`Type`: Тип данных

`Variable`: Имя действительной переменной

```
#include "systemc.h"

SC_MODULE (counter) {
  sc_signal <bool>         reset ;
  sc_signal <bool>         enable;
  sc_signal <sc_uint<4> > counter_out;

  // Rest of body
}
```

Типы данных и операторы SystemC подробно будут рассмотрены позже. Для справки смотрите раздел 3.18.

5.6.3. Создание экземпляров модулей

Для создания экземпляров модулей (Instanciating) в SystemC так же, как в Verilog, соблюдаются те же правила. Модуль SystemC, подобный Verilog, позволяет использовать два способа соединения портов.

По положению;

По имени.

По положению

Здесь порядок должен совпадать. Обычно не рекомендуется подключать порты по положению. Может вызвать проблемы при отладке (например: трудно найти порт, вызывающий ошибку компиляции), когда новый порт добавляется или удаляется. Ниже приведен пример подключения портов по позиции.

```
include "systemc.h"
#include "first_counter.cpp"

int sc_main (int argc, char* argv[]) {
  sc_signal<bool>   clock;
  sc_signal<bool>   reset;
  sc_signal<bool>   enable;
  sc_signal<sc_uint<4> > counter_out;
  int i = 0;
  // Connect the DUT (design under test)
  first_counter counter("COUNTER");
    // Here ports are connected by position
    counter(clock,reset,enable,counter_out);

  // Rest of the body of the testbench

  return 0;// Terminate simulation
 }
```

По имени:

Здесь имя должно совпадать с исходным модулем, порядок не имеет значения. Ниже приведен пример соединения портов по имени.

```
#include "systemc.h"
#include "first_counter.cpp"

int sc_main (int argc, char* argv[]) {
  sc_signal<bool>   clock;
  sc_signal<bool>   reset;
  sc_signal<bool>   enable;
  sc_signal<sc_uint<4> > counter_out;
  int i = 0;
  // Connect the DUT
  first_counter counter("COUNTER");
    // Here ports are connected by name
    counter.enable(enable);
    counter.reset(reset);
    counter.clock(clock);
    counter.counter_out(counter_out);

  // Rest of the body

  return 0;// Terminate simulation
 }
```

5.6.4. Внутренние переменные

SystemC позволяет использовать локальную переменную, так же Verilog или любой другой язык программирования. Локальные переменные

могут быть любого допустимого C ++ или SystemC типа или пользовательских типов.

```
#include "systemc.h"

SC_MODULE (local_variable) {
  sc_in_clk     clock ;      // Clock input of the design
  sc_in<bool>   reset ;      // active high, synchronous
Reset input
  sc_in<bool>   enable;      // Active high enable signal
for counter
  sc_out<sc_uint<4> > counter_out; // 4 bit vector output
of the counter

  //------------Local Variables Here--------------
  sc_uint<4> count;

  //------------Code Starts Here----------------
  // Below function implements actual counter logic
  void incr_count () {
    // Body of the function
  } // End of function incr_count

  // Constructor for the counter
  SC_CTOR(local_variable) {
    cout<<"Executing new"<<endl;
    SC_METHOD(incr_count);
    sensitive << reset;
    sensitive << clock.pos();
  } // End of Constructor
```

```
}; // End of Module counter
```

Фактическая функциональность модуля SystemC реализована в процессах. Процесс может быть либо сделан чувствительным к уровню модельной логики, либо может быть чувствительным к фронту последовательной логики модуля.

Процесс можно заставить работать вечно. Такие процессы называют потоками в SystemC. Или можно запускать процесс каким-либо событием Например, положительным фронтом тактового сигнала (posedge of clock).

Примечание. Сигналы чувствительного списка, которые запускают процесс / поток, должны быть портами и не могут быть сигналом или локальной переменной.

```
#include "systemc.h"

SC_MODULE(dff) {
  sc_in  <bool> din; // Data input of FF
  sc_in  <bool> clk; // Clock input of FF
  sc_out <bool> out; // Q output of FF

  void implement();  // Process that implements DFF

  SC_CTOR(dff) {
    SC_METHOD(implement);
    sensitive_pos << clk; /* Call implements() at every
pos edge of clk*/
  }
};
```

5.7. Конструктор SC_CTOR

КОНСТРУКТОР МОДУЛЯ SC_CTOR создает и инициализирует экземпляр МОДУЛЯ в симуляционном ядре SystemC. КОНСТРУКТОР создает внутренние структуры данных в ядре, которые используются МОДУЛЕМ и инициализирует их.

Синтаксис:

```
SC_CTOR(module_name)
{
тип ПРОЦЕССА(method_name)
}
```

Здесь:

SC_CTOR – Макрос;

module_name: Имя МОДУЛЯ, указанное в SC_MODULE («module_name»);

тип ПРОЦЕССА: Один из типов ПРОЦЕССОВ: SC_THREAD, SC_METOD, SC_CTHREAD;

method_name: Имя метода, реализованного в МОДУЛЕ.

Конструктор модуля создает и инициализирует экземпляр модуля. Конструктор создает внутренние структуры данных, которые используются для модуля, и инициализирует эти структуры данных до известных значений. Конструкторы модулей в SystemC реализованы таким образом, что имя экземпляра модулей и Hierarchy передается конструктору во время создания (создания). Это помогает идентифицировать модуль при возникновении ошибок или при сообщении информации из модуля. Это только разница между конструктором C ++ и SystemC.

```
#include "systemc.h"

SC_MODULE(dff) {
  sc_in  <bool> din; // Data input of FF
```

```
  sc_in  <bool> clk; // Clock input of FF
  sc_out <bool> dout;// Q output of FF

  void implement() {  // Process that implements DFF
    dout = din;
  }

  SC_CTOR(dff) {
    SC_METHOD(implement); // Contains a process named
implement
    sensitive_pos << clk; // Execute implements() at
every posedge of clk
  }
};
```

5.8. Альтернативные конструкторы: SC_HAS_PROCESS

SystemC имеет и другой подход к созданию конструкторов. Альтернативный подход использует макрос cpp SC_HAS_PROCESS. Вы можете использовать этот макрос в двух ситуациях. Во-первых, используйте SC_HAS_PROCESS, когда вам требуются конструкторы с аргументами, выходящими за пределы имени экземпляра строки, которая передается в SC_CTOR (например, для предоставления конфигурируемых модулей). Во-вторых, используйте SC_HAS_PROCESS, если вы хотите поместить конструктор в выполняемый файл (файл .cpp). Аргументы конструктора можно использовать, чтобы указать размеры подключенной памяти, диапазоны адресов для декодеров, глубину FIFO, делители тактовых импульсов, FFT глубину и другую конфигурационную информацию. Например, дизайн памяти может позволить выбор различных размеров памяти с аргументом:

```
My_memory instance("instance", 1024);
```

Чтобы использовать этот альтернативный подход, вызовите SC_HAS_PROCESS, а затем создайте обычные конструкторы. Применяется одна оговорка. Вы должны построить или ициализировать базовый класс модуля sc_module с помощью строки имени строки. Это требование является причиной того, что SC_CTOR нуждается в аргументе. Синтаксис этого стиля при использовании в заголовочном файле следующий:

```
//FILE: module_name.h
SC_MODULE(module_name) {
SC_HAS_PROCESS(module_name);
module_name(sc_module_name instname[, other_args…])
: sc_module(instname)
[, other_initializers]
{
CONSTRUCTOR_BODY
}
};
```

Синтаксис для использования SC_HAS_PROCESS в отдельной реализации (то есть при отдельной ситуации компиляции) аналогична.

```
//FILE: module_name.h
SC_MODULE(module_name) {
SC_HAS_PROCESS(module_name);
module_name (sc_module_name instname[,other_args…]);
};
//FILE: module_name.cpp
module_name::module_name(
sc_module_name instname[, other_args…])
: sc_module(instname)
[, other_initializers]
{
```

```
CONSTRUCTOR_BODY
}
```

В предыдущих примерах, другие аргументы являются необязательными.

5.9. Процессы

ПРОЦЕСС - это встроенная функция (метод) класса SC_MODULE, которая запускается программой-планировщиком в ядре SystemC.

Синтаксис:

```
Process declaration:
void process_name();

Process registration:
SC_METHOD(process_name);
SC_THREAD(process_name);
SC_CTHREAD(process_name, clock_edge_reference);
SC_SLAVE(process_name, slave_port);
```

ПРОЦЕССЫ необходимы также для моделирования параллельного функционирования физических узлов изделия.

В ПРОЦЕССАХ декларируются методы `.write` (для записи данных в ПОРТЫ) и `.read` (для чтения данных из ПОРТОВ), а также внутренние сигналы (Integral Signals) – для передачи данных от одного ПРОЦЕССА к другому внутри МОДУЛЯ.

Для ПРОЦЕССА должен быть задан список чувствительности – перечень портов МОДУЛЯ, изменения на которых влияют на процесс, и список внутренних СОБЫТИЙ, наступление которых могут изменять ход его исполнения. Списки определяются в КОНСТРУКТОРЕ.

Процесс является основной единицей выполнения в SystemC. Всё выполнение кода инициируется из одного или нескольких процессов. Процессы, по всей видимости выполняться параллельно.

ОПРЕДЕЛЕНИЕ: процесс SystemC является методом или функцией-членом класса SC_MODULE, который вызывается планировщиком ядра моделирования в SystemC. Прототип функции- члена для процесса SystemC является:

```
void PROCESS_NAME(void);
```

Процесс SystemC не принимает никаких аргументов и ничего не возвращает. Этот синтаксис делает его простым для вызова из ядра моделирования.

В SystemC предопределены три типа ПРОЦЕССОВ:

```
SC_THREAD; SC_METOD; SC_CTHREAD.
```

Процесс в SystemC объявляется в теле модуля и регистрируется как процесс внутри конструктора. Необходимо объявлять процесс как функцию void, не содержащую аргументов. При регистрации функции как SC_METHOD процесс, необходимо использовать конструкцию SC_METHOD, которая имеет один аргумент – имя процесса.

Пример описания процесса на SystemC:

```
SC_MODULE(my_module)
 {
sc_in<int> a;
sc_in<bool> b;
sc_out<int> x;
sc_out<int> y;
sc_signal<bool>c;
sc_signal<int> d;
void my_method_proc();
SC_CTOR(my_module)
{
 SC_METHOD(my_method_proc);
// Объявление списка чувствительных сигналов
```

```
}
};
```

Процессы реагируют на изменение сигналов, которые находятся в списке чувствительных входов. Возможно использование функций `sensitive()`, `sensitive_pos()` или `sensitive_neg()` или потоков `sensitive`, `sensitive_pos`, `sensitive_neg` при описании списка чувствительности (`sensitivity list`).

Для комбинаторной логики, спискок чувствительных входов приравнивает все входные порты (`input и inout ports`) и сигналы (`signals`) к входным сигналам процесса. Для реализации `level-sensitive` входов необходимо использовать метод `sensitive` так, как показано в примере:

```
SC_MODULE(my_module)
{ sc_in<int> a;
sc_in<bool> b;
sc_out<int> y;
sc_signal<bool>c;
void my_method_proc();
SC_CTOR(my_module)
{
SC_METHOD(my_method_proc);
// Объявление списка чувствительных сигналов
sensitive << a << c << d;
// Потоковое описание
 sensitive(b); //Функциональное описание
 sensitive(e); //Функциональное описание
 }
 };
```

Примечание: Во избежание риска возникновения ошибок на этапе симуляции проекта, включайте все входные порты в список чувствительных портов при реализации комбинаторной логики. В примере показана ситуация с неполным перечнем входов, отображенном в списке чувствительных портов.

```
void comb_proc ()
{
out_x = in_a & in_b & in_c;
}
SC_CTOR( comb_logic_complete )
{
SC_METHOD( comb_proc);
sensitive << in_a << in_b; // пропущен in_c
}
```

Пример описания заголовочного файла модели элемента 3И на SystemC:

```
#ifndef AND3_H
#define AND3_H
#include "systemc.h"

SC_MODULE(and3)
{
sc_in <bool> A, B, C;
sc_out <bool> F;
void do_and3()
{
F.write( A.read() && B.read() && C.read() );
}
SC_CTOR(and3)
{
```

```
SC_METHOD(do_and3);
sensitive << A << B << C;
}
};
#endif
```

Конструкция Edge-Sensitive используется для реализации последовательной логики, при моделировании триггеров. Для этого необходимо использовать такие потоки как sensitive_neg (срез), sensitive_pos (фронт). Входные порты должны иметь тип sc_in<bool>, ниже приведен пример использования конструкции edge-sensitive:

```
SC_MODULE(my_module)
{
 sc_in<int> a;
 sc_in<bool> b;
 sc_in<bool> clock;
 sc_out<int> y;
 sc_in<bool> reset;
 sc_signal<bool>c;
 void my_method_proc();
 SC_CTOR(my_module)
{
 SC_METHOD(my_method_proc);
 sensitive_pos (clock); // Функциональное описание
 sensitive_neg << b << reset; // Потоковое описание
}
};
```

Ограничения при использовании Sensitivity Lists:

- Нельзя совмещать конструкции `Level-Sensitive` и `Edge-Sensitive` в одном процессе;
- Нельзя применять тип `sc_logic` для реализации синхросигнала `(clock)` или других `Edge-Sensitive's` входов. Допустимым является только тип `sc_in<bool>`.

Листинг 5.1

Пример описания заголовочного файла JK-триггера на SystemC

```
#ifndef jk_H
#define jk_H

#include "systemc.h"
SC_MODULE(jk)
{
sc_in <bool> J, K, Clock, Reset;
 sc_out <bool> F, NF;
void do_jk()
{
if(!Clock.read())
{
if(J.read() == true && K.read() == true)
{ if(F.read())
{ F.write(false);
NF.write(true);
}
else
{
F.write(true);
NF.write(false);
}
```

```
}
if(J.read() == true && K.read() == false)
 {
F.write(true);
NF.write(false);
}
if(J.read() == false && K.read() == true)
 {
F.write(false); NF.write(true);
 }
 }
if(!Reset.read())
{
F.write(false);
NF.write(true);
}
}
SC_CTOR(jk)
{ SC_METHOD(do_jk);
sensitive << Clock.neg();
sensitive << Reset.neg();
}
};
#endif
```

Определения списка чувствительности процесса:

- чувствительный к оператору ();
- принимает один порт или сигнал в качестве аргумента;
- sensitive (sig1); sensitive (Sig2); sensitive (Sig3);

- чувствительный к обозначению потока;
- принимает произвольное количество аргументов;
- sensitive << sig1 << sig2 << sig3;
- sensitive_pos с помощью () или << operator
- определяет чувствительность к положительному фронту булевого сигнала или такта:
- sensitive_pos << clk;
- sensitive_neg с () или << оператором;
- определяет чувствительность к отрицательному фронту булевого сигнала или такта sensitive_neg << clk;

5.9.1. Процесс SC_THREAD

Самый простой тип процесса, чтобы понять это поток SystemC, названный SC_THREAD. Концептуально поток SystemC идентичен программному обеспечению потока. В простых C / C ++ программах есть только один поток, работающий для всей программа. Ядро SystemC позволяет выполнять много потоков параллельно (это называется параллелизм) Простой SC_THREAD начинает выполняться, когда планировщик называет его и заканчивается, когда происходит выход или возврат. SC_THREAD вызывается только один раз, так же, как простая программа C/C ++. SC_THREAD может также приостановить себя.

Регистрация простого процесса: SC_THREAD

После того, как вы определили тип процесса, необходимо определить и зарегистрировать его с ядром моделирования. Этот шаг позволяет планировщику симуляции ядра вызывать поток. Регистрация происходит в модуле конструктора класса SC_CTOR. Регистрация потока SystemC кодируется с помощью макроса C++ SC_THREAD внутри конструктора следующим образом:

```
SC_THREAD(process_name);//Must be INSIDE constructor
```

`Process_name` - это имя соответствующего метода члена класса.

С ++ позволяет конструктору появляются до или после провозглашения процесса метода. Ниже приведен полный пример определения SC_THREAD в модуле:

```
//FILE: simple_process_ex.h
SC_MODULE(simple_process_ex) {
SC_CTOR(simple_process_ex) {
SC_THREAD(my_thread_process);
}
void my_thread_process(void);
};
```

Member function – функция- член: функция, которая является элементом класса и которая оперирует с объектами этого класса, адресуясь через указатель this.

Изучать методы и процессы мы будем, используя в примерах программу простого счетчика first_counter, детально описанную на сайте компании ASIC. Схема счетчика показана на рис. 3.4. Управляя входными сигналами clock, reset, enable и условиями реакции счетчика на эти сигналы, можно проектировать различные варианты процессов и моделированием проверять правильность их работы. Управляющие сигналы создает программа Testbench, которая служит главной программой `main.cpp`.

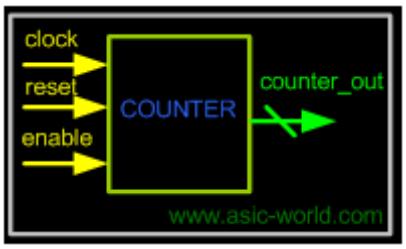


Рис. 3.4. Первый счетчик-first counter

Листинг 5.2

Программа sc_counter_threads.cpp

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

```
SC_MODULE (first_counter) {
  sc_in_clk     clock ;  // Clock input of the design
  sc_in<bool>   reset ;    /* active high, synchronous
Reset input*/
  sc_in<bool>   enable;      /* Active high enable signal
for counter*/
  sc_out<sc_uint<4> > counter_out; /* 4 bit vector output
of the counter*/

  //--------Local Variables Here---------------------
  sc_uint<4> count;

  //--------Code Starts Here-----------------------
  // Below function implements actual counter logic
  void incr_count () {
    // For threads, we need to have while true loop
    while (true) {
      // Wait for the event in sensitivity list to occure
      // In this example - positive edge of clock
      wait();
      if (reset.read() == 1) {
        count =  0;
        counter_out.write(count);
      /* If enable is active, then we increment the
counter*/
      } else if (enable.read() == 1) {
        count = count + 1;
        counter_out.write(count);
      }
```

```
    }
  } // End of function incr_count

  /* Below functions prints value of count when ever it
changes*/
  void print_count () {
    while (true) {
      wait();
      cout<<"@" << sc_time_stamp() <<
        " :: Counter Value "<<counter_out.read()<<endl;
    }
  }

  // Constructor for the counter
  /* Since this counter is a positive edge trigged one,*/
  /* We trigger the below block with respect to
positive*/
  // edge of the clock
  SC_CTOR(first_counter) {
    // Edge sensitive to clock
    SC_THREAD(incr_count);
    sensitive << clock.pos();
    // Level Sensitive to change in counter output
    SC_THREAD(print_count);
    sensitive << counter_out;
  } // End of Constructor

}; // End of Module counter

//Testbench for sc_counter_threads
```

```
int sc_main (int argc, char* argv[]) {
  sc_signal<bool>   clock;
  sc_signal<bool>   reset;
  sc_signal<bool>   enable;
  sc_signal<sc_uint<4> > counter_out;
  int i = 0;
  // Connect the DUT
  first_counter counter("COUNTER");
    counter.clock(clock);
    counter.reset(reset);
    counter.enable(enable);
    counter.counter_out(counter_out);
      sc_start(1, SC_MS);

  // Initialize all variables
  reset = 0;       // initial value of reset
  enable = 0;      // initial value of enable
  for (i=0;i<5;i++) {
    clock = 0;
    sc_start(1, SC_MS);
    clock = 1;
    sc_start(1, SC_MS);
  }
  reset = 1;    // Assert the reset
  cout << "@" << sc_time_stamp() <<" Asserting reset\n"
<< endl;
  for (i=0;i<10;i++) {
    clock = 0;
    sc_start(1, SC_MS);
    clock = 1;
```

```
    sc_start(1, SC_MS);
  }
  reset = 0;    // De-assert the reset
  cout << "@" << sc_time_stamp() <<" De-Asserting
reset\n" << endl;
  for (i=0;i<5;i++) {
    clock = 0;
    sc_start(1, SC_MS);
    clock = 1;
    sc_start(1, SC_MS);
  }
  cout << "@" << sc_time_stamp() <<" Asserting Enable\n"
<< endl;
  enable = 1;  // Assert enable
  for (i=0;i<20;i++) {
    clock = 0;
    sc_start(1, SC_MS);
    clock = 1;
    sc_start(1, SC_MS);
  }
  cout << "@" << sc_time_stamp() <<" De-Asserting
Enable\n" << endl;
  enable = 0; // De-assert enable
    cout << "@" << sc_time_stamp() <<" Terminating
simulation\n" << endl;
  return 0;// Terminate simulation

 }
```

В модели счетчик счётчик выполняет счет при отсутствии сигнала reset и наличии сигнала enable. Программа testbench создает входные сигналы,

выводит выходное значение со счетчика, регистрирует текущее время. Результат моделирования, полученные нами в среде Eclipse, показаны на рис. 5.5

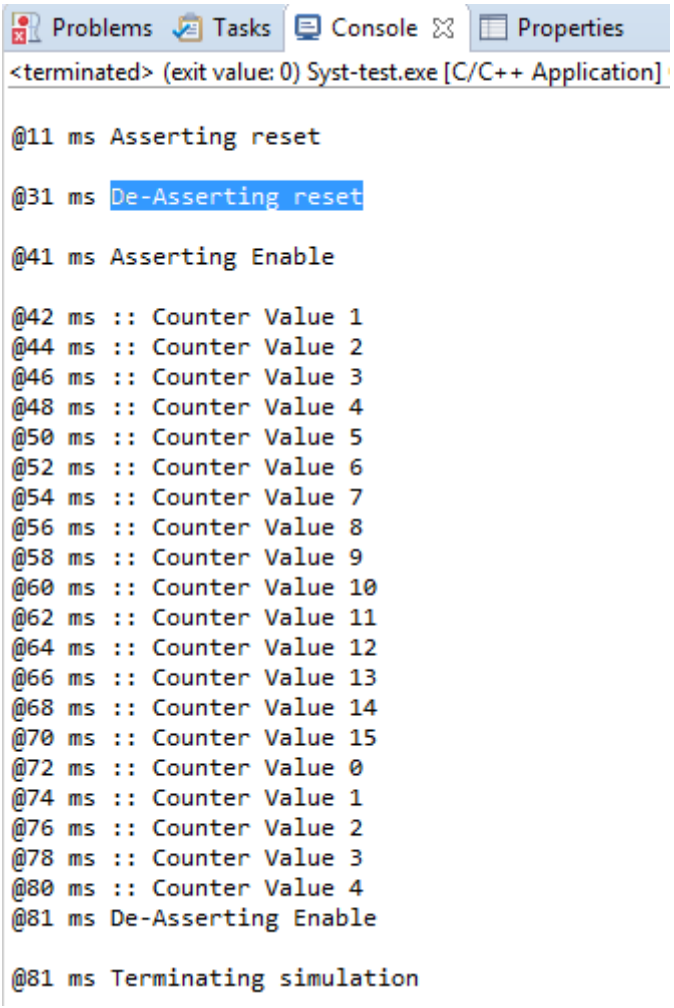


Рис. 5.5. Результаты моделирования в среде Eclipse

5.9.2. Процесс SC_METHOD

Процесс SC_METHOD в некотором смысле проще, чем SC_THREAD. Однако, эта простота делает его использование более трудным для некоторых стилей моделирования. По возможностям SC_METHOD является более эффективным, чем SC_THREAD. Одним из основных отличий является вызов. Процессы SC_METHOD никогда не приостанавливаются внутренне (то есть, они никогда не могут вызывать ожидание `wait()`). Вместо этого SC_METHOD процессы выполняются полностью с возвратом.

Симулятор вызывает их неоднократно на основе динамической или статическая чувствительность.

Поскольку процессам SC_METHOD запрещаются приостановления изнутри, они не могут вызвать метод ожидания. Попытка вызова ожидания либо непосредственно, либо из результатов SC_METHOD приводит к ошибкам во время выполнения. Они известны как блокирование методов. Методы чтения и записи типа `sc_fifo` данных, которые обсуждается далее, являются примерами способов блокировки. Таким образом, SC_METHOD процессы должны избегать использования вызовов методов блокирования. Синтаксис SC_METHOD процессов практически идентичен SC_THREAD за исключением ключевого слова SC_METHOD:

`SC_METHOD(`*process_name*`);//Located INSIDE constructor`

Методы ведут себя как функции. Когда функция вызывается, она запускается и выполняет и возвращает исполнение обратно в механизм вызова. Метод вызывается, когда изменяется какое-либо событие в списке чувствительности. Запуск события в чувствительном списке может быть либо чувствительным к краю (фронту), либо чувствительным к уровню.

Примечание: сигналы чувствительных списков, которые запускают процесс, могут быть сигналами, или локальной переменной, или портом.

Входные сигналы, вызывающие повторную активацию процесса, задаются списком чувствительности. Список чувствительности указан в конструкторе модуля

Листинг 5.3

Программа sc_counter_method.cpp

```
#include "systemc.h"

int sc_main (int argc, char* argv[]) {
SC_MODULE (first_counter) {
```

```
  sc_in_clk     clock ;       /* Clock input of the
design*/
  sc_in<bool>   reset ;       /* active high, synchronous
Reset input*/
  sc_in<bool>   enable;       /* Active high enable signal
for counter*/
  sc_out<sc_uint<4> > counter_out; /* 4 bit vector output
of the counter*/

  //------------Local Variables Here--------------
  sc_uint<4> count;

  //------------Code Starts Here-----------------
  // Below function implements actual counter logic
  void incr_count () {
    /* At every rising edge of clock we check if reset is
active*/
    /* If active, we load the counter output with
4'b0000*/
    if (reset.read() == 1) {
      count =  0;
      counter_out.write(count);
    /* If enable is active, then we increment the
counter*/
    } else if (enable.read() == 1) {
      count = count + 1;
      counter_out.write(count);
    }
  } // End of function incr_count
```

```
  /* Below functions prints value of count when ever it
changes*/
  void print_count () {
    cout<<"@" << sc_time_stamp() <<
      " :: Counter Value "<<counter_out.read()<<endl;
  }

  // Constructor for the counter
  /* Since this counter is a positive edge trigged one,*/
  /* We trigger the below block with respect to
positive*/
  /* edge of the clock and also when ever reset changes
state*/
  SC_CTOR(first_counter) {
    // Edge and level sensitive
    SC_METHOD(incr_count);
    sensitive << reset;
    sensitive << clock.pos();
    // Level Sensitive method
    SC_METHOD(print_count);
    sensitive << counter_out;
  } // End of Constructor

}; // End of Module counter
};

Testbench for sc_counter_method
int sc_main (int argc, char* argv[]); {
  sc_signal<bool>   clock;
  sc_signal<bool>   reset;
```

```
  sc_signal<bool>   enable;
  sc_signal<sc_uint<4> > counter_out;
  int i = 0;
  // Connect the DUT
  first_counter counter("COUNTER");
    counter.clock(clock);
    counter.reset(reset);
    counter.enable(enable);
    counter.counter_out(counter_out);
      sc_start(1, SC_MS);


  // Initialize all variables
  reset = 0;       // initial value of reset
  enable = 0;      // initial value of enable

  reset = 1;    // Assert the reset
  cout << "@" << sc_time_stamp() <<" Asserting reset\n"
<< endl;
  for (i=0;i<3;i++) {
    clock = 0;
    sc_start(1, SC_MS);
    clock = 1;
    sc_start(1, SC_MS);
  }
  reset = 0;    // De-assert the reset
  cout << "@" << sc_time_stamp() <<" De-Asserting
reset\n" << endl;
  for (i=0;i<4;i++) {
    clock = 0;
```

```
    sc_start(1, SC_MS);
    clock = 1;
    sc_start(1, SC_MS);
  }
  cout << "@" << sc_time_stamp() <<" Asserting Enable\n"
<< endl;
  enable = 1;  // Assert enable
  for (i=0;i<5;i++) {
    clock = 0;
    sc_start(1, SC_MS);
    clock = 1;
    sc_start(1, SC_MS);
  }
  cout << "@" << sc_time_stamp() <<" De-Asserting
Enable\n" << endl;
  enable = 0; // De-assert enable
    cout << "@" << sc_time_stamp() <<" Terminating
simulation\n" << endl;
    return 0;// Terminate simulation

};
};
```

Результаты моделирования:

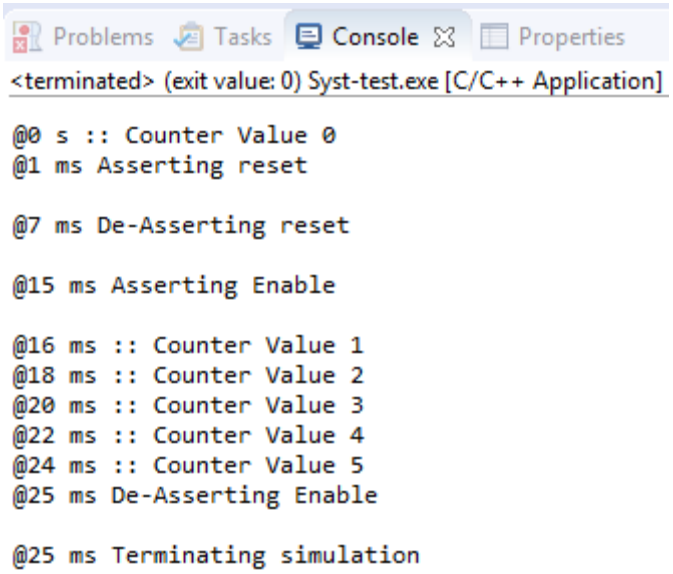


Рис. 3.6. Результаты моделирования в Eclipse

Рекомендуем сверить текст программы с результатом моделирования и убедиться в соответствии результатов коду программы.

5.9.3. Процесс SC_CTHREAD

Вариацией процесса SC_THREAD с тактовой частотой является популярный для поведенческих инструментов синтеза SC_CTHREAD,. Эта популярность отчасти потому, что синтезированные логические инструменты в настоящее время производят для полностью синхронного кода, и SC_CTHREAD предоставляет некоторые новые возможности для упрощения кодирования.

```
SC_CTOR(module_name) {
SC_CTHREAD(NAME_cthread, clock_name.edge()) ;
}
```

Процесс SC_CTHREAD отличается от процесса SC_THREAD несколькими способами. Сначала процесс SC_CTHREAD указывает объект тактирования. Когда другие типы процессов описывают в конструкторе модуля, они имеют только имя указанного процесса, а процесс SC_CTHREAD имеет имя процесса и такты, которые запускают процесс. У SC_CTHREAD нет отдельного списка чувствительности, как у других типов процессов. Список чувствительности - это только заданный фронт тактового

сигнала. Процесс SC_CTHREAD будет активизироваться всякий раз, когда происходит указанный фронт тактового сигнала. В этом примере указан положительный фронт такта, поэтому процесс `incr_count` будет выполняться на каждом положительном фронте тактового сигнала.

Примечание: SC_CTHREAD может иметь только три бита портов как триггер.

Пример программы с процессом SC_CTHREAD приведен в листинге 3.4

Листинг 5.4

Программа sc_counter_cthread.cpp

```
//------------------------------------------------
// This is my second Systemc Example
// Design Name : first_counter
// File Name : first_counter.cpp
// Function : This is a 4 bit up-counter with
// Synchronous active high reset and
// with active high enable signal
//------------------------------------------------
#include "systemc.h"

SC_MODULE (first_counter) {
  sc_in_clk     clock ;   /* Clock input of the design*/
  sc_in<bool>   reset ;      /* active high, synchronous
Reset input*/
  sc_in<bool>   enable;      /* Active high enable signal
for counter*/
  sc_out<sc_uint<4> > counter_out; /* 4 bit vector output
of the counter*/
```

```
  //---------Local Variables Here-----------------
  sc_uint<4> count;

  //----------Code Starts Here----------------------
  // Below function implements actual counter logic
  void incr_count () {
    // For threads, we need to have while true loop
    while (true) {
      /* Wait for the event in sensitivity list to
occure*/
      // In this example - positive edge of clock
      wait();
      if (reset.read() == 1) {
        count =  0;
        counter_out.write(count);
      /* If enable is active, then we increment the
counter*/
      } else if (enable.read() == 1) {
        count = count + 1;
        counter_out.write(count);
      }
    }
  } // End of function incr_count

  /* Below functions prints value of count when ever it
changes*/
  void print_count () {
    while (true) {
      wait();
      cout<<"@" << sc_time_stamp() <<
```

```
        " :: Counter Value "<<counter_out.read()<<endl;
    }
  }

  // Constructor for the counter
  /* Since this counter is a positive edge trigged one,*/
  /* We trigger the below block with respect to
positive*/
  // edge of the clock
  SC_CTOR(first_counter) {
    /* cthreads require to have thread name and
triggering*/
    // event to passed as clock object
    SC_CTHREAD(incr_count, clock.pos());
    // Level Sensitive to change in counter output
    SC_THREAD(print_count);
      sensitive << counter_out;
  } // End of Constructor

}; // End of Module counter


//Testbench for sc_counter_cthread.cpp
int sc_main (int argc, char* argv[]) {
  sc_signal<bool>   clock;
  sc_signal<bool>   reset;
  sc_signal<bool>   enable;
  sc_signal<sc_uint<4> > counter_out;
  int i = 0;
  // Connect the DUT
```

```
  first_counter counter("COUNTER");
    counter.clock(clock);
    counter.reset(reset);
    counter.enable(enable);
    counter.counter_out(counter_out);
      sc_start(1, SC_MS);


  // Initialize all variables
  reset = 0;       // initial value of reset
  enable = 0;      // initial value of enable
  for (i=0;i<2;i++) {
    clock = 0;
    sc_start(1, SC_MS);
    clock = 1;
    sc_start(1, SC_MS);
  }
  reset = 1;    // Assert the reset
  cout << "@" << sc_time_stamp() <<" Asserting reset\n"
<< endl;
  for (i=0;i<2;i++) {
    clock = 0;
    sc_start(1, SC_MS);
    clock = 1;
    sc_start(1, SC_MS);
  }
  reset = 0;    // De-assert the reset
  cout << "@" << sc_time_stamp() <<" De-Asserting
reset\n" << endl;
  for (i=0;i<3;i++) {
```

```
    clock = 0;
    sc_start(1, SC_MS);
    clock = 1;
    sc_start(1, SC_MS);
  }
  cout << "@" << sc_time_stamp() <<" Asserting Enable\n"
<< endl;
  enable = 1;  // Assert enable
  for (i=0;i<5;i++) {
    clock = 0;
    sc_start(1, SC_MS);
    clock = 1;
    sc_start(1, SC_MS);
  }
  cout << "@" << sc_time_stamp() <<" De-Asserting
Enable\n" << endl;
  enable = 0; // De-assert enable
    cout << "@" << sc_time_stamp() <<" Terminating
simulation\n" << endl;

  return 0;// Terminate simulation

 }
```

Результаты моделирования;

```
@5 ms Asserting reset

@9 ms De-Asserting reset

@15 ms Asserting Enable

@16 ms :: Counter Value 1
@18 ms :: Counter Value 2
@20 ms :: Counter Value 3
@22 ms :: Counter Value 4
@24 ms :: Counter Value 5
@25 ms De-Asserting Enable

@25 ms Terminating simulation
```

Рис. 5.7. Результаты моделирования в среде Eclipse

Процесс и потоки (не cthreads) автоматически запускаются в конструкторе, даже если событие в списке чувствительности не возникает. Чтобы предотвратить это неумышленное выполнение, используйте функцию `dont_initialize ()`, как показано в примере.

Листинг 5.5

Программа sc_dont_initialize.cpp

```
#include "systemc.h"

SC_MODULE (tff_sync_reset) {
  sc_in    <bool> data, clk, reset  ;
  sc_out   <bool> q;

  bool q_l ;

  void tff () {
    if (reset.read()) {
      q_l = 0;
    } else if (data.read()) {
```

```
      q_l = !q_l;
    }
    q.write(q_l);
  }

  SC_CTOR(tff_sync_reset) {
    SC_METHOD (tff);
      dont_initialize();
      sensitive << clk.pos();
  }

};

SC_MODULE (tff_tb) {
  sc_in<bool> clk;

  sc_signal <bool> data, reset  ;
  sc_signal   <bool> q;
  tff_sync_reset *dut;

  void do_test() {
    cout << "@" << sc_time_stamp() <<" Starting
test"<<endl;
    wait();
    cout << "@" << sc_time_stamp() <<" Asserting
reset"<<endl;
    reset = true;
    wait(4);
    cout << "@" << sc_time_stamp() <<" De-Asserting
reset"<<endl;
```

```
    reset = false;
    wait(3);
    cout << "@" << sc_time_stamp() <<" Asserting Data
input"<<endl;
    data = true;
    wait(3);
    data = false;
    cout << "@" << sc_time_stamp() <<" De-Asserting Data
input"<<endl;
    wait(3);
    cout << "@" << sc_time_stamp() <<" Terminating
simulation"<<endl;
    sc_stop();
  }

  SC_CTOR(tff_tb) {
    dut = new tff_sync_reset ("TFF");
    dut->clk    (clk);
    dut->reset  (reset);
    dut->data   (data);
    dut->q      (q);
    SC_THREAD (do_test);
      dont_initialize();
      sensitive << clk.pos();
  }
};

int sc_main (int argc, char* argv[]) {
  sc_clock clock ("my_clock",1,0.5);
```

```
  tff_tb  object("TFF_TB");
    object.clk (clock);
  sc_trace_file *wf = sc_create_vcd_trace_file("tff");
    sc_trace(wf, object.clk, "clk");
    sc_trace(wf, object.reset, "reset");
    sc_trace(wf, object.data, "data");
    sc_trace(wf, object.q, "q");

  sc_start(0);
  sc_start();
  sc_close_vcd_trace_file(wf);
  return 0;// Terminate simulation
}
```

Результаты моделирования:

```
Info: (I702) default timescale unit used for tracing: 1 ps (tff.vcd)
@0 s Starting test
@1 ns Asserting reset
@5 ns De-Asserting reset
@8 ns Asserting Data input
@11 ns De-Asserting Data input
@14 ns Terminating simulation

Info: /OSCI/SystemC: Simulation stopped by user.
```

Рис. 5.8. Результаты моделирования

Некоторыми из более простых объектов, предоставляемых этим новым процессом, являются новая форма ожидания `wait(N)` и уровня чувствительных ожиданий, которая называется `wait_until ()`. Синтаксисы выглядят так:

```
wait(N); // delay N clock edges
wait_until(delay_expr);//until expr true @ clock
```

5.10. Глобальное и локальное наблюдение

Наибольший интерес в том, что SC_CTHREAD обеспечивает концепцию наблюдения сигналов, которые эффективно изменяют поведение

от wait(). Когда наблюдают активный сигнал, выполнение переходит к новой области после возвращения из ожидания (), а не переходит к следующей команде. SystemC имеет две формы наблюдения: глобальную и локальную. Самая простая форма наблюдения носит глобальный характер. Глобальное наблюдение следит за активностью сигнала, чтобы перезапустить тактируемый поток с самом начала.

Более управляемые формы наблюдения предполагает использование четырех макросов: W_BEGIN, W_DO, W_ESCAPE и W_END. Эти макросы всегда используется в качестве набора в указанном порядке, и они определяют три области в коде наблюдения.

В первой области сигналы для слежения объявляются. В второй области, код слежения кодируется. В последнем области обеспечивается обрабатка наблюдаемой активации сигнала. Вот краткий обзор синтаксиса:

```
W_BEGIN
watching(delay_expr);
W_DO
Code_being_watched
W_ESCAPE
Code_handling_escape_condition
W_END
```

В наших программах, основванных на библиотеке SystemC-2.3.1, эти макросы отсутствуют.

Синтаксис wait () и wait_until () требует выражения задержки, delay_expr, который должен быть выражен с использованием сигналов с задержкой. В версиях SystemC перед стандартизацией существует еще один синтаксис для wait (N) и чувствительный к уровню ожидания, называемый wait_until (). Вы можете увидить это в устаревшем коде.

Это метод delayed () – специальный метод, который дает значение в конце дельта-цикла. Имейте в виду, что все этот метод устарел, отсутствует в

новом стандарте, и этот синтаксис применим только к версиям до версии OSCI 2.2. Метод delayed (), можно заменить почти эквивалентным следующим кодом SC_THREAD, предполагая, что поток статически чувствительный к краю тактового сигнала:

```
for(i=0;i!=N;i++) wait();//similar as wait(N)
do wait() while(!expr);// sames as
// wait_until(dexpr)
```

Глобальное наблюдение

Процессы SC_CTHREAD обычно имеют бесконечные циклы, которые будут выполняться непрерывно. Дизайнер обычно хочет каким-то образом инициализировать поведение цикла или выйти из цикла при возникновении условия. Это достигается с помощью наблюдательной конструкции. Наблюдающая конструкция будет контролировать указанное условие. Когда возникает это условие, управление переносится из текущей точки выполнения в начало процесса, где можно наблюдать появление наблюдаемого условия. Наблюдающая конструкция доступна только для процессов SC_CTHREAD.

Одним из неожиданных последствий выхода из цикла `while` и начала в начале процесса является то, что все переменные, локально определенные внутри процесса, потеряют свое значение. Если значение переменной необходимо сохранить между вызовами процесса, объявите переменную в модуле процесса, а не локальную для процесса.

Пример глобального просмотра

В конструкторе примера приведен следующий оператор: watching(reset == true). Этот оператор указывает, что для этого процесса будет наблюдаться сброс сигнала. Если сброс сигнала изменится на `true`, наблюдающее выражение будет истинным, и планировщик SystemC остановит выполнение цикла `while` для этого процесса и начнет выполнение в первой строке процесса.

Локальное наблюдение

Локальное наблюдение позволяет вам точно указать, какая часть процесса наблюдает какие-либо сигналы и где расположены обработчики событий. Эта функциональность задается 4 макросами, которые определяют границы каждой из областей. Ниже приведен псевдосинтаксис локального просмотра.

```
W_BEGIN
  // put the watching declarations here
  watching(...);
  watching(...);
W_DO
  // This is where the process functionality goes
  ...
W_ESCAPE
  // This is where the handlers for the watched events go
  if (..) {
    ...
  }
W_END
```

Макрос `W_BEGIN` отмечает начало локального блока наблюдения. Между макросами `W_BEGIN` и `W_DO` находятся все объявления просмотра. Эти объявления выглядят так же, как глобальные события просмотра. Между макросом `W_DO` и макросом `W_ESCAPE` находится место, где размещаются функциональные возможности процесса. Это код, который выполняется, пока не происходит ни одно из событий наблюдения. Между макросами `W_ESCAPE` и `W_END` находятся обработчики событий. Обработчики событий проведут проверку, чтобы убедиться, что произошло соответствующее событие, а затем выполните необходимые действия для этого события. Макрос `W_END` завершает локальный блок наблюдения.

Ранее мы отмечали, что указанные макросы отсутствуют в нашей версии SystemC-2.3.1.

Есть несколько интересных вещей, которые возможны при локальном наблюдении:

Все события в блоке объявления имеют одинаковый приоритет. Если нужен другой приоритет, тогда локальные блоки просмотра должны быть вложены.

Локальный просмотр работает только в процессах `SC_CTHREAD`.

Сигналы в наблюдающих выражениях отбираются только по активным границам процесса. В процессе `SC_CTHREAD` это означает, что процесс чувствителен к изменениям только в тактах.

Наблюдаемые в глобальном масштабе события имеют более высокий приоритет, чем локально наблюдаемые события.

Без макросов W_BEGIN, W_DO, W_ESCAPE, W_END не работает оператор watching (). Отредактированная нами программа, в которой исключены неработающие операторы и использованы условные операторы и дает те же результаты моделирования.

5.11. Порты и сигналы

Порты модуля являются внешним интерфейсом, который передает информацию из модуль и в модуль и переключает действия внутри модуля. Сигналы создают связи между портами модулей, что позволяет модулям взаимодействовать.

Порт может иметь три различных режима работы:

• Ввод

• Вывод

• Ввод-Вывод.

Входной порт (In) передает данные в модуль. Выходной порт (Out) передает данных из модуля и порт InOut передает данные как в модуль, так и из него в зависимости от работа модуля.

Сигнал присоединяет порт одного модуля к порту другого модуля. Сигнал передает данные от одного порта к другому, как если бы порты были непосредственно связаны. Когда порт считывающий, значение сигнала, подключенного к порту, является возвращаемым. Когда порт записывающий, новое значение будет записано в сигнал, когда процесс выполнения операции записи завершится или будет приостановлен. Это сделано, чтобы все операции в рамках процесса работали с тем же значением сигнала. Это должно предотвратить то, что некоторые процессы наблюдают старое значение, в то время как другие процессы наблюдают новое значение сигнала во время выполнения. Все процессы, выполняемые в течение временного шага, будут видеть старое значение сигнала. Эти семантика сигналов такая же, как операции с сигналами в VHDL и отсроченное поведение присваивания в Verilog.

Порты всегда связаны с сигналом за исключением одного частного случая, когда порт привязывается непосредственно к другому порту. Порты всегда связаны только с одним сигналом. Этот сигнал может быть комплексным сигналом, таким как структура, но он по-прежнему рассматривается как один сигнал. Соединение сигналами происходит во время конкретизации модуля.

При построении иерархической структуры проектирования модули инстанцируются в пределах других модулей, чтобы сформировать иерархию дизайна. Особый случай соединения, упомянутый раньше, происходит, когда порт модуля верхнего уровня непосредственно связан с портом модуля более низкого уровня в момент создания конструкции. Это показано на рисунке ниже:

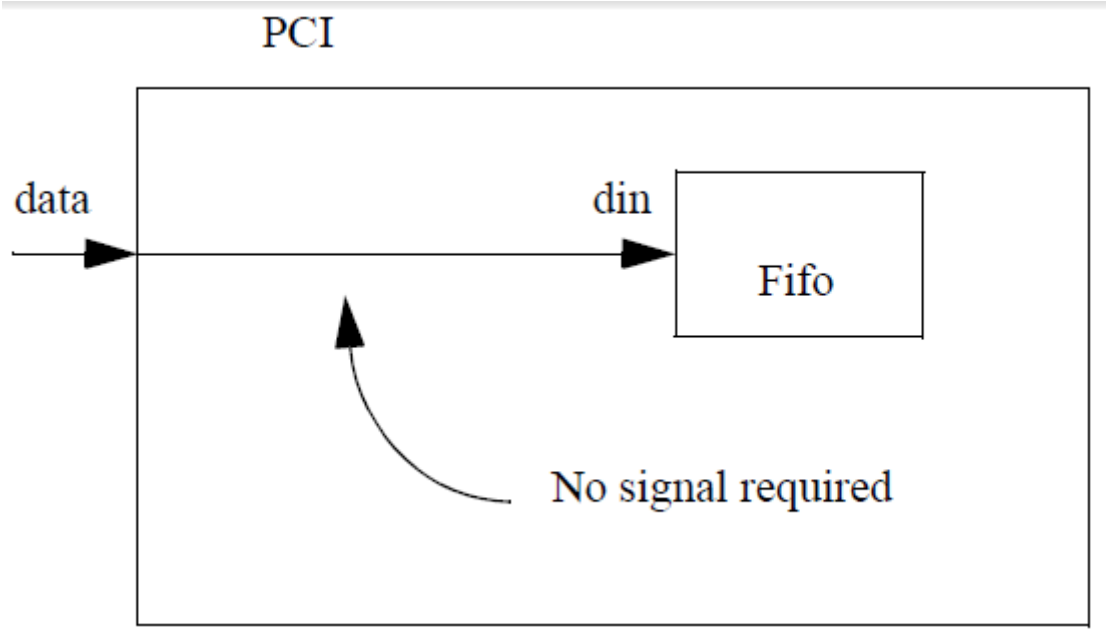


Рис. 5.11

В этом примере порт данных модуля PCI напрямую подключен к порту din модуля FIFO. Для этого случая не требуется локального сигнала.

Порты и сигналы также бывают разных размеров. Скалярные порты имеют одно измерение. Скалярный порт может быть одним из следующих типов:

C++ встроенные типы

- long
- int
- char
- short
- float
- double

SystemC типы

- sc_int<n>
- sc_uint<n>
- sc_bigint<n>
- sc_biguint<n>

- sc_bit
- sc_logic
- sc_bv<n>
- sc_lv<n>
- sc_fixed
- sc_ufixed
- sc_fix
- sc_ufix
- User defined structs

Входные, выходные и InOut порты описаны с использванием следующиго синтаксиса:

```
sc_in<porttype> // input port of type porttype
sc_out<porttype> // output port of type porttype
sc_inout<porttype> // inout port of type porttype
```

5.11.1. Чтение и запись портов и сигналов

Вы можете использовать методы read() и write() или оператор присваивания для чтения и записи портов и сигналов. Использование оператора присваивания делает ваш код более кратким и больше подобным коду HDL, потому что вы считываете и записываете непосредственно в портах.

Используйте read() и write() методы в явном виде для уточнения цели вашего кода, даже, если ваш код будет немного более многословным. Методы read() и write() вызываются неявным преобразованием, определенным в классе порта.

Если вам нужно неявное преобразование типов, так как тип, который вы читаете или записываете отличается от типа порта (например, если порт является BOOL и вы читаете или записываете INT), важно, чтобы вы

использовали методы read() и write(). C ++ автоматически применяет только одно неявное преобразование типов в любом конкретном месте, и вам нужны два неявных преобразования для чтения и записи другого типа, чем тип порта.

Не рекомендуется напрямую обращаться к портам, как для чтения, так и для записи. Типы данных порта должны использовать следующие методы для доступа к ним:

`Port_name.write ('value')`: для записи значения в порт.

`Port_name.read ()`: для чтения значения из порта

Методы write () и read () выполняют автоматическое преобразование типов из других типов данных в типы данных порта. Не всегда возможно использовать одни и те же типы данных портов или сигналов внутри процессов. При использовании разных типов данных всегда полезно использовать `port_name.read ()` и `port_name.write` («значение») для доступа к портам, и одно и то же правило применяется для сигналов.

Листинг 5.9

Программа sc_ ports_access

```
#include <systemc.h>

SC_MODULE(ports_access) {
  sc_in<sc_bit> a;
  sc_in<sc_bit> b;
  sc_in<bool>   en;
  sc_out<sc_lv<2> >  out;

  // Method to manipulate output
  void body () {
    // Ports should use read() method to read values
```

```
    if (en.read() == 1) {
      // Should use write() method of write values
      out.write(a.read() + b.read());
    }
  }
  // Method to monitor ports
  void monitor () {
    cout << "@" << sc_time_stamp() <<" a : " << a
      << " b : " << b << " en : " << " out : "
      << out.read() <<endl;
  }

  SC_CTOR(ports_access) {
    SC_METHOD(body);
      sensitive << a << b << en;
    SC_METHOD(monitor);
      sensitive << a << b << en << out;
  }

};

// Testbench to generate test vectors
int sc_main (int argc, char* argv[]) {
  sc_signal <sc_bit> a;
  sc_signal <sc_bit> b;
  sc_signal <bool>   en;
  sc_signal <sc_lv<2> >  out;

  ports_access prt_ac("PORT_ACCESS");
    prt_ac.a(a);
```

```
    prt_ac.b(b);
    prt_ac.en(en);
    prt_ac.out(out);

  sc_start(0, SC_MS);
  // Open VCD file
  sc_trace_file *wf =
sc_create_vcd_trace_file("ports_access");
    sc_trace(wf, a, "a");
    sc_trace(wf, b, "b");
    sc_trace(wf, en, "en");
    sc_trace(wf, out, "out");
  // Start the testing here
  sc_start(1, SC_MS);
  a = sc_bit('0');
  b = sc_bit('0');
  en = 1;
  sc_start(1, SC_MS);
  a = sc_bit('1');
  sc_start(1, SC_MS);
  b = sc_bit('1');
  sc_start(2, SC_MS);

  sc_close_vcd_trace_file(wf);
  return 0;// Terminate simulation
}
```

Результаты моделирования:

```
@1 ns a : 0 b : 0 en :  out : XX
 @1 ns a : 0 b : 0 en :  out : 00
```

```
@2 ns a : 1 b : 0 en :  out : 00
@2 ns a : 1 b : 0 en :  out : 01
@3 ns a : 1 b : 1 en :  out : 01
@3 ns a : 1 b : 1 en :  out : 10
```

```
@0 s a : 0 b : 0 en :  out : XX

Info: (I702) default timescale unit
@1 ms a : 0 b : 0 en :  out : XX
@1 ms a : 0 b : 0 en :  out : 00
@2 ms a : 1 b : 0 en :  out : 00
@2 ms a : 1 b : 0 en :  out : 01
@3 ms a : 1 b : 1 en :  out : 01
@3 ms a : 1 b : 1 en :  out : 10
```

На рис. 5.12 показана трассировка сигналов в GTKWave.

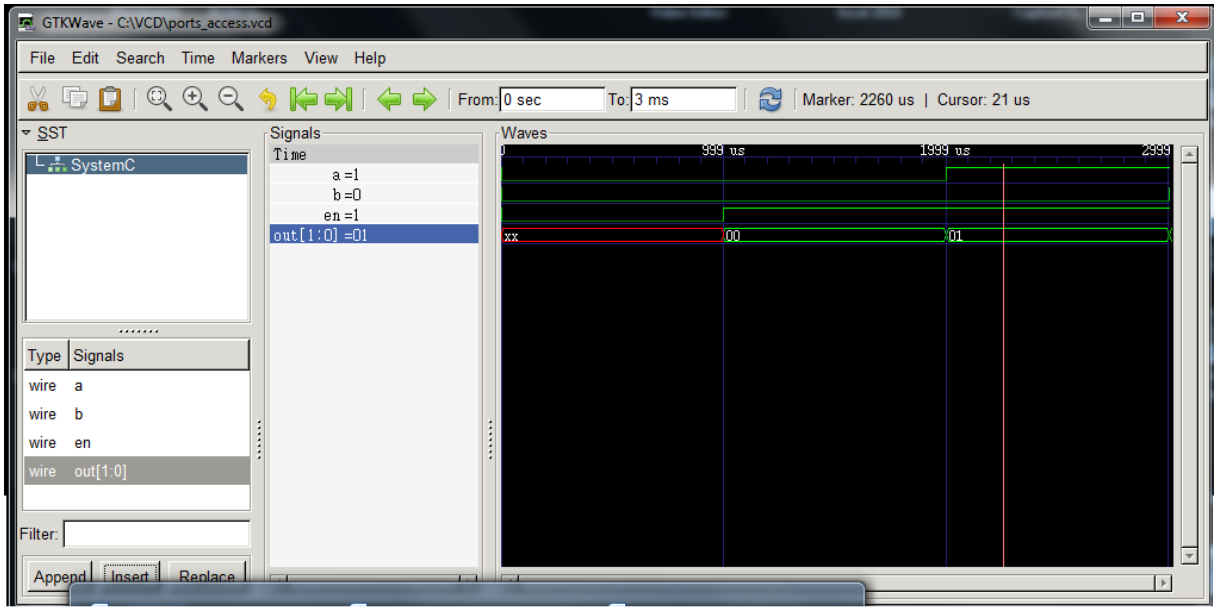


Рис. 3.12. Трассировка сигналов

5.11.2. Привязка сигналов

Каждый порт привязывается к одному сигналу. При чтении порта переменная, назначенная к порту, должна иметь тот же тип, что и тип порта. Для примера порт типа `sc_logic` не может быть прочитан в `int` переменной или сигналом.

Когда порты связаны с другими сигналами или портами, оба типа должны совпадать.

Ниже показан порт, связанный с другим портом (частный случай), и порт связанный с сигналом.

```
// statemach.h
#include "systemc.h"
SC_MODULE(state_machine) {
sc_in<sc_logic> clock;
sc_in<sc_logic> en;
sc_out<sc_logic> dir;
sc_out<sc_logic> status;
// ... other module statements
};

// controller.h
#include "statemach.h"
SC_MODULE(controller) {
sc_in<sc_logic> clk;
sc_out<sc_logic> count;
sc_in<sc_logic> status;
sc_out<sc_logic> load;
sc_out<sc_logic> clear
sc_signal<sc_logic> lstat;
sc_signal<sc_logic> down;
state_machine *s1;
SC_CTOR(controller) {
// .... other module statements
s1 = new state_machine ("s1");
s1->clock(clk); // special case port to
// port binding
s1->en(lstat); // port en bound to signal lstat
s1->dir(down); // port dir bound to signal down
```

```
s1->st(status); // special case port to
// port binding
}
};
```

В этом примере показан модуль контроллера с некоторым числом входных и выходных портов. Модуль также включает в себя локальные сигналы `lstat` и `down`. Модуль контроллера инстанцирует модуль `state_machine` с меткой `s1`.

Модуль `state_machine` должен содержать следующие операторы.

Первый оператор:

```
s1->clock(clk);
```

связывает тактирование порта с меткой `s1` к внешнему портом CLK контроллера. Это пример специального случая связывания, в котором порт привязывается непосредственно к другому порту вместо сигнала. Вторая привязка порта показан ниже:

```
s1->en(lstat);
```

Этот оператор привязывает порт с меткой `s1` к локальному сигналу `lstat`. Это является примером «именнованных соединений»-Named Mapping из раздела "Позиционные соединения".

Программа sc_signal_bind на листинге 3.12 иллюстрирует вопросы привязки портов.

Листинг 5.12

Программа sc_signal_bind

```
#include <systemc.h>

SC_MODULE (some_block) {
  sc_in<bool>   clock;
  sc_in<sc_bit> data;
```

```
  sc_in<sc_bit> reset;
  sc_in<sc_bit> inv;
  sc_out<sc_bit> out;

  void body () {
    if (reset.read() == 1) {
      out = sc_bit(0);
    } else if (inv.read() == 1) {
      out = ~out.read();
    } else {
      out.write(data.read());
    }
  }

  SC_CTOR(some_block) {
    SC_METHOD(body);
      sensitive << clock.pos();
  }
};


SC_MODULE (signal_bind) {
  sc_in<bool> clock;

  sc_signal<sc_bit> data;
  sc_signal<sc_bit> reset;
  sc_signal<sc_bit> inv;
  sc_signal<sc_bit> out;
  some_block *block;
  int  done;
```

```
  void do_test() {
    while (true) {
      wait();
      if (done == 0) {
        cout << "@" << sc_time_stamp() <<" Starting
test"<<endl;
        wait();
        wait();
        inv = sc_bit('0');
        data = sc_bit('0');
        cout << "@" << sc_time_stamp() <<" Driving 0 all
inputs"<<endl;
        // Assert reset
        reset = sc_bit('1');
        cout << "@" << sc_time_stamp() <<" Asserting
reset"<<endl;
        // Deassert reset
        wait();
        wait();
        reset = sc_bit('0');
        cout << "@" << sc_time_stamp() <<" De-Asserting
reset"<<endl;
        // Assert data
        wait();
        wait();
        cout << "@" << sc_time_stamp() <<" Asserting
data"<<endl;
        data = sc_bit('1');
        wait();
```

```
        wait();
        cout << "@" << sc_time_stamp() <<" Asserting
inv"<<endl;
        inv = sc_bit('1');
        wait();
        wait();
        cout << "@" << sc_time_stamp() <<" De-Asserting
inv"<<endl;
        inv = sc_bit('0');
        wait();
        wait();
        done = 1;
      }
    }
  }

  void monitor() {
    cout << "@" << sc_time_stamp() << " data :" << data
      << " reset :" << reset << " inv :"
      << inv << " out :" << out <<endl;
  }

  SC_CTOR(signal_bind) {
    block = new some_block("SOME_BLOCK");
    block->clock       (clock)  ;
    block->data        (data)   ;
    block->reset       (reset)  ;
    block->inv         (inv)    ;
    block->out         (out)    ;
    done = 0;
```

```
    SC_CTHREAD(do_test,clock.pos());
    SC_METHOD(monitor);
      sensitive << data << reset << inv << out;
  }
};

int sc_main (int argc, char* argv[]) {
  sc_signal<bool> clock;
  int i;

  signal_bind  object("SIGNAL_BIND");
    object.clock (clock);
  sc_trace_file *wf =
sc_create_vcd_trace_file("signal_bind");
    sc_trace(wf, object.clock, "clock");
    sc_trace(wf, object.reset, "reset");
    sc_trace(wf, object.data, "data");
    sc_trace(wf, object.inv, "inv");
    sc_trace(wf, object.out, "out");

  sc_start(0, SC_MS);
  for(i=0;i<100;i++) {
    if (object.done == 0) {
      clock = 0;
      sc_start(1, SC_MS);
      clock = 1;
      sc_start(1, SC_MS);
    }
  }
```

```
  sc_close_vcd_trace_file(wf);

  cout<<"Terminating Simulation"<<endl;
  return 0;// Terminate simulation
    }
```

Результаты моделирования:

```
 @3 ns Starting test
 @7 ns Driving 0 all inputs
 @7 ns Asserting reset
 @7 ns data :0 reset :1 inv :0 out :0
 @11 ns De-Asserting reset
 @11 ns data :0 reset :0 inv :0 out :0
 @15 ns Asserting data
 @15 ns data :1 reset :0 inv :0 out :0
 @17 ns data :1 reset :0 inv :0 out :1
 @19 ns Asserting inv
 @19 ns data :1 reset :0 inv :1 out :1
 @21 ns data :1 reset :0 inv :1 out :0
 @23 ns De-Asserting inv

 @23 ns data :1 reset :0 inv :0 out :1
 Terminating Simulation
```

```
<terminated> (exit value: 0) Syst-test.exe [C/C++ Application] C:\SystemC\workspace\Syst-test\Debu

Info: (I702) default timescale unit used for tracing: 1 ps (signal_bind.vcd)
@3 ms Starting test
@7 ms Driving 0 all inputs
@7 ms Asserting reset
@7 ms data :0 reset :1 inv :0 out :0
@11 ms De-Asserting reset
@11 ms data :0 reset :0 inv :0 out :0
@15 ms Asserting data
@15 ms data :1 reset :0 inv :0 out :0
@17 ms data :1 reset :0 inv :0 out :1
@19 ms Asserting inv
@19 ms data :1 reset :0 inv :1 out :1
@21 ms data :1 reset :0 inv :1 out :0
@23 ms De-Asserting inv
@23 ms data :1 reset :0 inv :0 out :1
```

Трассировка сигналов показана на рис. 5.13

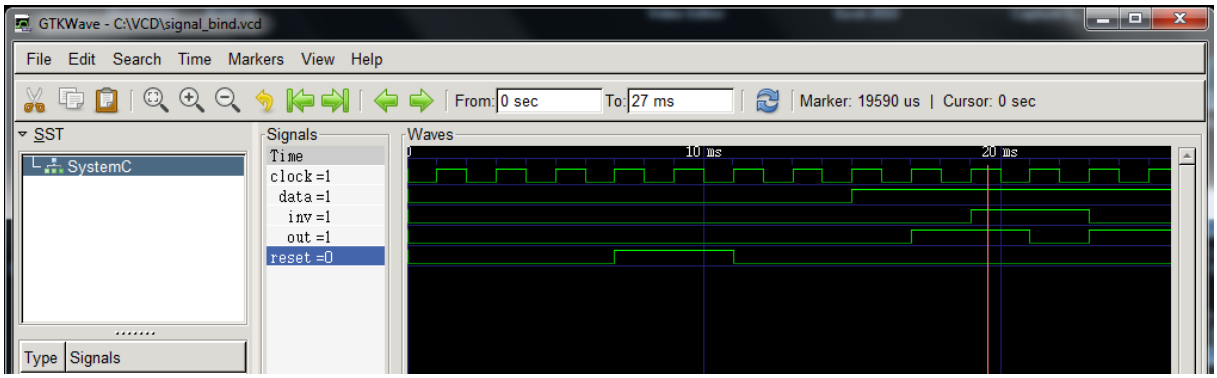


Рис. 5.13. Трассировка сигналов программы SC_SIGNAL_BIND

5.12. Тактирование

Объекты Clock являются специальными объектами в SystemC. Они генерируют сигналы синхронизации, используемые, чтобы синхронизировать события в симуляции. Вызов событий `clock` во времени, требуется, чтобы параллельные события в аппаратных средствах правильно моделировались симулятором на последовательном компьютере. Объект clock имеет ряд элементов данных для хранения настроек, а также методов для выполнения тактирования. Для создания объекта синхронизации используйте следующий синтаксис:

```
sc_clock CLOCK1 ("CLOCK1", 20, 0,5, 2, true);
```

Эта декларация создаст объект с именем clock с периодом 20 единиц, рабочий цикл 50%, первый фронт будет происходить на 2-ой единице

времени, и первое значение будет `true` (1). Все эти аргументы имеют значения по умолчанию, за исключением имя `clock`. По умолчанию период равен 1, рабочий цикл до 0,5, первый фронт на «0», а первое значение `true`.

Обычно `Clock` создается на верхнем уровне дизайна в Testbench и опускается вниз по иерархии модулей для остальной части конструкции. Это позволяет синхронизировать все участки конструкции или всю конструкцию одними тактами.

В примере программа `sc_main` конструкции создает `clock` и соединяет его со всеми компонентами основного модуля:

```
int sc_main(int argc, char*argv[]) {
sc_signal<int> val;
sc_signal<sc_logic> load;
sc_signal<sc_logic> reset;
sc_signal<int> result;
sc_clock ck1("ck1", 20, 0.5, 0, true);
filter f1("filter");
f1.clk(ck1.signal());
f1.val(val);
f1.load(load);
f1.reset(reset);
f1.out(result);
// rest of sc_main not shown
}
```

В этом примере процедура `sc_main` верхнего уровня конкретизирует модуль с названием фильтр и объявляет некоторые локальные сигналы, которые будут подключать фильтр к другим конкретным модулям. Обратите внимание на то, что сигнал CLK не объявлен, вместо этого, чтобы объект CLK инстанциировался, его параметры настройки установливаются, и его

`signal_method` используется для обеспечения сигнала синхронизации. Функция `ck1.signal ()` отображается на CLK порт объекта фильтра. В этом примере `clock` называют CK1 и тактовая частота определяется как 20 единиц измерения времени. Каждые 20 единиц времени `clock` сделают полный переход от `true` к `false` и обратно.

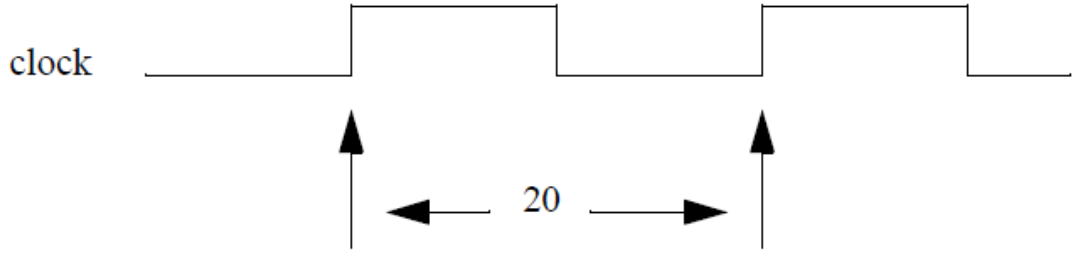


Рис. 3.16

Для модуля тактирования заданного так:

```
sc_clock ck1("ck1", 20, 0.5, 2, true);
```

получим:

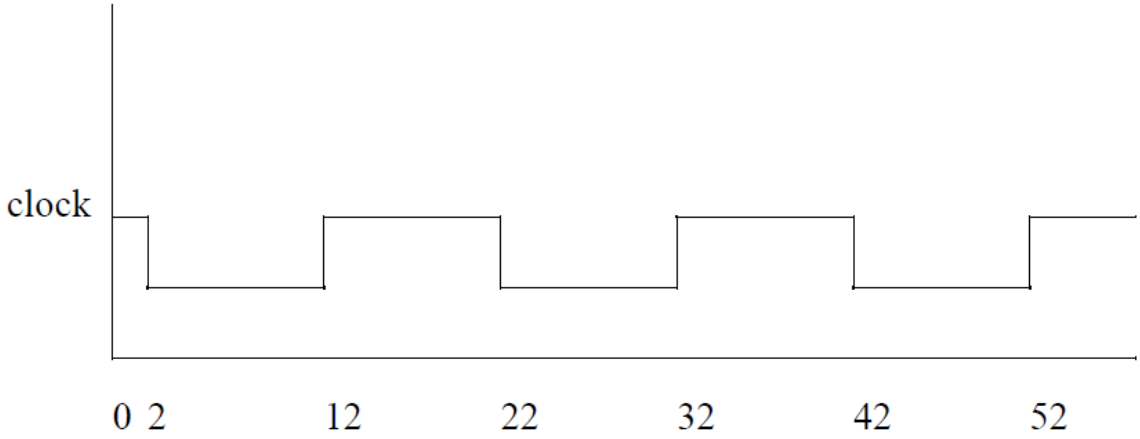


Рис. 5.17

При привязке такта к порту дизайнер должен использовать тактовый сигнал, который генерируется объектом `clock` в обозначении порта. Это делается с помощью метода сигнала, характеризующего объект `clock`. Обратите внимание на то, что CLK порта фильтра отображается на ck1.signal (). Этот тактовый сигнал генерируется объектом синхронизации.

Для процессов SC_CTHREAD `clock` объект отображен непосредственно на тактовом входе процесса и `signal_method` не требуется. Часто требуется указать размерность времени (например, SC_MS).

При привязке тактов к порту разработчик должен использовать тактовый сигнал, генерируемый тактируемым объектом. Это делается с использованием метода сигнала объекта clock.

Для процессов SC_CTHREAD объект тактированяи напрямую отображается на тактовый вход процесса, а метод `signal ()` не требуется.

5.13. Время

Различие между HDL и языком программирования SystemC - это понятие времени и совпадения.

Рассмотрим тип данных времени. Этот тип данных включает:

```
Sc_start ()
Sc_time_stamp ()
Wait (sc_time)
Sc_simulation_time ()
sc_event
```

`Sc_time` - специальный тип данных, который используется для представления временных задержек и временных интервалов моделирования, включая задержки и тайм-ауты. Объект класса `sc_time` строится из `double` и `sc_time_unit`. Время должно быть представлено внутренне как целое число без знака, по меньшей мере, 64 бита. Как и любой другой тип данных в SystemC, sc_time также позволяет выполнять арифметические операции.

Типы перечисления, которые обозначают различные единицы времени, приведены ниже

```
SC_FS = 0
SC_PS = 1
```

```
SC_NS = 2
SC_US = 3
SC_MS = 4
SC_SEC = 5
```

SC_SET_TIME_RESOLUTION

Время должно быть представлено внутренне как целое кратное временному разрешению. Временное разрешение по умолчанию составляет 1 пикосекунду. Временное разрешение можно изменить только вызовом функции sc_set_time_resolution. Эта функция вызывается только во время разработки, вызывается не более одного раза и не вызывается после создания объекта типа sc_time с ненулевым значением времени. Функция sc_get_time_resolution должна вернуть разрешение по времени.

SC_ZERO_TIME

SC_ZERO_TIME представляет время задержки 0. Хорошей практикой является использование этой константы при записи нулевого значения времени, например, при создании дельта-уведомления или дельта-тайм-аута.

SC_START

Sc_start () является ключевым методом в SystemC. Этот метод запускает фазу моделирования, которая состоит из инициализации и выполнения. Методы sc_start () в разных ситуациях выполняют операции, перечисленные ниже.

- Вызывается в первый раз: sc_start запускает планировщик, который запускается до времени моделирования, переданного в качестве аргумента (если передан аргумент).
- Когда вызывается во второй и последующие моменты, sc_start возобновит работу планировщика с момента, который он достиг в конце предыдущего вызова sc_start. Планировщик будет выполняться за время,

пришедшее в качестве аргумента (если был передан аргумент), относительно текущего времени моделирования.

- Когда в качестве аргумента передается время, планировщик будет выполнять моделирование «до» и, включив фазу уведомляемого времени, которая увеличивает время моделирования, будет действовать до конечного момента времени (рассчитывается путем добавления времени, указанного в качестве аргумента в момент моделирования, когда вызывается функция sc_start).
- Вызывается без аргументов: планировщик будет запускаться до тех пор, пока он не завершится.
- Вызывается с нулевым аргументом времени: планировщик должен запускаться для одного дельта-цикла.

После запуска планировщик будет запущен до тех пор, пока моделирование не завершится, или приложение вызовет функцию `sc_stop`, или произойдет другое исключение. Как только функция `sc_stop` была вызвана, функция sc_start больше не вызывается. Функция `sc_start` может вызываться из функции sc_main, и только из функции `sc_main`.

SC_TIME_STAMP()

Функция sc_time_stamp возвращает текущее время моделирования. Во время разработки и инициализации функция вернет нулевое значение.

5.14. События

Многие мероприятия в реальной системе происходят одновременно или параллельно. SystemC для моделирования процессов использует параллелизм. В большинстве управляемых событиями симуляторов, параллелизм не является истинным параллельным выполнением. На самом деле, параллельное моделирование работает подобно кооперативной многозадачности. Каждый процесс в симуляторе выполняет небольшой кусок кода, а затем добровольно отдает управление, чтобы другие процессы выполнялись в том же моделируемом временном пространстве.

Ядро симулятора отвечает за запуск процессов и управление тем, какой процесс выполняется следующим. Из-за кооперативного характера симулятора, процессы должны приостановливать себя, чтобы позволить выполнение других параллельных процессов.

SystemC в настоящее время предусматривает два основных типа процессов: SC_THREAD процессы и процессы SC_METHOD. Третий тип SC_CTHREAD, имеет незначительные изменения в процессе SC_THREAD и допускает тактирование.

Переключением выпполняемых процессов могут управлять *события(event)*

SystemC использует класс sc_event для моделирования событий. Этот класс позволяет выполнить явный запуск или переключение процессов от событий с помощью метода уведомления.

ОПРЕДЕЛЕНИЕ: Событие в SystemC – это появление sc_event уведомления, которое происходит в один момент времени. Событие не имеет длительность или значения.

Событие – это нечто, происходящее в определённое время. У события нет никакого значения и продолжительности, оно либо произошло, либо ещё нет. В SystemC событие определяется с помощью класса `sc_event`. Процессы могут реагировать на события, но для этого в списке чувствительности процесса должно быть явно указано событие, к которому он будет чувствителен, например,

```
sc_event delay; …
SC_METHOD(do_delay);
sensitive << delay;
```

Событие можно вызвать с помощью метода `notify()`, в качестве параметра может быть указан промежуток времени в формате `sc_time`, через которое событие должно произойти (в таком случае, событие будет запланированным), например,

```
// Событие произойдет немедленно delay.notify();
// Событие произойдет через промежуток времени 0
delay.notify(SC_ZERO_TIME);
// Событие произойдет через промежуток времени 20 нс
(запланированное событие)
delay.notify(20, SC_NS);
```

Отменить вызов запланированного события можно с помощью метода `cancel(): delay.cancel();`

Ниже приведен пример модели логического элемента НЕ7404, задержка срабатывания которого реализовано с помощью события:

```
SC_MODULE(not)
{ sc_in <bool> A;
sc_out <bool> F;
 sc_event delay;
SC_CTOR(not)
{
SC_METHOD(do_delay);
sensitive << A;
SC_METHOD(do_not);
sensitive << delay;
}
void do_delay()
{
delay.notify(22, SC_NS);
}
void do_not(){F.write(!A.read());}
};
```

В данном примере временная задержка реализована с использованием события. Событие `delay`, объявленное в теле модуля происходит тогда,

когда, на входе элемента НЕ меняется входной сигнал (sc_in <bool> A;), метод do_delay чувствителен к изменению сигнала А. Команда delay.notify(22, SC_NS) активизирует событие delay спустя 22 ns, в свою очередь метод do_not, чувствительный к событию delay, реагирует и меняет значение на выходе элемента НЕ.

Для того, чтобы поймать событие, надо наблюдать за ним. SystemC позволяет процессам ждать событие, используя динамическую или статическую чувствительность. Если событие происходит, а никакие процессы не ждут, чтобы поймать его, это событие проходит незамеченным.

Синтаксис для объявления события:

```
sc_event event_name1 [, event namei ]...;
```

Тип события sc_event обеспечивает следующие функциональные возможности.

Конструктор - Объект «события» может быть создан путем вызова конструктора без любых аргументов.

Например:

```
sc_event my_event;
```

Notify – о событие может быть уведомлено путем вызова метода уведомления объекта события. Например:

```
my_event.notify(); // notify immediately
my_event.notify( SC_ZERO_TIME ); // notify next delta
cycle
my_event.notify( 10, SC_NS ); // notify in 10 ns
sc_time t( 10, SC_NS );
my_event.notify( t ); // same
```

Кроме того, функции позволяют иметь функциональную нотацию для уведомления о событии. Например:

```
notify( my_event ); // notify immediately
```

```
notify( SC_ZERO_TIME, my_event ); // notify next delta
cycle
notify( 10, SC_NS, my_event ); // notify in 10 ns
sc_time t( 10, SC_NS );
notify( t, my_event ); // same
```

Cancel - уведомление о событии может быть отменено путем вызова Cancel - метода () объекта события. Например:

```
my_event.cancel(); // cancel a delayed notification
```

Конструктор копирования и оператор присваивания типа события отключены.

Каналами можно построить любое количество объектов событий - по одному для каждого типа события, которое оно может генерировать. Канал может уведомить о событии, вызвав один из методов уведомления объекта о события. Тем не менее, создание и уведомления о событиях не ограничивается каналами.

Event_name.notify () - это функция-член уведомления, которая создает немедленное уведомление. Все экземпляры процесса, чувствительные к событию, должны быть запущены до того, как управление будет возвращено из функции уведомления.

Event_name.cancel () is является функцией-членом, которая удалит все ожидающие уведомления для этого события.

Листинг 5.19

Программа sc_event

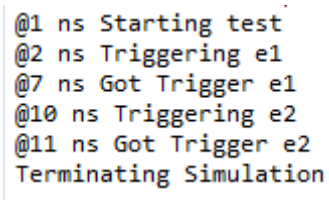
```
#include <systemc.h>

SC_MODULE (events) {
  sc_in<bool> clock;

  sc_event  e1;
```

```
  sc_event  e2;

  void do_test1() {
    while (true) {
      // Wait for posedge of clock
      wait();
      cout << "@" << sc_time_stamp() <<" Starting
test"<<endl;
      // Wait for posedge of clock
      wait();
      cout << "@" << sc_time_stamp() <<" Triggering
e1"<<endl;
      // Trigger event e1
      e1.notify(5,SC_NS);
      // Wait for posedge of clock
      wait();
      // Wait for event e2
      wait(e2);
      cout << "@" << sc_time_stamp() <<" Got Trigger
e2"<<endl;
      // Wait for posedge of clock
      wait();
      cout<<"Terminating Simulation"<<endl;
      sc_stop(); // sc_stop triggers end of simulation
    }
  }

  void do_test2() {
    while (true) {
      // Wait for event e2
```

```
      wait(e1);
      cout << "@" << sc_time_stamp() <<" Got Trigger
e1"<<endl;
      // Wait for 3 posedge of clock
      wait(3);
      cout << "@" << sc_time_stamp() <<" Triggering
e2"<<endl;
      // Trigger event e2
      e2.notify();
    }
  }

  SC_CTOR(events) {
    SC_CTHREAD(do_test1,clock.pos());
    SC_CTHREAD(do_test2,clock.pos());
  }
};

int sc_main (int argc, char* argv[]) {
  sc_clock clock ("my_clock",1,0.5);

  events  object("events");
    object.clock (clock);

  sc_start(0); // First time called will init schedular
  sc_start();  // Run the simulation till sc_stop is
encountered
  return 0;// Terminate simulation
}
```

```
@1 ns Starting test
 @2 ns Triggering e1
 @7 ns Got Trigger e1
 @10 ns Triggering e2
 @11 ns Got Trigger e2
 Terminating Simulation
 SystemC: simulation stopped by user.
```

Результаты моделирования



5.14.1. Функция wait ()

`Wait()` приостанавливает поток или экземпляр процесса с тактированием, из которого она вызывается в очередной раз, когда экземпляр этого процесса будет возобновлен и только в этом случае. Динамическая чувствительность определяется аргументами, переданными в функцию wait (пример: `clock`).

Вызов функции `wait` с пустым списком аргументов или с одним целым аргументом должен использовать статическую чувствительность экземпляра процесса. Это единственная форма ожидания, разрешенная в рамках процесса с тактируемым потоком. Вызов функции `wait` с одним или несколькими нецелыми аргументами должен переопределять статическую чувствительность экземпляра процесса.

При вызове функции `wait` с параметром, переданным по ссылке, приложение должно гарантировать, что время жизни любых фактических аргументов, переданных по ссылке, увеличивается с момента вызова функции до момента завершения вызова функции и, кроме того, в случае параметра типа sc_time, приложение не должно изменять значение фактического аргумента в течение этого периода.

Примечание. Нельзя вызывать функцию wait из процесса метода.

Wait принимает параметр, некоторые из которых перечислены ниже:

Wait (): дождитесь появления события чувствительного списка.

Wait (int): дождитесь появления n событий, события - один в чувствительном списке.

Wait (event): дождитесь события, указанного в качестве параметра.

Wait (double, sc_time_unit): подождите указанное время.

Wait (double, sc_time_unit, event): дождитесь заданного времени или события.

5.14.2. Next_trigger ()

Next_trigger() используется с методами процесса, которые не являются потоками. Функция next_trigger не приостанавливает экземпляр процесса метода: процесс метода не может быть приостановлен, но всегда выполняется до завершения, прежде чем вернуть управление ядру. Этим он отличается от метода wait (), который используется с потоками.

Next_trigger (): при отсутствии статической чувствительности для этого конкретного экземпляра процесса этот процесс не должен снова включаться во время текущего моделирования.

Next_trigger (event): процесс должен запускаться, когда о событии, переданном в качестве аргумента, получено уведомление.

Next_trigger (double, sc_time_unit): процесс должен запускаться по истечении заданного времени.

Next_trigger (double, sc_time_unit, event): Процесс должен быть инициирован, когда получено уведомление о событии или по истечении заданного времени, которое происходит первым.

5.15. Методы

Как упоминалось ранее, SystemC имеет более чем один тип процесса. SC_METHOD процесс более простой, чем SC_THREAD. Однако, эта простота делает его более трудным для некоторых стилей моделирования. По своим возможностям SC_METHOD является более эффективным, чем SC_THREAD. Одним из основных отличий является вызов. В SC_METHOD процессы никогда не приостанавливаются внутренне (то есть, они не могут никогда вызывать wait ()). Вместо этого SC_METHOD процессы выполняются полностью и с возвратом. Симулятор моделирования вызывает их неоднократно на основе динамической или статической чувствительности.

Поскольку процессам SC_METHOD запрещаются приостановления изнутри, они не могут вызвать метод ожидания. Попытка вызова ожидания либо непосредственно, либо косвенно по результатам SC_METHOD приводит к ошибке во время выполнения. Предполагаемые ожидания по результатам вызова других встроенных методов SystemC, определяются таким образом, что они могут выдать wait (). Они известны как блокирующие методы. Методы чтения и записи данных типа `sc_fifo`, обсуждаются далее и являются примерами способов блокировки. Таким образом, SC_METHOD процессы должны избегать использования вызовов методов блокирования.

Синтаксис SC_METHOD процессов следующий и практически идентичен SC_THREAD за исключением ключевого слова SC_METHOD за исключением:

```
SC_METHOD(process_name);//Located INSIDE constructor
```

Замечание о выборе этих ключевых слов может оказаться полезным. Сходство наименования между процессом SC_METHOD и обычным объектно-ориентированного методом передает свое название. Он выполняется без прерывания и возвращается к вызывающему

(планировщику). В противоположность этому, процесс SC_THREAD является более родственным отдельной операционной системе потоков с возможностью прерывания и возобновления. Переменные, выделенные в процессах SC_THREAD, являются неизменными. SC_METHOD процессы должны объявлять и инициализировать переменные каждый раз, когда метод вызывается. По этой причине процессы SC_METHOD обычно используют модуль локальных элементов данных, объявленых в SC_MODULE. SC_THREAD процессы как правило, используют локально объявленные переменные.

5.15.1. Методы sc_start() и sc_stop()

Метод sc_start() запускает фазу моделирования, которая состоит из инициализации и выполнения. Метод может принимать параметр типа sc_time, который является ограничением максимального времени моделирования. Без параметра sc_start() запускает фазу моделирования, которая будет протекать бесконечно, пока в программе не встретится метод sc_stop(), который принудительно завершает моделирование.

5.15.2. Метод wait()

Метод wait() используется в SystemC для того чтобы моделировать задержки реальных действий (например: механических воздействий, химических реакций или распространения сигнала). С помощью метода можно приостановить выполнение процесса SC_THREAD на промежуток времени или до появления какого-либо события:

```
sc_time t_delay(25, SC_FS);wait(t_delay);
```

5.15.3. Метод sc_time_stamp()

Данный метод позволяет определить в момент вызова, сколько времени прошло с начала моделирования, например:

```
sc_time t_delay1(10, SC_NS);
sc_time t_delay2(25, SC_NS);
```

```
cout << “Текущее время моделирования: ” <<
sc_time_stamp();
wait(t_dalay1);
cout << “Текущее время моделирования: ” <<
sc_time_stamp();
 wait(t_delay2);
cout << “Текущее время моделирования: ” <<
sc_time_stamp();
```

В результате исполнения данного проекта, мы получаем на экране следующие сообщения:

Текущее время моделирования: 0 s.

Текущее время моделирования: 10 ns.

Текущее время моделирования: 35 ns

5.16. Динамическая чувствительность для SC_METHOD: next_trigger ()

SC_METHOD процессы позволяют динамически определять их чувствительность с помощью метода next_trigger (). Этот метод имеет тот же синтаксис, что и `wait()` метод, но с несколько иным поведением.

```
next_trigger(time);
next_trigger(event);
next_trigger ;//any of these
next_trigger ;//all of these
//required
next_trigger(timeout, event); //event with timeout
next_trigger(timeout, ;//any + timeout
next_trigger(timeout, ; //all + timeout
next_trigger() ; //re-establish static sensitivity
```

Как и в случае ожидания, несколько синтаксисов событий не уточняют порядок событий. Так, с `next_trigger (evt1 & evt2)`, не представляется возможным узнать, какое событие произошло в первую очередь. Можно лишь утверждать, что оба `evt1` и `evt2` случилось.

Метод `wait` приостанавливает процессы SC_THREAD. Однако, процессы SC_METHOD не могут приостанавливаться. Метод `next_trigger` эффективен для временной настройки списка чувствительности, который влияет на SC_METHOD. Метод `next_trigger` можно вызывать несколько раз, и каждый новый вызов переопределяет предыдущие. Последний выполненный `next_trigger` перед возвращением из процесса определяет чувствительность для повторного вызова процесса. Вызов инициализации имеет жизненно важное значение для выполнения работы.

Следует отметить, что для каждого пути через SC_METHOD необходимо указать по крайней мере один `next_trigger` для того, чтобы процесс был вызван планировщиком. Без создание `next_trigger` или статической чувствительности SC_METHOD никогда не будет выполняться снова. Предостережение может относится к размещению `next_trigger` по умолчанию в качестве первой команды SC_METHOD, так как последующие `next_triggers` будут переписывать все предыдущие. Лучший способ справиться с этой проблемой существует и состоит в следующем.

5.17. Статическая чувствительность для процессов

Мы обсудили технику динамического (т.е. во время моделирование), определения, как возобновится процесс (либо как SC_THREAD, используя ожидание, или с помощью SC_METHOD, используя next_trigger). SystemC обеспечивает другой тип чувствительности для процессов, который называется статической чувствительностью. Статическая чувствительность устанавливает параметры для возобновления в процессе разработки (то есть, перед началом моделирования). После создания статические параметры

чувствительности не могут быть изменены (т.е. они статические). Как мы увидим, можно переопределить статическую чувствительность,.

Статическая чувствительность устанавливается с помощью вызова чувствительного метода sensitive() или перезагрузки оператора потока `operator <<`, который помещен сразу после регистрация процесса. Статическая чувствительность применяет только самую последнюю регистрацию процесса. Чувствительность `sensetive` может быть указана несколько раз. Есть два стили синтаксиса:

```
// IMPORTANT: Must follow process registration
sensitive << event [<< event] ;// streaming style
sensitive (event [, event] ); // functional style
```

Мы предпочитаем стиль потоковой передачи, так как он чувствует себя более объектно-ориентированным, уменьшает набор операторов и держит нас сосредоточеными на C ++.

Иногда возникает необходимость уточнить некоторые процессы, которые не следует инициализировать. Этой ситуацию в SystemC обеспечивает метод dont_initialize. Синтаксис имеет следующий вид:

```
// IMPORTANT: Must follow process registration
dont_initialize();
```

Использование `dont_initialize` требует списка статической чувствительности. В противном случае, не было бы ничего, чтобы начать процесс. Теперь наш модуль содержит:

```
SC_METHOD(attendant_method);
sensitive(event [, event);
dont_initialize() ;
```

В свете ограничения, что `sc_events` может иметь только одно невыполненное расписание, `sc_event_queues` были добавлены в SystemC версия 2.1. Эти дополнения позволяют одно событие запланировать

несколько раз даже в одно и то же время! Когда события запланированы на то же время, каждое происходит в отдельном дельта - цикле.

sc_event_queue немного отличается от sc_event. Во-первых, sc_event_queue объекты не поддерживают немедленное уведомление и, очевидно, нет необходимости создавать их очередь. Во-вторых, метод .cancel () заменяется .cancel_all (), чтобы подчеркнуть, что он отменяет все внешние sc_event_queue уведомления.

```
sc_event_queue action;
sc_time now(sc_time_stamp()); /*observe current time*/
action.notify(20, SC_MS);/*schedule for 20 ms from now*/
action.notify(1.5,SC_NS);/*another for 1.5 ns from now*/
action.notify(1.5,SC_NS);/*another identical action*/
action.notify(3.0,SC_NS);/*another for 3.0 ns from now*/
action.notify(SC_ZERO_TIME);/*for next delta cycle*/
action.notify(1,SC_SEC);//for 1 sec from now
action.cancel_all(); /* cancel all actions entirely*/
```

Метод .cancel () в настоящее время не реализуется; хотя, очевидно, расширение может быть использовано, чтобы отменить уведомления в определенное время. Еще одним расширением может быть получение информации о том, сколько невыполненных уведомлений существует (.pending ())

5.18. Типы данных и операторы

В таблице 5.1 представлены типы данных, поддерживаемые SystemC. В таблице 5.2 представлены операции над данными, поддерживаемые SystemC.

Таблица 5.1

Основные типы данных, поддерживаемые SystemC

Тип данных	Описание
sc_bit	Одиночный бит, принимающий значение true или false. Использовать данных тип не

	рекомендуется, более предпочтительно применение типа bool
sc_bv< n>	Вектор, содержащий n бит. Рекомендуется использовать sc_uint <n>, где это возможно
sc_logic	Одиночный бит, который принимает значения 0, 1, X, Z
sc_lv<n>	Вектор, содержащий n бит, типа sc_logic
sc_int<n>	Вектор, содержащий n целых чисел, размером 64 бит
sc_uint< n>	Беззнаковое sc_int <n>
sc_bigint<n>	Вектор, содержащий n целых чисел, размером более 64 бит
sc_biguint< n>	Беззнаковое sc_bigint <n>

К поддерживаемым типам данных также относятся другие типы данных C++, такие как bool, int, unsigned int, long, unsigned long, char, unsigned char, short, unsigned short, struct, enum.

Таблица 3.2.

Операции над данными, поддерживаемые SystemC

Операции	Описание
&(and), |(or), ^(xor), and ~(not)	Логические операции
<<(shift left) and >>(shift right)	Логические сдвиги
=, &=, |=, ^=, +=, -=, *=, /=, and %=	Префиксные операции
==, !=	Равенство

<, <=, >, >=	Сравнение
++ и --	Инкремент и декремент
[x]	Индексирование
(x-y)	Беззнаковое sc_bigint <n>
(x,y)	Конкатенация
to_uint( ) и to_int( )	Преобразование типов

Перечисленные операции над данными в основном совпадают с использоваанными в C++.

5.19. Каналы и интерфейсы

SystemC имеет встроенные механизмы, известные как каналы, для выполнения коммуникации и инкапсуляция в комплексе связей. SystemC имеет два типа каналов: примитивный и иерархический.

Примитивные каналы SystemC называются примитивными, потому что они не содержат никакой иерархии, не имеют процессов и предназначены быть очень быстрыми благодаря их простоте. Все примитивные каналы наследуются от базового класса `sc_prim_channel`.

SystemC содержит несколько примитивных каналов. Простейшие из них: `sc_mutex, sc_semaphore, sc_fifo.`

5.19.1. Канал `sc_mutex`

`mutex` является сокращением для объекта взаимного исключения. В компьютерном программировании `mutex` - программный объект, который позволяет нескольким программным потокам совместно использовать общий ресурс, такой как доступ к файлу, без конфликтов.

Во время разработки создается `mutex` с уникальным именем; впоследствии, любой процесс, который нуждается в ресурсе, должен

блокировать mutex, чтобы предотвратить использования общего ресурса другими процессами. Этот процесс должен отключать mutex, когда ресурс больше не нужен. Если другой процесс пытается получить доступ к заблокированному mutex, этот процесс преостанавливается до тех пор, пока mutex не станет доступным (разблокированным).

SystemC обеспечивает mutex через канал sc_mutex. Этот класс содержит несколько методов доступа, включая как заблокированные, так и разблокированные стили. Методы блокировки могут использоваться только в процессах SC_THREAD.

```
sc_mutex NAME;
// To lock the mutex NAME (wait until)
//// unlocked if in use)
NAME.lock();
/* Non-blocking, returns true if success, else false*/
NAME.try lock()
// To free a previously locked mutex
NAME.unlock();
```

Sc_mutex - это предопределенный примитивный канал, предназначенный для моделирования поведения блокировки взаимного исключения, используемого для управления доступом к ресурсу, совместно используемому параллельными процессами. mutex может быть в одном из двух эксклюзивных состояний: разблокирован или заблокирован. Только один процесс может заблокировать данный mutex за один раз. mutex может быть разблокирован только определенным процессом, который заблокировал его, но впоследствии может быть заблокирован другим процессом.

Класс sc_mutex поставляется с предопределенными методами, как показано ниже.

Int lock (): заблокируйте mutex, если он свободен, иначе подождите, пока mutex освободится.

Int unlock (): разблокировать mutex.

Int trylock (): проверить, свободен ли mutex, если свободен, то блокировать его и еще вернуть -1.

Char * kind (): Возвращает строку "sc_mutex"

5.19.2. Канал sc_semaphore

Для некоторых ресурсов может быть более одной копии или владельца. Для управления этим типом ресурсов SystemC предоставляет sc_semaphore. При создании объекта sc_semphore необходимо указать, сколько копий доступны. В некотором смысле mutex - это просто семафор со счетом один.

Доступ sc_semaphore состоит из ожидания доступного ресурса и затем отправки уведомления после завершения работы с ресурсом.

Важно понимать, что sc_semaphore ::wait () является явно отличающиймся методом от метода wait (), рассмотренного ранее совместно с SC_THREAD. На самом деле sc_semaphore ::wait () реализуется с помощью wait (event).

Sc_semaphore - предопределенный примитивный канал, предназначенный для моделирования поведения программного семафора, который используется для обеспечения ограниченного параллельного доступа к совместно используемому ресурсу. Семафор имеет целочисленное значение, которое устанавливается на допустимое количество одновременных обращений при создании семафора.

Sc_semaphore имеет следующие предопределенные методы.

Int wait (): если значение семафора равно 0, функция-член wait приостанавливает работу до тех пор, пока значение семафора не будет

увеличено (другим процессом), после чего работа должна возобновиться и попытаться уменьшить значение семафора.

`Int trywait ()`: если значение семафора равно 0, функция-член `trywait` должна немедленно вернуть значение -1 без изменения значения семафора.

`Int post ()`: сообщение функции-члена увеличивает значение семафора. Если существуют процессы, которые приостановлены и ждут увеличения значения семафора, то только одному из этих процессов будет разрешено уменьшать значение семафора (выбор процесса является недетерминированным), в то время как остальные процессы должны снова приостанавливаться.

`Int get_value ()`: функция-член `get_value` должна возвращать значение семафора.

`Char * kind ()`: возвращает строку "sc_semaphore"

5.19.3. Канал sc_fifo

Самый популярный канал для моделирования на архитектурном уровне это `sc_fifo`. Очереди «первым пришел-первым-вышел» (то есть FIFO) являются общими данными. Структура, используемая для управления потоком данных. FIFO являются одной из самых простых структур.

По умолчанию `sc_fifo <>` имеет глубину 16. Тип данных элементов также необходимо указать. `Sc_fifo` может содержат любой тип данных, включая большие и сложные структуры (например, TCP/IP пакет или блок диска).

Синтаксис:

```
sc_fifo<ELEMENT_TYPENAME> NAME(SIZE);
NAME.write (VALUE);
NAME.read (REFERENCE) ;
= NAME.read () /* function style */
if (NAME.nb_read (REFERENCE)) { // Non-blocking
```

```
// true if success
}
if (NAME.num_available() == 0)
wait(NAME.data_written_event());
if (NAME.num_free() == 0)
next_trigger(NAME.data_read_event());
```

Например, FIFO могут использоваться для буферизации данных между процессором изображений и шиной, или система связи может использовать FIFO для буферизации информационных пакетов при прохождении через сеть.

Sc_fifo - предопределенный примитивный канал, предназначенный для моделирования поведения fifo, то есть буфера first-in first-out. Fifo - это объект класса sc_fifo. Каждый fifo имеет несколько слотов для хранения значений. Количество слотов фиксируется при создании объекта. Размер слотов по умолчанию - 16.

Sc_fifo имеет следующие предопределенные методы.

Write (): этот метод записывает значения, переданные в качестве аргумента в fifo. Если fifo полный, то функция write () ожидает, пока не будет доступен слот fifo

Nb_write (): этот метод такой же, как write (); разница только в том, что когда fifo заполнен, функция nb_write () не ждет, пока не будет доступен слот fifo. Сразу возвращает false.

Read (): Этот метод возвращает самые последние записанные данные в fifo. Если fifo пуст, функция read () ожидает, пока данные не будут доступны в fifo.

Nb_read (): Этот метод аналогичен read (), только разница в том, что когда fifo пуст, функция nb_read () не ждет, пока fifo не получит некоторые данные. Сразу возвращает false.

`Num_available ()`: Этот метод возвращает числа значений данных, доступных в fifo в текущем дельта-времени.

`Num_free ()`: Этот метод возвращает количество свободных слотов, доступных в fifo в текущем дельта-времени.

5.20. Иерархические каналы

Каналы бывают двух типов: примитивные и иерархические. Основное место канала - это класс, который наследуется от интерфейса. Интерфейс делает канал удобным для использования с портами. Кроме того, каналы должны наследовать либо от `sc_prim_channel` или `sc_channel`. Это различие в последних двух базовых классах - есть одна из отличительных возможностей и функций. Другими словами, `Sc_prim_channel` имеет возможности, отсутствующие в `sc_channel` и наоборот.

Примитивные каналы предназначены для обеспечения очень простой и быстрой связи. Они не содержат иерархии, портов и SC_METHODs или SC_THREAD. Примитивные каналы имеют возможность реализовать парадигму оценки-обновления. Напротив, иерархические каналы могут обращаться к портам, они могут иметь процессы и содержать иерархию, как следует из названия. Фактически, иерархические каналы действительно являются просто модулями, реализующими один или несколько интерфейсов. Иерархические каналы предназначены для моделирования сложных коммуникационных шин таких как PCI, HyperTransport ™ или AMBA ™. Такие каналы требуют дополнительного изучения.

5.21. Моделирование уровня транзакций

Моделирование уровня транзакций - это высокоуровневый подход к моделированию цифровых систем, где детали взаимодействия между модулями отделены от деталей реализации функциональных блоков или архитектуры связи. Механизмы связи, такие как шины или FIFO, моделируются как каналы и представляются модулями с использованием классов интерфейса SystemC. Запросы транзакций выполняются

посредством вызова функций интерфейса этих моделей каналов, которые инкапсулируют детали низкого уровня обмена информацией.

Моделирование на уровне транзакций также обеспечивает более высокую скорость моделирования, чем интерфейсы на основе контактов. Модели уровня транзакций могут использоваться в любое время, когда их повышенный уровень абстракции выгоден.

Моделирование уровня транзакций (TLM) выдвигается как перспективное решение выше уровня передачи регистров (RTL) в проектном потоке SoC.

Моделирование уровня трансзакций мотивировано также по следующим причинам:

- Предоставление ранней платформы для разработки программного обеспечения.
- Развертывание и проверка дизайна системного уровня.
- Необходимость использования моделей системного уровня для проверки уровня блоков.

5.22. Моделирование и отладка с помощью SystemC

После того, как вы сделали описание системы в SystemC, как правило, вы хотите моделировать его в качестве следующего шага в процессе разработки. Здесь описывается возможности управления моделированием, предоставляемые SystemC для запуска и остановки моделирования, запроса текущего время, и понимания порядка, в котором выполняются различные процессы.

Описание системы в SystemC дает преимущество использования стандартных средства разработки C ++ для компиляции и отладки. Далее описывается дополнительные средства, которые могут помочь вам в отладке SystemC программ.

5.23. Планировщик SystemC

Моделирования в SystemC основно на циклах: выполняемые процессы и сигналы обновляются на тактовых переходах

. Библиотека SystemC включает в себя планировщик на основе цикла, который обрабатывает все события на сигналах и планирует процессы, когда соответствующие события происходят на их входах. Моделирование в SystemC следует парадигме оценки-обновления, где все процессы, которые готовы к исполнению выполняются, и только тогда их выходные сигналы обновляются.

Планировщик в SystemC выполняет следующие шаги в процессе моделирования.

1. Всем сигналам синхронизации, которые меняют свое значение в текущий момент времени, назначаются их новые значения.

2. Все SC_METHOD / SC_THREAD процессы с входами, которые изменились, выполняются. Все тело функции процесса SC_METHOD выполняется, в то время как SC_THREAD процессы не выполняются пока следующий оператор `wait()` приостанавливает выполнение процесса. SC_METHOD процессы / SC_THREAD не являются выполняемыми в определенном порядке.

3. Все SC_CTHREAD процессы, которые запускаются, имеют свои обновленные выходы, и они сохраняются в очереди, которая будет выполнена позже на шаге 5. Все выходы SC_METHOD / SC_THREAD процессов, которые были выполнены на шаге 1 также обновляются.

4. Шаги 2 и 3 повторяются до тех пор, пока сигнал не меняет свое значение.

5. Все SC_CTHREAD процессы, которые были запущены и поставлены в очередь на шаге 3, выполняются. Там нет фиксированного порядка выполнения этих процессов. Их выходы обновляются при следующем активном фронте сигнала (когда шаг 3 выполняется), и, следовательно, сохраняются внутри.

6. Время моделирования продвигается к следующему фронту тактового импульса и планировщик возвращается к шагу 1.

Если процессы взаимодействуют с помощью сигналов, порядок выполнения процесса не должен влиять на результаты моделирования. Тем не менее, если используются глобальные переменные и указатели, порядок выполнения процесса влияет на результаты моделирования. Обратите внимание, что эта семантика моделирования подобна Verilog семантике моделирования с отложенными заданиями сигналов и VHDL семантике моделирования.

5.24. Контроль моделирования

Вы можете начать моделирование только после того, как вы реализуете и правильно соедините все модули и сигналы. В SystemC моделирование начинается с вызова sc_start () с верхнего уровня, а именно из подпрограммы sc_main (). Функция sc_start () принимает переменную двойного типа качестве аргумента и моделирует систему столько стандартных единиц времени, каково значение переменной. Если вы хотите продолжать моделирование до бесконечности, для этого обеспечивают отрицательное значение для аргумента этой функции. Эта процедура генерирует все тактовые сигналы в соответствующие моменты времени и вызывает планировщик SystemC.

Моделирование можно остановить в любое время (из любого процесса) путем вызова sc_stop (). Функция не принимает аргументов.

Вы можете определить текущее время в процессе моделирования с помощью вызова `sc_simulation_time ()`. Эта функция возвращает текущее время моделирования в переменной двойного типа.

Чтобы помочь в отладке в процессе моделирования, переменные, порты и значения сигнала можно читать и распечатать. Печатное значение порта или сигнала является текущим значением порта или сигнала, а не просто значением, записанным в него.

5.24.1. Расширенные методы техники контроля моделирования

Есть возможность использовать другой метод для тактирования и моделирования управления, чем использование `sc_start ()`. Чтобы сделать это, вы должны сначала вызвать `sc_initialize()` для инициализации SystemC планировщика. После этого вы можете установить значение сигналов, путем записи в них, и вызовом процедуры `sc_cycle ()`, чтобы имитировать результат установки сигналов. Эта функция принимает переменную двойного типа в качестве аргумента. Она вызывает планировщик SystemC, моделирует, пока текущие эффекты записи сигнала не распространяются по всей системе. Затем она продвигает время моделирования на величину, заданную как аргумент функции. Например, если время по умолчанию в модуле будет 1 нс, sc_cycle (10) продвигает время моделирования на 10 нс.

Для примера, предположим, что вы определили такт, как:

```
sc_clock clk("my clock", 20, 0.5);
```

Вы можете смоделировать генерацию тактов для единиц времени 200, вызвав по умолчанию:

```
sc_start(200);
```

В последних версиях SystemC потребуется указать разметность времени.

С другой стороны, вы можете создать тактирование самостоятельно, выполнив следующие действия:

```
sc_signal<bool> clock;
sc_initialize();
for (int i = 0; i <= 200; i++)
clock = 1;
sc_cycle(10);
clock = 0;
sc_cycle(10);
}
```

Используя эту возможность, можно вводить события асинхронно по отношению к такту в систему, как показано на следующем рис. 3.22.

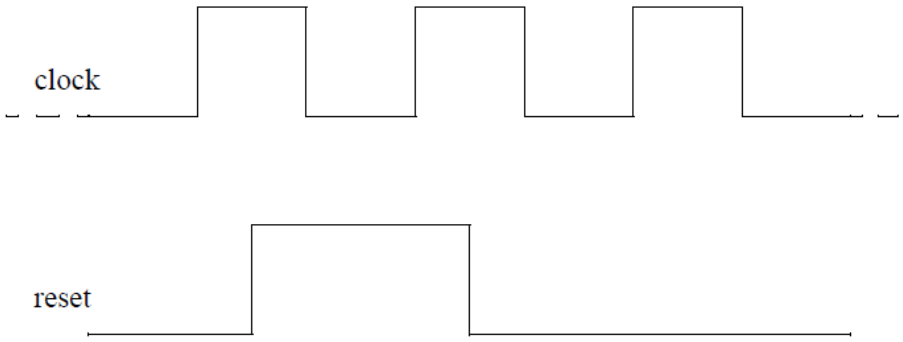


Рис. 5.22.

Для реализации этого, вы можете написать в sc_main ():

```
sc_initialize();
// Let the clock run for 10 cycles
for (int i = 0; i <= 200; i++)
clock = 1;
sc_cycle(10);
clock = 0;
sc_cycle(10);
}
// Inject asynchronous reset
clock = 1;
sc_cycle(5);
reset = 1;
sc_cycle(5);
clock = 0;
sc_cycle(10);
clock = 1;
sc_cycle(5);
reset = 0;
```

```
sc_cycle(5);
clock = 0;
sc_cycle(10);
// Now let the clock run indefinitely
for (;;)
clock = 1;
sc_cycle(10);
clock = 0;
sc_cycle(10);
}
```

Обратите внимание, что sc_cycle () может быть вызван только из верхнего уровня, подобного sc_start ().

5.25. Трассировка осциллограмм

SystemC предоставляет функции, позволяющие создать файлы VCD (Value Change Dump), ASCII WIF (Waveform Intermediate Format), или ISDB (Integrated Data Signal Base), которые содержат значения переменных и сигналов и показывают, как они изменяются во время моделирования. Формы волны, определенные в этих файлах можно просмотреть, используя стандартные средства просмотра осциллограмм, которые поддерживают форматы VCD, WIF или ISDB.

При генерации форм волны, обратите внимание на следующее:

• Можно наблюдать только переменные, которые находятся в области видимости во время всего моделирования. Это означает, что все сигналы и элементы данных модулей можно проследить. Локальные переменные функции не могут быть прослежены.

• Переменные и скалярные сигналы, массивы и агрегатные типы можно наблюдать.

• Различные типы файлов трассировки могут быть созданы в течение того же выполнения моделирования.

• Сигнал или переменная могут быть прослежены любое количество раз в различных форматах трассировки.

5.25.1. Создание файла трассировки

Первый шаг в отслеживании сигналов создает файл трассировки. Файл трассировки, как правило, создают на высшем уровне после того, как все модули и сигналы были смоделированы. Для отслеживание формы сигналов, используя формат VCD, файл трассировки создается с помощью вызова `sc_create_vcd_trace_file ()` с именем файла в качестве аргумента. Эта функция возвращает указатель на структуру данных, которая используется во время трассировки. Например,

```
sc_trace_file * my_trace_file;
my_trace_file = sc_create_vcd_trace_file(“my_trace”);
```

создает файл с именем VCD `my_trace.vcd` (.vcd расширение автоматически добавлено). Указатель на структуру данных файла трассировки возвращается. Вы должны хранить этот указатель, поэтому он может быть использован в вызовах подпрограмм трассировки.

Чтобы создать файл WIF, должна быть вызвана функция sc_create_wif_trace_file (). Для примера:

```
sc_trace_file *trace_file;
my_trace_file = sc_create_wif_file(“my_trace”);
```

создает файл с именем WIF `my_trace.awif` (.awif расширение автоматически добавлено). Аналогичным образом, может быть создан файл трассировки ISDB.

В конце моделирования файлы трассировки должны быть закрыты, иначе могут возникнуть ошибки. Закрыть файлы трассировки можно одной из следующих функций.

```
sc_close_isdb_trace_file(my_trace_file);
sc_close_wif_trace_file(my_trace_file);
sc_close_vcd_trace_file(my_trace_file);
```

Вызывать функции надо в соответствии с типом файла, который был создан. Вызывайте эту функцию как раз перед оператором возврата в вашей обычной sc_main.

5.25.2. Трассировка скалярной переменной и сигналов

SystemC обеспечивает трассировку функции для скалярных переменных и сигналов. Все трассировки функции имеют следующие общие черты:

• Функция называется sc_trace ().

• Их первый аргумент является указателем на файл трассировки структуры данных sc_trace_file.

• Их второй аргумент является ссылкой или указателем на переменную трассировки.

• Их третий аргумент является ссылкой на строку.

Например, далее показано, как трассируются сигнал типа int и переменная плавающего типа .

```
sc_signal<int> a;
float b;
sc_trace(trace_file, a, “MyA”);
sc_trace(trace_file, b, “B”);
```

В этом примере, trace_file является указателем типа sc_trace_file, который был создан ранее. "MyA"-это имя переменной int, как она будет отображаться в окне просмотра формы сигнала, и "B" - это имя переменной с плавающей точкой.

В регистрах функций трассировки создается список сигналов и переменных, которые будут прослежены. Фактическая трассировка происходит во время моделирования и обрабатывается планировщиком SystemC. Обратите внимание, что вызовы sc_trace () функции

выполняются только после того, как процессы и сигналы инстанцируются и после того, как файл трассировки открыт.

5.25.3. Трассировка переменных и сигналов совокупного типа

Функции трассировки, определенные в SystemC, могут принимать только сигналы или переменные скалярного типы. Для того, чтобы отслеживать переменные совокупного типа, нужно определить специальные функции трассировки для переменных этих типов, используя основные функции трассировки, которые обеспечиваются в SystemC.

Например, рассмотрим структуру

```
struct bus {
unsigned address;
bool read_write;
unsigned data;
};
```

Вам нужно определить функцию трассировки для этой структуры следующим образом:

```
void sc_trace(sc_trace_file *tf, const bus& v, const
sc_string& NAME)
{
sc_trace(tf, v.address, NAME + ".address");
sc_trace(tf, v.read_write, NAME + ".rw");
sc_trace(tf, v.data, NAME + ".data");
}
```

При вызове эта функция трассировки отслеживает структуру данных путем отслеживания отдельных полей структуры. Следует отметить, что каждому отдельному полю структуры присваивается уникальное имя, добавив имя поля к имени структуры.

5.25.5. Трассировка переменных и массивов сигналов

Для того, чтобы проследить переменную или массив сигнала, вам нужно определить специальную функцию трассировки, используя основные функции данных или трассировки сигнала, которые обеспечивает SystemC. Так, например, функцией трассировки для массивов типа `sc_signal <int>` являются

```
void sc_trace(sc_trace_file *tf, sc_signal<int> *v,const
sc_string& NAME, int len)
{
char stbuf[20];
for (int i = 0; i< len; i++) {
sprintf(stbuf, "[%d]", i);
sc_trace(tf, v[i], NAME + stbuf);
}
}
```

Эта функция трассировки имеет один дополнительный аргумент, которым является длина отслеживаемого массива.

SystemC имеет предопределенные векторные функции для векторных типов, определенных в SystemC.

`(sc_int<>, sc_uinit<>, sc_biginit<>, sc_bigunit<>, sc_lv<>, and so forth).`

5.25.5. Отладка SystemC

Поскольку каждый поток или тактированный процесс генерирует новый поток выполнения, отладка моделирования может быть более сложной, чем в типичной линейно исполняемой программе C ++. Потоки выполнения в моделировании означают, что моделирование происходит в нелинейной моде. Может быть трудно определить код, который будет выполняется следующим.

Вы можете отлаживать только код, а не библиотеки классов SystemC. Простейший способ отладки дизайна состоит в том, чтобы поместить

контрольную точку в начале процесса, который требует отладки. Когда моделирование останавливается на одной из этих точек останова, моделирование остановится, и вы можете отлаживать соответствующий процесс по мере необходимости.

Глава 6. Практическое программирование в SystemC

6.1. Введение

При освоении языков программирования весьма полезно детально изучать листинги образцов реальных отлаженных программ. Отдельные фрагменты можно будет с некоторой редакцией использовать при самостоятельном составлении новых программ. Особенно полезно проверить работоспособность образцов, выполнив их загрузку в среду разработки, компиляцию и решение.

В этой главе решения примеров мы будем проводить в среде разработки Eclipse IDE, а точнее в Eclipce CDT для C/C++с компилятором Cygwin. Перед началом работы необходимо выполнить компиляцию библиотек SystemC, провести настройки Eclipse для работы с этими библиотеками. В главе 2 приведены сведения о том, как это можно сделать.

При написании этой главы использовано большое количество материалов из различных зарубежных источников в виде книг, учебных материалов и слайдов, которые автор смог найти в Интернете. Все примеры программ опробованы и подтвердили правильность. Наиболее полезные источники перечислены в библиографии.

6.2. Два основных стиля

SystemC использует два основных шаблона для программирования проектов.

Первый - это более традиционный стиль, который в значительной степени опирается на заголовки.

Второй рекомендуемый стиль добавляет больше элементов в реализацию. Создание шаблонного модуля C ++ обычно исключает этот стиль из-за ограничений компилятора C ++.

Можно использовать любой из этих шаблонов для Вашего програмирования.

6.3. Традиционный шаблон

Традиционный шаблон, показанный на листингах 4.1 и 4.2, размещает все определения для создания примера и конструктора в файлах заголовка (.h). Только реализация процессов и вспомогательных функций откладывается до создания скомпилированного файла (.cpp). Напомним основные компоненты в каждом файле.

Во-первых, `# ifndef / # define / # endif` предотвращает проблемы, когда заголовочный файл включается несколько раз. Использование определения NAME_H вполне стандартно. Это определение сопровождается включением заголовочных файлов любого подмодуля с помощью `#include`.

Затем SC_MODULE {...}; охватывает определение класса. Не забывайте конечную точку с запятой, что является довольно распространенной ошибкой. В пределах определения класса порты обычно объявляются первым, потому что они представляют интерфейс к модулю. Затем следуют локальные каналы и подмодули.

Затем мы помещаем конструктор класса и, необязательно, деструктор. Для большинства случаев для этого достаточно макроса SC_CTOR () {...}. Тело конструктора обеспечивает инициализацию, связность подмодулей, и регистрации процессов. Все это будет подробно обсуждаться в примерах.

Заголовок заканчивается объявлениями процессов, помощником функций и, возможно, другими частными данными. Обратите внимание, что C ++ или SystemC не диктуют порядок этих элементов в объявлении класса.

Листинг 6.1

Традиционный стиль шаблона NAME.h

```
#ifndef NAME_H
#define NAME_H
#include "submodule.h"
SC_MODULE(NAME) {
Port declarations
Channel/submodule instances

SC_CTOR(NAME)
: Initializations
{
Connectivity
Process registrations
}
Process declarations
Helper declarations
};
#endif
```

Тело файла реализации традиционного стиля просто включает в себя заголовочный файл SystemC, и соответствующий модуль заголовка, только что описанный. Остальная часть этого файла просто содержит внешние реализации членов функций и процессов, которые будут также описаны в примерах. Обратите внимание, что возможно отсутствие файла реализации, если в модуле нет процессов или помощника функций.

Листинг 6.2

Традиционный стиль шаблона NAME.cpp

```
#include <systemc.h>
#include "NAME.h"
```

```
NAME::Process {implementations }
NAME::Helper {implementations }
```

6.4. Рекомендуемая алтернативная форма шаблона

По разным причинам рекомендуют другой подход, который в большей степени способствует независимому развитию модулей. Эти шаблоны представлены на листингах 4.3 и 4.4 и обратите внимание на различия.

Во-первых, заголовок содержит те же `#define` и `SC_MODULE` компоненты как традиционный стиль. Различия заключаются в том, как определяются каналы / подмодули и в перемещении конструктора в тело реализации. Обратите внимание, что канал / подмодули реализуются по-разному (с использованием указателей).

Листинг 6.3

Рекомендуемый стиль шаблона NAME.h

```
#ifndef NAME_H
#define NAME_H

Submodule forward class declarations
SC_MODULE(NAME) {
Port declarations
Channel/Submodule* definitions
// Constructor declaration:
SC_CTOR(NAME) ;
Process declarations
Helper declarations
};
#endif
```

Листинг 6.4

Рекомендуемый стиль шаблона NAME.cpp

```
#include <systemc.h>
#include "NAME.h"
NAME::NAME(sc_module_name nm)
: sc_module(nm)
, Initializations
{
Channel allocations
Submodule allocations
Connectivity
Process registrations
}
NAME::Process {implementations }
NAME::Helper {implementations }
```

6.5. Описание библиотек SystemC

Библиотека классов SystemC была разработана для поддержки проектирования на системном уровне. Она работает как на компьютерах, так и на платформах UNIX, и свободно загружается из Интернета.

Библиотека классов выпускается поэтапно. Первый этап - выпуск 1.0 (в настоящее время в версии 1.0.2) предоставляет все необходимые средства моделирования для описания систем, подобных тем, которые могут быть описаны с использованием языка описания аппаратных средств, такого как VHDL. В версии 1.0 представлено ядро моделирования, типы данных, подходящие для арифметики с фиксированной точкой, каналы связи, которые ведут себя как части провода (сигналы), и модули для разбивки конструкции на более мелкие части.

В версии 2.0 (в настоящее время мы используем версию 2.3.1) библиотека классов была сильно переписана, чтобы обеспечить путь

обновления до истинного дизайна системного уровня. Функции, встроенные в версию 1.0, такие как сигналы, теперь построены на базовой структуре каналов, интерфейсов и портов. События были представлены в качестве примитивного средства запуска поведения вместе с набором примитивных каналов, таких как FIFO и MUTEX. Версия 2.0 позволяет добиться гораздо более мощного моделирования путем моделирования на уровне транзакций.

В будущем версия 3.0 библиотеки классов будет расширена, чтобы охватить моделирование операционных систем, чтобы поддержать разработку моделей встроенного программного обеспечения. Также возможно предоставить дополнительные библиотеки для поддержки определенной методологии проектирования. Примерами этого являются библиотека связи «Ведущий-ведомый» и SystemC Verification Library (SCV).

Библиотеки SystemC были разработана группой компаний, входящих в Open SystemC Initiative (OSCI).

Можно описать простой флип-флоп (рис. 6.1) , листинг 6.5 и описать сложный дизайн. SystemC - один из доступных в отрасли языков моделирования. SystemC позволяет нам проектировать цифровые устройства на очень высоком уровне. SystemC позволяет разработчикам аппаратных средств выражать свои проекты с помощью поведенческих конструкций, отражая детали реализации на более поздней стадии проектирования в конечном проекте. С добавлением SystemC Verification, тот же язык может использоваться для проектирования, а также и для проверки.

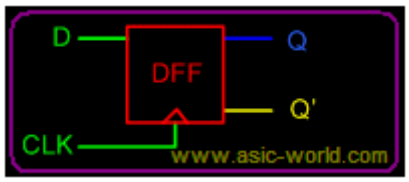


Рис. 6.1

Листинг 6.5

```
// D-FF Code
```

```
#include "systemc.h"

SC_MODULE(d_ff) {
  sc_in<bool> din;
  sc_in<bool> clock;
  sc_out<bool> dout;

  void doit() {
    dout = din;
  };

  SC_CTOR(d_ff) {
    SC_METHOD(doit);
    sensitive_pos << clock;
  }
};
```

6.6. Еще одно приветствие в SystemC

Любая книга по языку программирования обычно начинается с «Привет мир». Рассмотрим такую программу «hello world» в SystemC (листинг 6.6), решение (рис. 6.2).

Листинг 6.6

```
// All systemc modules should include systemc.h header
file
#include "systemc.h"
// Hello_world is module name
SC_MODULE (hello_world) {
  SC_CTOR (hello_world) {
    // Nothing in constructor
  }
```

```
  void say_hello() {
    //Print "Hello World" to the console.
    cout << "Hello World.\n";
  }
};

// sc_main in top level function like in C++ main
int sc_main(int argc, char* argv[]) {
  hello_world hello("HELLO");
  // Print the hello world
  hello.say_hello();
  return(0);
}
```

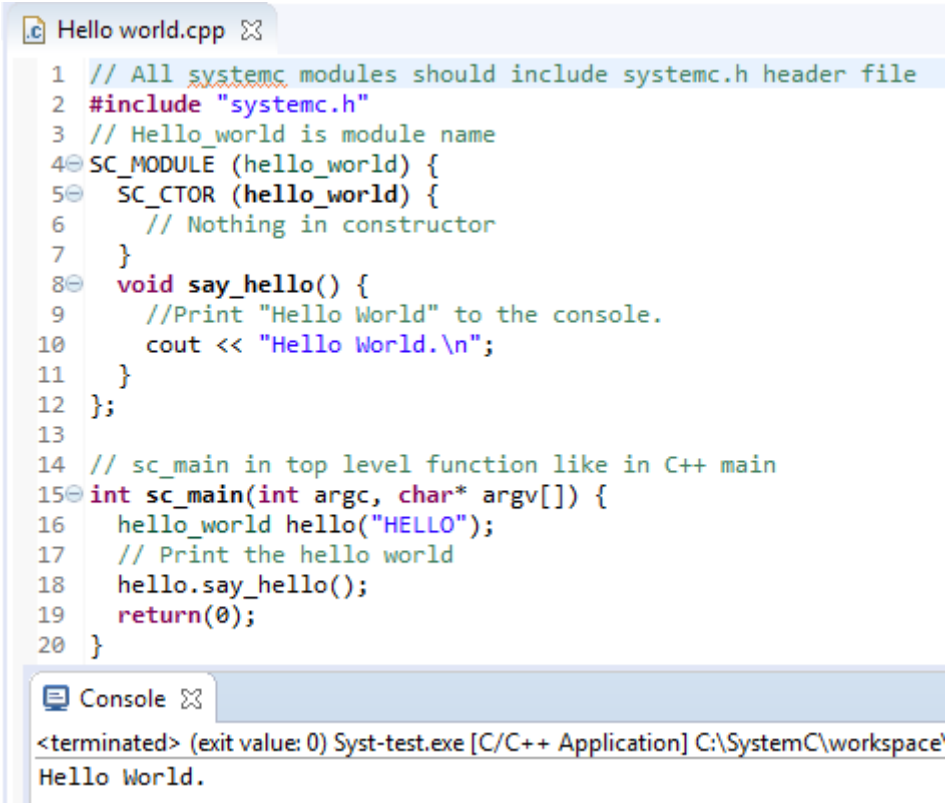


Рис. 6.2

Любая программа в SystenC должна включать заголовочный файл “systemc.h” в начале файла. Этот файл содержит все макросы и шаблоны SystemC. Любая программа в SystemC начинается с зарезервированного

слова SC_MODULE <имя_модуля>, в нашем примере строка 4 содержит модуль hello_world. В программе могут быть предпроцессорные инструкции для компилятора, такие как инструкции `#include`, `#define` перед объявлением модуля. Строка 5 содержит конструктор SC_CTOR, который похож на новый в C ++, строка 8 содержит функцию `void say_hello()`. Эта функция печатает текст «Hello World» в `cout<<` при вызове этой функции. В SystemC, если у вас есть несколько строк внутри блока, вам нужно использовать фигурные скобки `{..}`. Модуль заканчивается на `};`, в данном случае строка 12. Нельзя забывать поставить точку с запятой.

У нас есть `sc_main` в строке 15, которая является функцией верхнего уровня, такой как `main` в C ++. В `sc_main` мы инициализируем модуль `hello_world`. В строке 18 мы вызываем функцию say_hello, которая выполняет печать приветствия.

6.7. Базовый пример канала связи для сложных моделей

Это базовый пример, показывающий, как использовать SystemC, можно применять в качестве шаблона для создания более сложных моделей. Этот пример мы взяли из каталога …/Systemc-2.3.1/examples/syst/pipe.

Он состоит из 3 процессов (stage1, stage2, stage3), которые образуют отдельные этапы работы канала связи. Исходные числа формирует модуль numgen. Результаты отображает модуль Display.

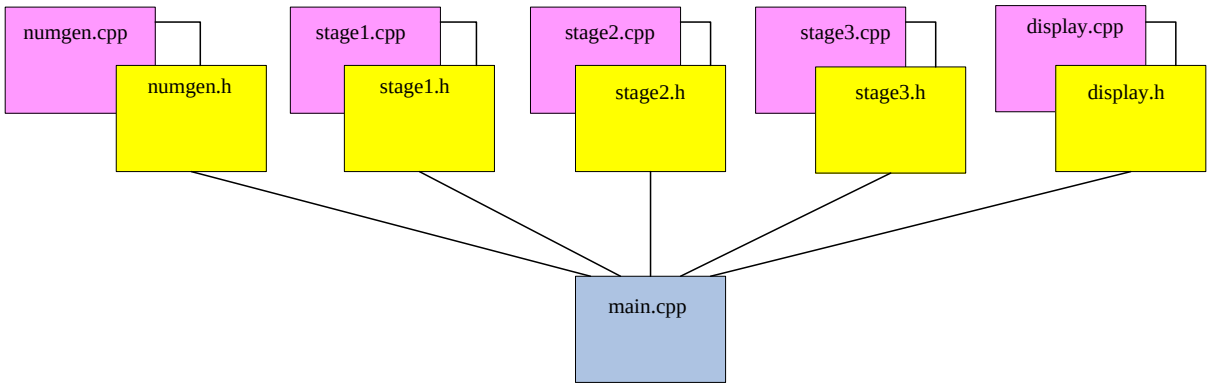


Рис. 6.3. Структура программы

В заголовочном файле numgen.h создана структура с именем numgen и определен модуль sc_module, содержащий порты out1, out2, clock. Функция void generate () описана в конструкторе методом SC_METHOD(generate) и чувствительна к положительному фронту сигнала clock.

В исполняемом файле Numgen.cpp при каждом импульсе clock происходит вычитание чисел 1,5 и 2,8 из исходных значений 134.56 и 98.24. Результаты поступают на выходные порты out1 и out2.

Первый этап конвейера принимает сигналы с двух входов и вычисляет их сумму и разность. Второй этап принимает результаты первой стадии и вычисляет их произведение и частное. Наконец, этап 3 принимает эти выходы со второго этапа и вычисляет первый вход, возведенный в степень второго входа.

Модуль Display выполняет печать результатов.

Чтобы скомпилировать эту модель, вам нужно выполнить gmake / make. После компиляции вы должны найти исполняемый файл run.x. запустить Run.x и распечатает результаты на вашем экране.

Если Вы работаете в среде Eclipce, то все нижеприведенные файлы с листингов 6.7 – 6.17 надо скорировать в папку src, открыть файл main.cpp, выполнить компиляцию и решение. Лучше сделать это непосредственно из примера SystemC-2.3.1, скорировав файлы с расширением .h и .cpp (рис. 6.4). Из программы main.cpp надо исключить директиву #include <conio.h> и последнюю команду _getch(), которые требуются для вывода результатов в Microsoft Visual Studio.

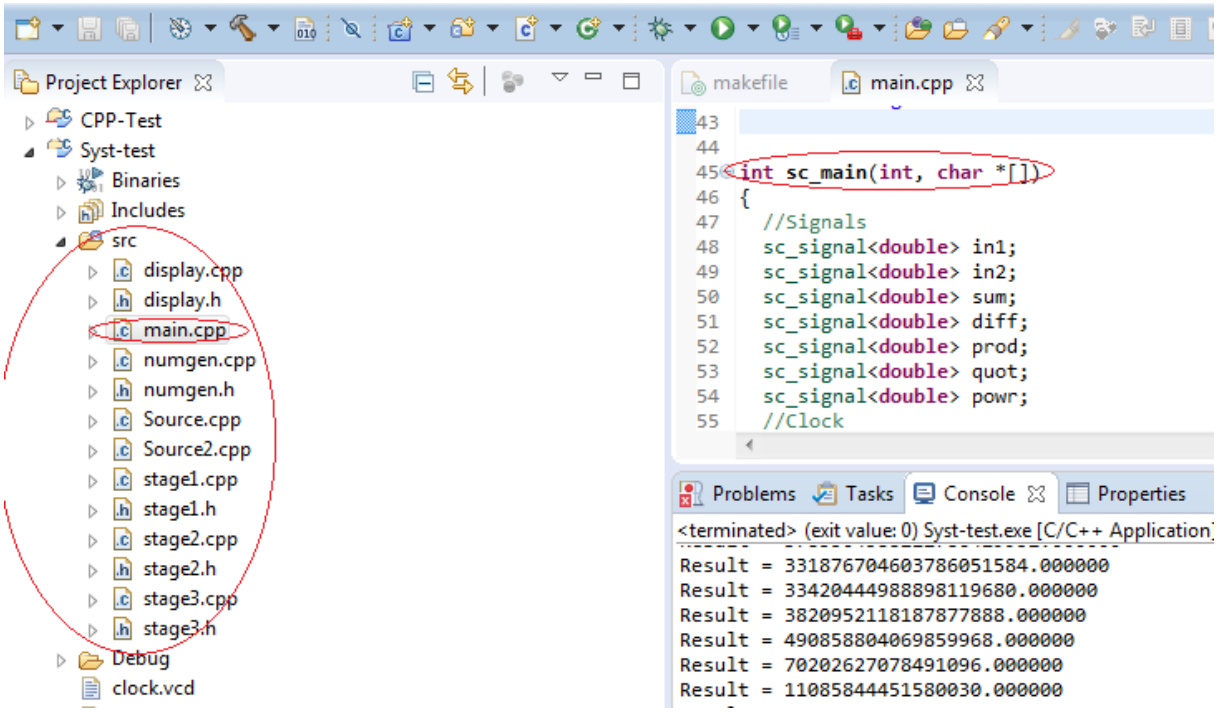


Рис. 6.4. Загрузка примера pipe в Eclipse

Программа построена по традиционному шаблону и содержит пять модулей и главную программу.

Заголовочный файл `Stage1.h` является интерфейсом для модуля stage1.

Листинг 6.7

Заголовочный файл `Numgen.h`

```
#ifndef NUMGEN_H
#define NUMGEN_H

struct numgen : sc_module {
    sc_out<double> out1;      //output 1
    sc_out<double> out2;      //output 2
    sc_in<bool>    clk;       //clock

    // method to write values to the output ports
```

```
    void generate();

    //Constructor
    SC_CTOR( numgen ) {
    SC_METHOD( generate );   /*Declare generate as
SC_METHOD and make it sensitive to positive clock edge*/

        dont_initialize();
        sensitive << clk.pos();
    }
};

#endif
```

Листинг 6.8

Файл реализации Numgen.cpp

```
#include "systemc.h"
#include "numgen.h"

// definition of the `generate' method
void numgen::generate()
{
  static double a = 134.56;
  static double b = 98.24;

  a -= 1.5;
  b -= 2.8;
```

```
  out1.write(a);
  out2.write(b);

} // end of `generate' method
```

Листинг 6.9

Заголовочный файл Stage1.h

```
#ifndef STAGE1_H
#define STAGE1_H

struct stage1 : sc_module {
    sc_in<double> in1;   //input 1
    sc_in<double> in2;   //input 2
    sc_out<double> sum;  //output 1
    sc_out<double> diff; //output 2
    sc_in<bool>    clk;  //clock

    void addsub();   /*method implementing
functionality*/

    //Counstructor
    SC_CTOR( stage1 ) {
     SC_METHOD( addsub );     /*Declare addsub as
SC_METHOD and make it sensitive to positive clock edge*/
     dont_initialize();
        sensitive << clk.pos();
```

```
    }

};

#endif
```

Листинг 6.10

Файл реализации Stage1.cpp

```
#include "systemc.h"
#include "stage1.h"

//Definition of addsub method
void stage1::addsub()
{
  double a;
  double b;

  a = in1.read();
  b = in2.read();
  sum.write(a+b);
  diff.write(a-b);

} // end of addsub method
```

Листинг 6.11

Заголовочный файл `Stage2.h`

```
#ifndef STAGE2_H
#define STAGE2_H
```

```
struct stage2 : sc_module {
    sc_in<double>  sum;      //input port 1
    sc_in<double>  diff;     //input port 2
    sc_out<double> prod;     //output port 1
    sc_out<double> quot;     //output port 2
    sc_in<bool>    clk;      //clock

    void multdiv();          /*method providing
functionality*/

    //Constructor
    SC_CTOR( stage2 ) {
    SC_METHOD( multdiv );     /*Declare multdiv as
SC_METHOD and make it sensitive to positive clock edge*/
        dont_initialize();
        sensitive << clk.pos();
    }

};

#endif
```

Листинг 6.12

Файл реализации Stage2.cpp

```
#include "systemc.h"
#include "stage2.h"

//definition of multdiv method
void stage2::multdiv()
```

```
{
  double a;
  double b;

  a = sum.read();
  b = diff.read();
  if( b == 0 )
      b = 5.0;

  prod.write(a*b);
  quot.write(a/b);

} // end of multdiv
```

Листинг 6.13

Заголовочный файл Stage3.h

```
#ifndef STAGE3_H
#define STAGE3_H

struct stage3: sc_module {
    sc_in<double>  prod;     //input port 1
    sc_in<double>  quot;     //input port 2
    sc_out<double> powr;     //output port 1
    sc_in<bool>    clk;      //clock

    void power();            /*method implementing
functionality*/
```

```
    //Constructor
    SC_CTOR( stage3 ){
        SC_METHOD( power );      /*declare power as
SC_METHOD and make it sensitive to positive clock edge */

        dont_initialize();
        sensitive << clk.pos();      }

};

#endif
```

Листинг 6.14

Файл реализации Stage3.cpp

```
#include <math.h>
#include "systemc.h"
#include "stage3.h"

//Definition of power method
void stage3::power()
{
  double a;
  double b;
  double c;

  a = prod.read();
  b = quot.read();
```

```
  c = (a>0 && b>0)? pow(a, b) : 0.;
  powr.write(c);

} // end of power method
```

Листинг 6.15

Заголовочный файл DISPLAY_H

```
#ifndef DISPLAY_H
#define DISPLAY_H

struct display : sc_module {
    sc_in<double> in;         // input port 1
    sc_in<bool>   clk;        // clock

    void print_result();      /* method to display input
port values*/

    //Constructor
    SC_CTOR( display ) {
    SC_METHOD( print_result ); /* declare print_result as
SC_METHOD and make it sensitive to positive clock edge*/
        dont_initialize();
    sensitive << clk.pos();        }

};

#endif
```

Листинг 6.16

Файл реализации Display.cpp

```
#include "systemc.h"
#include "display.h"

#include <stdio.h>

//Definition of print_result method
void display::print_result()
{
    printf("Result = %f\n", in.read());
} // end of print method
```

Листинг 6.17

Главная программа Main.cpp

Это файл верхнего уровня, создающий экземпляры модулей и связывающий порты с сигналами. Эта программа формирует 50 импульсов clock с периодом 20 нс.

```
#include "systemc.h"
#include "stage1.h"
#include "stage2.h"
#include "stage3.h"
#include "display.h"
#include "numgen.h"
#include <conio.h> //for MSVC

int sc_main(int, char *[])
```

```
{
  //Signals
  sc_signal<double> in1;
  sc_signal<double> in2;
  sc_signal<double> sum;
  sc_signal<double> diff;
  sc_signal<double> prod;
  sc_signal<double> quot;
  sc_signal<double> powr;
  //Clock
  sc_signal<bool>   clk;

  numgen N("numgen");       /*instance of `numgen'
module*/
  N(in1, in2, clk );    //Positional port binding

  stage1 S1("stage1");  /*instance of `stage1' module*/
  //Named port binding
  S1.in1(in1);
  S1.in2(in2);
  S1.sum(sum);
  S1.diff(diff);
  S1.clk(clk);

  stage2 S2("stage2");   /*instance of `stage2' module*/
  S2(sum, diff, prod, quot, clk );  /*Positional port
binding*/

  stage3 S3("stage3");    /*instance of `stage3' module*/
```

```
  S3( prod, quot, powr, clk);   /*Positional port
binding*/

  display D("display");  /*instance of `display' module
*/
  D(powr, clk);    /*Positional port binding

  sc_start(0, SC_NS);     //Initialize simulation
  for(int i = 0; i < 50; i++){
      clk.write(1);
      sc_start( 10, SC_NS );
      clk.write(0);
      sc_start( 10, SC_NS );
  }

  _getch();  //for MSVC
  return 0;
}
```

На рис. 6.5 показан фрагмент решения в среде Eclipce .

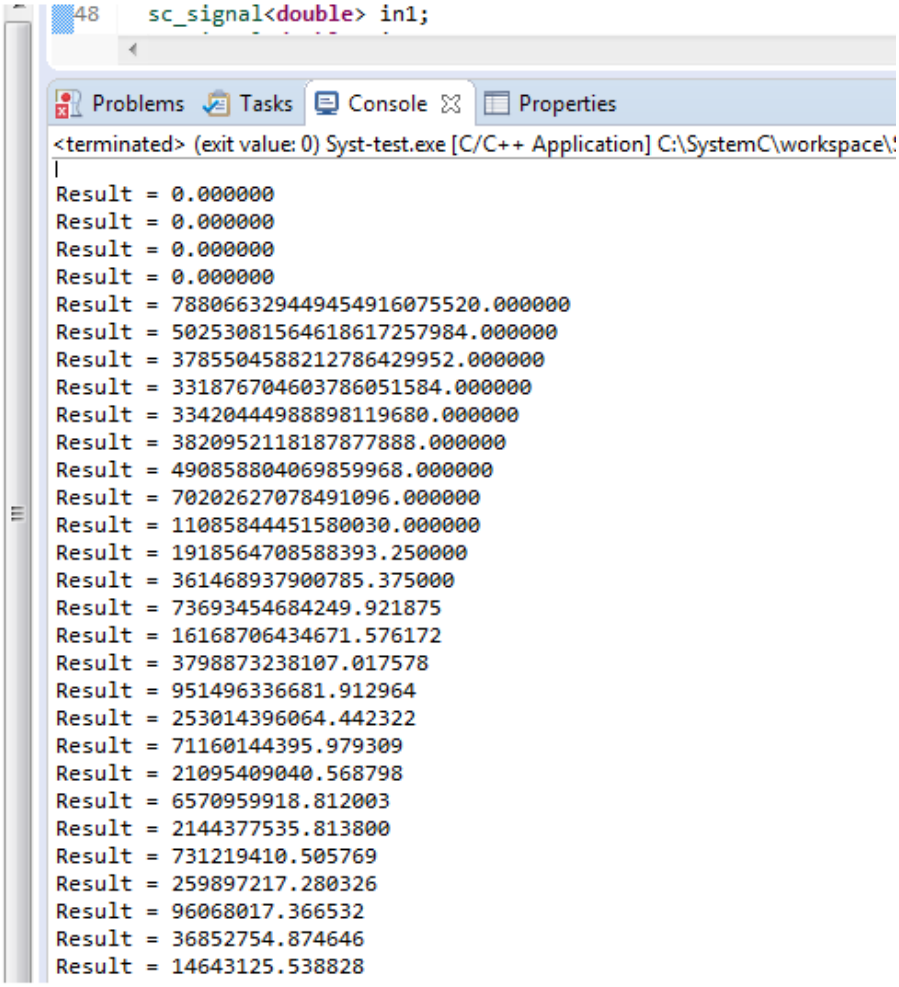


Рис. 6.5 . Фрагмент решения в среде Eclipse

На рис.4.6 показана фрагмент программы main.cpp в среде Visual Studio 2012. Для вывода результатов эта программа содержит директиву #include <conio.h> и оператор _getch().

Решение в Visual Studio 2012 показано на рис. 6.7.

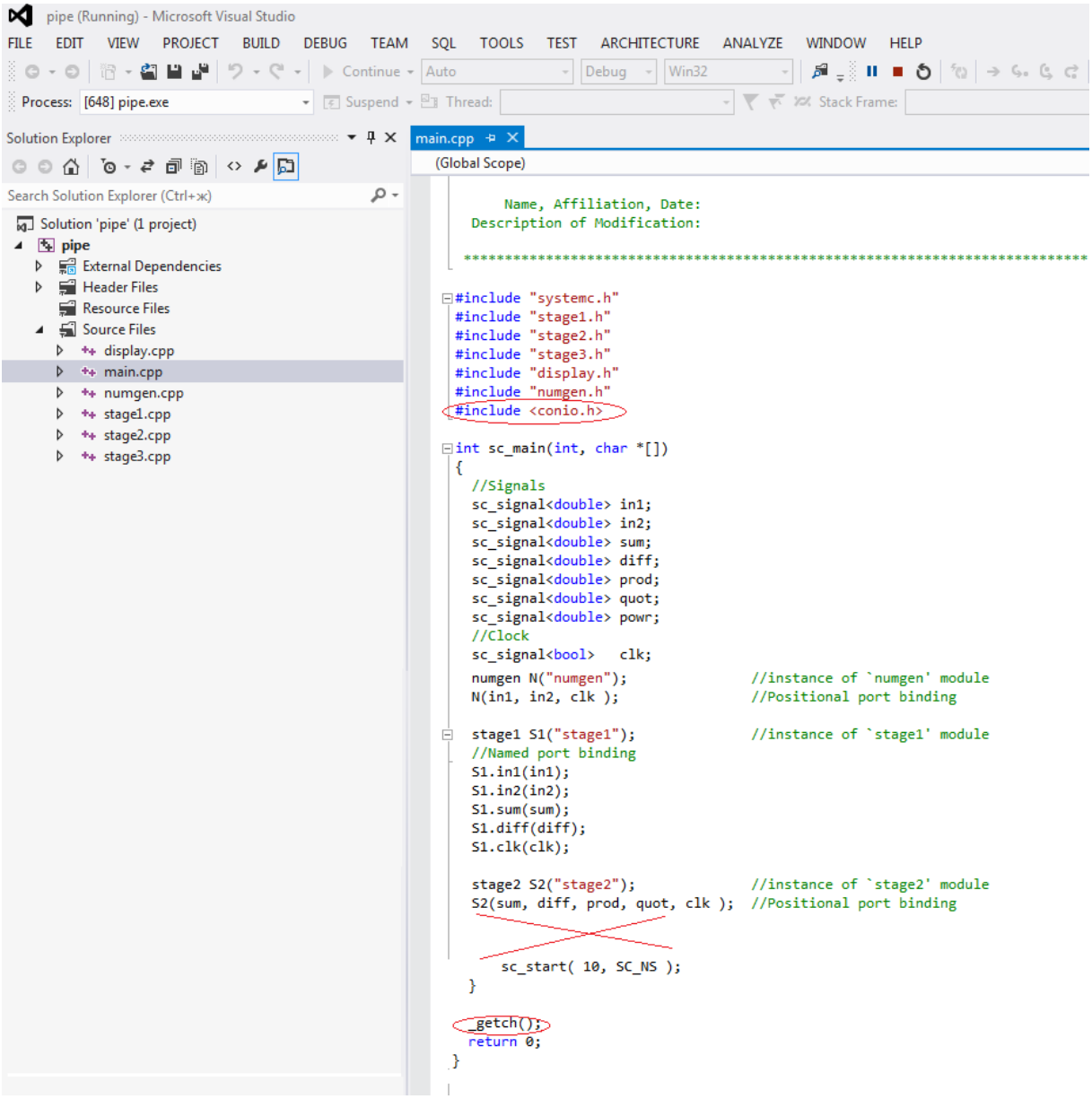


Рис. 6.6. Фрагмент программы main.cpp в среде Visual Studio 2012

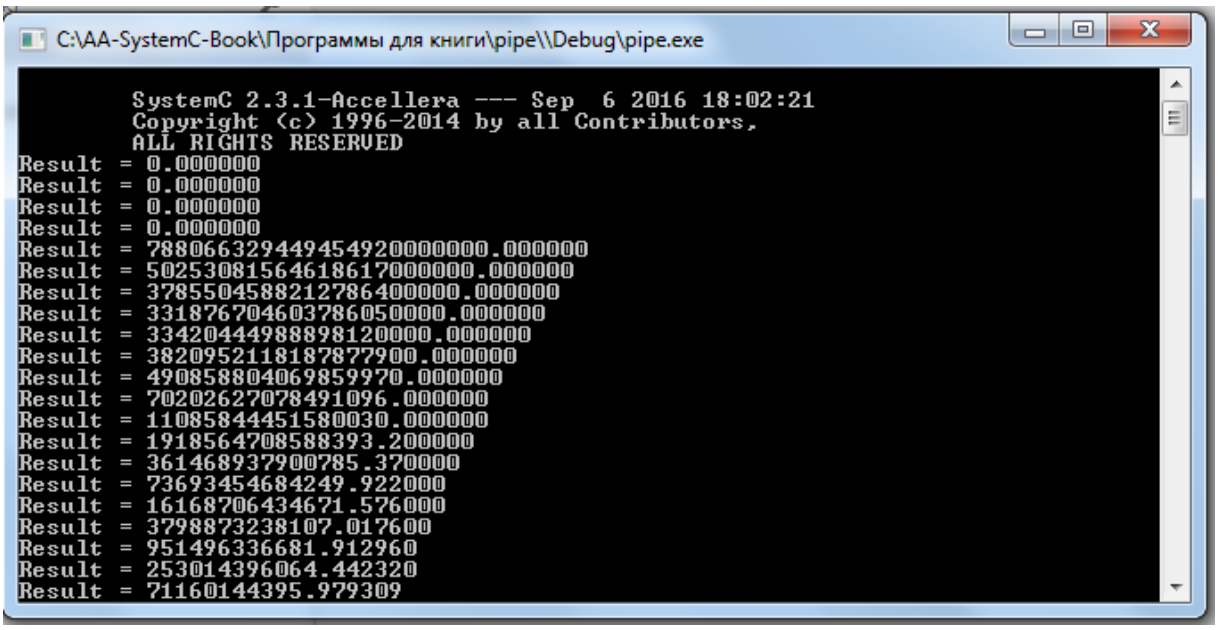


Рис. 6.7. Фрагмент решения в среде Visual Studio 2012

6.8. Использование SystemC для RTL синтеза устройств

SystemC компилятор синтезирует SystemC RTL модули или проекты с интегрированными RTL и поведенческими модулями в список соединений на уровне затворов. Он также может синтезировать системный модуль SystemC в RTL или Netlist-netlist. После синтеза вы можете использовать этот список соединений в качестве входных данных для других инструментов *компании Synopsys*, такие как Design Compiler и Physical Compiler.

Технология Synopsys лежит в основе инноваций, которые меняют то, как мы живем и работаем: интернет вещей, автономные машины, умные медицинские устройства, безопасные финансовые услуги, машинное и компьютерное зрение. Эти прорывы вступают в эпоху Smart, Secure Everything - где устройства становятся умнее, все подключено, и все должно быть безопасным.

Использование этой новой эры технологий дает усовершенствованные кремниевые чипы, которые сделаны более умными благодаря замечательному программному обеспечению, которое ими управляет.

Synopsys находится на переднем крае Smart, Secure Everything с самыми передовыми в мире инструментами для проектирования кремниевых чипов, проверки, интеграции IP и тестирования безопасности приложений. Технология помогает заказчикам внедрять инновации от Silicon до Software, поэтому они могут поставлять Smart, Secure Everything.

SystemC Compiler - это инструмент, который может принимать как поведенческие, так и RTL модели SystemC и выполнять поведенческий или RTL синтез, так как требуется, чтобы создать список соединений на уровне затворов. Вы также можете использовать SystemC компилятор для создания описания HDL для моделирования или использования с другими инструменты HDL в вашей работе.

На рис. 6.8 показаны процессы поведенческого синтеза и RTL синтеза для описания на уровне затворов с помощью SystemC Compiler

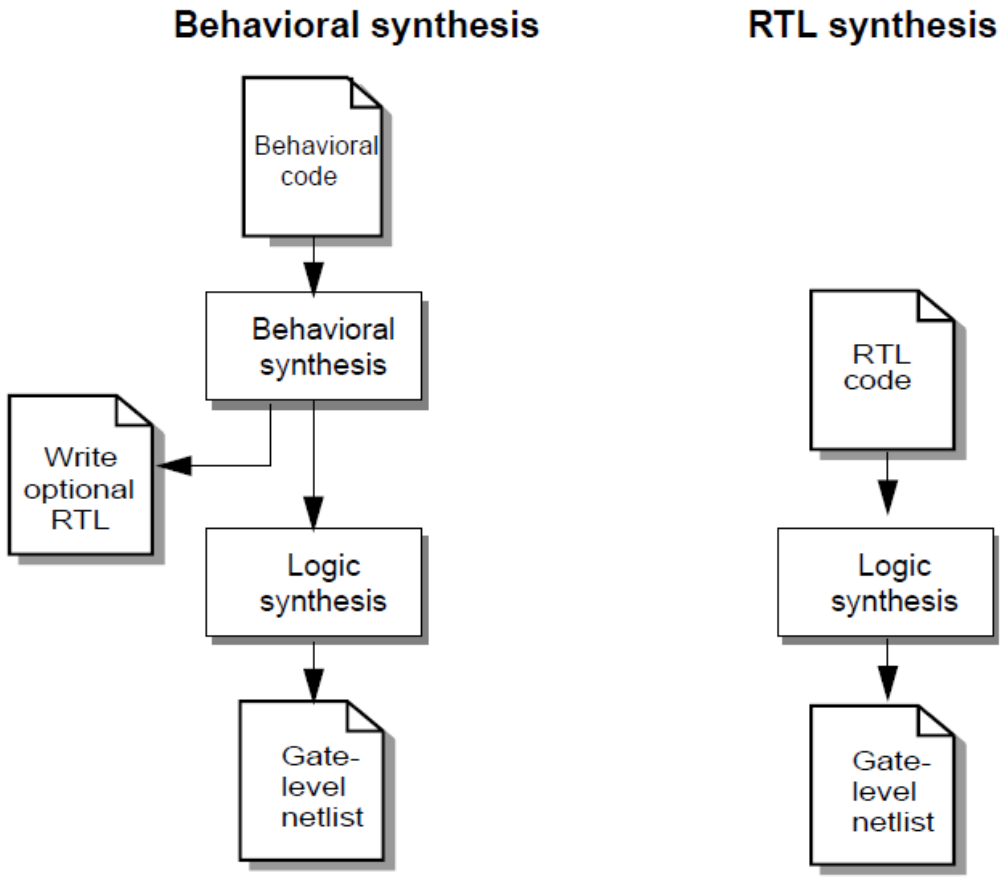


Рис. 6.8. Сравнение поведенческого синтеза и RTL-синтеза

Выбор правильной абстракции для синтеза

Вы можете реализовать аппаратный модуль, используя синтез RTL или синтез поведенческого уровня. Модель RTL описывает регистры в вашем проекте и комбинационную логику между регистрами. Вы можете указать функциональность вашей системы как конечного автомата (FSM-finite-state machin) и путь передачи данных. Поскольку обновления регистра привязаны к тактам, модель имеет точность цикла, как на интерфейсах, так и внутри. Точность внутреннего цикла означает, что вы указываете тактовый цикл, в котором выполняются каждая операция.

Поведенческая модель является алгоритмическим описанием. В отличие от полного программного описания поведение ввода-вывода модели описано в цикле точно. Поэтому ожидания операторов вставлено в алгоритмическое описание, чтобы четко очертить такты границ цикла и при выполнении операций ввода-вывода. В отличие от описаний RTL, поведение описывается алгоритмически, а не в терминах FSM и путей передачи данных. Оцените каждый модуль за модулем, рассмотрите каждый атрибутов модуля для определения применим ли RTL или поведенческий синтез.

Определение атрибутов, подходящих для синтеза RTL

При определении аппаратного обеспечения найдите следующие атрибуты дизайна

Модуль, подходящий для синтеза RTL с помощью SystemC Compiler:

• Легче понять дизайн как FSM (Finite State Machine – конечный автомат) и пути передачи данных, чем как алгоритм (например, микропроцессор).

• Дизайн очень высокопроизводительный, а дизайнер, поэтому должен выполнять полный контроль над архитектурой.

• Конструкция содержит сложную память, такую как SDRAM или RAMBUS.

• Дизайн асинхронный.

Процесс синтеза RTL использует специальные команды, описанные в руководствах:

• Компилятор проектирования Synopsys

• Synopsys HDL Compiler или компилятор VHDL

• Симулятор Synopsys Scirocco VHDL

• Synopsys Verilog Compiled Simulator (VCS)

Определение атрибутов, подходящих для поведенческого синтеза

При определении аппаратного обеспечения найдите следующие атрибуты дизайна модуля, который подходит для поведенческого синтеза с SystemC

Содержание атрибутов:

• Легче понять дизайн как алгоритм, чем как FSM и путь данных (например, быстрое преобразование Фурье, фильтр, обратное квантование или цифровой процессор сигналов).

• Конструкция имеет сложный поток управления - например, есть процессор.

• У дизайна есть доступ к памяти, и вам нужно синтезировать доступ к синхронной памяти.

Обзор усовершенствований

Чистая C / C ++ модель вашего оборудования описывает только то, что составляет аппаратное обеспечение, без предоставления информации о аппаратной структуре или архитектуре. Начиная с модели C/C ++, целью первого этапа разработки является создание аппаратной структуры. Чтобы синтезировать оборудование, вам необходимо:

• Определить порты ввода-вывода для аппаратного модуля

• Указать внутреннюю структуру как модулей

• Указать внутреннюю связь между модулями

Для каждого блока в дизайне вы начинаете с функционального уровня SystemC и уточняете его в RTL-модели для синтеза с компилятором SystemC.

Этапы совершенствования модели высокого уровня в RTL-модель для синтеза включают следующие шаги:

• Определить ввод / вывод в точном соответствии с циклом;

• Отделить логику управления и пути данных;

• Определение архитектуры путей данных;

• Определить явные FSM для логики управления.

Высокоуровневая система SystemC может содержать абстрактные порты, типы которых не могут быть переведены на оборудование. Для каждого абстрактного порта, вам необходимо определить порт или набор портов для замены каждого терминала абстрактного порта и заменить все обращения к абстрактному порту или терминалу с доступом к вновь определенным портам.

Входы и выходы для синтеза RTL

Для компилятора SystemC требуется иметь описание SystemC RTL, технологическую библиотеку и синтетическую библиотеку.

На рис. 6.9 показан процесс синтеза в компилятор SystemC и выход из него.

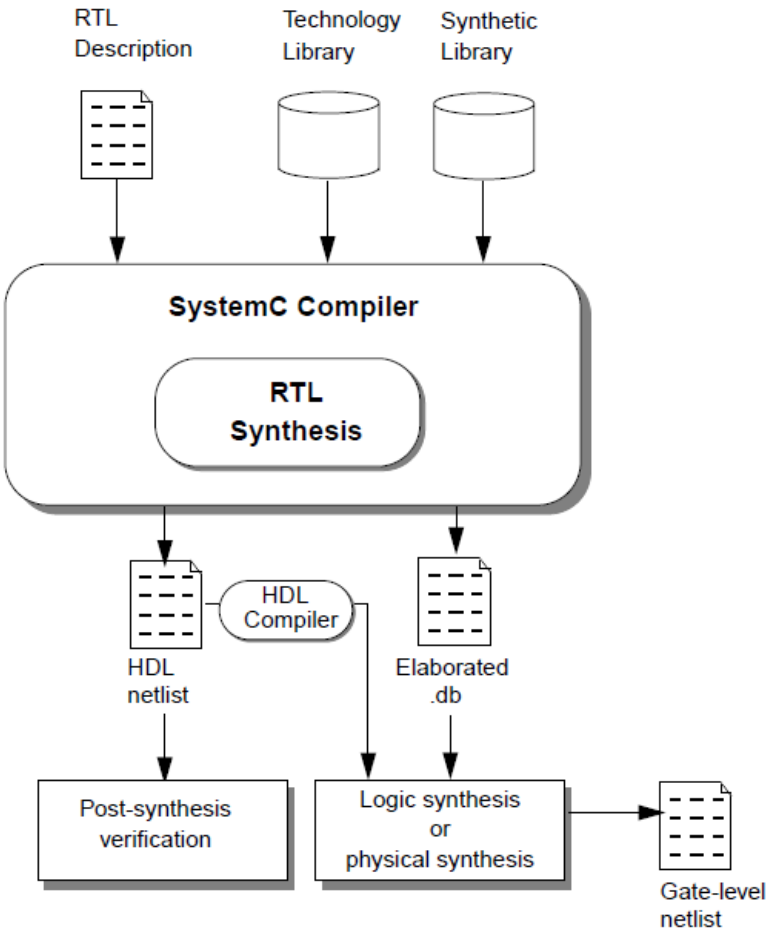


Рис. 6.9. Процесс RTL- синтеза в SystemC

Описание SystemC RTL делают, используя SystemC Class Library. Примеры конструкций будут рассмотрены далее.

Технологическая библиотека предоставляется поставщиком ASIC в Synopsys .db Format базы данных. Поставщик предоставляет области, сроки, модели загрузки соединений и условия эксплуатации. Вы предоставляете путь к выбранной вами Библиотеке технологий для вашего дизайна, определяя целевую библиотеку.

Синтетическая библиотека является технологически независимой библиотекой логики и включает такие компоненты, как сумматоры и множители. Компилятор SystemC сопоставляет ваши проектные операторы с синтетической библиотекой логических компонентов. Вы предоставляете путь к выбранным вами синтетическим библиотекам для вашего дизайна, определяя переменную синтетическую библиотеку в dc_shell.

SystemC Compiler создает расширенный .db-файл для ввода в инструмент проектирования компилятора. Он также генерирует RTL-файлы HDL (например, Verilog), которые могут использоваться в потоках на основе HDL.

6.9. Испытательные программы Testbench

Testbench (испытательный стенд) – это модель, которую используют для испытания и верификации проектов путем тестирования. В добавление к написанию проекта в SystemC можно написать testbench, используя разные языки, с том числе и SystemC.

Testbench имеет три главные цели:

1. Генерировать стимулы для моделирования (осциллограммы).
2. Подавать стимулы на тестируемое устройство и собирать выходные реакции.
3. Сравнивать выходные реакции с ожидаемыми результатами.

Есть много различных путей для написания testbench. Стимулы могут быть записаны в модуле stimulus.h и stimulus.cpp, проверка выходных результатов и сравнение можно записать в другом моделе monitor.h и monitor.cpp. Тестируемый проект будет в отдельном модуле dut.h и dut.cpp. Главная программа соединяет различные модули и подключает их к сформированному испытательному стенду. Другой путь состоит в дополнении генерации стимулов в главную программу во время проверки функционирования.

Рекомендуемы стиль создания конструкции testbench следующий. Сигналы и такты должны быть декларированы в первую очередь, они могут следовать за содержанием модуля. Затем должны быть открыты файлы трассировки и сделан вызов трассировки И, наконец, могут быть вызваны команды моделирования start и stop. Открытые файлы трассировки должны быть закрыты после окончания моделирования.

6.9.1. Основные конструкции испытательных программ

SystemC обеспечивает следующие конструкции, помогающие в моделировании:

Sc_clock: генерирует тактовый сигнал;

Sc_trace: сохраняет информацию трассировки в файл специального формата;

Sc_start: запускает симуляцию на определенное время;

Sc_stop: останавливает симуляцию;

Sc_time_stamp: получает текущее время моделирования с единицами времени;

Sc_simulation_time: получает текущее время моделирования без единиц времени.

Sc_cycle, sc_initialize: используется для выполнения циклического моделирования.

Sc_time: определяет значение времени.

Рассмотрим более детально эти конструкции, возможно, повторив пройденный материал.

`Sc_clock`

Позволяет создавать специальные тактовые объекты, которые содержат временные осциллограммы.

Объявление clock

Sc_clock rclk (“rclk”, 10, SC_NS);

создает тактовую осциллограмму с периодом 10 нс, с рабочим циклом 50% и начальным значением true (1).

Другая декларация:

Sc_clock mclk (“mclk”, 10, SC_NS, 0.2, 5, SC_NS, false);

Создает тактовую осциллограмму с периодом 10 нс, рабочим циклом 20%, первый форнт появляется после 5 нс и начальное значение на первом фронте false (0).

Первый аргумент – это имя тактов и должно быть объявлено. По умолчанию период равен 1 единице времени (по умолчанию 1 нс).

Например: sc_clock clka (“clka”);

- это тактовый сигнал с периодом 1 нс, раблчим циклом 50%, начальным значением true и первым фроном в момент 0.

`Sc_trace`

SystemC поддерживает три различных формата, в которых можно сохранять результаты моделирования:

VCD (Value Change Dump)

WIF (Waveform Interchange Format)

ISDB (Integrated Signal Data Base)

Чтобы сохранить значения в формате VCD, используют соответствующую функцию вызова: Sc_create_vcd_trace_file(). Расширение .vcd добавляется автоматически.

Заголовок файла декларирует тип указателя для sc_trace_file.

Например:

sc_trace_file *tfile=sc_create_vcd_trace_file (“myvcddump”);

Эта декларация создает VCD файл трассировки, названный myvcddump.vcd. Укакзатель, который надо использовать в testbench, tfile.

Используя функцию sc_trace() , вы можете специфицировать сигналы, значения которых вы хотите сохранить в файле трассировки.

Пример: sc_trace (tfile, signal_name, “signal_name”);

Эта функция сохраняет значение signal_name в файле трассировки tfile. Строка значений сигнала будет появлятся в файле трассировки.

Перед выходом из программы sc_main испытательного стенда надо закрыть файлы трассировки:

Sc_close_vcd_trace_file (pointer_to_trace_file);

Вызов функции sc_trace() должен появится после нового открытия файла трассировки и создания трассируемых сигналов.

`Sc_start ()`

Этот метод указывает ядру моделирования начать симуляцию. Он может быть специфицирован после всех конкретизаций и после вызова трассировок.

Пример:

Sc_start (100, SC_MS);

говорит ядру моделирования запустить симуляцию на 100 мс.

Метод вызова: sc_start (-1) ;

говорит симулятору запустить моделирование навсегда.

`Sc_stop ()`

Метод sc_stop () может быть использован в любых процессах, чтобы остановить моделирование:

Sc_stop ();

- Не имеет каких-либо аргументов.

`Sc_time_stamp`

Этот метод возвращает текущее время моделирования с единицами измерениюю.

Например:

cout <<"Current time is" <<sc_time_stamp ()<<endl;

напечатает, например : Current time is 25 ns

`sc_simulation_time`

Этот метод возвращает текущее время симуляции как целое значение двойной длины в терминах единиц времени по умолчанию. Наример:

```
Double curr_time=sc_simulation_time ();
```

Где выполняется, что curr_time будет содержать текущее время моделирования.

`Sc_cycle и sc_initialize`

Эти два метода используются для выполнения циклического моделирования. Наример, если Вы хотите оценивать Ваш проект через

каждые 10 единиц времени, Вы должны делать циклическую симуляцию. В таком случае Вы не используете sc_start, но используете sc_initialize() и sc_cycle ().

Метод sc_initialize () инициализирует ядро симуляции. Метод sc_cycle () выполняет все процессы , которые готовы к запуску, которые могли взять некоторое количество дельта-циклов, пока нет других процессов, готовых к запуску. Затем это трассирует необходимые сигналы перед расширением времени моделирования на указанную величину. Так:

Sc_cycle (10, SC_US); // 10 microsecond

будет моделировать все процессы и затем увеличит время моделирования на 10 мкс.

Sc_time

Декларацию sc_time используют, чтобы задать значение времени, например, 10 нс, 20 нс. Объект sc_time может быть затем использован всюду, где требуется значение времени, как в sc_clock и sc_start.

Например:

```
Sc_time t1 (100, SC_NS);/* Specifies the value 100 ns*/
Sc_time t2 (20, SC_PS); /* Specifies the value 20 ps*/

// Following two forms are equivalent:
Sc_start (t1); // Run simulation for 100 ns
Sc_start (100, SC_NS);

Sc_cycle (t2);

Sc_time period (10, SC_NS);
Sc_time start_time (2,SC_NS);
Sc_clock fclk ( “fclk”, period, 0.2, start_time, true);
```

Можно использовать одну из единиц времени: `SC_FS, SC_PS, SC_NS, SC_US, SC_MS, SC_SEC.`

По умолчанию время разрешения 1 ps. Оно может быть изменено, используя метод `sc_set_time_resolution()`, например так:

```
sc_set_time_resolution(100, SC_PS);
```

Временное разрешение будет 100 ps.

Если Вы установите:

Sc_clock c1 (“c1”, 20.26, SC_NS);

то будут созданы такты с периодом 20300 ps. Временное разрешение можно задавать только кратным 10 и обычно оно задается только один раз в самом начале программы `sc_main ()`.

6.9.2. Сигналы

Регулярные периодические сигналы (waveforms), такие как такты, можно генерировать, используя декларацию `sc_clock`. Рассмотрим различные способы создания стимулов.

Произвольный сигнал

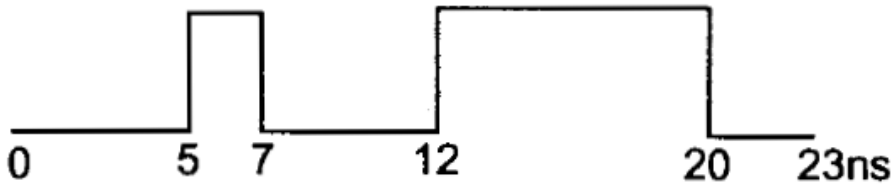


Figure 8-3 An arbitrary waveform

Рис. 6.10. Форма произвольного сигнала

Можно использовать процесс SC_THREAD. Этот процесс можно приостанавливать и перезапускать, основываясь на времени или возникающий событиях. Вот модуль, который генерирует произвольный сигнал:

```
//File: wave.h
#include “systemc.h”
```

```
SC_MODULE (wave) {
sc_out<bool> sig_out;
void prc_wave ();
SC_CTOR (wave) {
SC_THREAD (prc_wave);
};

//File: wave.cpp
#include “wave.h”

void wave::prc_wave() {
sig_out=0;
wait (5, SC_NS);
sig_out=1;
wait (2, SC_NS);
sig_out = 0;
wait (5, SC_NS);
sig_out=1;
wait (8, SC_NS);
sig_out = 0;
}
```

Процесс `prc_wave` начнет исполнение во время инициализации. Все процессы (SC_METHOD, SC_THREAD) выполняются один раз во время инициализации прямо перед началом симуляции. Перед началом выполнения выходному значению присвоен 0. Последующая команда ожидания wait приостанавливает процесс на 5 нс. Такая задержка не может появится в

процессе SC_METHOD. После задержки выходное значение становится равным 1 и процесс задерживается на 2 нс и т.д.

Сложный повторяющийся сигнал

Если Вы хотите повторять сигнал каждые 100 нс, это можно сделать, используя бесконечный цикл в процессе SC_THREAD.

```
// File: wave2.cpp
#include "wave.h"

void wave::prc_wave(){
while (1) {
sig_out=0;
wait (5, SC_NS);
sig_out=1;
wait (2, SC_NS);
sig_out=0;
wait (5, SC_NS);
sig_out=1;
wait (8, SC_NS);
sig_out=0;
wait (80, SC_NS);
}
}
```

Тактовый сигнал можно также моделировать следующим образом.

```
// File: myclock.h
#include "systemc.h"
const int start_value=0;
const int INITIAL_DELAY=5;
const int FIRST_DELAY=2;
const int SECOND_DELAY=3;
```

```
SC_MODULE (myclock) {
Sc_out<bool>clk_out;
void prc_myclock();

SC_CTOR (myclock) {
SC_THREAD (prc_myclock);
}
};

// File: myclock.cpp
#include "myclock.h"

Void myclock::prc_myclock() {
Clk_out=START_VALUE;
Wait  (INITIAL_DELAY, SC_NS);

While (1) {
Clk_out=!clk_out;
Wait (FIRST_DELAY, SC_NS);
Clk_out=!clk_out;
Wait (SECOND_DELAY, SC_NS);
}
}
```

Figure 8-4 My own clock generator.

Рис. 4.11. Сигнал собственного генератора

Генерация произвольных импульсов

Рассмотрим модель генератора импульсов, синхронизированных тактовым сигналом.

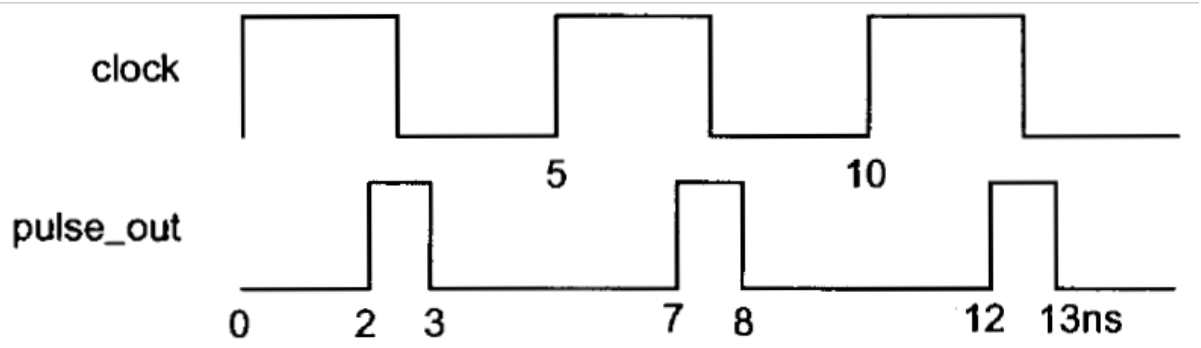


Рис. 6.12. Форма импульсного сигнала

Листинг 6.18

```
//File: pulse.h
#include "systemc.h"
#define DELAY 2, SC_NS
#define ON_DURATION 1, SC_NS

SC_MODULE (pulse) {
Sc_in<bool>clk;
Sc_out<bool>pulse_out;

Void prc_pulse();
```

```
SC_CTOR (pulse){
SC_THREAD (prc_pulse);
Sensitive_pos<<clk;
}
};

//File: pulse.cpp
#include “pulse.h”

Void pulse::prc_pulse(){
Pulse_out=0;
While (true) {
Wait (); //Wait for positive edge of clock
Wait (DELAY);
Pulse_out=1;
Wait (ON_DURATION);
Pulse_out=0;
}
}

// File: pulse_main.cpp
#include “pulse.h”

Int sc_main (int argc, char *argv[]) {
Sc_signal<bool>pout;
Sc_trace_file *tf;
Sc_clock clock (“masater_clk”, 5, SC_NS);

//Instantiate the pulse module:
```

```
pulse p1 (“pulse_p1”);
p1.clk (clock(;
p1.pulse_out (pout);


/* Specify trace file pulse.vcd and trace signals:*/
tf=sc_create_vcd_trace_file (“pulse”);
sc_trace (tf, clock, “clock”);
sc_trace (tf, pout, “pulse_out”);

sc_start (100, SC_NS);
sc_close_vcd_trace_file (tf);
cout<<”Finished at time” <<sc_time_stamp()<<endl;
return 0;
}
```

Процесс SC_THREAD может опционально иметь лист чувствительности. Команда `wait` в процессе должна ждать изменения сигнала из списка чувствительности. Это первая команда в цикле. Две другие команды ожидания выполняют только истекшее время и на связаны с листом чувствительности.

Реактивные стимулы

Реактивные стимулы генерируют на основе текущего состояния проекта, который тестируется, и `testbench`, тоже реактивная, основывается на состоянии проекта. Такое приближение полезно, если разные стимулы надо прикладывать в разных состояниях проекта. Такой подход полезен, если различные стимулы требуются, когда проект находится в различных стадиях. На рис. 4.13 показан механизм взаимодействия между устройством и испытательной моделью. Входной сигнал `resert` сбрасывает

модель расчета факториала в исходное состояние. Сигнал `start` устанавливается после появления входных данных `data`. Когда вычисления выполнены, выходной сигнал `done` показывает, что вычисленный сигнал появляется на выходах `fac_out` и exp_out. Результирующее значение факториала будет (fac_out*2 $^{exp\_out}$). Модель `testbench` предоставляет входной сигнал `data` от начального значения 1 до 20 с инкркментом 1. Testbench выдает входные данные, устанавливает сигнал `start`, ожидает сигнала `done`, и затем выдает следующее входное значение. Подтверждения используются, чтобы убедиться, что появившиеся на выходе значения корректны.

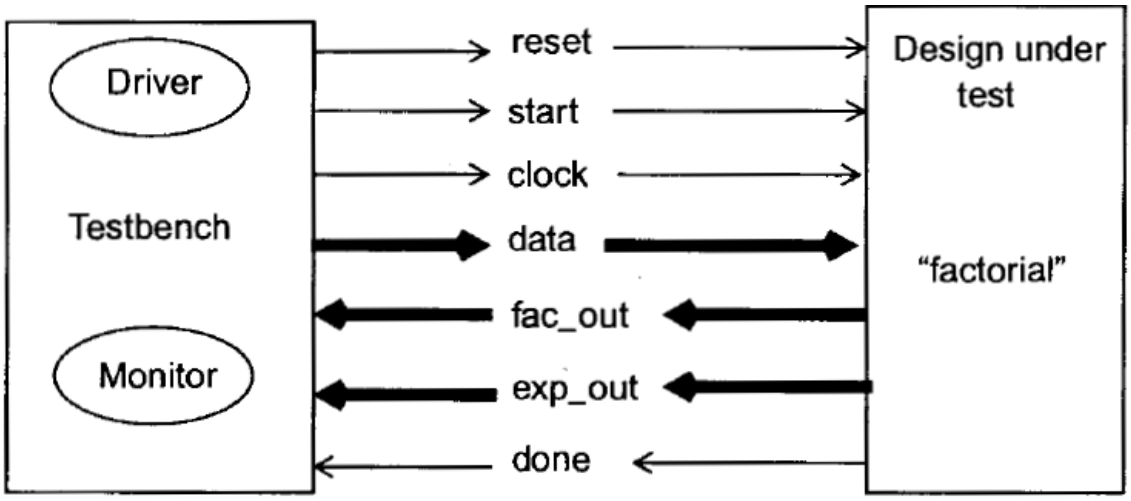


Figure 8-6 Handshake between the testbench and design under test.

Рис. 6.13. Взаимодействие между testbench и тестируемым проектом

6.10. Пример моделирования D-триггера с испытательной программой

D-триггер (рис. 6.14) имеет вход сброса (`reset`), тактовый вход (`clock`), вход данных (`data`) и выход (`data_out`). Испытательная программа `testbench` использует метод `wave()`, чтобы создать последовательность входных сигналов. Метод `wait()` приостанавливает процесс и ждет появления события из его листа статической

чувствительности (лист описан в блоке конструктора). В этом примере этим событием является положительный фронт сигнала `clk`. Модуль `check` просто передает выходные изменения на выходной терминал.

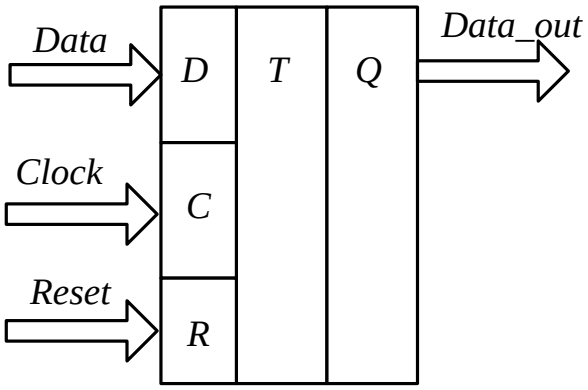


Рис. 4.14. Схема D-триггера

Листинг6.19

Программа ff.h

```
//File: ff.h
#include "systemc.h"


SC_MODULE(ff){

    sc_in<bool> clk;
    sc_in<bool> reset;
    sc_in<int> data;
    sc_out<int> data_out;

    void prc_ff();
    SC_CTOR (ff)
    {
        SC_METHOD (prc_ff);
        sensitive_pos<<clk<<reset;
```

```
    }
};
```

Листинг6.20

Программа ff.cpp

```
//File:ff.cpp
#include "ff.h"

void ff::prc_ff() {
    if (reset)
        data_out=0;
    else
        data_out=data;
}
```

Листинг 6.21

Программа ff_tb.h

```
//File:ff_tb.h
#include "systemc.h"


SC_MODULE(ff_tb) {
    sc_in<bool>clk;
    sc_in<int> data_out;
    sc_out<bool>reset;
    sc_out<int> data;
    void test();
    void check();
```

```
    SC_CTOR(ff_tb){
        SC_THREAD(test);
        sensitive_pos<<clk;
        SC_METHOD(check);
        sensitive<<data_out;
    }
};
```

Листинг 6.22

Программа ff_tb.cpp

```
//File:ff_tb.cpp
#include "ff_tb.h"

void ff_tb::test(){
    reset.write(1);
    wait(20, SC_MS);
    reset.write(0);
    data.write(1);
    wait(20, SC_MS);
    data.write(0);
    wait(20, SC_MS);
    data.write(1);
    wait(20, SC_MS);
    data.write(0);
    wait(20, SC_MS);
    data.write(1);
    wait(20, SC_MS);
    data.write(1);
    wait(20, SC_MS);
```

```
}

void ff_tb::check() {
    cout<<"Output data is (@"<<sc_time_stamp()<<
    "):"<<data_out.read()<<endl;
}
```

Листинг 6.23

Программа main.cpp

```
//File:main.cpp
#include "ff.h"
#include "ff_tb.h"


int sc_main(int argc, char *argv[]){
    sc_clock clk ("clk", 2, SC_MS);
    sc_signal<bool> reset;
    sc_signal<int> data, data_out;

    ff_tb tb("tb");
    tb.clk(clk);
    tb.reset(reset);
    tb.data_out(data_out);
    tb.data(data);

    ff f1 ("ff_f1");
    f1.clk(clk);
    f1.reset(reset);
    f1.data_out(data_out);
```

```
    f1.data(data);
        /* Start simulation and run forever. simulation
stops*/
    /*due to execution of sc_stop() in module ff_tb.*/
    sc_start(200, SC_MS);
    return 0;
}
```

Результаты моделирования

```
Info: (I804) /IEEE_Std_1666/deprecated: sc_sensitive_pos
Output data is (@0 s):0
Output data is (@20 ms):1
Output data is (@40 ms):0
Output data is (@60 ms):1
Output data is (@80 ms):0
Output data is (@100 ms):1
```

Рис. 6.15

6.11. Программы «First_counter» и «Testbench»

На рис. 6.16 показана схема счетчика, который надо запрограммировать в SystemC и проверить функционирование. Листинг 6.24 работы счетчика first_counter.cpp.

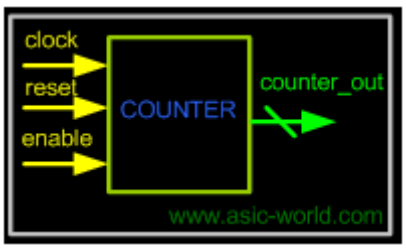


Рис. 4.16. Счетчик

Листинг 6.24

Программа first_counter.cpp

```
#include "systemc.h"

SC_MODULE (first_counter) {
  sc_in_clk     clock ;   /* Clock input of the design*/
```

```
  sc_in<bool>   reset ;      /* active high, synchronous
Reset input*/
  sc_in<bool>   enable;      /* Active high enable signal
for counter*/
  sc_out<sc_uint<4> > counter_out; /* 4 bit vector output
of the counter*/

  //----Local Variables Here---------------------
  sc_uint<4> count;

  //---------Code Starts Here-----------------
  /* Below function implements actual counter logic*/
  void incr_count () {
    /* At every rising edge of clock we check if reset is
active*/
    /* If active, we load the counter output with
4'b0000*/
    if (reset.read() == 1) {
      count =  0;
      counter_out.write(count);
    /* If enable is active, then we increment the
counter*/
    } else if (enable.read() == 1) {
      count = count + 1;
      counter_out.write(count);
      cout<<"@" << sc_time_stamp() <<" :: Incremented
Counter "
        <<counter_out.read()<<endl;
    }
  } // End of function incr_count
```

```
  // Constructor for the counter
  /* Since this counter is a positive edge trigged one,*/
  /* We trigger the below block with respect to
positive*/
  /* edge of the clock and also when ever reset changes
state*/
  SC_CTOR(first_counter) {
    cout<<"Executing new"<<endl;
    SC_METHOD(incr_count);
    sensitive << reset;
    sensitive << clock.pos();
  } // End of Constructor

}; // End of Module counter
```

В этой программе введен и описан модуль `SC_MODULE (first_counter)`. Модуль имеет три входных порта `(clock, resert, enable)`, выходной порт `(counter_out)`. Локальной переменной является 4-х битный вектор `counter_out`.

Функция `void incr_count ()` выполняет инкрементацию счетчика. Вначале счетчик обнуляется. Если сигнал `resert` активный, каждый нарастающий фронт тактового сигнала `clock` будет вызвать сброс счетчика. При этом, если сигнал `resert` неактивный, а сигнал `enable` активный, происходит инкрементация счетчика на единицу. Выводится время с начала моделирования и значание числа в счетчике.

Далее программа содержит конструктор `SC_CTOR(first_counter)`, в котором описан процесс `SC_METHOD (incr_count)` и чувствительность метода к сигналам `resert` и положительному фронту сигнала `clock`.

Любая цифровая схема, независимо от того, насколько сложна она, должна быть проверена. Для логики счетчика нам нужно предоставить логику синхронизации и сброса. Как только счетчик выходит из сброса, мы переключаем вход разрешения на счетчик и проверяем форму волны, чтобы проверить, правильно ли подсчитывает счетчик. То же самое делается в SystemC.

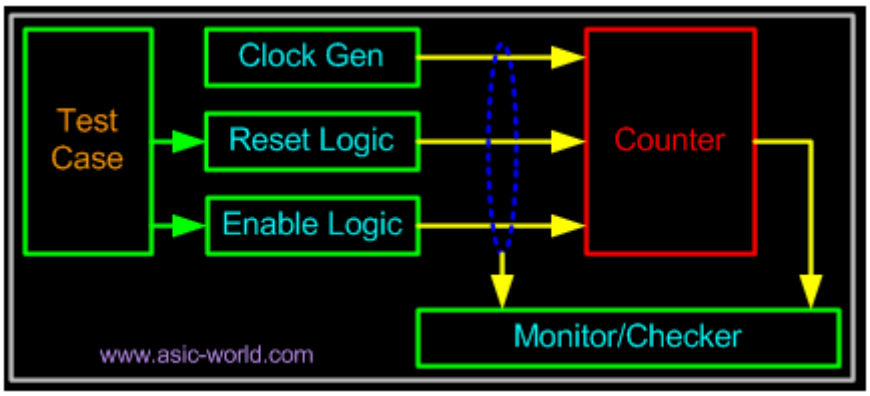


Рис. 6.17. Схема программы Testbench

Программа testbench состоит из генератора тактовых импульсов, управления сбросом, управления включением и логики монитора / контролера. Ниже на листинге 6.25 приведен простой код `testbench` без логики `monitor / checker`. Рекомендуем, пользуясь комментариями, внимательно изучить программу, скорировать файлы **first_counter.cpp** и **first_counter_tb.cpp** в папку src проекта в среде Eclipse C++, выполнить компиляцию и решение.

Листинг 6.25

Программа first_counter_tb.cpp

```
#include "systemc.h"
#include "first_counter.cpp"

int sc_main (int argc, char* argv[]) {
  sc_signal<bool>   clock;
```

```
  sc_signal<bool>   reset;
  sc_signal<bool>   enable;
  sc_signal<sc_uint<4> > counter_out;
  int i = 0;
  // Connect the DUT
  first_counter counter("COUNTER");
    counter.clock(clock);
    counter.reset(reset);
    counter.enable(enable);
    counter.counter_out(counter_out);

  sc_start(1, SC_MS);

  // Open VCD file
  sc_trace_file *wf =
sc_create_vcd_trace_file("counter");
  // Dump the desired signals
  sc_trace(wf, clock, "clock");
  sc_trace(wf, reset, "reset");
  sc_trace(wf, enable, "enable");
  sc_trace(wf, counter_out, "count");

  // Initialize all variables
  reset = 0;       // initial value of reset
  enable = 0;      // initial value of enable
  for (i=0;i<5;i++) {
    clock = 0;
    sc_start(1, SC_MS);
    clock = 1;
    sc_start(1, SC_MS);
```

```
  }
  reset = 1;    // Assert the reset
  cout << "@" << sc_time_stamp() <<" Asserting reset\n"
<< endl;
  for (i=0;i<10;i++) {
    clock = 0;
    sc_start(1, SC_MS);
    clock = 1;
    sc_start(1, SC_MS);
  }
  reset = 0;    // De-assert the reset
  cout << "@" << sc_time_stamp() <<" De-Asserting
reset\n" << endl;
  for (i=0;i<5;i++) {
    clock = 0;
    sc_start(1, SC_MS);
    clock = 1;
    sc_start(1, SC_MS);
  }
  cout << "@" << sc_time_stamp() <<" Asserting Enable\n"
<< endl;
  enable = 1;  // Assert enable
  for (i=0;i<20;i++) {
    clock = 0;
    sc_start(1, SC_MS);
    clock = 1;
    sc_start(1, SC_MS);
  }
  cout << "@" << sc_time_stamp() <<" De-Asserting
Enable\n" << endl;
```

```
  enable = 0; // De-assert enable

  cout << "@" << sc_time_stamp() <<" Terminating
simulation\n" << endl;
  sc_close_vcd_trace_file(wf);
  return 0;// Terminate simulation

    }
```

Результаты решения показаны на рис. 6.18.

```
Console
<terminated> (exit value: 0) Syst-test.exe [C/C++ Application] C:\SystemC\workspace\Syst-test\[
Executing new

Info: (I702) default timescale unit used for tracing: 1 ps (counter.vcd)
@11 ms Asserting reset

@31 ms De-Asserting reset

@41 ms Asserting Enable

@42 ms :: Incremented Counter 0
@44 ms :: Incremented Counter 1
@46 ms :: Incremented Counter 2
@48 ms :: Incremented Counter 3
@50 ms :: Incremented Counter 4
@52 ms :: Incremented Counter 5
@54 ms :: Incremented Counter 6
@56 ms :: Incremented Counter 7
@58 ms :: Incremented Counter 8
@60 ms :: Incremented Counter 9
@62 ms :: Incremented Counter 10
@64 ms :: Incremented Counter 11
@66 ms :: Incremented Counter 12
@68 ms :: Incremented Counter 13
@70 ms :: Incremented Counter 14
@72 ms :: Incremented Counter 15
@74 ms :: Incremented Counter 0
@76 ms :: Incremented Counter 1
@78 ms :: Incremented Counter 2
@80 ms :: Incremented Counter 3
@81 ms De-Asserting Enable

@81 ms Terminating simulation
```

Рис. 6.18. Результаты ммоделирования первого счетчика

Программа **first_counter_tb.cpp** создает VCD – файл и выполняет трассировку сигналов clock, resert, enable, count (листинг 6.26).

Листинг 6.26

```
  // Open VCD file
  sc_trace_file *wf =
sc_create_vcd_trace_file("counter");
  // Dump the desired signals
  sc_trace(wf, clock, "clock");
  sc_trace(wf, reset, "reset");
  sc_trace(wf, enable, "enable");
  sc_trace(wf, counter_out, "count");
```

По умолчанию файл VCD записан в папке Workspace. Полезно скопировать этот файл поближе с основной директрории, например, в папку C:\VCD и открыть в программе GTKWave. Как установить программу и как с ней работать описано во второй главе книги.

На рис. 6.19 показаны результаты трассировки.

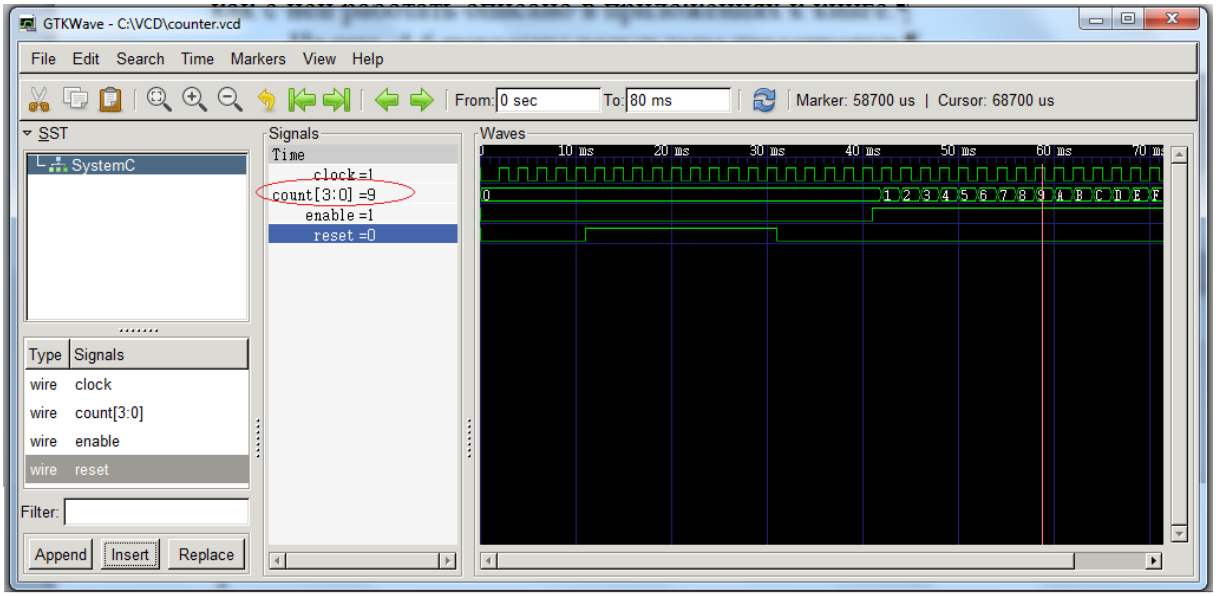


Рис. 6.19. Результаты трассировки моделирования счетчика

Установка маркера в нужной позиции показывает численные значения сигналов. Так в позиции от 58 до 60 мс выходное значение счетчика равно 9 и это значение выдается в момент 60 мс на печать.

6.12. Моделирование на системном уровне

Функции SystemC, рассмотренные в предыдущем разделе, делают его подходящим языком описания оборудования. Тем не менее, вышеупомянутые возможности сделают SystemC лишь незначительным улучшением по существующим языкам HDL.

Истинное преимущество SystemC как языка спецификации заключается в том, что он охватывает все важные аппаратные функции моделирования, а также предоставляет мощное моделирование конструкций для проектирования системного уровня.

Это гарантирует, что переход к методологии на основе SystemC не влечет за собой никаких компромиссов с точки зрения выразительной силы на более низких уровнях абстракции, и тем не менее, предоставляет полезную основу для моделирования на более высоких уровнях.

Нынешним HDL крайне не хватает последней способности.

Функции моделирования на уровне системы, представленные в SystemC 2.0 в основном включают в себя поддержку гораздо более общего и абстрактного средства связи между процессами и более общим механизмом синхронизации событий.

6.12.1. События и чувствительность

SystemC 2.0 представляет общий механизм для спецификации и уведомление о событиях. События больше не приравниваются к переключению одного бита, но это абстрактные типы, которые могут быть использованы для более общих и сложных взаимодействий. Тип события `sc event` может использоваться для объявления событий, которые могут быть созданы с помощью уведомления ключевым словом и быть синхронизированы в операторах ожидания, как показано ниже.

```
sc_event e1, e2; // declare events
sc_time t (5, SC_NS);
e1.notify (t); // notify event e1 after 5 ns
```

```
wait (e1); // suspend execution until
// event e1 occurs
wait (); // wait until an event occurs
// on the process sensitivity list
wait (10, SC_NS, e1 | e2);
// wait for events e1 or e2 to occur
// but for a maximum of 10 ns
```

6.12.2. Интерфейсы и каналы

Ранее мы рассматривали простой пример подключения модулей, используя сигналы. Это картина общения, где взаимодействие между модулями ограничивается передачей значений на отдельные провода, однако, на самом низком уровне абстракции. На уровне системы нужна способность моделировать более абстрактно и изощренно интеллектуальные коммуникационные парадигмы. SystemC 2.0 вводит понятие каналов и интерфейсов, которые обеспечивают эту способность моделирования.

Системный уровень дизайна SystemC состоит из набора модулей и каналов на верхнем уровне. Грубо говоря, модули покрывают функциональные аспекты системы, а каналы несут коммуникационные аспекты. Каналы могут быть очень общими и реализовывать сложные алгоритмы внутри себя, например, сложные протоколы шины с арбитражем. На самом деле каналы могут иметь иерархический состав.

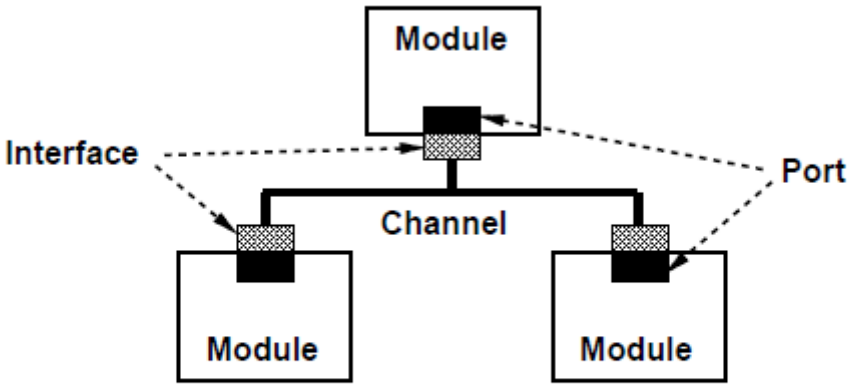


Figure 4: Interfaces and Channels. Ports of modules are connected to channels through interfaces.

Рис. 6.20.

На рисунке 6.20 показана простая конструкция с тремя подключенными модулями на канал. Интерфейс состоит из набора объявлений методов (не относится к методам SC), реализованным каналом.

Эти методы видны для порта, подключенного к каналу через интерфейс. Таким образом, порт (и, следовательно, модуль) изолирован от деталей реализации (таких как локальные данные) самого канала. Эта архитектура помогает сохранить связанные с функциональностью части дизайна, отделенные от коммуникации, насколько это возможно, и позволяет модифицировать одни без влияния на другие до тех пор, пока интерфейс остается неизменным.

Интерфейс и конструкции канала были *вдохновлены* похожими понятия на языке SpecC.

6.12.3. Примитивные и иерархические каналы

Каналы в SystemC могут быть либо примитивными, либо иерархическими.

Примитивные каналы относительно просты. SystemC 2.0 обеспечивает набор примитивных каналов, которые имеют широкое применение, таких как `sc signal` (классические сигналы) , sc mutex (используется для

моделирования взаимного исключения) и sc fifo (используется для моделирования очереди).

Иерархические каналы могут демонстрировать структуру. Они являются модулями сами, которые, в свою очередь, могут содержать процессы и другие каналы и модули. Сложные протоколы шины, которые имеют несколько подзадач, могут быть эффективно смоделированы с использованием иерархических каналов.

Простой пример двух модулей, соединенных каналом типа sc fifo показан ниже. Соединение FIFO установлено между выходным портом модуля M1 и входным портом M2.

```
SC_MODULE (A) {
sc_fifo_in<int> in;
sc_fifo_out<int> out;
// rest omitted
};
...
A *M1, *M2; // instances of module A
...
sc_fifo<int> q (5); // create FIFO channel
// with buffer size 5
M1->out (q); // connect port 'out' of M1 to q
M2->in (q); // connect port 'in' of M2 to q
```

6.12.4. Методологические библиотеки

Механизм моделирования, представленный выше, является достаточно общим для моделирование множества различных моделей общения и вычислений.

Различные методологии коммуникации могут быть построены на этой базовой инфраструктуре моделирования. Будущий выпуск SystemC будет содержать один такой пример: библиотека связи ведущий-ведомый. Элементы, представленные здесь, могут быть использованы для простого моделирования коммуникационных интерфейсов, основанных на протоколах master-slave.

6.13. Моделирование на уровне транзакций

Моделирование уровня транзакций (TLM - transaction-level modeling) в SystemC включает связь между процессами SystemC с использованием вызовов функций. В центре внимания TLM лежит связь между процессами, а не алгоритмы, выполняемые самими процессами, поэтому процессы, показанные в этой главе будет довольно тривиальным. Мы предполагаем, что в модели поведения системы некоторые процессы SystemC будут производить данные, другие будут потреблять данные, некоторые инициируют общение, иные будут пассивно реагировать на сообщение, инициированное другими. В центре внимания OSCI TLM-2.0, в частности, является моделирование встроенных микросхем с памятью. TLM-2.0 имеет многоуровневую структуру. Более низкие слои являются более гибкими и общими, а верхние слои являются специфичными для моделирования шины..

6.13.1. Инициаторы, цели и сокеты

В TLM-2.0 инициатор - это модуль, который инициирует новые транзакции, а цель (target) - это модуль, который реагирует на транзакции, инициированные другими модули. Транзакция представляет собой структуру данных (объект C ++), переданную между инициаторами и целевыми объектами с использованием вызовов функций. Тот же модуль может действовать как в качестве инициатора, так и в качестве цели, и это, как правило, будет иметь место для модели арбитра, маршрутизатора или шины.

Для передачи транзакций между инициаторами и целевыми объектами TLM-2.0 использует сокеты (гнезда). Инициатор отправляет транзакции через

сокет инициатора, а цель получает входящие транзакции через целевой сокет. Модуль, который просто пересылает транзакции без изменения их контента известен как компонент межсоединения. Компонент межсоединения будет иметь как целевой сокет, так и сокет инициатора.

Теперь давайте посмотрим на некоторый код SystemC. Модель TLM-2.0 должна включать стандартный заголовок SystemC, заголовок «tlm.h» и любые другие, которые вы используете в комплекте. В этом случае мы используем два сокета из каталога утилит (tlm_utils). Вы должны указать вложенный путь вашего компилятора C ++ (или makefile) в каталог ./include в реализации.

Листинг 5.1

```
#define SC_INCLUDE_DYNAMIC_PROCESSES
#include "systemc"
using namespace sc_core;
using namespace sc_dt;
using namespace std;
#include "tlm.h"
#include "tlm_utils/simple_initiator_socket.h"
#include "tlm_utils/simple_target_socket.h"
```

6.13.2. Общая полезная нагрузка и блокирующий транспорт

Тип транзакции по умолчанию для классов сокетов, подразумеваемый в отсутствие любых аргументов шаблона, - tlm_generic_payload. Общая полезная нагрузка является важной частью стандарта TLM-2.0, поскольку она является еще одним из ключей к достижению взаимодействия между уровнем транзакции моделей. Общая полезная нагрузка служит двум близким целям. Он может использоваться как тип транзакции общего назначения для абстрактной памяти при моделировании шины, когда вас не интересуют точные детали любого конкретного протокола шины, предполагая немедленную функциональную совместимость между

моделями. Альтернативно, общая полезная нагрузка может использоваться в качестве основы для моделирования широкого спектра конкретных протоколов на более подробном уровне. Красота этого подхода заключается в том, что относительно легко соединяться между различными протоколами, когда оба построены поверх того же общего типа полезной нагрузки. Наш модуль-инициатор имеет `thread process` для генерации потока общих транзакций полезной нагрузки.

Листинг 5.6

```
SC_CTOR(Initiator) : socket("socket")
{
SC_THREAD(thread_process);
}
void thread_process()
{
tlm::tlm_generic_payload* trans = new
tlm::tlm_generic_payload;
sc_time delay = sc_time(10, SC_NS);
for (int i = 32; i < 96; i += 4)
{
...
socket->b_transport( *trans, delay );
...
}
}
```

Транзакция отправляется через сокет, используя метод `b_transport` транспортного интерфейса блокировки TLM-2.0, который передает его аргумент транзакции по ссылке и не имеет возвращаемого значения. Инициатор несет ответственность за выделение и удаление хранилища для транзакции.

Вызов b_transport также содержит временную аннотацию, которая должна быть добавлена к текущему времени моделирования (как возвращено sc_time_stamp), чтобы определить время обработки транзакции. Временные аннотации синхронизации активны как для вызова, так и для возврата из b_transport.

Листинг 5.7

```
tlm::tlm_command cmd = static_cast(rand() % 2);
if (cmd == tlm::TLM_WRITE_COMMAND) data = 0xFF000000 | i;
trans->set_command( cmd );
trans->set_address( i );
trans->set_data_ptr( reinterpret_cast<unsigned
char*>(&data) );
trans->set_data_length( 4 );
trans->set_streaming_width( 4 );
trans->set_byte_enable_ptr( 0 );
trans->set_dmi_allowed( false );
trans->set_response_status( tlm::TLM_INCOMPLETE_RESPONSE
);
socket->b_transport( *trans, delay );
```

Каждая общая транзакция полезной нагрузки имеет стандартный набор атрибутов шины: команда, адрес, данные, байт, ширина потока и ответный статус. Общая полезная нагрузка также содержит подсказку и расширения DMI. Хотя каждый атрибут имеет значение по умолчанию, рекомендуется использовать не менее 8 из 10 атрибутов явно перед передачей транзакции на вызов метода интерфейса. Причина в том, что объекты транзакции обычно используются повторно из пула.

Общая полезная нагрузка поддерживает две команды: чтение и запись. Здесь атрибут команды настроен на чтение или запись произвольным образом. Атрибут адреса - это самое низкое значение адреса, по которому

данные должны быть прочитаны или записаны. Здесь адрес устанавливается в индекс цикла. Атрибут указателя данных указывает на буфер данных внутри инициатора, а атрибут длины данных дает длину массива данных в байтах. Здесь длина данных равна 4 байтам. В случае команды записи данные будут скопированы из массива данных в целевой объект, а в случае команды чтения, скопированы с целевого объекта в массив данных. В любом случае фактическая копия выполняется в цели. Атрибут ширины потоковой передачи определяет ширину потокового пакета, где адрес повторяется. Для транзакции без потоковой передачи ширина потока должна равняться длине данных, как это имеет место здесь. Хотя значение по умолчанию для атрибута ширины потоковой передачи равно 0, значение 0 не разрешается, когда транзакция приходит к отправке через вызов метода интерфейса. Этот же принцип применяется к указателю данных и данным длины.

Указатель включения байта установлен в 0, чтобы указать, что байтовые включения не используются. Существует также атрибут байтовой длины, который здесь не задан, потому что с указателем, установленным в 0, он будет проигнорирован. Метод `set_dmi_allowed` устанавливает подсказку DMI, которая всегда должна быть инициализирована на `false`. Атрибут подсказки DMI может быть установлен объектом, чтобы указывать, что он поддерживает интерфейс прямой памяти. Статус ответа всегда должен быть инициализирован значением `TLM_INCOMPLETE_RESPONSE`. Статус ответа может быть задан целью. Десятый общий атрибут полезной нагрузки, не упомянутый выше, представляет собой массив расширений. По умолчанию, любые расширения могут быть проигнорированы инициатором или целью. Способ блокирования транспорта реализуется в целевой памяти. Во-первых, набор из шести атрибутов, которые нельзя игнорировать, извлекается из общей полезной нагрузки транзакции. (Остальные атрибуты - это подсказка DMI и статус ответа, которые задаются целевым объектом, длина разрешения байта, которые можно игнорировать, если байтовые

включения не используются, и расширения, которые в любом случае можно игнорировать.)

Листинг 5.8

```
virtual void b_transport( tlm::tlm_generic_payload&
trans, sc_time& delay )
{
tlm::tlm_command cmd = trans.get_command();
sc_dt::uint64 adr = trans.get_address() / 4;
unsigned char* ptr = trans.get_data_ptr();
unsigned int len = trans.get_data_length();
unsigned char* byt = trans.get_byte_enable_ptr();
unsigned int wid = trans.get_streaming_width();
```

Затем атрибуты проверяются, чтобы гарантировать, что инициатор не пытается использовать функции, которые не может поддерживать целевой объект. В этом случае цель - память не поддерживает байтовые включения, ширину потоковой передачи или пакетные передачи. Следующее утверждение также проверяет, что адрес не выходит за пределы. Если транзакция не может быть выполнена, обработчик отчета SystemC вызывается для генерации ошибки.

Листинг 5.9

```
if (adr >= sc_dt::uint64(SIZE) || byt != 0 || len > 4 ||
wid < len)
SC_REPORT_ERROR("TLM-2",
"Target does not support given generic payload
transaction");
```

Конечно, если цель не может поддерживать определенные функции общей полезной нагрузки, это ограничит совместимость, но, по крайней

мере, наборы функций четко определены и существуют стандартные обязательства по проверке и сообщению о несовместимости. Затем цель реализует команду чтения и записи путем копирования данных в область или из области данных в инициаторе. Что касается суждения, то правило состоит в том, что общая полезная нагрузка принимает тот же смысл, что и главный компьютер. Пока целевая память также моделируется с использованием хоста Endianness, копирование данных может быть выполнено с помощью memcpy:

Листинг 5.10

```
if ( cmd == tlm::TLM_READ_COMMAND )
memcpy(ptr, &mem[adr], len);
else if ( cmd == tlm::TLM_WRITE_COMMAND )
memcpy(&mem[adr], ptr, len);
```

Последним действием метода блокирующего транспорта является установка атрибута статуса ответа общей полезной нагрузки для указания успешного завершения транзакции. Если не задано, статус ответа по умолчанию указывает инициатору, что транзакция неполна.

```
trans.set_response_status( tlm::TLM_OK_RESPONSE );
```

6.13.3. Временная аннотация

Поскольку метод блокирующего транспорта только моделирует функциональность цели, а не моделирует любую деталь времени, он просто игнорирует значение аргумента задержки и возвращает его инициатору нетронутым.

После вызова b_transport инициатор проверяет статус ответа:

```
if (trans->is_response_error() )
SC_REPORT_ERROR("TLM-2", "Response error from
b_transport");
```

Инициатору теперь нужно реализовать любые аннотированные сроки. Поскольку эта модель касается только функциональности, она может продолжать накапливать задержки до бесконечности. Идея заключается в том, чтобы имитационная модель вычислила функциональность инициатора и цели на полной скорости и отслеживала время, затрачиваемое на любые смоделированные ресурсы, просто увеличивая переменную «сбоку». Такой стиль кодирования называется слабоуровневым в TLM-2.0. Однако то, что на самом деле делает модель, - это просто ожидание указанной задержки при возврате из вызова `b_transport`. Это будет замедлять симуляцию, поскольку для транзакции требуется контекстный переключатель. Но этого достаточно для простого примера и делает журнал моделирования простым для интерпретирования.

6.13.4. Взаимодействие и базовый протокол

Таким образом TLM-2.0 обеспечивает совместимость между моделями, использующими стандартный набор API, предоставляет дополнительные классы полезности для улучшения производительности и поощряет последовательный стиль кодирования.

Ключами к совместимости в TLM-2.0 являются:

- использование стандартного инициатора и целевых сокетов, один из которых должен быть создан для каждого соединения с шиной памяти или другой коммуникационным ресурсом;
- использование общей полезной нагрузки, которая должна быть создана и задана для представления атрибутов каждого объекта транзакции;
- использование базового протокола.

Нам не нужно было изучать детали базового протокола в этой главе учебника, но это подразумевается при использовании простых сокетов и метода `b_transport`. Базовый протокол определяет правила использования

общей полезной нагрузки со стандартными интерфейсами TLM-2.0 и сокетами. Это будет описано в последующих руководствах. Блокирующий транспортный интерфейс должен использоваться везде, где транзакция может быть выполнена в одном вызове функции. Это эффективно, когда запрос транзакции переносится вызовом `b_transport`, и ответ передается с возвратом из `b_transport`. В этом примере один объект транзакции повторно используется через вызовы. Хранилище для объекта транзакции назначается инициатором в начале и после запуска. Это приемлемо, поскольку только одна транзакция «в полете» в любой момент времени. Управление памятью тривиально и обрабатывается инициатором.

Еще одна особенность этого конкретного примера заключается в том, что метод блокирования транспорта не используется, то есть не вызывает `wait.b_transport` , но может вызвать `wait` , однако, и в принципе мы могли бы иметь ситуацию, когда есть несколько одновременных вызовов `b_transport` через один и тот же сокет от нескольких потоков в инициаторе, возможно, с противоречивыми аннотациями времени. Такая ситуация разрешена в соответствии с правилами базового протокола.

Блокирующий транспортный интерфейс предназначен для поддержки слабоограниченного стиля кодирования, где основное внимание уделяется функциональному исполнению с минимальной детализация времени и минимальные накладные расходы на моделирование.

6.13.5. Интерфейсы TLM-2.0

TLM-2.0 состоит из набора основных интерфейсов, глобальных частей, инициаторных и целевых сокетов, общей полезной нагрузки, базового протокола и утилит. Также включены основные интерфейсы TLM-1, интерфейс анализа и порты анализа, хотя они отделены от основного тела стандарта TLM-2.0. Основные интерфейсы TLM-2.0 состоят из блокирующих и неблокирующих транспортных интерфейсов, интерфейса прямой памяти (DMI) и отладки транспортного интерфейса. Общая полезная

нагрузка поддерживает абстрактное моделирование подключенных к памяти шин вместе с механизмом расширения для поддержки моделирования конкретных протоколов шины, обеспечивая максимальную совместимость.

Классы TLM-2.0 накладываются поверх библиотеки классов SystemC, как показано на диаграмме (рис. 5.1). Для максимальной совместимости и, в частности, для моделирования шины с памятью рекомендуется, чтобы основные интерфейсы TLM-2.0, сокеты, общая полезная нагрузка и базовый протокол были вместе совместно использованы. Эти классы известны в совокупности как уровень взаимодействия. В случаях, когда общая полезная нагрузка неприемлема, основные интерфейсы, инициатор и целевые сокеты или основной интерфейс сам по себе могут использоваться с альтернативным типом транзакции. Даже технически возможно, что общая полезная нагрузка будет использоваться непосредственно с основными интерфейсами без инициатора и целевых сокетов, хотя этот подход не рекомендуется. Не обязательно использовать утилиты для обеспечения совместимости между моделями шин. Тем не менее, эти классы должны использоваться там, где это возможно, для согласованности стиля, документируются и поддерживаются как часть стандарта TLM-2.0.

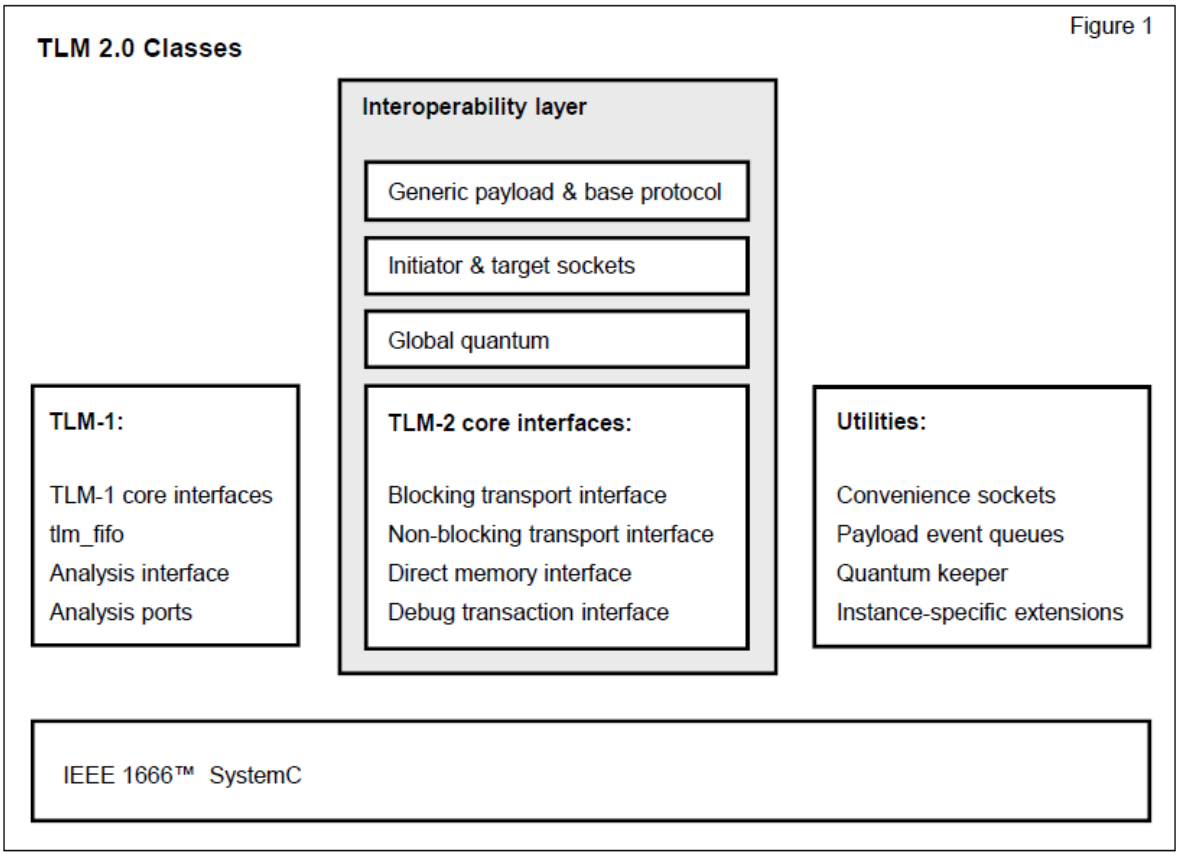


Рис. 5.1. Классы TLM 2.0

Рис. 6.21. Классы TLM 2.0

Общая полезная нагрузка в основном предназначена для моделирования шины с отображением памяти, но может также использоваться для моделирования других протоколов без шины с аналогичными атрибутами. Атрибуты и этапы общей полезной нагрузки могут быть расширены для моделирования конкретных протоколов, но такие расширения могут привести к сокращению функциональной совместимости в зависимости от степени отклонения от стандартной не расширенной общей полезной нагрузки. Ожидается, что быстрая, слабозатухающая модель использует блокирующий транспортный интерфейс, интерфейс прямой памяти и временную развязку. Ожидается, что более точная, ориентированная по времени модель будет использовать неблокирующий транспортный интерфейс и очереди событий полезной нагрузки. Однако эти

утверждения представляют собой только предложения стиля кодирования и не являются нормативной частью стандарта TLM-2.0.

6.13.6. Моделирование на уровне транзакций, варианты использования и абстракция

Стандарт TLM-2.0 определяет набор интерфейсов, которые следует рассматривать как низкоуровневые механизмы программирования для реализации моделей уровня транзакций, а затем описывает ряд стилей кодирования, которые подходят для различных вариантов использования, но не привязаны к ним.

Определения стандартных интерфейсов TLM-2.0 отличаются от описаний стилей кодирования. Это интерфейсы TLM-2.0, которые составляют нормативную часть стандарта и обеспечивают совместимость. Каждый стиль кодирования может поддерживать диапазон абстракции по функциональности, времени и связи. В принципе пользователи могут создавать свои собственные стили кодирования.

Вневременная функциональная модель, состоящая из одного программного потока, может быть записана как функция C или как один процесс SystemC и иногда называется алгоритмической моделью. Такая модель не является уровнем транзакции как таковой, поскольку по определению транзакция является абстракцией связи, а однопоточная модель не имеет межпроцессного взаимодействия. Модель уровня транзакции требует нескольких процессов SystemC для имитации одновременного выполнения и связи.

Абстрактная модель уровня транзакции, содержащая несколько процессов (несколько программных потоков), требует некоторого механизма, с помощью которого эти потоки могут обеспечивать контроль над собой. Это связано с тем, что SystemC использует совместную многозадачную модель, в которой процесс выполнения не может быть предотвращен каким-либо другим процессом. Процессы SystemC контролируют выход, вызывая `wait`

в случае *процесса потока* или возвращаясь к ядру в случае *процесса метода*. Вызовы ожидания обычно скрываются за интерфейсом программирования (API), который может моделировать специфический абстрактный или конкретный протокол, который может или не может опираться на информацию о времени.

Синхронизация может быть сильной в том смысле, что последовательность событий связи точно определяется заранее или слабой в том смысле, что последовательность событий связи частично определяется подробным сроком отдельных процессов. Сильная синхронизация легко внедряется в SystemC с использованием FIFO или семафоров, что позволяет полностью исключить стиль моделирования, где в принципе симуляция может работать без увеличения времени моделирования. В этом смысле вневременное (`untimed`) моделирование выходит за рамки TLM-2.0. С другой стороны, быстрая виртуальная платформа, позволяющая параллельно запускать несколько встроенных программных потоков, может использовать сильную или слабую синхронизацию. В этом стандарте подходящий стиль кодирования для такой модели называется слабо синхронизированным (*loosely-timed*.)

Для более подробной модели уровня транзакции может потребоваться связать несколько временных точек протокола с каждой транзакцией, например, точки синхронизации, чтобы отметить начало и конец каждой фазы протокола. Выбирая подходящее количество точек синхронизации, можно моделировать связь с высокой степенью точности синхронизации без необходимости выполнять модели компонентов в каждом отдельном такте. В этом стандарте такой стиль кодирования называется приблизительным (*approximately-timed*.).

6.13.7. Стили кодирования

Стиль кодирования - это набор идиом языка программирования, которые хорошо работают вместе, а не конкретный уровень абстракции или

интерфейс программного обеспечения. Для простоты и ясности мы ограничим разработку двух конкретных названных стилей кодирования: *loosely-timed* and *approximately-timed.* По своей природе стили кодирования точно не определены, и правила, определяющие основные интерфейсы TLM-2.0, определяются независимо от этих стилей кодирования. В принципе, можно было бы определить другие стили кодирования на основе механизмов TLM-1 и TLM-2.0.

На рис. 5.2 показаны различные задачи и случаи использования стилей кодирования и применяемых механизмов.

6.13.8. Стиль untimed

TLM-2.0 не дает четкого представления о стиле несвязанного кодирования, поскольку для всех современных систем на основе шины требуется некоторое понятие времени для моделирования программного обеспечения, работающего на одном или нескольких встроенных процессорах. Однако несвязанное моделирование поддерживается основными интерфейсами TLM-1. Термин `untimed` иногда используется для обозначения моделей, которые содержат ограниченную информацию о времени неопределенной точности. В TLM-2.0 такие модели будут называться временными.

6.13.9. Стиль Loosely-timed и временная развязка

В слабосвязанном стиле кодирования используется блокирующий транспортный интерфейс. Этот интерфейс позволяет связать только две точки синхронизации с каждой транзакцией, соответствующей вызову и возврату из функции блокирования транспорта. В случае базового протокола первая точка синхронизации отмечает начало запроса, а вторая знаменует начало ответа. Эти два момента времени могут возникать при одном и том же времени моделирования или в разное время.

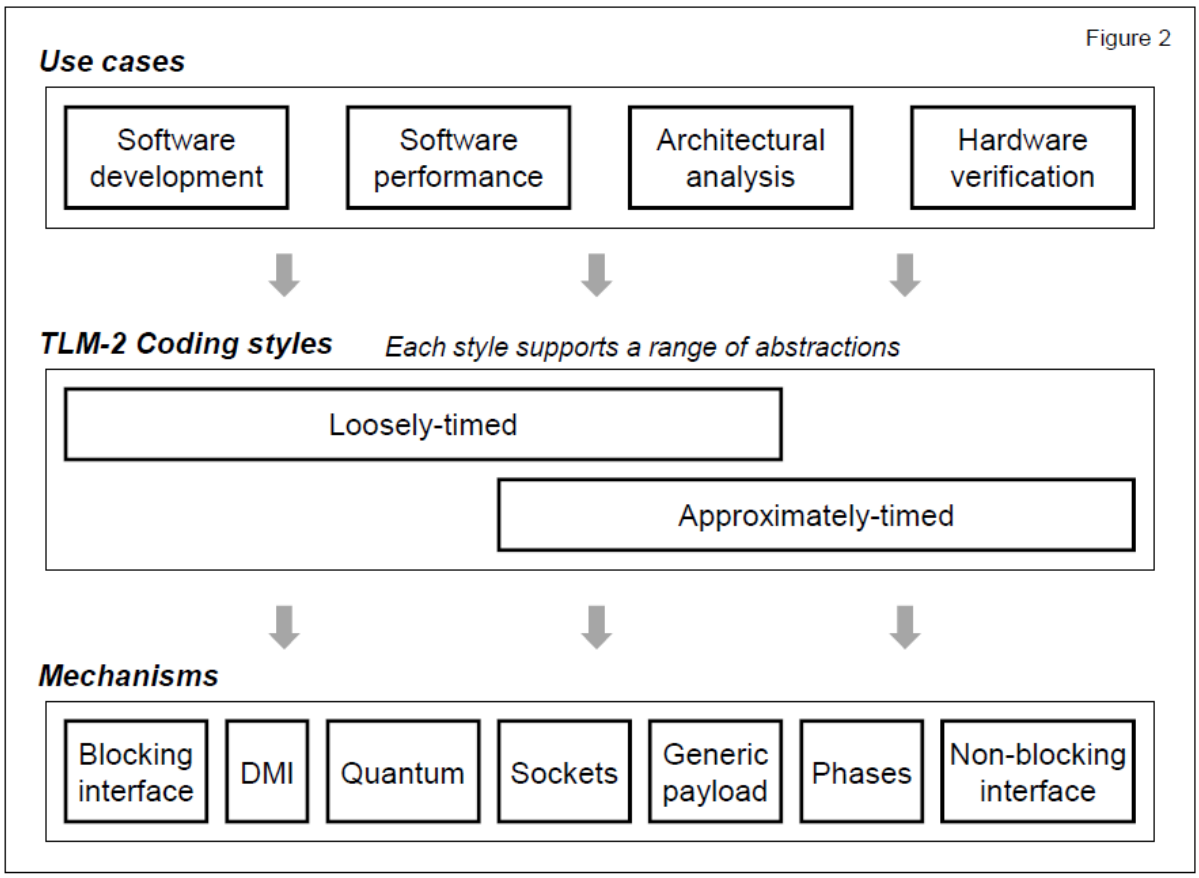


Рис. 5.2. Задачи, стили кодирования и механизмы моделей

Рис. 6.22. Задачи, стили моделирования и механизмы моделей

Слабо-временный стиль кодирования подходит для использования в разработке программного обеспечения с использованием модели виртуальной платформы MPSoC, где содержание программного обеспечения может включать одну или несколько операционных систем. Слабо-временный стиль кодирования поддерживает моделирование таймеров и прерываний, достаточное для загрузки операционной системы и запуска произвольного кода на целевой машине.

Слабо-синхронизированный стиль кодирования также поддерживает временную развязку, когда отдельные процессы SystemC разрешены для запуска в локальном временном режиме без фактического увеличения времени моделирования до тех пор, пока они не достигнут точки, когда им необходимо синхронизировать с остальной системой. Временная развязка может привести к очень быстрой симуляции для определенных систем,

поскольку она увеличивает местоположение данных и кода и снижает затраты на планирование симулятора. Каждому процессу разрешается запускать определенный фрагмент времени или квант, прежде чем переключиться на следующий, или вместо этого может получить контроль, когда он достигнет явной точки синхронизации.

Просто рассматривая SystemC, планировщик SystemC не задерживает время моделирования. Планировщик ускоряет время моделирования до момента следующего события, затем запускает любые процессы из-за запуска в это время или по событию, чувствительному для процесса. Процессы SystemC выполняются только в текущее время моделирования (полученное вызовом метода `sc_time_stamp`), и всякий раз, когда процесс SystemC читает или записывает переменную, он обращается к состоянию переменной, как это было бы при текущем времени моделирования. Когда процесс завершается, он должен передать управление обратно в ядро моделирования. Если имитационная модель написана на мелкодетальном уровне, то накладные расходы на планирование событий и переключение контекста процесса становятся доминирующим фактором в скорости моделирования. Один из способов ускорить моделирование - позволить процессам работать впереди текущего времени моделирования или временной развязки.

При реализации временной развязки в SystemC процесс может быть запущен до времени моделирования, пока ему не потребуется взаимодействовать с другим процессом, например, чтобы прочитать или обновить переменную, принадлежащую другому процессу. В этот момент процесс может либо получить доступ к текущему значению, либо продолжиться (с некоторой возможной потерей точности синхронизации) или может вернуть управление ядру моделирования, только возобновив процесс, когда время моделирования затягивается с локальной временной деформацией. Каждый процесс отвечает за определение того, может ли он

работать до времени моделирования без нарушения функциональности модели. Когда процесс встречается с внешней зависимостью, он имеет два варианта: либо принудительную синхронизацию, что означает предоставление возможности запускам всех других процессов в нормальном режиме до тех пор, пока время моделирования не достигнет, или не проверит или не обновит текущие значения и продолжит процесс.

Если бы процесс был разрешен для запуска до времени моделирования без ограничений, планировщик SystemC не смог бы работать, а другие процессы никогда не получили бы возможности выполнения. Этого можно избежать с помощью ссылки на глобальный квант, который накладывает верхний предел на время, когда процессу разрешено работать до времени моделирования. Квант устанавливается приложением, а квантовое значение представляет собой компромисс между скоростью и точностью моделирования. Слишком маленькие квантовые шаги выдают синхронизацию очень часто, но замедляют моделирование. Слишком большой квант может ввести временные несоответствия в системе, возможно, до такой степени, когда система перестает функционировать.

Например, рассмотрим моделирование системы, состоящей из процессора, памяти, таймера и некоторых медленных внешних периферийных устройств. Программное обеспечение, выполняемое на процессоре, тратит большую часть времени на получение и выполнение инструкций из системной памяти и взаимодействует только с остальной частью системы, когда она прерывается таймером, скажем, каждые 1 мс. ISS, которая моделирует процессор, может быть разрешена для запуска до времени моделирования SystemC с квантом до 1 мс, что обеспечивает прямой доступ к модели памяти, но только для синхронизации с периферийными моделями с частотой прерываний таймера. Дело здесь в том, что ISS не нужно запирать на время моделирования аппаратной части системы, как это было бы при более традиционном совместном симуляторе

аппаратного обеспечения. В зависимости от детализации моделей, временная развязка сама по себе может дать улучшение скорости моделирования примерно 10X или 100X в сочетании с DMI.

В TLM-2.0 временная развязка поддерживается классом `tlm_global_quantum` и аннотацией времени блокирующего и неблокирующего транспортного интерфейса. Класс утилиты `tlm_quantumkeeper` обеспечивает удобный способ доступа к глобальному кванту.

6.13.10. Характеристика слабо-временных и приближенно-временных стилей кодирования

Стили кодирования можно охарактеризовать в терминах временных точек и временной развязки.

Слабовременные. Каждая транзакция имеет только два момента времени, обозначая начало и конец транзакции. Используется время моделирования, но процессы могут быть временно отключены от времени моделирования. Каждый процесс ведет подсчет того, как далеко он прошел впереди времени моделирования, и может появиться, потому что он достигает явной точки синхронизации или потому, что он потребляет квант времени.

Приближенно-временные. Каждая транзакция имеет несколько временных точек. Обычно процессы должны запускаться в режиме блокировки с временем моделирования SystemC. Задержки, аннотированные для взаимодействия процессов, реализуются с использованием тайм-аутов (ожидание) или уведомлений о временном событии.

Untimed. Понятие времени моделирования не требуется. Производятся процессы при явных предопределенных точках синхронизации.

6.13.11. Переключение между слабо-временным и приближенно- временным моделированием.

Модель может переключаться между слабо-временным и приближенно-временным стилем кодирования во время моделирования.

Идея состоит в том, чтобы быстро запускать последовательность сброса и загрузки на слабо-временном уровне, а затем переключиться на приближенно временное моделирование для более подробного анализа, как только симуляция достигнет интересной стадии.

В приведенной ниже таблице (рис. 6.23) представлено сопоставление между вариантами использования для моделирования уровня транзакций и стилей кодирования:

Use Case	**Coding style**
Software application development	Loosely-timed
Software performance analysis	Loosely-timed
Hardware architectural analysis	Loosely-timed or approximately-timed
Hardware performance verification	Approximately-timed or cycle-accurate
Hardware functional verification	Untimed (verification environment), loosely-timed or approximately-timed

Рис. 5.3. Сопоставление вариантов задач и кодирования

Рис. 6.23. Сопоставление вариантов и задач

6.13.12. Мосты транзакций

Интерфейсы ядра TLM-2.0 передают транзакции между инициаторами и целями. Напомним, что инициатор - это модуль, который может инициировать транзакции, т.е. создавать новые объекты транзакций и передавать их, вызывая метод одного из основных интерфейсов. Цель - это модуль, который выступает в качестве конечного пункта назначения для транзакции. В случае записи транзакции, инициатор (например, процессор) записывает данные в цель (например, память). В случае транзакции чтения инициатор считывает данные из целевого объекта. Компонент межсоединений - это модуль, который обращается к транзакции, но не выступает в качестве инициатора или цели для этой транзакции, типичными

примерами являются арбитры и маршрутизаторы. Роли инициатора, межсоединения и цели могут динамически меняться. Например, данный компонент может выступать в качестве межсоединения для некоторых транзакций, но как цель для других транзакций.

Чтобы проиллюстрировать эту идею, будет описано время жизни типичного объекта транзакции. Объект транзакции создается инициатором и передается в качестве аргумента метода транспортного интерфейса (блокирующего или неблокирующего). Этот метод реализуется компонентом межсоединения, таким как арбитр, который может считывать атрибуты объекта транзакции, прежде чем передавать его на другой транспортный вызов. Второй транспортный метод реализуется вторым компонентом межсоединений, таким как маршрутизатор, который, в свою очередь, передает транзакцию через третий транспортный вызов к цели, такой как память, конечный пункт назначения для объекта транзакции. (Фактическое количество компонентов межсоединений будет варьироваться от транзакции к транзакции. Их может и не быть.) Эта последовательность вызовов методов известна как прямой путь. Транзакция выполняется в целевом объекте, и объект транзакции может быть возвращен инициатору одним из двух способов: либо переносом с возвратом из вызовов метода транспорта, когда они разветвляются, (это называются обратным путем), либо передаются путем явной транспортной передачи. Метод вызывает противоположный путь от цели до инициатора, известный как обратный путь. Этот выбор определяется возвращаемым значением из неблокирующего транспортного метода. (Строго говоря, есть два пути возврата, соответствующие прямым и обратным путям, но смысл обычно ясен из контекста.)

Figure 4

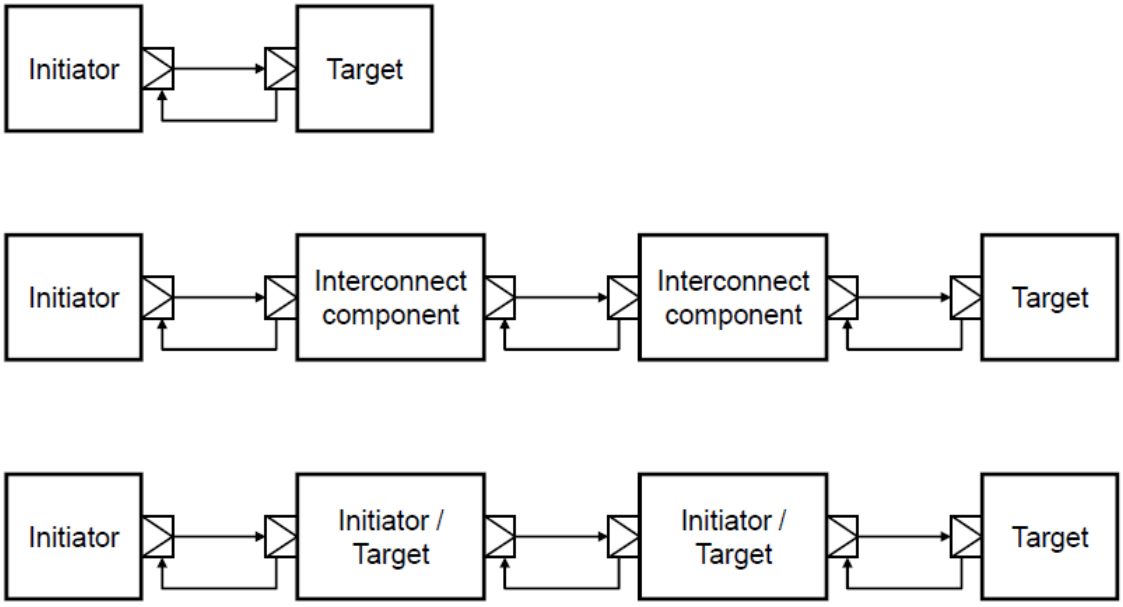


Рис. 6.24. Различные варианты соединений

Прямой путь - это путь вызова, по которому компонент инициатора или межсоединения делает вызовы метода интерфейса вперед в направлении другого компонента межсоединения или цели. Обратный путь - это путь вызова, с помощью которого компонент цели или межсоединения делает обратные вызовы метода интерфейса в направлении другого компонента межсоединения или инициатора. Весь путь между инициатором и мишенью состоит из нескольких переходов, каждый из которых соединяет два соседних компонента. Количество переходов от инициатора к цели больше, чем количество компонентов межсоединений на этом пути. При использовании общей полезной нагрузки путь вперед и назад всегда должен проходить через тот же набор компонентов и сокетов, очевидно, в обратном порядке.

Чтобы поддерживать как прямой, так и обратный пути, для каждого соединения между компонентами требуется порт и экспорт, оба из которых должны быть связаны. Этому способствует сокет инициатора и целевой сокет. Инициатор-сокет предоставляет порт для вызовов метода интерфейса для прямого пути и экспорт для вызовов метода интерфейса на обратном

пути. Целевой разъем обеспечивает обратное. (Более конкретно, сокет инициатора выводится из класса sc_port и имеет sc_export и наоборот для целевого сокета.) Инициатор и классы сокетов-мишеней перегружают оператор привязки порта SystemC, чтобы неявно связывать как прямые, так и обратные пути. Как и транспортные интерфейсы, сокеты также инкапсулируют транспортные интерфейсы DMI и отладки (см. ниже). При использовании сокетов компонент-инициатор будет иметь по меньшей мере один сокет-инициатор, целевой компонент, по меньшей мере, один целевой сокет и компонент межсоединения, по меньшей мере, один из них. Компонент может иметь несколько сокетов, переносящих разные типы транзакций, и в этом случае один компонент может выступать в качестве инициатора или цели для нескольких независимых транзакций.

Для моделирования шинного моста есть две альтернативы. Либо смоделируйте мост шины как компонент межсоединения, либо смоделируйте мост шины как мост транзакции между двумя отдельными транзакциями TLM-2.0. Компонент межсоединения передавал бы указатель на один объект транзакции, что является наилучшим подходом для скорости моделирования. Для моста транзакции требуется копирование объекта транзакции, что дает гораздо большую гибкость, поскольку две транзакции могут иметь разные атрибуты. Использование сокетов TLM-2.0 рекомендуется для максимальной совместимости, удобства и последовательного стиля кодирования. Несмотря на то, что компоненты могут использовать порты SystemC и экспортировать напрямую с основными интерфейсами TLM-2.0, это не рекомендуется.

6.13.13. Интерфейсы DMI и отладки

Интерфейс прямой памяти (DMI) и интерфейс отладки - это специализированные интерфейсы, отличные от транспортного интерфейса, обеспечивающие прямой доступ и отладочный доступ к области памяти, принадлежащей цели. После предоставления запроса на DMI интерфейс

DMI позволяет инициатору обходить обычный путь через компоненты межсоединений, используемые транспортным интерфейсом. DMI предназначен для ускорения транзакций с регулярной памятью в свободно-синхронизированной симуляции, тогда как интерфейс отладки транспорта предназначен для отладочного доступа без задержек или побочных эффектов, связанных с регулярными транзакциями. DMI имеет интерфейсы прямой (инициатор-к-цели) и обратный (цель к инициатору), тогда как отладка имеет только прямой интерфейс.

6.13.14. Комбинированные интерфейсы и сокеты

Блокирующие и неблокирующие транспортные интерфейсы объединяются с интерфейсами DMI и отладки транспорта в стандартном инициаторе и целевых сокетах. Все четыре интерфейса (два транспортных интерфейса, DMI и debug) могут использоваться параллельно для доступа к заданной цели (в соответствии с правилами, описанными в этом стандарте). Эти объединенные интерфейсы являются одним из ключей к обеспечению взаимодействия между компонентами с использованием стандарта TLM-2.0, а другой ключ - общая полезная нагрузка.

Стандартные целевые сокеты обеспечивают все четыре интерфейса, поэтому каждый целевой компонент должен эффективно реализовывать методы всех четырех интерфейсов. Тем не менее, конструкция блокирующих и неблокирующих транспортных интерфейсов вместе с предоставлением удобных сокетов для преобразования между двумя средствами, которые необходимы для данной цели, только реализуют либо блокирующий, либо неблокирующий транспортный метод, а не оба, в соответствии с требованиями скорости и точности модели. Данный инициатор может выбрать способ вызова через любой или все основные интерфейсы, опять же в соответствии с требованиями скорости и точности. Стили кодирования, упомянутые выше, помогают выбрать подходящий набор функций интерфейса. Как правило, слабо-временной инициатор будет вызывать

блокировку транспорта, DMI и отладки, тогда как инициатор с минимальным временем ожидания вызовет неблокирующий транспорт и отладка.

6.13.15. Примеры использования основных интерфейсов TLM-2.0

В дополнение к основным интерфейсам TLM-1, TLM-2.0 добавляет блокирующие и неблокирующие транспортные интерфейсы, интерфейс прямой памяти (DMI) и интерфейс отладки транспорта.

Транспортные интерфейсы

Транспортные интерфейсы являются первичными интерфейсами, используемыми для транспортировки транзакций между инициаторами, целями и компонентами межсоединений. Как блокирующие, так и неблокирующие транспортные интерфейсы поддерживают аннотацию времени и временную развязку, но только неблокирующий транспорт поддерживает несколько фаз в течение срока действия транзакции. Блокирующий транспорт не имеет явного аргумента фазы, и любая связь между блокирующим транспортом и фазами неблокирующего транспортного интерфейса является чисто условным. Только неблокирующий метод транспорта возвращает значение, указывающее, был ли использован обратный путь.

Транспортные интерфейсы и общая полезная нагрузка были разработаны для совместного использования в задачах быстрого абстрактного моделирования распределенных по памяти шин. Шаблоны транспортного интерфейса специализируются на типе транзакции, позволяющем использовать их отдельно от общей полезной нагрузки, хотя многие из преимуществ функциональной совместимости будут потеряны. Правила управления памятью объекта транзакции, упорядочения транзакций и допустимой последовательности вызова функций зависят от конкретного типа транзакции, переданного в качестве аргумента шаблона в транспортный интерфейс, который, в свою очередь, зависит от класса признаков протокола,

переданного в качестве аргумента шаблона для сокета (если используется сокет).

Блокирующий транспортный интерфейс

Транспортный интерфейс блокировки TLM-2.0 предназначен для поддержки слабо-временного стиля кодирования. Блокирующий транспортный интерфейс является подходящим, когда инициатор хочет завершить транзакцию с целью в течение одного вызова функции и единственными моментами, представляющими интерес, являются те, которые отмечают начало и конец транзакции. Транспортный интерфейс блокировки использует только прямой путь от инициатора к цели.

Транспортный интерфейс блокировки TLM-2.0 использует новый метод `b_transport`, который имеет один аргумент транзакции, переданный неконстантной ссылкой, и второй аргумент для аннотирования времени. Этот единственный аргумент используется как для вызова, так и для возврата из `b_transport`, чтобы указать время начала и конца транзакции, соответственно, относительно текущего времени моделирования

Метод блокирующего транспорта может немедленно возвращаться (а именно, на текущую фазу оценки SystemC) или может дать управление планировщику и вернуться к инициатору только в более позднюю точку во время моделирования. Хотя поток инициатора может быть заблокирован, другому потоку в инициаторе может быть разрешено вызывать `b_transport` до того, как первый вызов вернется, в зависимости от протокола.

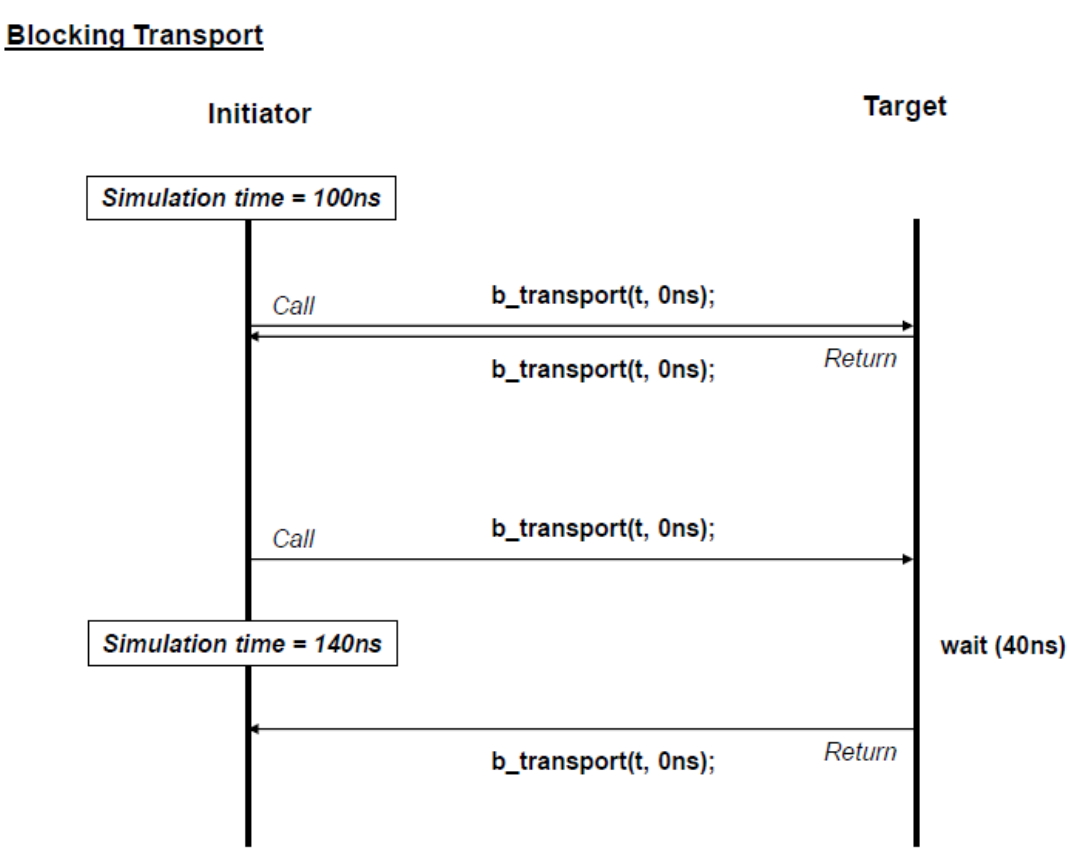


Рис. 5.5. Диаграмма работы блокирующего интерфейса

Рис. 6.25 Диаграмма работы блокирующего интерфейса

6.13.16. Пример неблокирующего интерфейса `at_1_phase`

Рассмотрим в качестве примера использования TLM-2.0 программу, приведенную в примерах SystemC-2.3.1. Это пример системы AT – (Approximately Timed – Примерное время).

Цель состоит в том, чтобы проиллюстрировать:

- Применение TLM 2.0 в реальной системе;

-Аннотированную неблокирующую (NB) опцию неблокирующего стиля;

- NB аннотированное время, которое называется «1 фазой»,

- Простейшую версию неблокирующего / AT стиля.

Возможные применения:

- исследование архитектуры;

- предварительная разработка программного обеспечения

Блок-схема примера показана на рис. 6.26.

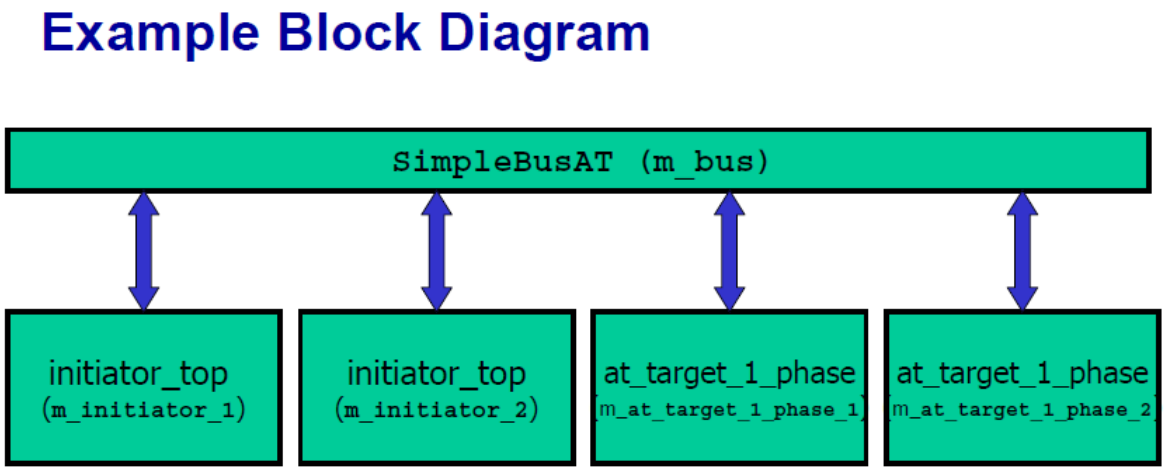


Рис. 6.26. Блок-схема примера

Модель имеет два инициатора, две цели и маршрутизатор. Цели являются разными фазовыми объектами (1 и 2).

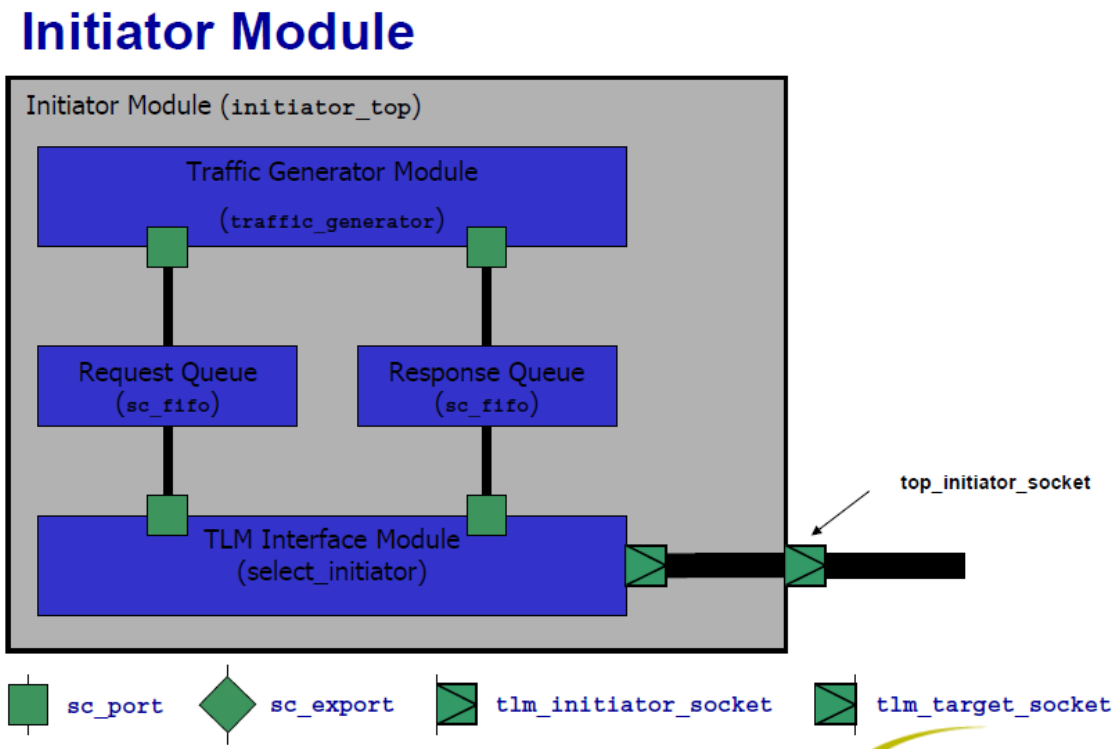


Рис. 6.27. Модуль инициатора

Инициатор содержит модель генератора передачи (Traffic Generator Module), очередь запросов (Request Queue) и очередь ответов (Response

Queue), выполненные как sc_fifo, и интерфейсный TLM модуль с выбором инициатора.

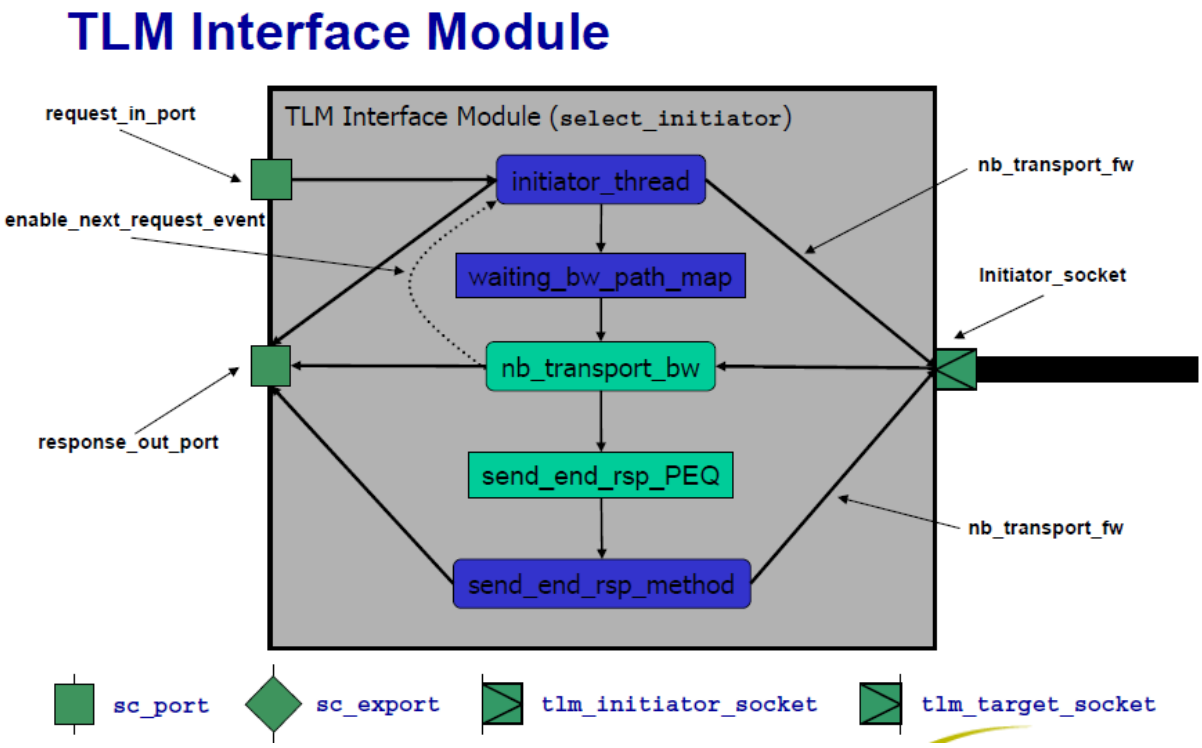


Рис. 6.28. Модуль интерфейса

Модуль TLM интерфейса содержит входной порт запросы, выходной порт ответа, сокет инициатора. Входному запрос инициирует поток инициатора и вызывает метод **nb_transport_fw**, который передается на сокет инициатора, на выходной порт ответа, и на метод ожидания по карте прямой передачи. С сокета модуля интерфейса поступает ответный сигнал на и запускает метод обратного интерфейса nb_transport_bw, с которого выдается сигнал на выходной порт ответа, разрешение на следующий запрос и на метод send_end_rsp_method, посылающий на сокет и на выходной порт сигнал конца ответа.

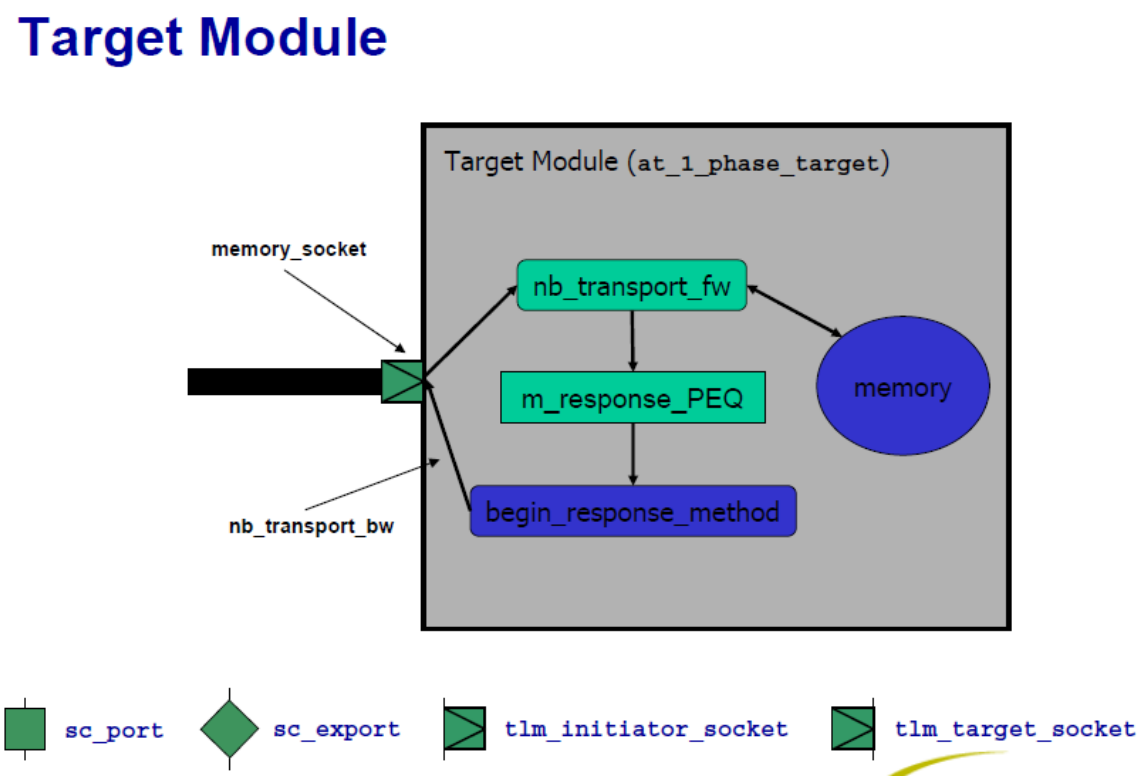


Рис. 6.29. Модуль цели

Модуль цели имеет сокет, через который по интерфейсу nb_transport_fw принятые данные передаются в память и формируется ответ для посылки инициатору.

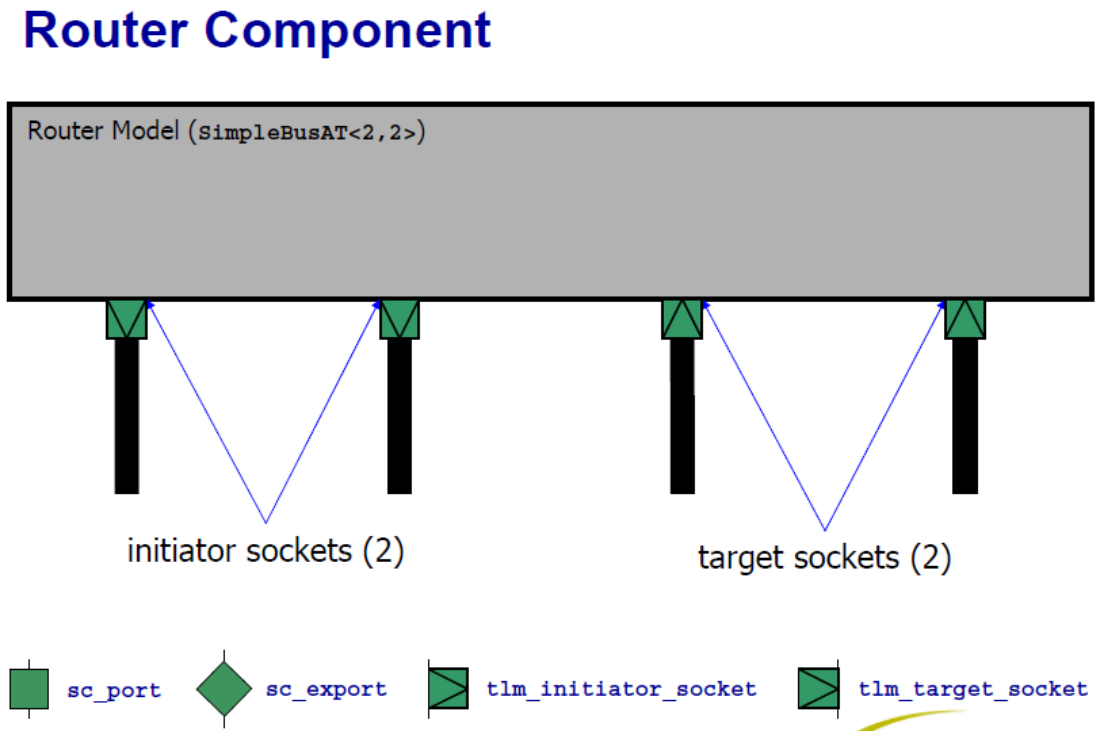


Рис. 6.30. Компонент маршрутизатор

Маршрутизатор построен по принципу простой шины с примерным временем и сокеты инициаторов и целей.

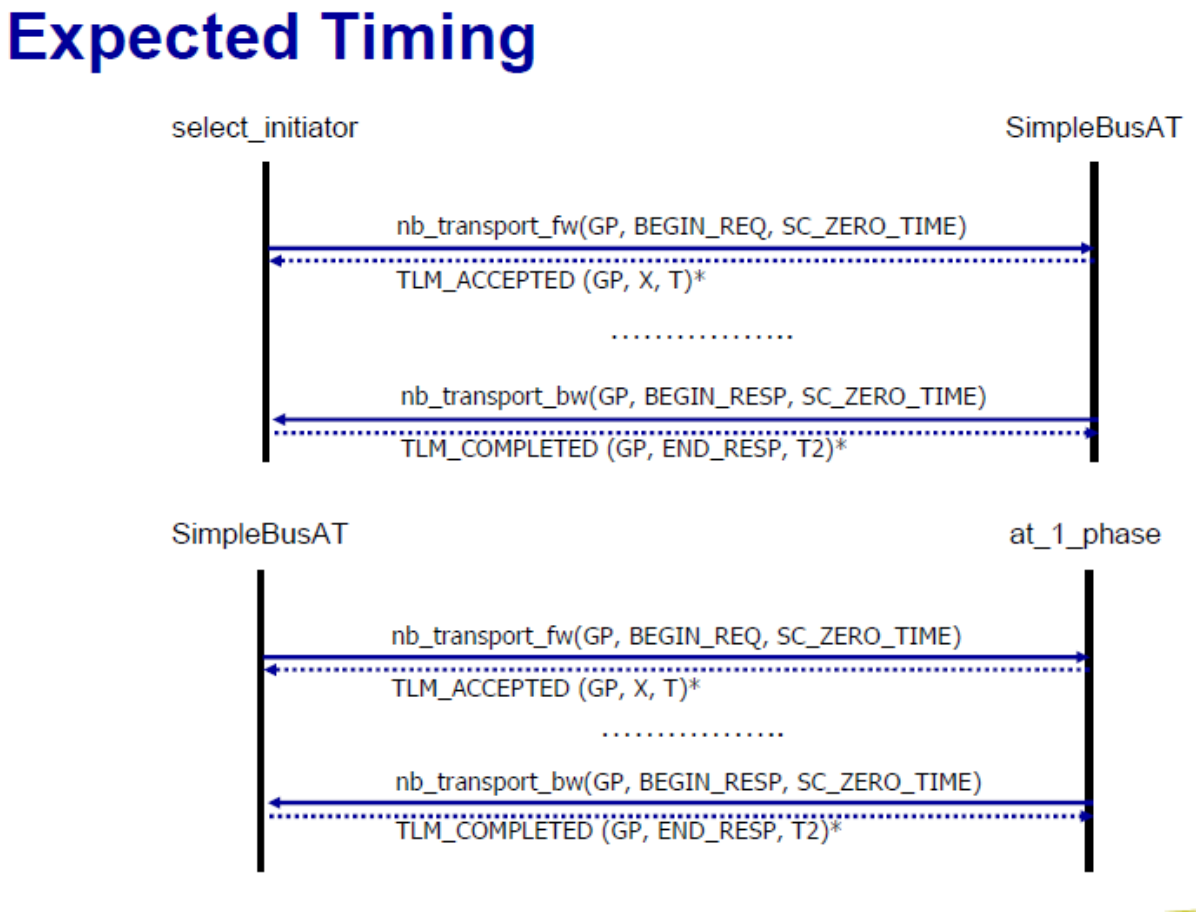


Рис. 6.31. Ожидаемая временная диаграмма

Основная литература:

1. Адамов Ю.Ф. Проектирование систем на кристалле. - М.: МИЭТ, 2005. – 112 с.

2. Алехин В.А. SystemC. Моделирование электронных систем. – М.: Горячая линия – Телеком, 2018. – с. 320.

3. Алехин В.А. OrCAD 17.2. Анализ и проектирование электронных устройств. – М.: Горячая линия – Телеком, 2019. – с. 328.

4. Алехин В.А. Микроконтроллеры PIC. Основы программирования и моделирования в интерактивных средах MPLAB IDE, microC, TINA, Proteus. Практикум. – М.: Горячая линия – Телеком, 2016. – с. 248.

5. Быковский С.В., Горбачев Я.Г., Ключев А.О., Пенской А.В., Платунов А.Е. Сопряжённое проектирование встраиваемых систем. (Hardware/Software Co-Design). Часть 1. Учебное пособие. – СПб.: Университет ИТМО, 2016. – 109 с.

6. Быковский С.В., Горбачев Я.Г., Ключев А.О., Пенской А.В., Платунов А.Е. Сопряжённое проектирование встраиваемых систем. (Hardware/Software Co-Design). Часть 2. Учебное пособие. – СПб.: Университет ИТМО, 2016. – 106 с.

7. Платунов А.Е, Постников Н.П. Высокоуровневое проектирование встраиваемых систем. Часть 1. – СПб.: НИУ ИТМО, 2011. – 121 с.

8. Платунов А.Е, Постников Н.П. Высокоуровневое проектирование встраиваемых систем. Часть 2. – СПб.: НИУ ИТМО, 2013. – 172 с.

9. Ключев А.О., Кустарев П.В., Ковязина Д.Р., Петров Е.В. Программное обеспечение встроенных вычислительные систем. – СПб.: СПбГУ ИТМО, 2009. – 212 с.

10. Гончаровский О.В. Проектирование встроенных управляющих систем реального времени: учеб. пособие / О.В. Гончаровский, Н.Н.Матушкин, А.А. Южаков – Пермь: Изд-во Перм. нац. исслед. политехн. ун-та, 2013. – 165 с

11. Губарев В.А., Воронков С.О. Технология проектирования устройств и систем ВТ средствами САПР (высокоуровневое системное моделирование), МГТУ МИРЭА, 2012.

12. Рубанов В.В. Обзор методов описания встраиваемой аппаратуры и построения инструментария кросс-разработки. Сборник Трудов Института системного программирования РАН. – М. :ИСП РАН. – с.7-40.

13. Суворова Е.А., Шейнин Ю.Е. Проектирование цифровых систем на VHDL– СПб: БХВ – Санкт-Петербург, 2003.

14. В. Немудров, Г.Мартин. Системы на кристалле. Проектирование и развитие. Техносфера, М. 2004.

Дополнительная литература

1. Thorsten Grotker, Stan Liao, Grant Martin, Stuart Swan. System Design with SystemC. Created in the USA, 2002.

2. David C. Black, Jack Donovan, Bill Bunton, Anna Keist. SystemC: From the Ground Up. Second Edition. -New York , Dordrecht Heidelberg London.: Springer. - 2010, p.291.

3. SystemC Compiler RTL User and Modeling Guide. Version 2001.08. - Printed in the U.S.A. : Synopsys, 2001. –p. 222.

4. Abderazek Ben Abdallah. Advanced Multicore Systems-On-Chip Architecture, On-Chip Network, Design: Springer Nature Singapore Pte Ltd. 2017. – p.292.

5. Embeded software for SoC. Edited by Ahmed Amine Jerraya. University of Kaiserlautern, Germany. 2004 Springer Science + Business Media, Inc.

6. Brian Bailey · Grant Martin. ESL Models and their Application Electronic System Level Design and Verification in Practice. : Springer Science+Business Media, LLC 2010. – p. 466

7. Changyi Gu. Building Embedded Systems. Programmable Hardware. : San Diego, California, USA, 2016. – p. 337

Электронные ресурсы:

1. Алехин В.А. Технологии проектирования устройств и систем вычислительной техники в среде САПР. Часть 2. SystemC. М. : МИРЭА, 2016.

Доступ: https://yadi.sk/d/KtNTEgobxD3xg