

**К. Д. Любавин, Д. В. Тельпухов,
А. А. Беляев, П. А. Кузьмин**

**Лабораторный практикум по курсу
«Проектирование СнК с
программируемой архитектурой»**

Утверждено редакционно-издательским советом университета

Москва 2023

УДК 621.3.049.77+004.3'12 (076.5)

И46

Рецензент: канд. техн. наук *Д. А. Булах*

Любавин К. Д., Тельпухов, Беляев А. А., Д. В., Кузьмин П. А.

И46 Лабораторный практикум по курсу «Проектирование СнК с программируемой архитектурой». - М.: МИЭТ, 2023.- с.: ил.

Предназначен для приобретения практического опыта при функционально-логическом проектировании систем на кристалле с программируемой архитектурой в САПР Synopsys VCS и навыков разработки с использованием языка описания аппаратуры SystemVerilog и языка программирования С.

В лабораторных работах рассматривается один из базовых системных интерфейсов – AMBA APB, методы проектирования периферийных блоков, обладающих системными интерфейсами, а также базовая концепция взаимодействия между встроенными процессорными исполнительными ядрами и периферийными модулями. Затрагиваются вопросы подготовки исходных данных, файлов и модулей, интеграция разработанного модуля в исходный проект СнК, написание исполняемой программы для встроенного процессорного ядра, компиляция проекта в САПР Synopsys VCS, а также анализа результатов при работе с системой на кристалле.

Для студентов, изучающих дисциплину «Проектирование СнК с программируемой архитектурой».

© МИЭТ, 2023

Введение

В настоящий момент при разработке систем на кристалле (СнК) широко распространена практика интеграции в систему одной или нескольких процессорных подсистем (или отдельных процессорных исполнительных ядер).

Данный подход обладает существенными преимуществами, поскольку позволяет изменять логику работы устройства путём изменения программного кода, исполняемый процессорными подсистемами, тем самым, не затрагивая изменения аппаратной части устройства. К минусам данного подхода относится снижение производительности устройства, которое обуславливается исполнением части логических функций не оптимизированную под конкретную задачу аппаратных блоков процессорных структур.

При проектировании таких СнК необходимо разрабатывать все периферийные модули устройства с добавлением системных интерфейсов, через которые интегрированный процессор сможет взаимодействовать с разработанными модулями. В данный момент одними из самых распространённых системных интерфейсов общего назначения являются интерфейсы семейства Advanced Microcontroller Bus Architecture (AMBA). К данному семейству относятся интерфейсы серии AXI, APB, ACE, ATB, и AHB. При этом модули могут иметь отличные друг от друга интерфейсы, в зависимости от требуемой пропускной способности и типа транзакций. Для подключения различных интерфейсов друг к другу существуют специальные модули, которые транслируют транзакции из одного протокола в другой. Такие модули называют bridge-модулями.

В начале разработки таких СнК закладывается так называемая адресная карта устройства, регламентирующая базовый адрес и размер выделяемой памяти для адресации для каждого из присутствующих периферийных модулей. Размер выделяемой памяти для каждого из модулей зависит от самого модуля и должен быть соответствующего размера, чтобы иметь доступ до всех функциональных узлов, управление которыми производится через системный интерфейс модуля.

Исключением из правил для расчёта размера выделяемой памяти под адресацию до конкретного блока являются внешние периферийные модули, которые используются для подключения к внешним устройствам (например, к ПК или серверным станциям) и позволяют взаимо-

действовать с памятью внешнего устройства. Примером таких внешних периферийных модулей является PCI-Express, широко используемый для подключения различных носителей информации, вычислительных сопроцессоров и других устройств.

После того как спроектирована адресная карта и определены размеры памяти для адресации до всех необходимых периферийных модулей реализуется один или несколько специальных interconnect-модулей (модулей межсоединений), которые транслируют транзакцию процессора в нужный модуль, опираясь на адрес этой самой транзакции процессора.

С появлением HDL SystemVerilog процесс разработки цифровых интегральных схем существенно упростился за счёт нововведений данного языка. К ним можно отнести следующие нововведения:

- 1) Введение ООП;
- 2) Введение C-based конструкций: структур, перечислений (enum), динамических массивов;
- 3) Приведение типов;
- 4) Дополнительные циклы и операторы: foreach, return, break, continue и т.д.;
- 5) Появились интерфейсы как отдельная конструкция языка;
- 6) Утверждения (assertions);
- 7) Новые типы данных: logic, void, bit, byte, longint, shortint и т.д.

Благодаря данным нововведениям процесс проектирования ЦИС и СнК с программируемой архитектурой перешёл на новый уровень.

В рамках данного лабораторного практикума студенты ознакомятся с одним из базовых интерфейсов семейства AMBA – APB3, разработают периферийный модуль с данным интерфейсом с использованием языка описания аппаратуры (*Hardware Description Language, HDL*) SystemVerilog, произведут подключение в TOP-уровень учебного проекта СнК, а также реализуют исполняемую программу на языке программирования (ЯП) C для взаимодействия процессора RISC-V с разработанным модулем.

Лабораторная работа № 1

Основы проектирования ведомых модулей обладающих системным интерфейсом

Цель работы: Изучить особенности описания интерфейсов с использованием HDL SystemVerilog. Изучить протокол системного интерфейса AMBA APB3. Разработать HDL модуль с использованием HDL SystemVerilog, реализующий APB3-Slave (ведомое) устройство. Приобрести практические навыки для работы в САПР Synopsys VCS.

Продолжительность работы: 4 ч.

Теоретические сведения

Конструкция Interface в HDL SystemVerilog

Применительно к HDL SystemVerilog конструкция `interface` является методом инкапсуляции сигналов в логическую группу с целью упрощения взаимодействия с необходимой группой сигналов, объединённых по критерию функционального назначения. Данная конструкция используется для как плавного перехода между верификационным окружением и самим устройством, так и для соединения различных модулей внутри устройства, позволяя тем самым облегчить повторное использование кода в рамках одного устройства.

На самом низком уровне конструкция `interface` представляет собой именованный набор переменных. Интерфейс инстанцируется в проекте и может быть подключен к интерфейсным портам других созданных модулей, интерфейсов и программ. Доступ к интерфейсу можно получить через порт как к отдельному элементу, а при необходимости можно ссылаться на отдельные его компоненты.

При использовании конструкции `interface` в синтезируемом подмножестве необходимо указывать направление подключения сигналов, содержащихся внутри интерфейса. Для этого используется конструкция `modport`. Синтаксис конструкции `interface` и примеры использования представлены на рисунках 1.1 – 1.3.

```

1  interface SPI_if#(
2      parameter    SLAVES_NUM    = 2
3  )(
4      input        CLK,
5      input        RSTN
6  );
7
8      /* --- Interface signals --- */
9      logic        MOSI;
10     logic        MISO;
11     logic    [SLAVES_NUM-1:0]    SS;
12
13     /* --- Modports --- */
14     modport Master(
15         input        MISO,
16         output       MOSI,
17         output       SS
18     );
19
20     modport Slave(
21         input        MOSI,
22         input        SS,
23         output       MISO
24     );
25
26     endinterface : SPI_if

```

Рисунок 1.1 – Пример реализации конструкции interface

Конструкция interface синтаксически похожа на конструкцию module в HDL Verilog/SystemVerilog, обладает аналогичным механизмом параметризации.

Относительно конструкции interface, модули делятся на ведущих (Master или Manager) и ведомых (Slave или Subordinate).


<pre> 1 module Slave (2 input CLK, 3 input RSTn, 4 input MOSI, 5 output MISO, 6 input SS 7); 8 9 /* -- Slave logic --*/ 10 11 endmodule : Slave </pre>		<pre> 1 module Slave (2 SPI_if SPI 3); 4 5 /* -- Slave logic --*/ 6 7 endmodule : Slave </pre>
--	---	--

Рисунок 1.2 – Пример использования interface в порт-листе модуля

```

1  always_comb rx_write  = mosi_reg[4];
2  always_comb rx_read   = mosi_reg[5];
3  always_comb addr      = mosi_reg[7:6];
4
5  /* --- Receive data --- */
6  always_ff@(posedge SPI.CLK or negedge SPI.RSTN)
7  if (!SPI.RSTN)
8      mosi_reg <= 8'h00;
9  else if (rx_active)
10     mosi_reg <= {mosi_reg[7:0], SPI.MOSI};
11
12  /* --- Write Transaction --- */
13  always_ff@(posedge SPI.CLK or negedge SPI.RSTN)
14  if (!SPI.RSTN)
15     memory <= '0;
16  else if (rx_done && rx_write)
17     memory[addr] <= mosi_reg[3:0];
18
19  /* --- Read Transaction --- */
20  always_ff@(posedge SPI.CLK or negedge SPI.RSTN)
21  if (!SPI.RSTN)
22     miso_reg <= 8'h0;
23  else if (rx_done && rx_read)
24     miso_reg <= {mosi_reg[7:4], memory[addr]};
25  else
26     miso_reg <= miso_reg << 1;
27
28  always_comb SPI.MISO = miso_reg[7];

```

Рисунок 1.3 – Пример использования interface на примере SPI-slave логики

Системные интерфейсы

При проектировании СнК с программируемой архитектурой для взаимодействия процессоров с периферийными устройствами, такими как DDR, UART, SPI, PCI-Express и т.д. используются системные интерфейсы. Системные интерфейсы являются совокупностью нескольких сигналов, объединённых в логические группы и обладающие строго описанными правилами поведения, называемыми протоколом системного интерфейса. Данный подход позволяет достаточно быстро организовать эффективное взаимодействие компонентов устройств друг с другом, что является одним из основных вопросов при проектировании архитектуры любого устройства.

Одним из самых распространённых семейств интерфейсов, является семейство AMBA (Advanced Microcontroller Bus Architecture), разработка компании ARM, включающая в себя такие системные интерфейсы как AXI (Advanced eXtensible Interface), ACE (AXI Coherency Extensions), APB (Advanced Peripheral Bus), AHB (Advanced High-Performance Bus), ATB (Advanced Trace Bus), ASB (Advanced System Bus).

Семейство AMBA обладает открытым набором требований и правил проектирования (протоколом), что позволяет беспрепятственно реализовать данные системные интерфейсы в любом устройстве, без ограничений по лицензированию. Данные системные интерфейсы обладают следующими преимуществами:

1. Масштабируемость. Интерфейсы семейства AMBA разработаны с учетом возможности масштабирования, что позволяет легко интегрировать дополнительные компоненты в систему.
2. Сокращение времени разработки: интерфейсы AMBA имеют детально описанный стандарт и правила проектирования, которые упрощают процесс проектирования.
3. Эффективная передача данных: интерфейсы AMBA используют архитектуру на основе шины, которая поддерживает передачу данных с высокой пропускной способностью и эффективную связь между компонентами.
4. Гибкость. Интерфейсы AMBA обладают гибкой параметризацией и поддерживают широкий перечень устройств, от встраиваемых периферийных модулей с низким энергопотреблением до высокопроизводительных DSP блоков.

Протокол системного интерфейса AMBA APB3

AMBA APB3 (*Advanced Peripheral Bus 3*) — это протокол шины с низким энергопотреблением, малой задержкой и низкой пропускной способностью, используемый для подключения низкоскоростных периферийных устройств внутри СнК или к микроконтроллеру. В связи с простотой проектирования и маленьким количеством используемых логических элементов, он используется для таких устройств, как таймеры, GPIO и регистры управления системой. Протокол APB3 поддерживает пакетную передачу, несколько ведомых шин и обеспечивает простую интеграцию с другими протоколами AMBA.

Таблица 1.1

Назначение сигналов AMBA APB3 интерфейса

Сигнал	Master	Slave	Описание
PCLK	input	input	Тактовый сигнал
PRESETn	input	input	Сигнал сброса (активный уровень 0)
PADDR	output	input	Адрес
PSEL	output	input	Выбор определенного slave устройства
PENABLE	output	input	Инициализация транзакции
PWDATA	output	input	Данные для записи
PWRITE	output	input	Строб записи (1 – запись, 0 – чтение)
PREADY	input	output	Флаг окончания обработки транзакции
PRDATA	input	output	Данные для чтения
PSLVERR	input	output	Флаг наличия ошибки

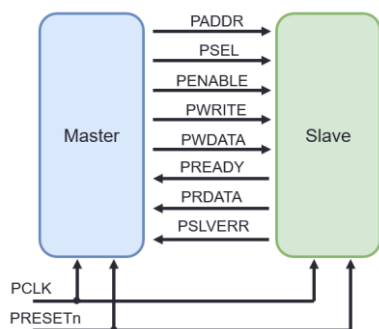


Рисунок 1.4 – Подключение APB3-совместимых устройств

Для проведения транзакций в AMBA APB3 существуют определенные требования для каждого сигнала. При необходимости передачи данных, ведущее (master) устройство должно выбрать необходимое ему ведомое устройство путём поднятия нужного бита сигнала PSEL. Во все ведомые устройства подключается сигнал PSEL, имеющий разрядность 1 бит, однако на стороне ведущего устройства размерность сигнала PSEL соответствует количеству ведомых устройств, подключенных к данному экземпляру интерфейса, но не более 16 бит. Также существует ограничение и на размерность сигналов PRDATA, PWDATA. Они должны быть эквиваленты друг другу по размеру, а также их размер должен соответствовать 2^N , где N может принимать значение от 3 до 9, иначе говоря, от 8 бит до 512 бит.

После того, как ведущим устройством определено целевое конечное устройство, оно должно через 1 период тактового сигнала PCLK поднять в активный уровень сигнал PENABLE, сигнализируя о начале транзакции между ведущим и целевым ведомым устройством. Одновременно с фиксацией выбора целевого устройства, ведущее устройство должно установить и зафиксировать до окончания транзакции адрес обращения, тип обращения, данные для записи (если тип обращения соответствует записи) и ожидать сигнала окончания обработки транзакции со стороны ведомого устройства.

Сигналом окончания обработки транзакции является поднятый в активный уровень сигнал PREADY, которым управляет ведомое устройство. После того, как ведомое устройство сообщило об окончании транзакции, ведущее устройство должно опустить сигналы PSEL, PENABLE минимум на 1 период тактового сигнала PCLK.

Для детектирования успешности транзакции существует флаг наличия ошибки PSLVERR. Данный флаг имеет валидное значение только во время активного уровня сигнала PREADY. Он используется для передачи информации о некорректной транзакции ведущему устройству. Под некорректной транзакцией подразумевается транзакция, совершенная по некорректному адресу, либо по запрошенному адресу обращения запрещено совершать транзакции чтения или записи.

При совершении транзакции чтения, данные передаются в сигнале PRDATA и являются валидными только во время активного уровня сигнала PREADY и одновременно низкого уровня сигнала PSLVERR.

Примеры транзакций чтения и записи по системе интерфейсу AMBA APB3 представлены на рисунках 1.5 – 1.6.

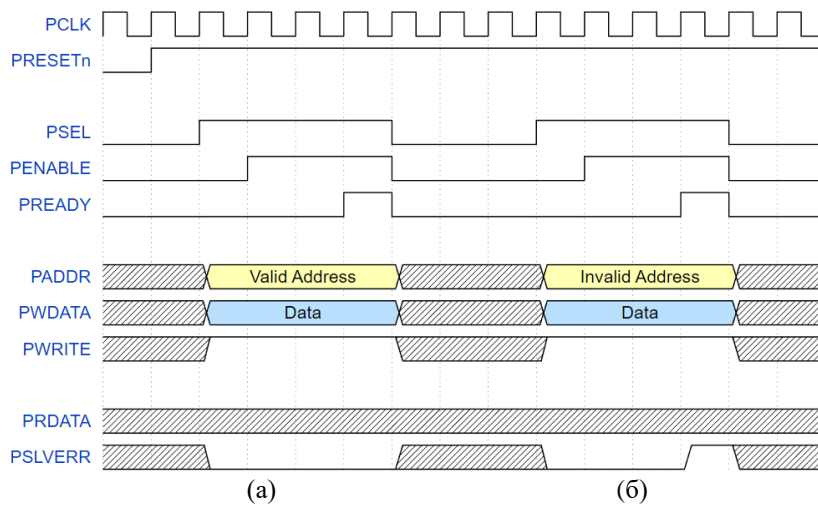


Рисунок 1.5 – Транзакция записи:
(а) – корректная, (б) – с флагом ошибки

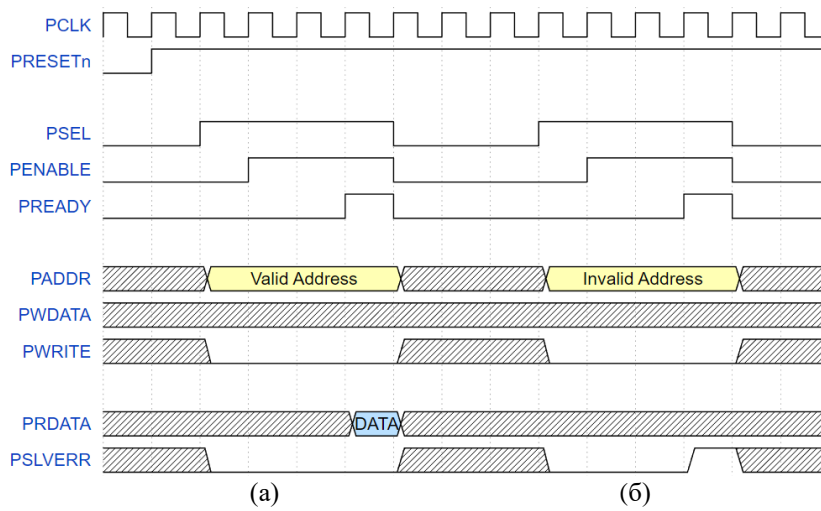


Рисунок 1.6 – Транзакция чтения:
(а) – корректная, (б) – с флагом ошибки

Моделирование в САПР Synopsys VCS

Synopsys VCS (*Verification Compiler Simulator*) – это высокопроизводительный САПР, используемый для верификации цифровых схем и систем. Он позволяет производить функциональную и временную верификацию, а также верификацию соответствия стандартам.

Данный САПР поддерживает все актуальные языки описания аппаратуры, такие как Verilog, SystemVerilog и VHDL, а также имеет интеграцию с другими инструментами Synopsys, такими как Design Compiler, PrimeTime и SpyGlass, что позволяет проводить более точную верификацию.

Среди возможностей Synopsys VCS можно выделить:

- 1) Быструю симуляцию цифровых схем и систем различной сложности и размера;
- 2) Поддержку всех основных языков описания аппаратуры и стандартов верификации;
- 3) Интеграцию с другими инструментами Synopsys;
- 4) Возможность симуляции на уровне системы (SystemC);
- 5) Поддержку параллельной симуляции для ускорения процесса верификации.

В Synopsys VCS также есть возможность работы с Assertion-Based Verification (ABV) – это методология, которая позволяет определить формальные спецификации для верификации цифровых схем. ABV используется для проверки того, что цифровые схемы соответствуют требованиям и спецификациям, и позволяет выявить ошибки и проблемы на ранних стадиях разработки.

Synopsys VCS широко используется в индустрии разработки цифровых схем и систем, а также в учебных целях. Он является одним из наиболее популярных и мощных симуляторов для верификации цифровых схем.

Для запуска симуляции в САПР Synopsys VCS в ОС Linux необходимо скомпилировать исходные HDL SystemVerilog файлы используя команду `vcs`, а также аргументы, указывающие на необходимые файлы для компиляции, шаг и точность моделирования (`timescale`), тип языка описания аппаратуры и необходимость графического интерфейса. Пример команды запуска компиляции:

```
vcs -sverilog -RI -PR -timescale=1ns/1ps  
hdl_file1.sv hdl_file2.sv testbench.sv
```

После запуска данной команды пройдет процесс компиляции предоставленных САПР HDL файлов. После успешной компиляции исходных файлов необходимо открыть пользовательский графический интерфейс (GUI) для проведения симулятора. Для этого используется команда

```
./simv -gui
```

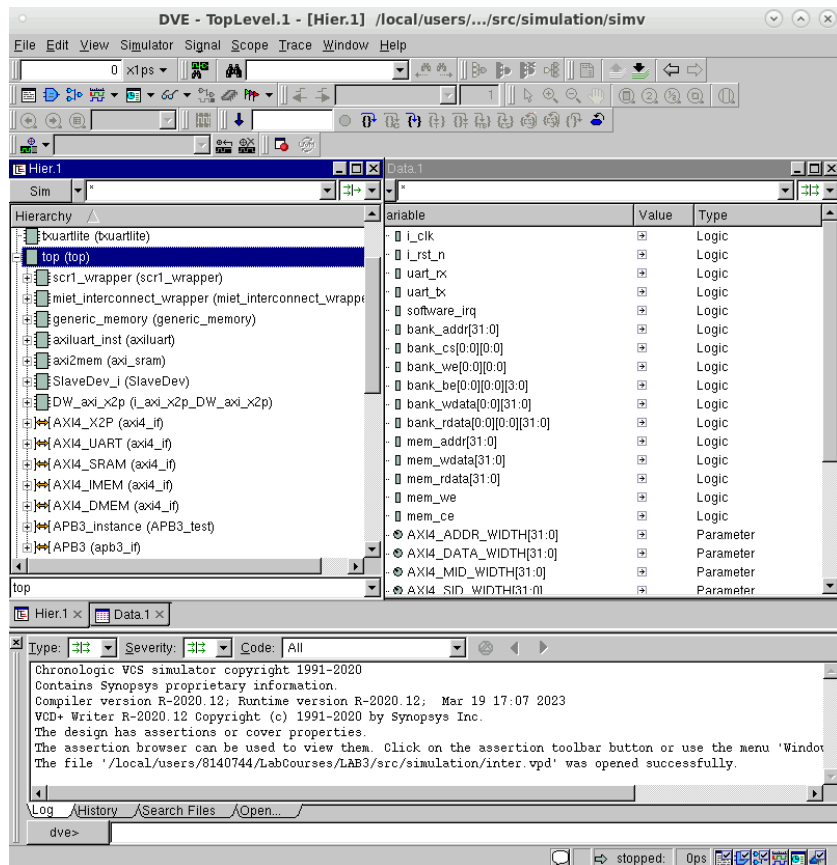


Рисунок 1.7 – Общий вид графического интерфейса VCS с демонстрацией иерархии запущенного проекта

Для проведения симуляции, необходимо выбрать необходимые для наблюдения сигналы и добавить их на временную диаграмму (Wave-Form). Выбираем необходимый модуль в окне Hierarchy, затем выделяем необходимые сигналы в окне Variable и в контекстном меню выбираем Add To Waves → New Wave View (рисунок 1.8).

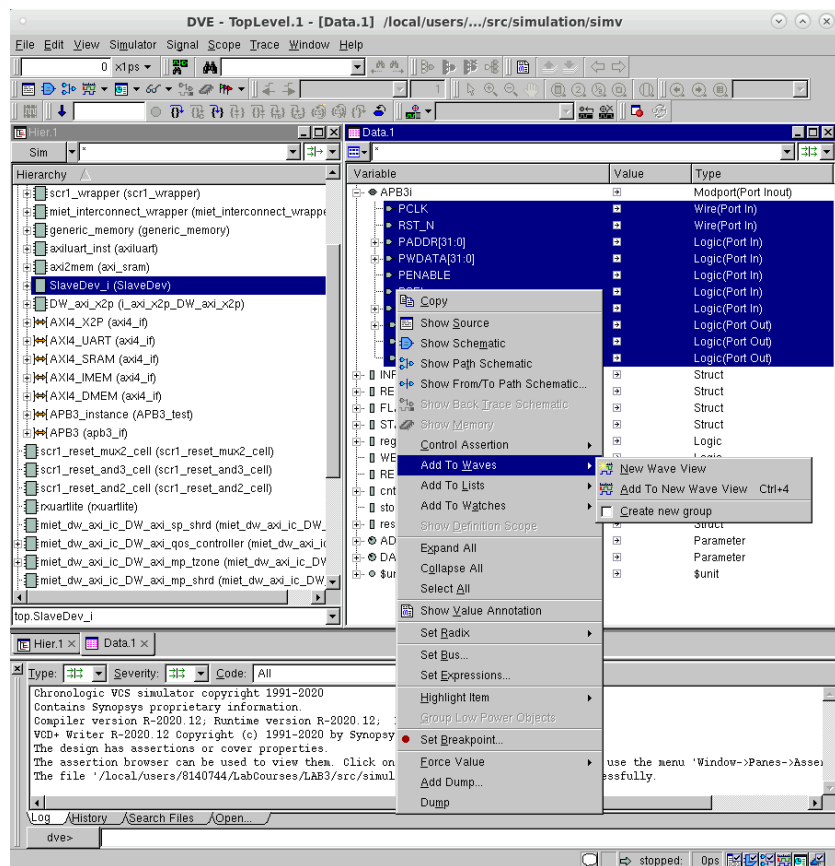


Рисунок 1.8 – Добавление сигналов для моделирования на временные диаграммы CAIP Synopsys VCS

После того, как необходимые сигналы были добавлены, необходимо нажать **Start / Continue** (рисунок 1.9) для начала симуляции проекта.

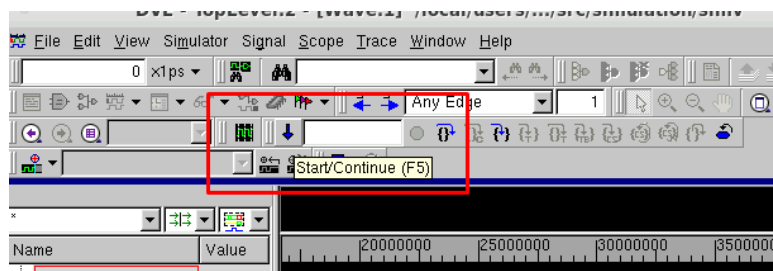


Рисунок 1.9 – Кнопка начала симуляции проекта

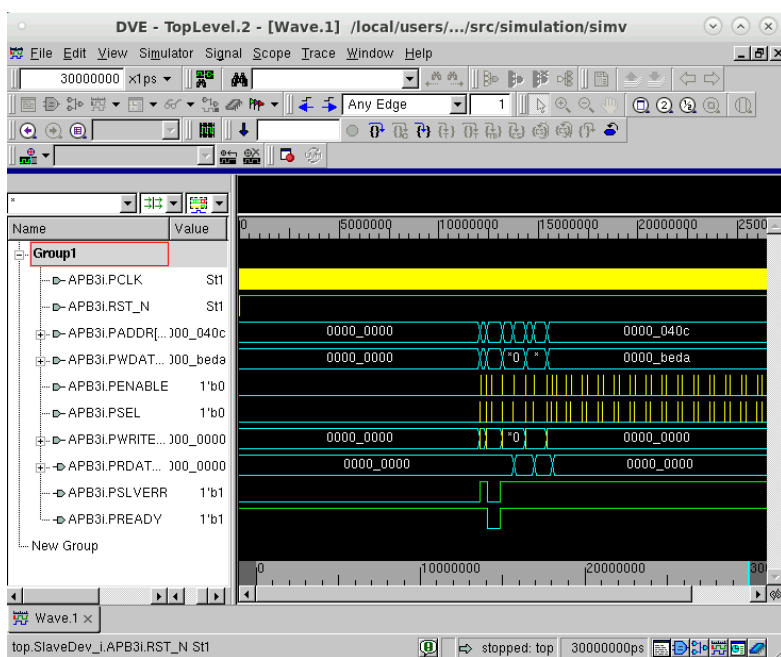


Рисунок 1.10 – Пример графического интерфейса после симуляции проекта

В случае, если пользователю необходимо внести правки в исходное HDL описание проекта, необходимо, после внесения правок перекомпилировать проект. Для этого в панелью меню необходимо вызвать раздел **Simulator** → **Rebuild and Start**, затем в вызванном окне установить “**Use script generated by VCS**”, убедиться, что в выпадающем меню “**Action on OK**” установлен вариант “**Rebuild and Start**” и нажать **OK** (Рисунок 1.11).

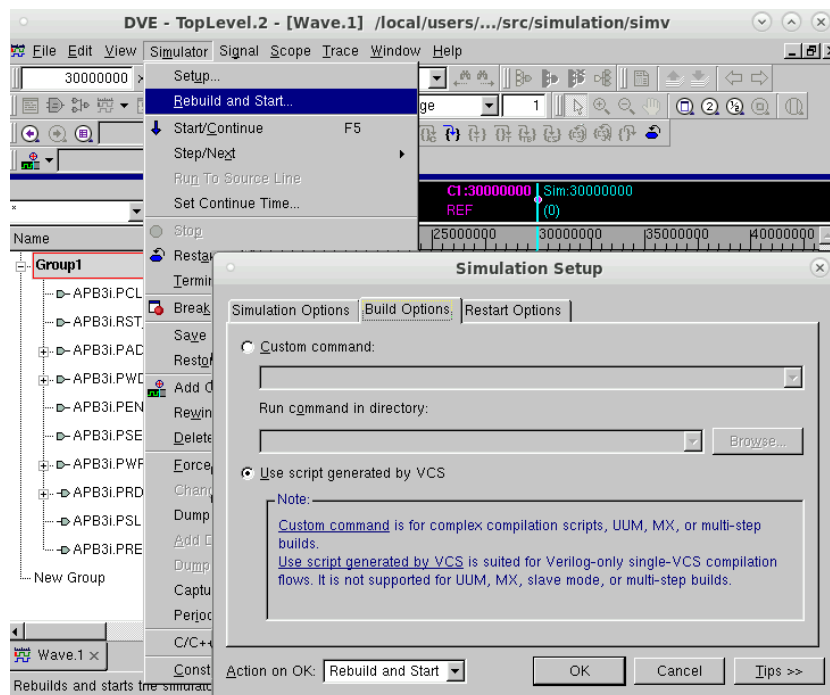


Рисунок 1.11 – Алгоритм перекомпиляции проекта при внесении правок без закрытия графического интерфейса САПР Synopsys VCS

Методика выполнения лабораторной работы

1. Изучить применение конструкции `interface` в HDL SystemVerilog;
2. Изучить перечень и назначение сигналов интерфейса AMBA APB3 и его протокол для взаимодействия с модулями;
3. Самостоятельно описать интерфейс для AMBA APB3 с использованием HDL SystemVerilog и конструкции `interface`;
4. Разработать APB3-Slave модуль, который будет выполнять следующие задачи через интерфейс APB3 в соответствии с протоколом:
 - a. Записывать данные в регистр памяти, основываясь на адрес регистра;
 - b. Считывать данные из регистра памяти, основываясь на адрес регистра.
 - c. Количество регистров – 4. При попытке записать в 5 регистр и более использовать флаг PSLVERR.
5. Разработать тестовое окружение с использованием HDL SystemVerilog;
6. Провести моделирование в САПР Synopsys VCS.

Лабораторное задание

1. Выполнить и освоить действия в соответствии с методикой выполнения лабораторной работы, в процессе которой необходимо:
 - Провести подготовительные действия по созданию исходных файлов для выполнения лабораторной работы;
 - Подготовить по предлагаемой методике выполнения лабораторной работы HDL файлы проекта с использованием SystemVerilog;
 - Произвести компиляцию проекта с использованием САПР Synopsys VCS; при появлении ошибок внести необходимые исправления и повторить компиляцию, добиться безошибочного результата компиляции;
 - Выполнить симуляцию устройства;
 - Получить результаты симуляции в виде временных диаграмм.

2. Проанализировать полученные результаты симуляции; убедиться в правильности работы устройства в соответствии с APB3 протоколом и требованиями к разработке из п. 4 методики выполнения лабораторной работы. При выявлении ошибок попытаться их выявить и устранить; повторить симуляцию, добиться положительных результатов;
3. Составить отчёт по работе;
4. Представить результаты работы преподавателю.

Требования к отчету

Отчет должен содержать:

1. Название и цель работы;
2. Файл интерфейса AMBA APB3 с использованием конструкции `interface HDL SystemVerilog`;
3. Файлы реализованных модулей и тестового окружения на языке SystemVerilog;
4. Ответы на контрольные вопросы.

Контрольные вопросы

1. Зачем используется конструкция `interface` в HDL SystemVerilog?
2. Для чего необходимо указывать `modport` внутри конструкции `interface`?
3. Как используется конструкция `interface` внутри модуля?
4. Как происходит процесс чтения и записи данных в системном интерфейсе AMBA APB3?
5. Когда считанные данные через интерфейс AMBA APB3 считаются валидными?

Лабораторная работа № 2

Проектирование регистровой карты устройства, обладающего системным интерфейсом

Цель работы: Изучить конструкции `struct`, `typedef`, `package` в HDL SystemVerilog и приобрести навыки их корректного использования. Изучить методику разработки регистровых карт. Реализовать регистровую карту для управления устройством, предназначенного для расчёта циклически избыточного кода (CRC).

Продолжительность работы: 4 ч.

Теоретические сведения

Конструкция `struct`

Конструкция `struct` в HDL SystemVerilog позволяет объединять сигналы по логическому признаку в структуры. На всю структуру можно ссылаться как на одно целое, так и на отдельные её части по имени поля, используя «иерархическое» обращение (аналогичное обращение используется для конструкций `interface`). Это существенно увеличивает удобство, читаемость и скорость написания RTL-кода.

Существуют структуры типа `packed` и `unpacked`. Применительно к SystemVerilog массивы объектов, которые обладают признаком `packed` представлены в виде единого вектора, в котором находятся все переменные из подмножества массива. В случае `unpacked`, каждая переменная является отдельным независимым вектором. При объявлении структуры без присваивания конкретного типа она по умолчанию будет представлена как `unpacked`. К синтезируемому подмножеству конструкций HDL SystemVerilog относится только лишь структура типа `packed`. Структуры типа `unpacked` являются не синтезируемыми, однако широко распространены в тестовых окружениях, необходимых для функциональной верификации устройства.

Синтаксис конструкции `struct` представлен ниже.

```
struct [тип структуры (packed/unpacked)] {  
    список переменных  
} [название структуры];
```

В представленном ниже примере (рисунок 2.1), переменные, которые относятся к одной логической группе (в данном случае – параметры фруктов в магазине) объединены в структуры типа `packed`. Данный пример демонстрирует упрощение взаимодействия с логическими группами сигналов.

```
1  localparam  YES = 1'b1,  
2             NO  = 1'b0;  
3  
4  struct packed {  
5      logic  [31:0] fruit_id;  
6      bit    is_on_sale;  
7      byte   expiry_date;  
8  } apple, orange;  
9  
10 always_comb apple.expiry_date  = 8'h7;  
11 always_comb apple.is_on_sale   = YES;  
12 always_comb apple.fruit_id     = 32'h0000_01FCD;  
13  
14 always_comb orange.expiry_date  = 8'h9;  
15 always_comb orange.is_on_sale   = NO;  
16 always_comb orange.fruit_id     = 32'h0000_01FCE;
```

Рисунок 2.1 – Пример использования конструкции `struct` в HDL SystemVerilog

Конструкция struct

В языке SystemVerilog, также, как и в C/C++ существует возможность создания дополнительного имени (псевдонима) для типов данных и структур. Данная конструкция широко применяется для упрощения синтаксиса объявления сложных структур данных, состоящих из struct и типов объединения.

Синтаксис конструкции выглядит следующим образом:

```
typedef [исходный_тип_данных] [новый_тип_данных];
```

Поскольку данная конструкция не создаёт новых типов данных, а лишь используется для создания псевдонимов уже существующих типов данных, возможность синтеза кода с использованием структуры typedef зависит исключительно от типов данных, для которых создаются их псевдонимы.

```
1  typedef enum logic [1:0] {
2      ADD = 2'd0,
3      SUB = 2'd1,
4      MUL = 2'd2,
5      DIV = 2'd3
6  } exp_type_t;
7
8  typedef struct packed {
9      logic [13:0] calculation_result;
10     exp_type_t expression_type;
11     logic [7:0] value_2;
12     logic [7:0] value_1;
13 } summ_reg_t;
14
15 summ_reg_t summ_reg;
16
17 always_ff@(posedge clk or negedge nrst)
18 if(!nrst)
19     summ_reg.calculation_result <= '0;
20 else case(summ_reg.expression_type)
21
22     ADD: summ_reg.calculation_result <= summ_reg.value_1 + summ_reg.value_2;
23     SUB: summ_reg.calculation_result <= summ_reg.value_1 - summ_reg.value_2;
24     MUL: summ_reg.calculation_result <= summ_reg.value_1 * summ_reg.value_2;
25     DIV: summ_reg.calculation_result <= summ_reg.value_1 / summ_reg.value_2;
26
27 endcase
```

Рисунок 2.2 – Пример использования конструкции typedef в сочетании с конструкцией struct

Конструкция package

Package реализуют механизм хранения и передачи структур, функций, переменных, параметров и других конструкций HDL SystemVerilog между модулями для дальнейшего использования. Размещение таких объектов и объявлений внутри структуры package позволяет избежать загромождения глобальной области имен, тем самым увеличивая качество кода, его читабельность и модульность. Затем package могут быть импортированы в необходимые модули, где могут использоваться его объекты.

Элементы внутри пакетов не могут иметь иерархических ссылок на идентификаторы, кроме тех, которые созданы внутри пакета или стали видимыми при импорте другого пакета.

Синтаксис конструкции package:

```
package [название];  
    [содержимое/контент];  
endpackage
```

При импортировании package существует возможность импортировать не все объекты (в этом случае указывается wildcard * как идентификатор объекта), а лишь необходимые.

Синтаксис импортирования package:

```
import <название_package>::<имя_объекта>;
```

Конструкция package входит в синтезируемое подмножество HDL SystemVerilog, однако на результаты синтеза влияет содержимое package. Так, например, в package нельзя добавлять циклы и процедурные блоки, такие как `always`, `assign`, `initial`. Иначе говоря, для корректной работы и предсказуемого поведения структура package должна содержать только объявление объектов, которые могут быть переиспользованы в различных источниках, в которых соответствующий package был импортирован.

Примеры использования package показаны на рисунках 2.3 и 2.4.

```

1 package registers_pkg;
2
3     localparam DATA_WIDTH = 32;
4     localparam HASH_SIZE = 5;
5
6     typedef struct packed {
7         logic [DATA_WIDTH - 1 : 0] data;
8         logic [1:0] expression_type;
9         logic [HASH_SIZE - 1 : 0] hash;
10    } data_packet_t;
11
12    typedef enum logic [1:0] {
13        IDLE = 2'd0,
14        START = 2'd1,
15        PROCESSING = 2'd2,
16        FINISH = 2'd3
17    } fsm_states_t;
18
19    function automatic logic [4 : 0] calc_5bit_hash(
20        input [DATA_WIDTH - 1 : 0] data,
21        input [4 : 0] init
22    );
23
24        static logic [HASH_SIZE - 1 : 0] hash;
25
26        hash[0] = (init[0] ^ init[3] ^ data[0] ^ data[5] ^ data[6]);
27        hash[1] = (init[1] ^ init[3] ^ init[4] ^ data[0] ^ data[1]);
28        hash[2] = (init[2] ^ init[4] ^ init[2] ^ data[1] ^ data[2]);
29        hash[3] = (init[0] ^ init[3] ^ init[2] ^ data[2] ^ data[3]);
30        hash[4] = (init[1] ^ init[4] ^ data[3] ^ data[4] ^ data[8]);
31
32        return hash;
33
34    endfunction : calc_5bit_hash
35
36
37 endpackage : registers_pkg

```

Рисунок 2.3 – Пример объявления package, содержащего локальные параметры, конструкции typedef и функцию

```

1  module example
2      import registers_pkg::*;
3  (
4      input                                clk,
5      input                                nrst,
6      input                                start,
7      input                                [1 : 0] exp_type,
8      input                                [DATA_WIDTH - 1 : 0] data,
9
10     output data_packet_t                  packet
11 );
12     localparam logic [4 : 0] HASH_INIT_VALUE = 5'h1D;
13
14     always_ff@(posedge clk or negedge nrst)
15     if(!nrst)
16         packet <= '0;
17     else if(start) begin
18         packet.expression_type <= exp_type;
19         packet.data <= data;
20         packet.hash <= calc_5bit_hash(data, HASH_INIT_VALUE);
21     end
22
23 endmodule : example

```

Рисунок 2.4 – Подключение package внутри модуля и использование его содержимого

Методика проектирования регистровых карт

Регистровые карты являются одним из основных методов управления периферийным устройством. Через них периферийное устройство может конфигурироваться, получать исходные данные для проведения функциональных операций над ними, а также предоставлять для чтения как сервисную информацию, так и результат работы функционального блока, к которым они подключены. Для данных целей существует несколько типов доступа к регистру или его полям:

- 1) *R/W* (Read/Write) – позволяет как записывать данные через системную шину, так и читать их;
- 2) *R/O* (Read Only) – позволяет только читать данные через системную шину, запись запрещена;
- 3) *W/O* (Write Only) – позволяет только писать данные через системную шину, чтение запрещено.

Также, регистры могут отличаться друг от друга по признаку волатильности. Волатильные (volatile) регистры могут изменять своё значение не только при их модификации через системную шину, но также и со стороны устройства, к которому подключена регистровая карта. Например, при записи значения в регистр оно может со временем измениться при выполнении каких-либо условий со стороны подключенного устройства.

Помимо типов доступа, существуют еще атрибуты доступа полей:

- 1) *WIS* (Write '1' to Set) – в данное поле может быть записана логическая '1' для модификации значения из '0' в '1', однако значение не может быть модифицировано из '1' в '0' через взаимодействие по системной шине.
- 2) *WIC* (Write '1' to Clear) – в поле, обладающее таким атрибутом, при записи '1' через системную шину модифицируется значение поля из '1' в '0'. Запись '0' через системную шину не производит поле никакого эффекта. Данный атрибут широко используется для регистров, которые управляют прерываниями. Этот атрибут позволяет упростить программный алгоритм управления таким регистром.
- 3) *WIT* (Write '1' to Toggle) – в поле, обладающее таким атрибутом, при записи '1' через системную шину инвертируется текущее значение на противоположное. Запись '0' не оказывает какое-либо эффекта на значение в поле.

- 4) *W0S* (Write '0' to Set) – аналогично *W1S*, только управляющим значением является '0'. Запись '1' не оказывает эффекта на значение в поле.
- 5) *W0C* (Write '0' to Clear) – аналогично *W1C*, только управляющим значением является '0'. Запись '1' не оказывает эффекта на значение в поле.
- 6) *W1T* (Write '0' to Toggle) – аналогично *W1T*, только управляющим значением является '0'. Запись '1' не оказывает эффекта на значение в поле.

В рамках одного регистра могут находиться поля с различными атрибутами доступа.

При проектировании регистровой карты можно использовать регистры различной размерности, в том числе и комбинировать их внутри одной регистровой карты. Однако, в этом случае необходимо соблюдать правило выровненности адреса регистра устройства под размерность этого регистра.

Правило выровненности адреса регистра – регистр должен располагаться по адресу, значение которого кратно количеству байт в данном регистре. Например, 32-битные регистры должны иметь адреса, кратные 4-ём (0×0 , 0×4 , 0×8 , $0 \times C$). Для 64-битных регистров значение их адреса должно быть кратно 8-ми (0×0 , 0×8). Аналогичное правило используется в алгоритмах обращения к памяти, которые подразумевают, что адреса переменных в памяти процессора или вычислительного блока должны быть выровнены под размерность этих переменных. В противном случае, если это правило не соблюдается, процессор при обращении к переменной может получить значение части другой переменной.

Одной из основных методик проектирования регистровых карт (с использованием HDL SystemVerilog) является составление одной или нескольких структур, состоящих из вложенных полей внутри package. Далее, после того как структуры регистровых карт описаны, внутри модуля, обладающего логикой взаимодействия с системным интерфейсом, объявляется экземпляр структуры, для которого описывается логика и взаимодействие в соответствии с типом доступа и атрибутами для каждого из полей.

Методика выполнения лабораторной работы

1. Изучить теоретический материал лабораторной работы;
2. Реализовать структурную схему (таблицу) регистровой карты. Регистровая карта должна состоять из 4-х 32-битных регистров: регистра входящих данных для вычисления, регистра результата вычислений, регистра системной информации и регистра управления, в котором должны быть 4 функциональных бита: начало вычисления, сброс результата вычисления и флаги окончания вычисления и состояния процесса вычисления. Сигналы начала вычисления и сброса должны автоматически переходить из '1' в '0' на следующий такт после записи. Флаг окончания вычисления и состояния процесса вычисления, как и оставшиеся 28 бит не должны модифицироваться при записи через системную шину.
3. Описать для каждого регистра и поля их тип доступа и атрибуты доступа (при наличии);
4. Реализовать package с использованием `typedef` и `struct`, которые будут содержать объявление данной регистровой карты;
5. Реализовать AVR3-ведомое устройство, содержащее данную регистровую карту и описанный функционал поведения регистров и их полей;
6. Провести функциональную верификацию работоспособности разработанного устройства;
7. Представить результаты преподавателю;
8. Ответить на вопросы по изучаемой теме и выполненной работе.

Лабораторное задание

В лабораторной работе необходимо составить регистровую карту устройства, описать тип доступа каждого регистра и его полей. Реализовать полученную регистровую карту на базе AVR3-ведомого устройства с использованием конструкций `typedef`, `struct`, `package`. За основу можно использовать модуль устройства, разработанного в Л/Р №1.

Требования к отчету

Отчет должен содержать:

1. Название и цель работы;
2. Структурная схема полученной регистровой карты с типами доступа полей их атрибутами и назначением;
3. Файлы реализованных модулей и тестового окружения на языке SystemVerilog.

Контрольные вопросы

1. Поясните спектр применения конструкций struct, typedef?
2. Зачем используется конструкция package? Какие объекты может содержать package? Какие объекты нельзя добавлять в package?
3. Опишите, какие бывают регистры и поля в регистрах по типу доступа в регистровых картах? В чём отличие W1S полей регистров от W1C?
4. Что такое признак волатильности регистра?
5. Можно ли добавить в регистровую карту регистры, размерность которых отличается в размерности шины данных системного интерфейса?
6. Почему необходимо выравнивать адреса регистров под их размерность?

Лабораторная работа № 3

Проектирование модуля аппаратного расчёта CRC для использования в ведомых устройствах с системной шиной

Цель работы: Изучить методику вычисления циклически избыточного кода (Cyclic Redundancy Check, CRC). Реализовать алгоритм расчёта CRC с использованием языка описания аппаратуры SystemVerilog. Подключить разработанный модуль к ведомому модулю, разработанному в процессе выполнения предыдущих лабораторных работ.

Продолжительность работы: 4 ч.

Теоретические сведения

Введение в CRC

Циклически избыточный код служит для обнаружения ошибок целостности при передаче набора данных между устройствами. Является одним из самых популярных и не требовательных по ресурсопотреблению методом нахождения контрольной суммы пакетов данных. Широко применяется в Ethernet, RFID и т.п. устройствах.

Основная идея алгоритма CRC состоит в представлении сообщения в виде огромного двоичного числа, делении его на другое фиксированное двоичное число и использовании остатка этого деления в качестве контрольной суммы. Получив сообщение, приёмник может выполнить аналогичное действие и сравнить полученный остаток с "контрольной суммой" (переданным остатком).

Например, предположим, что сообщение состоит из 2 байт (0x6, 0x23). Их можно рассматривать, как шестнадцатеричное число 0x0167, или как двоичное число 16'b0000_0110_0001_0111. Предположим, что ширина регистра контрольной суммы составляет 1 байт, а в качестве

делителя используется 4'b1001, тогда сама контрольная сумма будет равна остатку от деления 16'b0000_0110_0001_0111 на 4'b1001.

Пример расчёта контрольной суммы для данного примера:

```

4'b1001 = Делитель
16'b00000011000010111 = 0617 = 1559 = Делимое
0000.,.,.,.,.,.,.,.
-----.,.,.,.,.,.,.
0000.,.,.,.,.,.,.,.
0000.,.,.,.,.,.,.,.
-----.,.,.,.,.,.,.
0001.,.,.,.,.,.,.,.
0000.,.,.,.,.,.,.,.
-----.,.,.,.,.,.,.
0011.,.,.,.,.,.,.,.
0000.,.,.,.,.,.,.,.
-----.,.,.,.,.,.,.
0110.,.,.,.,.,.,.,.
0000.,.,.,.,.,.,.,.
-----.,.,.,.,.,.,.
1100.,.,.,.,.,.,.,.
1001.,.,.,.,.,.,.,.
=====.,.,.,.,.,.,.
0110.,.,.,.,.,.,.,.
0000.,.,.,.,.,.,.,.
-----.,.,.,.,.,.,.
1100.,.,.,.,.,.,.,.
1001.,.,.,.,.,.,.,.
=====.,.,.,.,.,.,.
0111.,.,.,.,.,.,.,.
0000.,.,.,.,.,.,.,.
-----.,.,.,.,.,.,.
1110.,.,.,.,.,.,.,.
1001.,.,.,.,.,.,.,.
=====.,.,.,.,.,.,.
1011.,.,.,.,.,.,.,.
1001.,.,.,.,.,.,.,.
=====.,.,.,.,.,.,.
0101.,.,.,.,.,.,.,.
0000.,.,.,.,.,.,.,.
-----.,.,.,.,.,.,.
1011
1001
=====
0010 = 02 = 2 = Остаток

```

На основе вышесказанного, можно утверждать, что метод расчёта CRC заключается в нахождении остатка от деления исходного набора

данных на так называемый «полином» (делитель). Все CRC алгоритмы основаны на полиномиальных вычислениях, и для любого алгоритма CRC можно указать, какой полином он использует.

Реализация алгоритма вычисления CRC

Вместо представления делителя, делимого (сообщения), частного и остатка в виде положительных целых чисел (как это было сделано в предыдущем разделе), можно представить их в виде полиномов с двоичными коэффициентами или в виде строки бит, каждый из которых является коэффициентом полинома. Например, десятичное число 9 в двоичном коде представляет – 4'b1001, что совпадает с полиномом:

$$1 \cdot x^3 + 0 \cdot x^2 + 0 \cdot x^1 + 1 \cdot x^0$$

или, упрощенно:

$$x^3 + x^0$$

И сообщение, и делитель могут быть представлены в виде полиномов, с которыми можно выполнять любые арифметические действия. Размером полинома считается степень самого старшего разряда с прибавлением 1 и имеет обозначение W. В приведённом выше примере размер полинома W равен 4. Данную величину также принято отображать в названии конкретного алгоритма CRC, например CRC-16 имеет полином с W равным 16.

CRC-арифметика, которая реализует деление исходного пакета данных на требуемый полином соответствует операции «сложение по модулю 2», которое также называется «исключающее или» (XOR).

Основываясь на данной информации, методика вычисления CRC аппаратным способом сводится к следующему порядку действий:

1. Создаётся сдвиговый регистр общей размерностью W;
2. Регистр инициализируется начальным значением (для каждого типа CRC начальное значение индивидуальное);
3. Исходные данные дополняются логическими нулями в количестве W;
4. В регистр исходные данные «затягиваются» по одному биту за такт;
5. В случае, если «затянутое» значение соответствует логической единице, производится операция XOR между полиномом и содержимым сдвигового регистра;

6. После того, как были «задвинуты» все биты исходного сообщения в сдвиговый регистр, в нём будет содержаться остаток от деления, который и будет вычисленным CRC.

Программный алгоритм выглядит следующим образом:

```
Загрузим регистр нулевыми битами
Дополним хвостовую часть сообщения W нулевыми битами
While (пока еще есть необработанные биты)
Begin
    Сдвинем регистр на 1 бит влево и поместим очередной
    еще не обработанный бит из сообщения в 0 позицию реги-
    стра.
    If (из регистра был выдвинут бит со значением "1")
        Регистр = Регистр XOR Полином
End
Теперь в регистре содержится остаток.
```

Одним из самых широко применяемых типов циклически избыточного кода является CRC-16-CCITT. Он широко применяется для передачи данных в оборудовании для телекоммуникаций и СВЧ RFID меток. Полином CRC-16-CCITT соответствует числу 0x1021. Сам полином выглядит следующим образом:

$$x^{15} + x^{12} + x^5 + x^0$$

Схемотехнически устройство вычисления CRC-16-CCITT можно представить следующим образом:

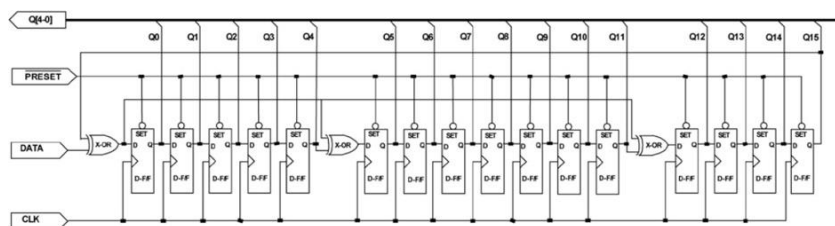


Рисунок 3.1 – Пример аппаратной схемы вычисления CRC-16

Пример схемы на HDL SystemVerilog

На рисунке 3.2 приведён пример расчёта CRC-16-CCITT на языке описания аппаратуры SystemVerilog.

```
1  always_ff@(posedge clk or negedge nrst)
2  if(!nrst)
3      crc_reg    <= 16'hFFFF;
4  else if (init)
5      crc_reg    <= 16'hFFFF;
6  else begin
7      crc_reg[0] <= data_in ^ crc_reg[15];
8      crc_reg[1] <= crc_reg[0];
9      crc_reg[2] <= crc_reg[1];
10     crc_reg[3] <= crc_reg[2];
11     crc_reg[4] <= crc_reg[3];
12     crc_reg[5] <= crc_reg[4] ^ data_in ^ crc_reg[15];
13     crc_reg[6] <= crc_reg[5];
14     crc_reg[7] <= crc_reg[6];
15     crc_reg[8] <= crc_reg[7];
16     crc_reg[9] <= crc_reg[8];
17     crc_reg[10] <= crc_reg[9];
18     crc_reg[11] <= crc_reg[10];
19     crc_reg[12] <= crc_reg[11] ^ data_in ^ crc_reg[15];
20     crc_reg[13] <= crc_reg[12];
21     crc_reg[14] <= crc_reg[13];
22     crc_reg[15] <= crc_reg[14];
23 end
```

Рисунок 3.2 – Пример реализации расчёта CRC-16-CCITT с использованием HDL SystemVerilog

Методика выполнения лабораторной работы

1. Получить у преподавателя задание в соответствии с вариантом;
2. Реализовать модуль с использованием языка описания аппаратуры SystemVerilog. Перечень управляющих сигналов модуля и их разрядность соответствует управляющим сигналам из регистровой карты, разработанной в лабораторной работе №2;
3. Подключить разработанный модуль к APB3-ведомому устройству из Л/Р №2;
4. Реализовать тестовое окружение и сценарий тестирования, который использует различные сценарии взаимодействия с устройством через регистровую карту устройства;
5. Подготовить отчёт о выполнении лабораторной работы;
6. Представить результаты преподавателю;
7. Ответить на вопросы по изучаемой теме и выполненной работе.

Таблица 3.1

Перечень управляющих сигналов модуля

Название	Направление	Разрядность, бит	Описание
clk	input	1	Тактовый сигнал.
nrst	input	1	Асинхронный сброс с низким активным уровнем.
init	input	1	Инициализировать регистр в начальное состояние. Ширина импульса – 1 такт.
start	input	1	Флаг начала расчёта CRC. Ширина импульса – 1 такт.
data	input	32	Данные для расчёта CRC.
done	output	1	Флаг окончания расчёта CRC. Ширина импульса – 1 такт.
busy	output	1	Признак активного процесса расчёта CRC.
result	output	32	Результат расчёта CRC.

Лабораторное задание

В лабораторной работе необходимо в соответствии с вариантом реализовать устройство на HDL SystemVerilog и провести его функциональную верификацию. Составить краткий отчет по результатам выполнения работы.

Таблица 3.2

Варианты заданий

№	Название	Полином	Исходное значение	RefIn	RefOut	XorOut
1	CRC-32	0x04C11DB7	0xFFFFFFFF	Да	Да	0xFFFFFFFF
2	CRC-32/BZIP2	0x04C11DB7	0xFFFFFFFF	Нет	Нет	0xFFFFFFFF
3	CRC-32/JAMCRC	0x04C11DB7	0xFFFFFFFF	Да	Да	0x00000000
4	CRC-32/MPEG-2	0x04C11DB7	0xFFFFFFFF	Нет	Нет	0x00000000
5	CRC-32/POSIX	0x04C11DB7	0x00000000	Нет	Нет	0xFFFFFFFF
6	CRC-32/SATA	0x04C11DB7	0x52325032	Нет	Нет	0x00000000
7	CRC-32/XFER	0x000000AF	0x00000000	Нет	Нет	0x00000000
8	CRC-32C	0x1EDC6F41	0xFFFFFFFF	Да	Да	0xFFFFFFFF
9	CRC-32D	0xA833982B	0xFFFFFFFF	Да	Да	0xFFFFFFFF
10	CRC-32Q	0x814141AB	0x00000000	Нет	Нет	0x00000000

RefIn – Признак инверсии входящих байт данных. При наличии данного параметра для расчёта CRC необходимо изменить порядок расположения бит для каждого байта перед расчётом контрольной суммы.

RefOut – Признак инверсии исходящих байт данных. При наличии данного параметра после расчёта CRC необходимо инвертировать содержимое регистра.

XorOut – W-битное значение, которое должно комбинироваться с конечным содержимым регистра для получения окончательного значения контрольной суммы.

Требования к отчету

Отчет должен содержать:

1. Название и цель работы;
2. Вариант устройства и его параметры;
3. Исходные данные и рассчитанные для них эталонные результаты вычислений CRC;
4. Файлы реализованных модулей и тестового окружения на языке SystemVerilog.

Контрольные вопросы

1. Для чего используется расчёт циклически избыточного кода (CRC)?
2. Как происходит расчёт CRC?
3. Какие бывают типы CRC и чем они отличаются?
4. В каких случаях необходимо не сбрасывать исходное значение для просчёта нового значения CRC?
5. На какой параметр CRC влияет размерность полинома (W)? Зачем используют полиномы большой размерности ($W \geq 32$)?

Лабораторная работа № 4

Реализация программы взаимодействия встроенного процессорного ядра с периферий- ными устройствами СнК

Цель работы: Изучить теоретическую часть по основам разработки ПО для встраиваемых систем. С использованием языка программирования Си разработать алгоритм взаимодействия RISC-V процессора с разработанным ведомым устройством. Подключить разработанный модуль к верхнему уровню проекта СнК.

Продолжительность работы: 4 ч.

Теоретические сведения

Программирование систем на кристалле

С появлением систем на кристалле (СнК) произошло существенное уменьшение размеров электронных устройств и повышение их производительности. СнК объединяет в себе несколько функциональных блоков, таких как процессор, память, контроллеры периферийных устройств и другие, на одном кристалле. Это позволяет значительно уменьшить размеры устройств, упростить их проектирование и снизить стоимость производства. Кроме того, СнК обеспечивает более высокую производительность и энергоэффективность по сравнению с традиционными решениями на отдельных компонентах.

Однако, для работы с СнК требуются специалисты, которые знают особенности аппаратной части и могут создавать оптимизированный и эффективный код для встраиваемых систем. Такие программисты называются программистами встраиваемых систем и при работе им необходимы знания в области как аппаратной части, так и низкоуровневого программирования для создания программного обеспечения СнК.

Для проектирования встраиваемого ПО на языке Си необходимо знать основные конструкции языка, которые позволяют работать с ап-

паратным обеспечением и создавать оптимизированный и эффективный код. Это обуславливается тем, что СнК достаточно часто обладают ограниченным количеством выделенной под программы памяти, поэтому крайне важно писать оптимизированный с точки зрения используемой памяти код и эффективно использовать имеющиеся объёмы памяти.

Также, при проектировании СнК необходимо использовать RISC (Reduced Instruction Set Computing) архитектуры процессоров, например ARM, MIPS, RISC-V. Эта необходимость обуславливается тем, что RISC процессоры обладают большей энергоэффективностью, производительностью, масштабируемостью и надёжностью, в сравнении с CISC (Complex Instruction Set Computing) процессорами.

Основы разработки ПО встраиваемых систем с использованием языка программирования C

Основными конструкциями кода, которые широко используются при проектировании ПО для встраиваемых систем на ЯП C (Си) являются макросы, указатели, стандартные беззнаковые типы данных и структуры.

Макросы используются для определения символьных констант, функций и операций, которые могут быть использованы в программе. В программировании встроенных систем они необходимы для оптимизации кода по используемым ресурсам. Например, макросы могут заменять длинные выражения на более короткие, что уменьшает размер программы, ускоряет её выполнение и упрощает восприятие кода разработчиком, за счёт уменьшения объёма кода программы. Пример использование макросов:

```
#define CRC_MODULE_BASE_ADDRESS ( 0x11000 )
```

В данном примере через макрос создаётся константа CRC_MODULE_BASE_ADDRESS, которая соответствует базовому адресу устройства расчёта CRC в системе, который соответствует значению 0x11000.

Указатели в языке C используются для работы с памятью напрямую и управлению аппаратными ресурсами. Они позволяют использовать динамическое выделение памяти, создание массивов и структур данных. Использование указателей также помогает оптимизировать код, уменьшить размер программы и ускорить ее выполнение. Например,

указатели могут использоваться для передачи больших блоков данных между функциями без копирования этих данных в память.

Указатель в языке С – это переменная, которая хранит адрес в памяти на объект своего типа. Указатели пишутся с помощью символа звёздочки (*). Для объявления указателя необходимо указать тип данных, на который он будет указывать, например:

```
int* p;    // указатель на целое число
char* s;   // указатель на символ
float* f;  // указатель на число с плавающей точкой
```

Пример использования указателя в коде:

```
int  x = 10;
int *p = &x; // теперь содержит адрес переменной x
*p = 20;    // изменяем значение x на 20
```

В приведённом выше примере создаётся переменная `x`, которой присваивается значение 10. Далее создаётся указатель `p` на объект типа `int`, которому присваивается адрес переменной `x`. После этого, появляется возможность поменять значение переменной `x`, используя указатель на адрес в памяти, в котором хранится значение переменной `x`.

Также, для сохранения кроссплатформенности разрабатываемого программного обеспечения для встраиваемых систем, в ЯП Си используют библиотеку `stdint.h`, которая предоставляет набор типов данных с фиксированным размером:

- `int8_t`, `uint8_t` – знаковый и беззнаковый целочисленный тип данных размером 8 бит;
- `int16_t`, `uint16_t` – знаковый и беззнаковый целочисленный тип данных размером 16 бит;
- `int32_t`, `uint32_t` – знаковый и беззнаковый целочисленный тип данных размером 32 бит;
- `int64_t`, `uint64_t` – знаковый и беззнаковый целочисленный тип данных размером 64 бит.

Помимо переносимости кода, использование таких типов данных обладает следующими преимуществами:

1. Явное указание размера переменных: использование типов данных из `stdint.h` позволяет явно указывать размер переменных, что упрощает написание кода и делает его более понятным.

2. Удобство работы с битами: такие типы данных используются для работы с битами, так как они имеют фиксированный размер и могут быть легко преобразованы в битовые поля.

3. Безопасность: использование типов данных из `stdint.h` обеспечивает безопасность при работе с целочисленными значениями, так как они имеют фиксированный размер и не могут быть переполнены или недостаточны по размеру.

4. Совместимость с другими библиотеками: многие библиотеки для встраиваемого ПО используют типы данных из `stdint.h`, что обеспечивает совместимость и упрощает интеграцию с другими библиотеками.

Помимо приведённых выше синтаксических конструкций, в разработке ПО для встраиваемых систем используются структуры. Структуры (`struct`) используются для организации и хранения данных, в том числе и различных типов. Эта конструкция позволяет объединить несколько переменных в одну структуру, что упрощает работу с данными и повышает читабельность кода. Благодаря упрощению работы с аппаратными устройствами в СнК структуры позволяют более эффективно управлять предоставленным объёмом памяти. Синтаксис конструкции `struct` схож с аналогичной конструкций в HDL SystemVerilog. Пример использования конструкции `struct` представлен на рисунке 4.1.

Также, при разработке ПО для встраиваемых решений, необходимо обязательно указывать волатильность для тех регистров, значение в которых может измениться не только за счёт модификации значения процессором, но и самой аппаратной логикой, либо другим процессором. Эта необходимость обуславливается тем, что без указания волатильности регистра или участка памяти, процессор, обладающий кэш-памятью, может кэшировать полученное значение, что в дальнейшем может привести к некорректной работе разрабатываемого устройства. Для того, чтобы указать, что регистр или участок памяти является волатильным, используется конструкция `volatile`, которая определяется перед объявлением указателя на объект (рисунок 4.1).


```

1 // Подключение библиотеки stdint.h
2 #include <stdint.h>
3
4 // указание базового адреса устройства
5 #define BASE_ADDRESS ( 0x11000 )
6
7 // создание структуры регистровой карты
8 // apb3 slave устройства
9 typedef struct {
10     uint32_t      crc_data_in;
11     uint32_t      crc_data_out;
12     uint32_t      control;
13 } regmap_t;
14
15 int main() {
16
17     // создание экземпляра регистровой карты
18     // с указанием адреса, где он находится
19     volatile regmap_t* regmap = (regmap_t*) BASE_ADDRESS;
20
21     // загрузка исходных данных для расчёта
22     regmap->data_in = 0xABCDEF01;
23
24     return 0;
25 }

```

Рисунок 4.1 – Пример использования конструкции struct в Си

Методика выполнения лабораторной работы

1. Получить у преподавателя необходимый для выполнения лабораторной работы материал:

- Исходный проект СнК на языке SystemVerilog, состоящий из процессора RISC-V, SRAM памяти для исполняемой программы и модулем межсоединений для системных интерфейсов (рисунок 4.2);
- Базовый адрес свободного APB3 устройства в исходном проекте;
- Компилятор для процессоров RISC-V (riscv64-unknown-elf);
- Шаблон программы и инструкции по компиляции программы для предоставленного проекта;
- Инструкции по запуску HDL проекта совместно с пользовательской программой.

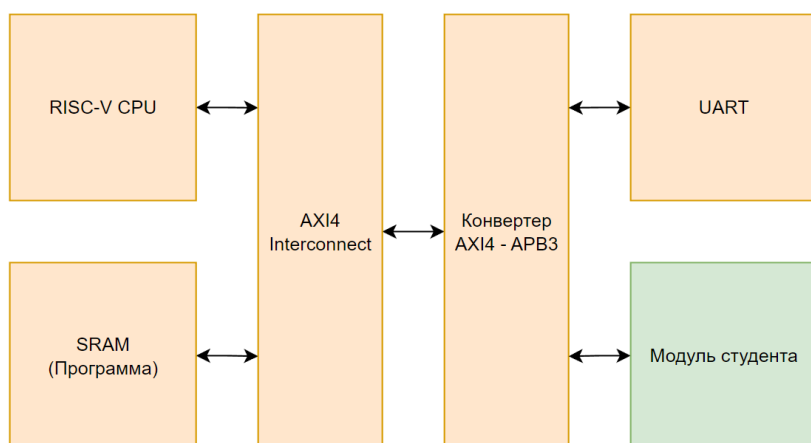


Рисунок 4.2 – Структурная схема исходного проекта

2. Инстанцировать разработанный в ходе лабораторной работы №3 модуль в верхнем уровне HDL описания проекта.

3. Произвести компиляцию HDL проекта в САПР Synopsys VCS; при появлении ошибок внести исправления и повторить компиляцию, добиться корректной компиляции.

4. Реализовать алгоритм взаимодействия с разработанным модулем с использованием языка программирования С. В алгоритме должны быть заложены функции корректного и безопасного обращения к модулю, с проверкой текущего состояния модуля, т.е. необходимо учитывать флаги текущее значение флагов `busy` и `calculation done` в регистровой карте устройства.

5. Провести компиляцию полученной программы с использованием компилятора GCC для архитектуры RISC-V.

6. Запустить HDL симуляцию проекта с разработанной программой.

7. Провести функциональную симуляцию алгоритма с использованием САПР Synopsys VCS.

8. Проанализировать полученные результаты; убедиться в правильности работы разработанного алгоритма. При появлении ошибок попытаться их выявить и устранить; затем повторить функциональную симуляцию, добиться положительных результатов.

9. Проанализировать пройденный маршрут; разобраться в производимых действиях.

10. Составить отчёт по работе.

11. Представить результаты работы преподавателю; ответить на вопросы по теоретическому и практическому материалу лабораторной работы, а также по полученным результатам.

Лабораторное задание

В лабораторной работе необходимо подключить разработанный в процессе выполнения предыдущих лабораторных работ модуль вычисления CRC-32, обладающего системным интерфейсом APB3 в верхний уровень HDL проекта СнК. После чего реализовать алгоритм взаимодействия с подключенным модулем. Алгоритм должен быть реализован в программе с использованием языка программирования С. Провести функциональную симуляцию проекта СнК с разработанным программным алгоритмом. Составить краткий отчёт по результатам выполнения работы.

Требования к отчету

Отчет должен содержать:

1. Название и цель работы;
2. Блок-схему разработанного алгоритма;
3. Код на языке программирования C для разработанного алгоритма;
4. Код на языке HDL SystemVerilog верхнего уровня проекта;
5. Результаты функциональной симуляции верхнего уровня с подтверждением корректной работы алгоритма взаимодействия с периферийным модулем.

Контрольные вопросы

1. Для чего необходимо указывать признак волатильности участка внешней памяти в ЯП C?
2. Для чего используются указатели?
3. Как узнать адрес переменной?
4. Как можно реализовать взаимодействие с регистровой картой устройства без использования структур?
5. Какие существуют ограничения при написании ПО для встро-енных систем?
6. Для чего используется библиотека `stdint.h` в ЯП C?

Рекомендуемая литература

1. *Харрис Д. М.* Цифровая схемотехника и архитектура компьютера: RISC-V / *Д.М. Харрис, С.Л. Харрис*. – 2-е изд. – М.: ДМК Пресс, 2022. – 810 с.
2. "IEEE Standard for SystemVerilog – Unified Hardware Design, Specification, and Verification Language", in IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012), P. 1-1315.
3. *Бродин В.Б., Калинин А.В.* Системы на микроконтроллерах и БИС программируемой логики. – М.: ЭКОМ, 2002. – 400 с.
4. *Томас Д.* Логическое проектирование и верификация систем на SystemVerilog. – М.: ДМК Пресс, 2019. – 384 с.
5. *Авдеев В. А.* Периферийные устройства: интерфейсы, схемотехника, программирование. – М.: ДМК Пресс, 2012. – 848 с.
6. AMBA® APB Protocol Specification. Version 2.0. ARM Limited. 2010. P. 1-28.

Оглавление

Введение	3
Лабораторная работа № 1. Основы проектирования ведомых модулей обладающих системным интерфейсом	5
Лабораторная работа № 2. Проектирование регистровой карты устройства обладающего системным интерфейсом	19
Лабораторная работа № 3. Проектирование модуля аппаратного расчёта CRC для использования в ведомых устройствах с системной шиной.....	29
Лабораторная работа № 4. Реализация программы взаимодействия встроенного процессорного ядра с периферийными устройствами СнК.....	37
Рекомендуемая литература	45

Учебное издание



Любавин Кирилл Дмитриевич

Тельпухов Дмитрий Владимирович

Беляев Андрей Александрович

Кузьмин Павел Андреевич

**Лабораторный практикум по курсу «Проектирование систем на кристалле
с программируемой архитектурой»**

Редактор  Технический редактор .

Верстка авторов.

Подписано в печать с оригинал-макета . .2023. Формат 60х84 1/16. Печать
офсетная. Бумага офсетная. Гарнитура Times New Roman. Усл. печ. л. .

Уч.-изд. л. . Тираж экз. Заказ .

Отпечатано в типографии ИПК МИЭТ.

124498, Москва, Зеленоград, площадь Шокина, дом 1, МИЭТ.