Разработка систем на кристалле
Лабораторный практикум

Анисимова М.
Анисимова Н.
Примаков Е.
Рыжкова Д.
Силантьев А.
Солодовников А.

573b311d (fpga)

Оглавление

Лабораторная работа 1

Введение в проектирование СнК

Улучшение процессов производства интегральных схем (ИС), в соответствии с законом Мура, ведет к непрерывному росту уровня их интеграции. Быстродействие ИС также увеличивается и характеризуется рабочими частотами эквивалентными нескольким сотням Мегагерц. На кристаллах, обладающих подобными характеристиками, можно построить систему, выполняющую целый ряд сложных задач обработки информации.

Такие системы обладают большим количеством преимуществ по сравнению с реализацией системы на плате с несколькими устройствами, обладающей идентичным функционалом. Они обеспечивают большее быстродействие и безопасность передачи данных между различными элементами системы, имеют более низкое энергопотребление, меньший физический размер, а также более низкую стоимость конечного продукта. Но существует и ряд недостатков, связанных с данной реализацией. Во-первых, они обладают меньшей гибкостью, т.е. устройство нельзя модифицировать после изготовления, но в случае необходимости его можно дополнить сторонними внешними устройствами, при условии, что для их сопряжения существуют внешние интерфейсы. Во-вторых, такие системы являются специфичными для конкретных приложений, и вследствие невозможности расширения функционала их редко можно применить для широкого круга задач.

Подобные системы принято называть **системами на кристалле** (СнК). СнК — это интегральная схема, выполняющая функции целого устройства. Основным и обязательным компонентом СнК является процессор. Для реализации функционала, возложенного на СнК, также необходимы периферийные модули, такие как различные интерфейсы (GPIO, UART, и т. д.), блоки памяти, периферийные устройства (таймеры, встроенный ЦАП, АЦП, и т. д.). Взаимодействие между процессором (мастером – устройством, управляющим транзакциями на системной шине) и всеми остальными блоками СнК (слейвами – ведомыми устройствами) осуществляется посредством системной шины (рис. 1.1).

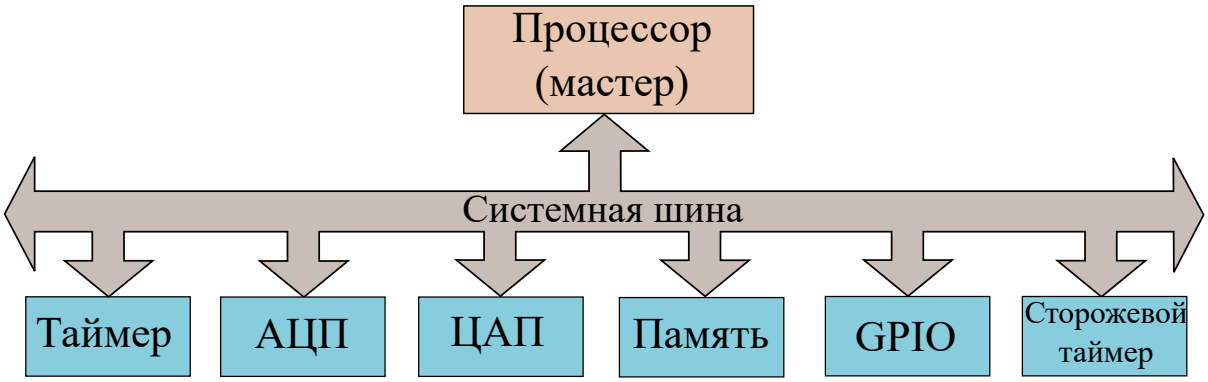

Рис. 1.1: Пример структурной схемы СнК.

Классифицировать СнК можно по различным признакам, далее приведем две наиболее широко используемые классификации.

По признаку ориентации на потребителя (или по назначению) СнК можно разделить на стандартные – микроконтроллеры, а также на специализированные СнК. Микроконтроллеры – это ИС, с размещенными на одном кристалле вместе с процессором внутренней памятью и периферийными устройствами. Они представляют собой законченные устройства и ориентированы на массовое потребление , вследствие чего имеют низкую стоимость (пример структурной схемы микроконтроллера приведен на рис. 1.2). Специализированные СнК разрабатывают по конкретному заказу и имеют более низкий тираж, а, следовательно, более высокую стоимость конечного продукта.

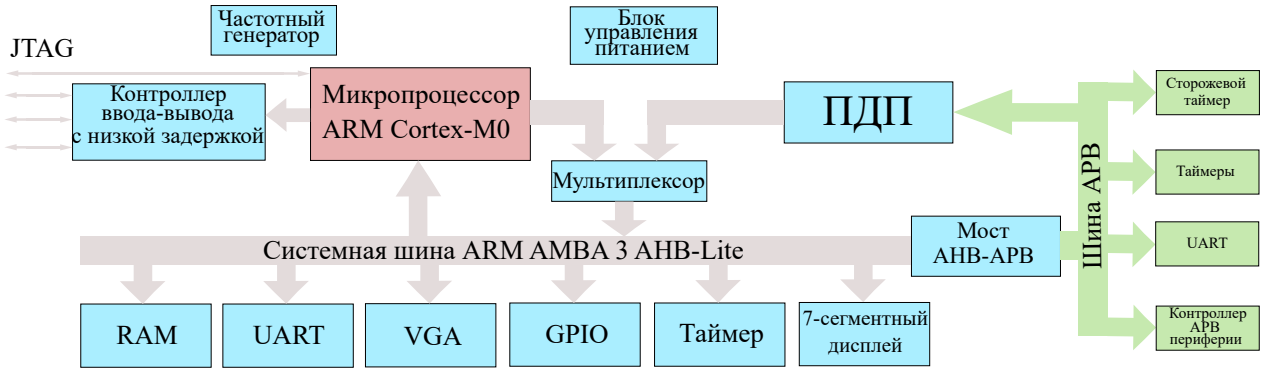

Рис. 1.2: Пример структурной схемы микроконтроллера.

Также СнК можно поделить на две группы по типу реализации: ASIC (специализированные ИС) и FPGA (ИС программируемой логики).

Очевидно, что процесс проектирования и изготовления ASIC является более трудоемким, длительным и дорогим. Как результат, изготовление ASIC является экономически выгодным решением только при производстве изделий большим тиражом, а также их длительного срока эксплуатации. В случае средних и малых объемов тиража, а также существования тенденции к быстрому обновлению реализуемых в конкретной системе алгоритмов или стандартов более выгодным является реализация СнК на базе FPGA. Явным преимуществом данной реализации также является возможность в случае возникновения неисправности быстрого внесения корректив даже в полевых условиях. В случае реализации в виде ASIC цена любой ошибки

(как во временном, так и в ценовом эквиваленте) очень велика, поэтому прежде, чем начать ее производство для верификации используют прототип на базе FPGA.

Процесс производства СнК в виде ASIC обычно разделяют на ряд ключевых шагов. Причем исходя из требования к увеличению скорости выпуска конечного изделия, а также уменьшения вероятности возникновение ситуации, требующей возвращение к предыдущим этапам производства (из-за невозможности полного сопряжения программной и аппаратной составляющей) процессы разработки и верификации ПО и АО происходят параллельно. Данный подход к проектированию называется сопряженным и позволяет практически на любом этапе оценить работу системы в целом (рис. 1.3).

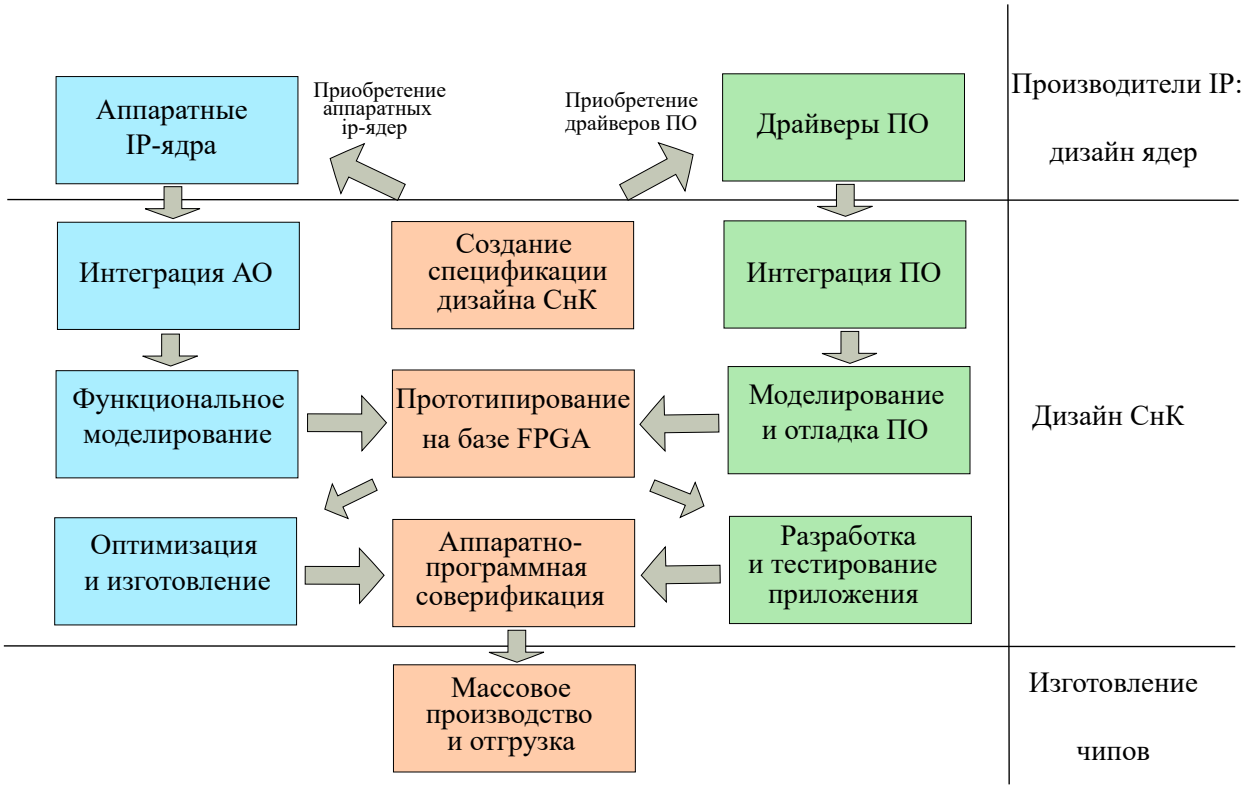

Рис. 1.3: Процесс производства и проектирования СнК.

1.1 Архитектура микропроцессора

Как было упомянуто ранее, неотъемлемой частью СнК является микропроцессор. **Микропроцессор** – программно-управляемое устройство, осуществляющее цифровую обработку информации и управление этим процессом. Микропроцессор “понимает” только нули и единицы, поэтому команды закодированы двоичными числами в формате, который называется машинным кодом. Но для программиста чтение и написание компьютерных программ на машинном языке представляется весьма сложной задачей, он рассматривает процессор с точки зрения его архитектуры. Архитектура системы команд (ISA или просто архитектура) – это система команд и средства для их выполнения: форматы данных, регистры, способы адресации, память. Существует множество разных архитектур, таких как: x86, MIPS, SPARC и PowerPC.

В данном курсе будет рассмотрена архитектура **RISC-V** [1], которая в сравнении с другими имеет несколько преимуществ. Первым и главным из которых является открытость спецификации и свобода использования, что послужило причиной создания большого количества процессорных ядер на ее основе. Архитектура RISC-V является load-store архитектурой, т. е. все инструкции можно разделить на два типа: инструкции доступа к памяти (для загрузки данных

в регистры (load) и сохранения их в памяти (store)), а также инструкции для работы с АЛУ, которые адресуются только к регистрам (рис. 1.4).

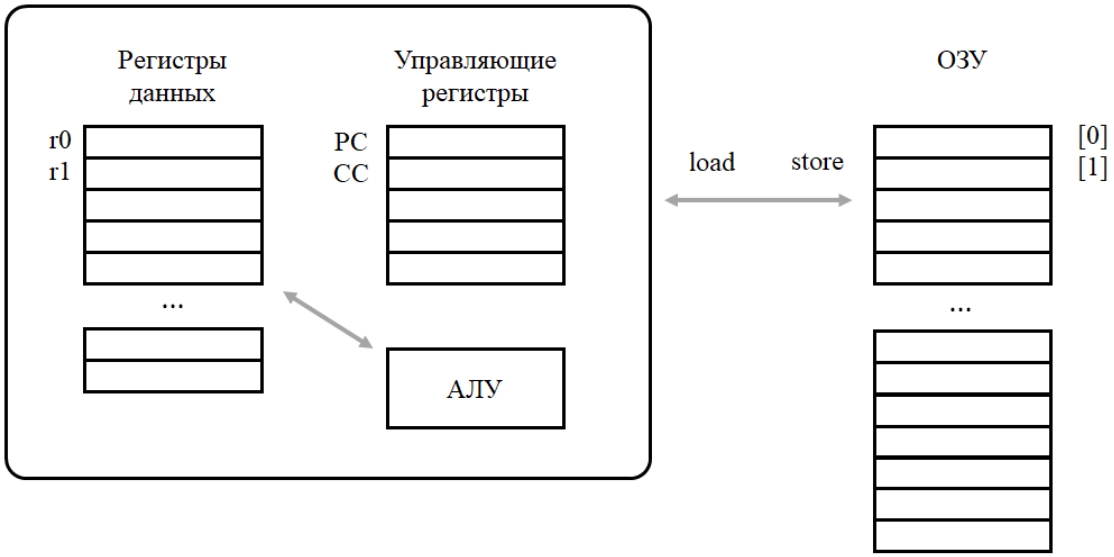

Рис. 1.4: Доступ к памяти в архитектуре RISC-V.

Также важным преимуществом архитектуры RISC-V является модульность ISA, т. е. помимо нескольких базовых наборов инструкций (которые характеризуются разрядностью регистров и соответствующим размером адресного пространства) существуют также опциональные расширения. Рассмотрим базовый набор инструкций RV32I ISA RISC-V [2] (приложение, таблица 1.3).

В базовом наборе для работы программиста существует 31 регистр общего назначения (РОН) *x1-x31*, один регистр *x0*, хранящий нулевую константу, а также регистр, хранящий текущее значение счетчика команд (PC). Помимо наименования регистров по их порядковому номеру в блоке РОН существует так же наименование по назначению (таблица 1.1). Так, например, второе наименование регистра *“x1” – “ra”* (return address), так как он используется для хранения обратного адреса при вызове подпрограммы.

Таблица 1.1: Наименования регистров и их назначения.

Регистр	ABI имя	Описание
x0	zero	Нулевая константа
x1	ra	Обратный адрес, при вызове подпрограммы
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5-7	t0-2	Временные регистры, значения которых валидны только во время вызова функции
x8	s0/fp	Регистры, значения которых сохраняются между вызовами функций/ Frame pointer
x9	s1	Регистры, значения которых сохраняются между вызовами функций
x10-11	a0-1	Аргументы функций / возвращаемые значения
x12-17	a2-7	Аргументы функций
x18-27	s2-11	Регистры, значения которых сохраняются между вызовами функций
x28-31	t3-6	Временные регистры, значения которых валидны только во время вызова функции

В архитектуре RV32I каждая инструкция (рис. 1.5) представлена 32-разрядным словом и весь набор можно разделить на 6 типов (каждая инструкция в свою очередь состоит из полей (таблица 1.2)):

1. R (инструкции с 2 регистровыми входами, например сложение);
2. I (инструкции с одним операндом, например прибавление константы);
3. S (инструкции работы с памятью);
4. SB (инструкции ветвления для выполнения команд условного перехода, изменяющих значение PC);
5. U (инструкции с длинным операндом);
6. UJ (инструкции безусловного перехода, изменяющие значение PC).

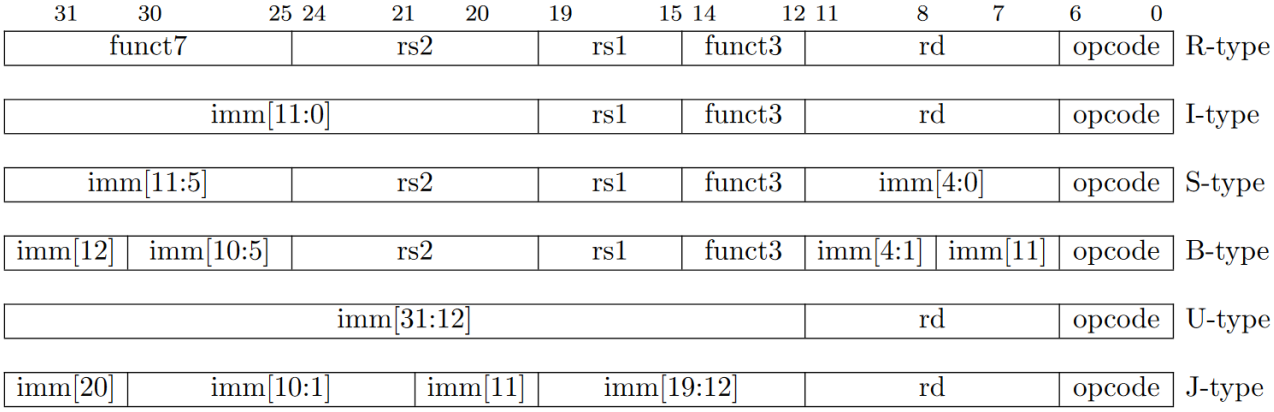

Рис. 1.5: Инструкции базового набора команд архитектуры RISC-V.

Таблица 1.2: Поля инструкций архитектуры RISC-V и их назначение.

rs1, rs2	Регистры, содержащие операнды для операции
func7, func3	Выбор типа операции
rd	Регистр для записи результата операции
opcode	Для пользовательского расширения и выбора типа операции
imm	Поле для констант

1.2 Ассемблер RISC-V

Чтобы понять архитектуру любого компьютера, нужно в первую очередь изучить его ассемблер. **Ассемблер** – язык программирования низкого уровня, представляющий собой формат записи машинных команд, удобный для восприятия человеком. Каждая команда ассемблера задает операцию, которую необходимо выполнить, а также операнды, над которыми производится эта операция. Операнды – это входные данные, с которыми производится операция, и получаемые результаты. Операнды могут находиться в памяти, в регистрах или внутри самой инструкции (константы). Таким образом, каждую инструкцию из ISA можно описать с помощью языка ассемблер.

Не существует общего синтаксиса языка ассемблер для различных архитектур, однако, есть некоторый стандартный подход: сначала записывается операция, потом операнды. Для ассемблера RISC-V (далее просто ассемблер) инструкция будет иметь следующий вид:

[Команда] [Регистр-результат] , [Регистры-операнды через “,”]

Например, инструкция для сложения двух операндов и записи результата в регистр *“add”* на языке ассемблера архитектуры RISC-V будет выглядеть следующим образом (рис. 1.6). Полный список возможных инструкций описан в документе The RISC-V Instruction Set Manual [1].

Рис. 1.6: Соответствие инструкции RISC-V и команды на языке ассемблера на примере операции сложения.

Помимо основного набора инструкций ассемблер реализует ряд удобных **псевдо-инструкций** [1], которые формируются из базового ISA, но имеют неявные аргументы, которые приводят к определенной семантике (приложение, таблица 1.4). Наглядным примером удобства использования таких модифицированных инструкций является операция “nop” - операция, цель которой состоит в том, чтобы затратить 1 такт процессорного времени. Чтобы затратить 1 такт, не выполнив ни одной операции, меняющей контекст микропроцессора (состояние его регистров, памяти), можно выполнить операцию прибавление нулевой константы к регистру *x0 (addi x0, x0, 0)*, однако более наглядным и понятным будет использование псевдо-инструкции nop.

Помимо инструкций и псевдо-инструкций важным элементом синтаксиса языка ассемблера являются **директивы**, которые управляют сборкой инструкций в объектный файл (приложение, таблица 1.5). Директивы не транслируются в инструкции и дают возможность включать произвольные данные в объектный файл, управлять экспортом символов, выбором разделов, выравниванием данных, опциями сборки для сжатия, позиционно-зависимым и позиционно-независимым кодом. Все директивы предваряются точкой.

Первая подгруппа директив – это директивы разделов (или просто разделы). Объектный файл состоит из нескольких разделов, причем каждый раздел соответствует отдельным типам исполняемого кода или данных. Существуют различные типы разделов, наиболее часто используемыми являются:

1. .text – раздел только для чтения, содержащий исполняемый код;
2. .data – раздел для чтения и записи, содержащий глобальные или статические переменные;
3. .rodata – раздел только для чтения, содержащий константные переменные;
4. .bss – раздел для чтения и записи, содержащий неинициализированные переменные;
5. и так далее.

Вторая крупная подгруппа – это директивы передачи данных (или выравнивания данных). Например, директива *.half* с последующими после нее аргументами, перечисленными через запятую, передает в блок памяти данные, соответствующие перечисленным аргументам размером по 16 бит каждый. Директива *.string* используется для передачи в память строки. Ниже приведен пример использования директивы *.data* для инициализации некоторых ячеек памяти переменными различного типа (листинг кода 1.1).

```
.data
  num1: .word 1          # 32-битное слово
  num2: .byte  3         # 8-битное слово
  arr1: .word 5 7 1      # массив трех 32-битных слов
  s1:   .string "AB\n"   # строка
```

Листинг кода 1.1: Пример использования директивы *.data*.

Отдельно стоит отметить директиву *.globl* из подгруппы директив для определения и экспорта символов. Директива *.globl* делает символ видимым для компоновщика, она указывает на символ *__start* (листинг кода 1.2), так как требуется сообщить компоновщику, чтобы тот поместил код, на который указывает символ, в корректное место в памяти. *__start* – это “label”, или символ, значение которого равно адресу в памяти ассемблерного кода, который начинается после объявления *__start*.

```
.globl:
.globl __start
.text
__start:
# исполняемый код
```

Листинг кода 1.2: Пример использования директивы .globl.

Как и в языках высокого уровня, в языке ассемблер возможна реализация подпрограмм или функций с целью устранения ситуаций многократного повторения одной и той же последовательности действий, а также оптимизации кода и простоты восприятия. Для изменения значения счетчика команд на адрес первой инструкции подпрограммы используется команда безусловного перехода – инструкция *jal*.

jal rd , <Адрес первой инструкции подпрограммы>

jal сохраняет адрес следующей за ней инструкции в регистр *ra* (*pc*+4) для возможности продолжения основной программы после выхода из функции. Если функции нужно передать аргументы, то их следует записать в регистры *a0-a7* перед выполнением команды безусловного перехода. Если функция возвращает значения, то они окажутся в регистрах *a0-a1* (таблица 1.1) после возвращения в основную программу. Ниже приведен пример вызова подпрограммы на языке ассемблер (листинг 1.3). Для выхода из подпрограммы используется псевдо-инструкция *ret*.

```
.globl __start

.data
  msgt:  .string "\nmsgt "
  msgs:  .string "\nmsgs "

.text
  target:
    ecall
    ret             # выход из подпрограммы

  __start:
    li  a0, 4       # запись аргументов
    la  a1, msgt
    jal ra, target  # команда безусловного перехода
```

Листинг кода 1.3: Пример реализации подпрограмм.

1.3 Симуляторы ассемблера RISC-V

Для начинающего программиста проверка правильности работы программы, написанной на языке ассемблера, через систему с софт-процессором с помощью специальных САПР на временных диаграммах является сложной задачей. Поэтому для удобства сепаратной отладки ПО от аппаратной части удобно использовать симуляторы ассемблера конкретной архитектуры. **Jupiter** – это симулятор ассемблера RISC-V с открытым исходным кодом, созданный в образовательных целях. Он написан на Java и способен моделировать все инструкции базового ISA (расширение I) плюс расширения M и F (RV32IMF), включая все псевдо-инструкции, описанные в руководстве по набору инструкций. Стоит отметить, что Jupiter не является единственным симулятором ассемблера RISC-V (существуют такие симуляторы как qemu, ANGEL, Spike и т. д.), однако его интерфейс является интуитивно понятным и максимально удобным для специалистов, начинающих изучение ассемблера.

Отдельно стоит отметить возможности работы с консолью Jupiter. RISC-V предоставляет отдельный код операции (opcode) для вызова операционной системы – инструкция *“ecall”*. Для вывода информации на консоль нужен код вызова (для записи код равен 4), строка для записи и длина в байтах, которые записываются в регистры *a0, a1* и *a2* соответственно перед вызовом *ecall*. Такой подход отличается от большинства других архитектур, которые используют механизм прерывания для подобных целей. Для чтения информации с консоли, необходимо записать код вызова в регистр *a0* (для чтения 5), полученная в ходе операции строка будет записана в регистр *a0*. Пример вывода и чтения информации с консоли приведен в листинге кода 1.4.

```
# вывести сообщение
li a0, 4
la a1, msg
ecall
# считать данные с консоли
li a0, 5
ecall
mv t3, a0
```

Листинг кода 1.4: Пример вывода и чтения информации с консоли.

Симулятор Jupiter предоставляет возможность пошагового выполнения работы программы (*step*), на каждом шаге (такте процессора) можно посмотреть текущее состояние всех регистров процессора (*Registers*), памяти (областей *text*, *stack*, *heap*), а также кэша (*cache*). Для сборки кода и последующего запуска необходимо выбрать в меню *Run* пункт *Assemble*. В листинге 1.5 приведен пример простого кода, осуществляющего сортировку массива, лежащего в памяти по возрастанию, с дальнейшей записью его в ту же область памяти.

```
.globl __start

.data
array: .word 7, 2, 3, 4, 20, 0, 0xA, 10 # массив для сортировки
_i: .word 0  # начальные значения итератора 1 элемента сравнения
_j: .word 4  # начальные значения итератора 2 элемента сравнения
_i_max: .word 32
_j_max: .word 28

.text
change:  # поменять элементы в регистрах местами
  mv a2, t6
  mv t6, t5
  mv t5, a2
  j  return

exit:
  li   a0, 10
  ecall

__start:
  lw   s9, _i_max
  lw   s8, _j_max
  lw   t0, _i
  lw   t1, _j
  la   t2, array
  add  t3, t2, t0  # адрес i-ого элемента сравнения
  add  t4, t2, t1  # адрес j-ого элемента сравнения

```

```
30 internal_cycle:
31   lw   t5, 0(t3)  # i-ый элемент сравнения - загружаем в регистр t5 (адрес
     ↪  лежит в регистре t3, смещение 0)
32
33   lw   t6, 0(t4)      # j-ый элемент сравнения
34   bge  t5, t6, change  # если i-ый элемент больше либо равен j-ого элемента
35
36 return:
37   sw   t5, 0(t3)  # сохранить в память по адресу t3 значение в t5
38   sw   t6, 0(t4)
39   addi t4, t4, 4  # сдвигаем указатель на адрес j-ого элемента сравнения
40   addi t1, t1, 4
41   blt  t1, s9, internal_cycle  # если сравнили i-ый элемент со всеми
     ↪  последующими элементами
42
43   addi t3, t3, 4  # сдвигаем указатель на адрес i-ого элемента сравнения
44   addi t0, t0, 4
45   addi t4, t3, 4
46   addi t1, t0, 4
47   blt  t0, s8, internal_cycle
48
49   jal  ra, exit
```

Листинг кода 1.5: Пример реализации алгоритма сортировки пузырьком.

1.4 Практическая часть

1.4.1 Контрольные вопросы

1. Классификации СнК.
2. Преимущества и недостатки СнК.
3. Перечислить этапы процесса производства и проектирования СнК.
4. Перечислить типы инструкций базового набора RV32I с примерами к каждому типу.
5. Для чего используются директивы ассемблера RISC-V?
6. Написание подпрограмм на ассемблере RISC-V, привести пример программы.

1.5 Полезные ссылки

1. The RISC-V Instruction Set Manual:
   `https://riscv.org/specifications/`
2. RISC-V Reference Card:
   `https://www.cl.cam.ac.uk/teaching/1617/ECAD+Arch/files/docs/RISCVGreenCardv8-20151013.pdf`

3. The RISC-V Instruction Reference:
   `https://rv8.io/isa`

4. RISC-V Assembler Reference:
   `https://rv8.io/asm.html`

1.6 Приложение к лабораторной работе

Таблица 1.3: Базовый целочисленный набор инструкций RV32I .

Начало таблицы 1.3		
Format	**Name**	**Pseudocode**
LUI rd,imm	Load Upper Immediate	$rd \leftarrow imm$
AUIPC rd,offset	Add Upper Immediate to PC	$rd \leftarrow pc + offset$
JAL rd,offset	Jump and Link	$rd \leftarrow pc + length(inst)$ $pc \leftarrow pc + offset$
JALR rd,rs1,offset	Jump and Link Register	$rd \leftarrow pc + length(inst)$ $pc \leftarrow (rs1 + offset) \wedge -2$
BEQ rs1,rs2,offset	Branch Equal	$if(rs1 = rs2)\ then$ $pc \leftarrow pc + offset$
BNE rs1,rs2,offset	Branch Not Equal	$if(rs1 \neq rs2)\ then$ $pc \leftarrow pc + offset$
BLT rs1,rs2,offset	Branch Less Than	$if(rs1 < rs2)\ then$ $pc \leftarrow pc + offset$
BGE rs1,rs2,offset	Branch Greater than Equal	$if(rs1 \geq rs2)\ then$ $pc \leftarrow pc + offset$
BLTU rs1,rs2,offset	Branch Less Than Unsigned	$if(rs1 < rs2)\ then$ $pc \leftarrow pc + offset$
BGEU rs1,rs2,offset	Branch Greater than Equal Unsigned	$if(rs1 \geq rs2)\ then$ $pc \leftarrow pc + offset$
LB rd,offset(rs1)	Load Byte	$rd \leftarrow s8[rs1 + offset]$
LH rd,offset(rs1)	Load Half	$rd \leftarrow s16[rs1 + offset]$
LW rd,offset(rs1)	Load Word	$rd \leftarrow s32[rs1 + offset]$
LBU rd,offset(rs1)	Load Byte Unsigned	$rd \leftarrow u8[rs1 + offset]$
LHU rd,offset(rs1)	Load Half Unsigned	$rd \leftarrow u16[rs1 + offset]$
SB rs2,offset(rs1)	Store Byte	$u8[rs1 + offset] \leftarrow rs2$
SH rs2,offset(rs1)	Store Half	$u16[rs1 + offset] \leftarrow rs2$
SW rs2,offset(rs1)	Store Word	$u32[rs1 + offset] \leftarrow rs2$
ADDI rd,rs1,imm	Add Immediate	$rd \leftarrow rs1 + sx(imm)$
SLTI rd,rs1,imm	Set Less Than Immediate	$rd \leftarrow sx(rs1) < sx(imm)$
SLTIU rd,rs1,imm	Set Less Than Immediate Unsigned	$rd \leftarrow ux(rs1) < ux(imm)$
XORI rd,rs1,imm	Xor Immediate	$rd \leftarrow ux(rs1) \oplus ux(imm)$
ORI rd,rs1,imm	Or Immediate	$rd \leftarrow ux(rs1) \vee ux(imm)$
ANDI rd,rs1,imm	And Immediate	$rd \leftarrow ux(rs1) \wedge ux(imm)$
SLLI rd,rs1,imm	Shift Left Logical Immediate	$rd \leftarrow ux(rs1) \ll ux(imm)$
SRLI rd,rs1,imm	Shift Right Logical Immediate	$rd \leftarrow ux(rs1) \gg ux(imm)$
SRAI rd,rs1,imm	Shift Right Arithmetic Immediate	$rd \leftarrow sx(rs1) \gg ux(imm)$

Продолжение таблицы 1.3		
Format	**Name**	**Pseudocode**
ADD rd,rs1,rs2	Add	$rd \leftarrow sx(rs1) + sx(rs2)$
SUB rd,rs1,rs2	Subtract	$rd \leftarrow sx(rs1) - sx(rs2)$
SLL rd,rs1,rs2	Shift Left Logical	$rd \leftarrow ux(rs1) \ll rs2$
SLT rd,rs1,rs2	Set Less Than	$rd \leftarrow sx(rs1) < sx(rs2)$
SLTU rd,rs1,rs2	Set Less Than Unsigned	$rd \leftarrow ux(rs1) < ux(rs2)$
XOR rd,rs1,rs2	Xor	$rd \leftarrow ux(rs1) \oplus ux(rs2)$
SRL rd,rs1,rs2	Shift Right Logical	$rd \leftarrow ux(rs1) \gg rs2$
SRA rd,rs1,rs2	Shift Right Arithmetic	$rd \leftarrow sx(rs1) \gg rs2$
OR rd,rs1,rs2	Or	$rd \leftarrow ux(rs1) \vee ux(rs2)$
AND rd,rs1,rs2	And	$rd \leftarrow ux(rs1) \wedge ux(rs2)$
FENCE pred,succ	Fence	
FENCE.I	Fence Instruction	
Окончание таблицы		

Таблица 1.4: Псевдо-инструкции архитектуры RISC-V.

Начало таблицы 1.4		
Pesudo-instruction	**Expansion**	**Description**
nop	addi zero,zero,0	No operation
li rd, expression	(several expansions)	Load immediate
la rd, symbol	(several expansions)	Load address
mv rd, rs1	addi rd, rs, 0	Copy register
not rd, rs1	xori rd, rs, -1	One's complement
neg rd, rs1	sub rd, x0, rs	Two's complement
negw rd, rs1	subw rd, x0, rs	Two's complement Word
sext.w rd, rs1	addiw rd, rs, 0	Sign extend Word
seqz rd, rs1	sltiu rd, rs, 1	Set if = zero
snez rd, rs1	sltu rd, x0, rs	Set if ≠ zero
sltz rd, rs1	slt rd, rs, x0	Set if < zero
sgtz rd, rs1	slt rd, x0, rs	Set if > zero
fmv.s frd, frs1	fsgnj.s frd, frs, frs	Single-precision move
fabs.s frd, frs1	fsgnjx.s frd, frs, frs	Single-precision absolute value
fneg.s frd, frs1	fsgnjn.s frd, frs, frs	Single-precision negate
fmv.d frd, frs1	fsgnj.d frd, frs, frs	Double-precision move
fabs.d frd, frs1	fsgnjx.d frd, frs, frs	Double-precision absolute value
fneg.d frd, frs1	fsgnjn.d frd, frs, frs	Double-precision negate
beqz rs1, offset	beq rs, x0, offset	Branch if = zero
bnez rs1, offset	bne rs, x0, offset	Branch if ≠ zero
blez rs1, offset	bge x0, rs, offset	Branch if ≤ zero
bgez rs1, offset	bge rs, x0, offset	Branch if ≥ zero
bltz rs1, offset	blt rs, x0, offset	Branch if < zero
bgtz rs1, offset	blt x0, rs, offset	Branch if > zero
bgt rs, rt, offset	blt rt, rs, offset	Branch if >
ble rs, rt, offset	bge rt, rs, offset	Branch if ≤
bgtu rs, rt, offset	bltu rt, rs, offset	Branch if >, unsigned

Продолжение таблицы 1.4		
Pesudo-instruction	**Expansion**	**Description**
bleu rs, rt, offset	bltu rt, rs, offset	Branch if ≤, unsigned
j offset	jal x0, offset	Jump
jr offset	jal x1, offset	Jump register
ret	jalr x0, x1, 0	Return from subroutine
Окончание таблицы		

Таблица 1.5: Директивы ассемблера RISC-V.

Начало таблицы 1.5		
Directive	**Arguments**	**Description**
Directives for emitting data		
.2byte		16-bit comma separated words (unaligned)
.4byte		32-bit comma separated words (unaligned)
.8byte		64-bit comma separated words (unaligned)
.half		16-bit comma separated words (naturally aligned)
.word		32-bit comma separated words (naturally aligned)
.dword		64-bit comma separated words (naturally aligned)
.byte		8-bit comma separated words
.dtpreldword		64-bit thread local word
.dtprelword		32-bit thread local word
.sleb128	expression	signed little endian base 128, DWARF
.uleb128	expression	unsigned little endian base 128, DWARF
.asciz	“string”	emit string (alias for .string)
.string	“string”	emit string
.incbin	“filename”	emit the included file as a binary sequence of octets
.zero	integer	zero bytes
Directives for control of alignment		
.align	integer	align to power of 2 (alias for .p2align)
.balign	b,[pad_val=0]	byte align
.p2align	p2,[pad_val=0],max	align to power of 2
Directives for definition and exporing of symbols		
.globl	symbol_name	emit symbol_name to symbol table (scope GLOBAL)
.local	symbol_name	emit symbol_name to symbol table (scope LOCAL)
.equ	name, value	constant definition
Directives for selection of sections		
.text		emit .text section (if not present) and make current
.data		emit .data section (if not present) and make current
.rodata		emit .rodata section (if not present) and make current

Продолжение таблицы 1.5		
Directive	**Arguments**	**Description**
.bss		emit .bss section (if not present) and make current
.comm	symbol_name,size,align	emit common object to .bss section
.common	symbol_name,size,align	emit common object to .bss section
.section	[.text,.data,.rodata,.bss]	emit section (if not present, default .text) and make current
Directives includes options, macros and other miscellaneous functions		
.option	rvc,norvc,pic,nopic,push,pop	RISC-V options
.macro	name arg1 [, argn]	begin macro definition argname to substitute
.endm		end macro definition
.file	“filename”	emit filename FILE LOCAL symbol table
.ident	“string”	accepted for source compatibility
.size	symbol, symbol	accepted for source compatibility
.type	symbol, @function	accepted for source compatibility
Окончание таблицы		

Лабораторная работа 2

Системные шины

Системная шина (system bus) – совокупность сигнальных линий, служащих для обмена информацией между элементами системы на кристалле или на печатной плате. Сигналы системной шины в зависимости от назначения можно разделить на три группы. Линии системной шины, отвечающие за передачу данных, называется шиной данных. Линии, передающие адрес, называются шиной адреса, а прочие управляющие сигналы – шиной управления. Как было замечено ранее, основной функцией системной шины является обмен информацией. В простейшем случае происходит обмен информацией между одним процессорным ядром и множеством контроллеров. Обобщённая структурная схема подключения процессора и контроллеров к системной шине показана на рисунке 2.1.

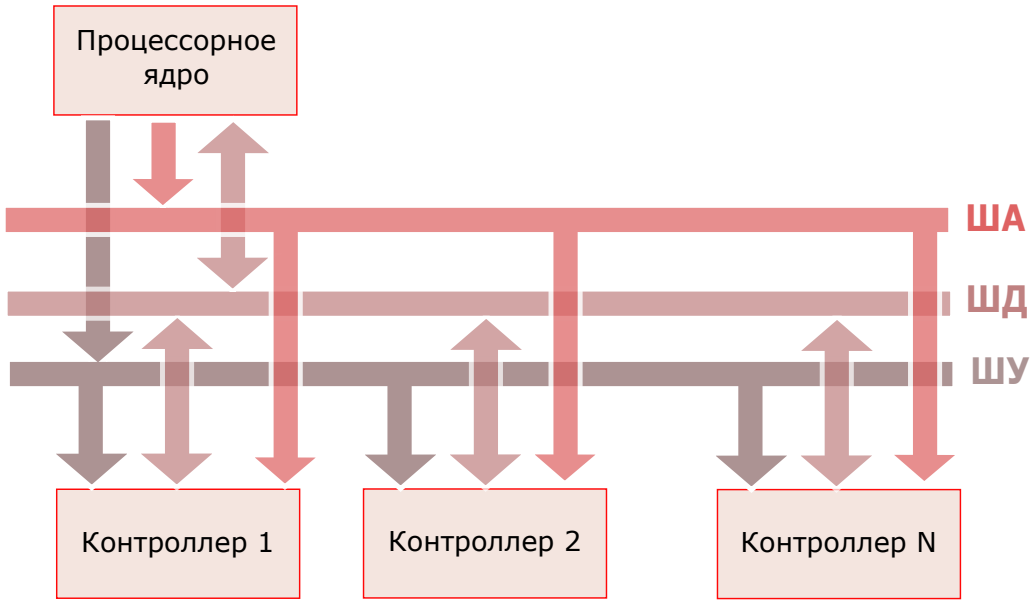

Рис. 2.1: Обобщённая структурная схема подключения процессора и контроллеров к системной шине.

На рисунке 2.2 приведена диаграмма обобщенной конструкции системной шины.

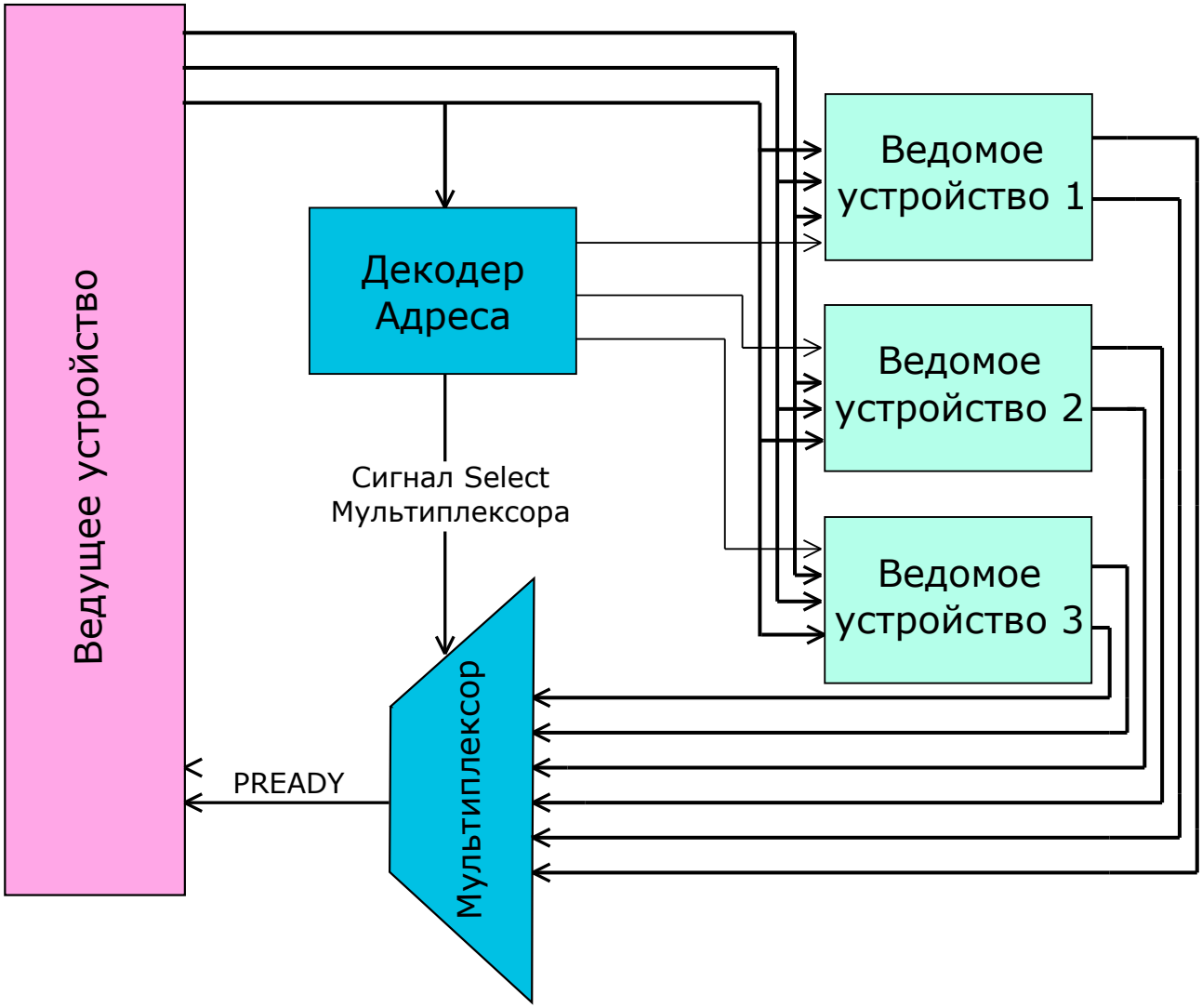

Рис. 2.2: Обобщённая конструкция системной шины.

Существует ряд терминов, которые обычно используются в спецификациях шин для систем на кристалле.

- **Ведущее устройство (Master)** – устройство, которое инициирует передачу данных для чтения или записи по системной шине (например, процессор или ЦОС-процессор).
- **Ведомое устройство (Slave)** – устройство, которое не инициирует обмен данными и отвечает только на входящие запросы на передачу.
- **Декодер** относится к логическому блоку, который дешифрует адрес назначения передачи данных, инициированной ведущим устройством, и выбирает соответствующее ведомое устройство для приема данных.
- **Мультиплексор** используется для мультиплексирования шины считанных данных и ответных сигналов от подчиненных устройств к ведущему. Декодер обеспечивает управление мультиплексором.

Теперь вы знаете, что такое системная шина, ее обобщенную организацию и основные термины, относящееся к понятию системная шина. Реализация СнК не возможна без системной шины. Однако действительно ли использование стандартизированных системной шины оправдано? Да, использование стандартизированных системных шин имеет ряд преимуществ, среди которых:

- Существует огромное количество IP-блоков, которые мы можем подключить в проект, который использует стандартизированную системную шину, без изменения исходных файлов IP-блока;
- Масштабируемость проекта – возможность с легкостью подключать новые IP-блоки или даже несколько одинаковых;
- Простота ПО для систем со стандартизированными системными шинами, так как организация адресного пространства таких шин достаточно простая.

Операции чтения и записи на системной шине называются транзакциями. Существует три вида транзакций, каждая из которых используется в определенных стандартах системных шин, в зависимости от назначения:

1. Single

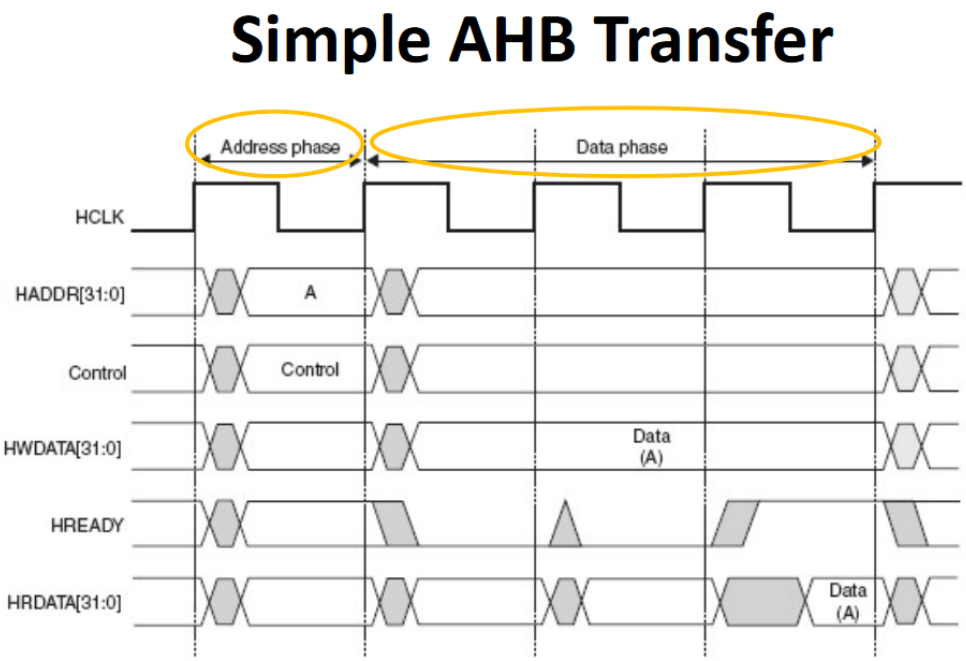

Рис. 2.3: Single транзакция.

2. Pipeline

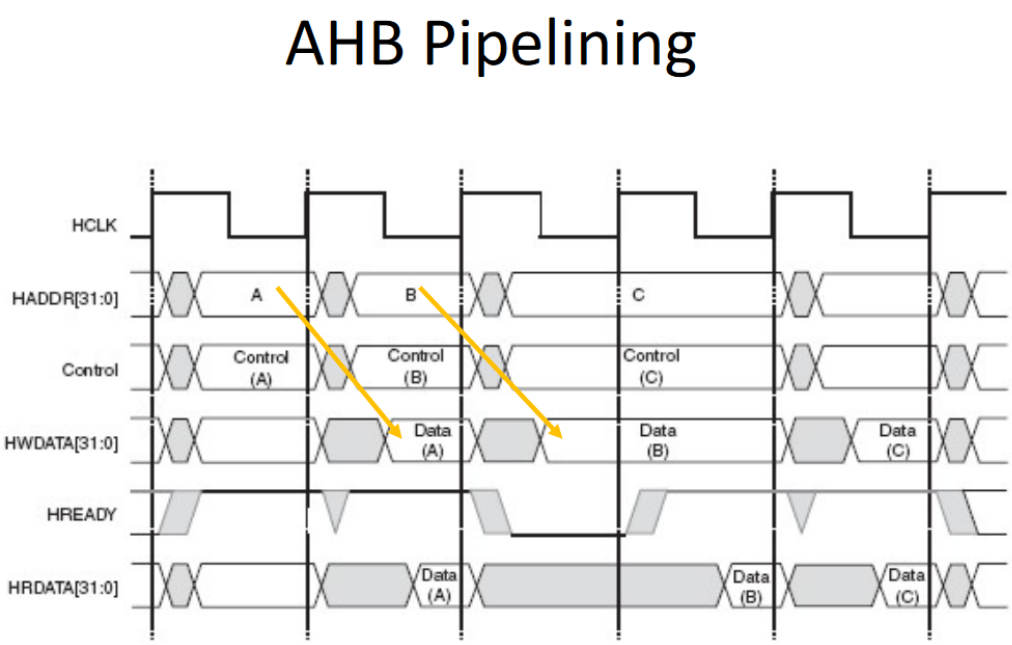

Рис. 2.4: Pipeline транзакция.

3. Burst

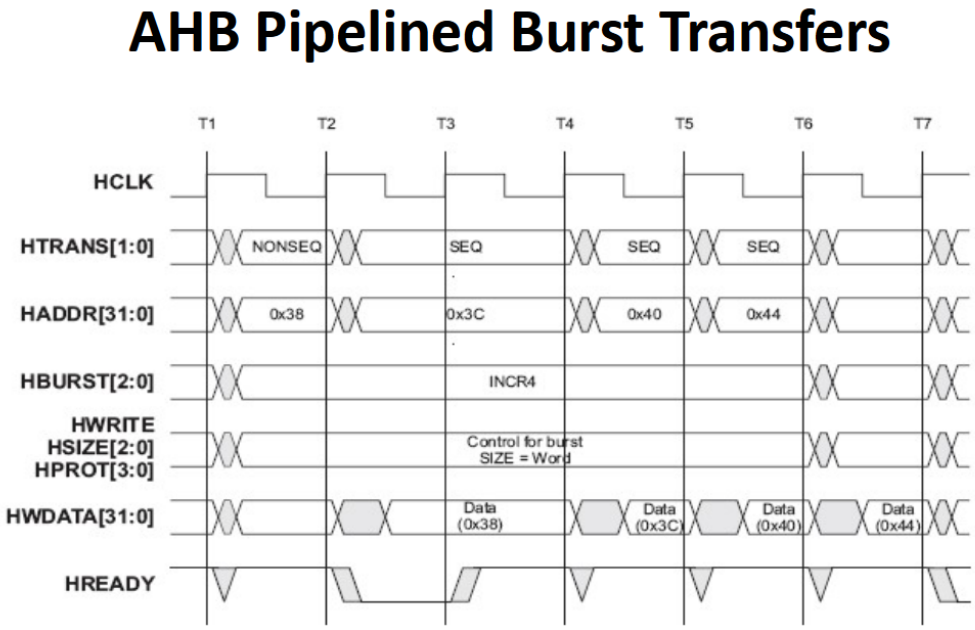

Рис. 2.5: Burst транзакция.

Транзакции отличаются скоростью передачи данных, сложностью арбитража и реализацией. Pipeline является промежуточным вариантом между двумя другим и используется для повышения производительности. Burst транзакция используется для обмена большим количеством данных, например для передачи данных во внешнюю память.

Рассмотрим пример транзакции по системной шине:

1. Ведущее устройство выбирает одно периферийное устройство и назначает адрес системной шине. В тоже время он устанавливает на шину управляющие сигналы (например, сигнал чтения);

2. Ведущее устройство ожидает ответа ведомого устройства (например, периферийного устройства).

3. Как только ведомое устройство готово, оно отправляет запрошенные данные ведущему устройству. Одновременно ведомое устройство устанавливает сигнал готовности на шине управления.

4. Наконец, мастер считывает переданные данные и может инициировать следующую транзакцию.

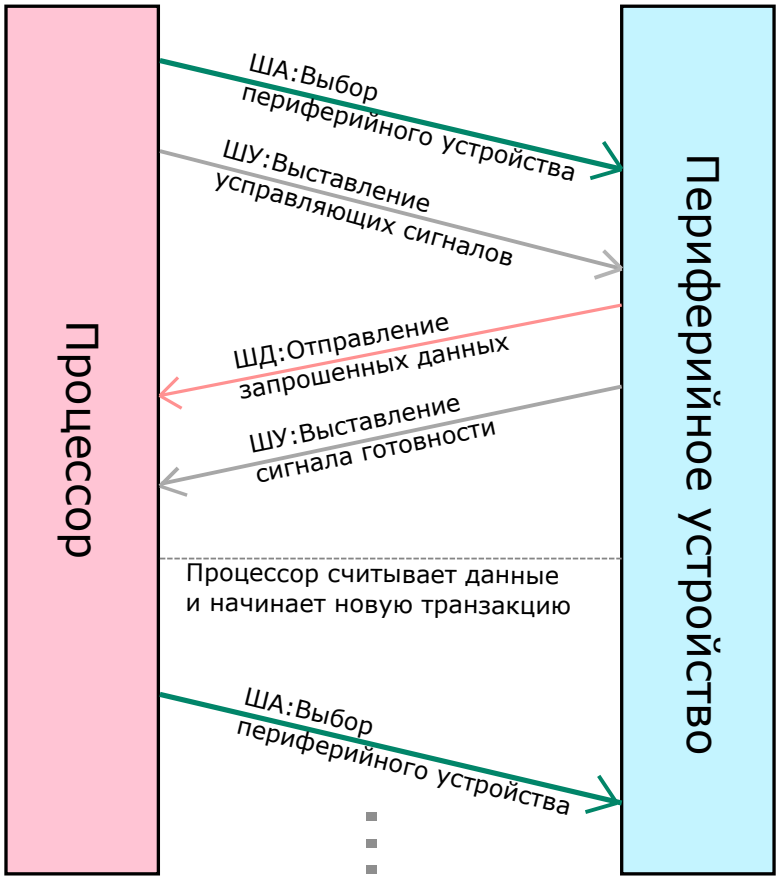

Рис. 2.6: Пример транзакции по системной шине.

2.1 Классификация системных шин и существующие стандарты

Используемые в настоящее время шины отличаются по:

- **Разрядности** (8, 16, 32, 64 бит). Чем больше разрядность шины, тем больше информации может быть передано за один цикл чтения или записи по каналу. Разрядность шины адреса можно определять независимо от разрядности шины данных. Разрядность шины адреса показывает, сколько ячеек памяти можно адресовать при передаче данных.

- **Способу передачи сигнала** (последовательные или параллельные)

- **Пропускной способности**. Ширина полосы пропускания называется также пропускной способностью и показывает общий объем данных, который можно передать по шине за данную единицу времени.

- Шины могут быть **синхронными** (осуществляющими передачу данных только по тактовым импульсам) и **асинхронными** (осуществляющими передачу данных в произвольные моменты времени).

Далее рассмотрим существующие стандарты системных шин.

Advanced Microcontroller Bus Architecture (AMBA) – это открытый стандарт требований к внутрикристалльным межсоединениям для соединения и управления функциональными блоками в системах на кристалле. Она облегчает развитие многопроцессорных разработок с большим числом контроллеров и периферии. Несмотря на название, с самого своего начала, AMBA имела виды, уходящие далеко за границы микроконтроллерных устройств. Сегодня AMBA широко применяется в ряде частей ASIC и SoC, включая прикладные процессоры, применяемые в современных небольших переносных устройствах вроде смартфонов.

Существует несколько разновидностей стандарта AMBA. На рисунке 2.7 представлены первые 4 стандарта AMBA. В таблице 2.1 представлены наиболее популярные шины стандартов AMBA.

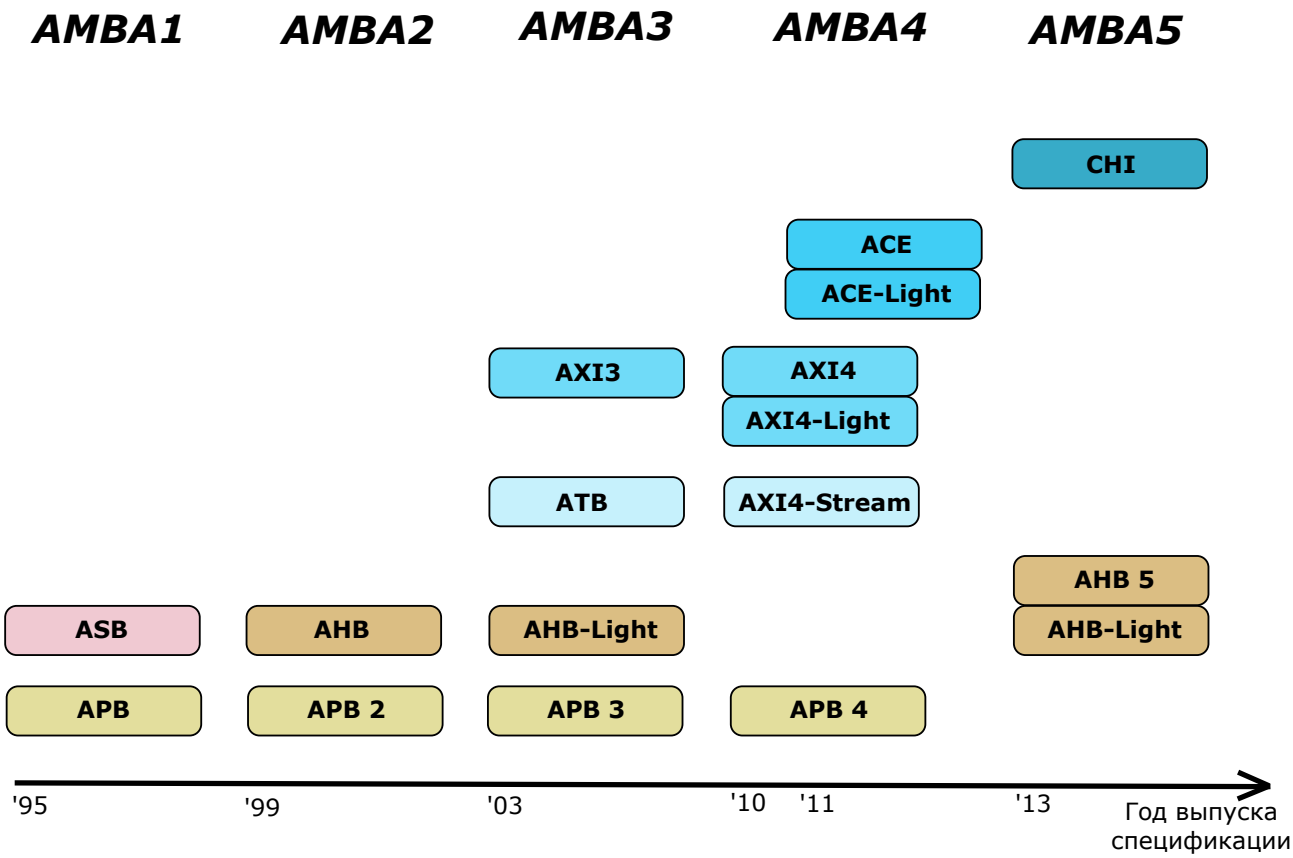

Рис. 2.7: Стандарты AMBA.

Таблица 2.1: Шины AMBA.

Начало таблицы 2.1		
Стандарт	**Шина**	**Описание**

Продолжение таблицы 2.1

Стандарт	**Шина**	**Описание**
AMBA1	Advanced System Bus (ASB)	ASB – это первое поколение системной шины AMBA. ASB реализует функции, необходимые для высокопроизводительных систем, включая: пакетные передачи, конвейерные передачи, режим мульти-мастера (Возможность более 1 ведущего устройства). В качестве ведущих устройств могут быть, например, процессорное ядро, ЦОС-процессор. В качестве ведомых устройств мост APB, устройства памяти, а также любые другие. Однако периферийные устройства с низкой пропускной способностью обычно находятся на шине APB.
AMBA1, AMBA2, AMBA3, AMBA4	Advanced Peripheral Bus (APB, APB3, APB4)	APB является частью иерархии шин AMBA и оптимизирована для минимального энергопотребления и снижения сложности интерфейса. APB выглядит как локальная вторичная шина, которая инкапсулирована как одно ведомое устройство AHB или ASB. Мост APB выглядит как подчиненный модуль, который обрабатывает передачу управляющих сигналов от имени локальной периферийной шины. APB следует использовать для взаимодействия с любыми периферийными устройствами, которые имеют низкую пропускную способность и не требуют высокой производительности интерфейса конвейерной шины.
AMBA2, AMBA5	Advanced High-performance Bus (AHB, AHB5)	AHB – это шина второго поколения шин AMBA, предназначенная для удовлетворения требований высокопроизводительных синтезируемых конструкций. Это высокопроизводительная системная шина, которая поддерживает несколько ведущих устройств шины и обеспечивает работу с высокой пропускной способностью. AHB реализует функции, необходимые для высокопроизводительных систем с высокой тактовой частотой, включая: пакетные передачи, разделенные транзакции, однотактная процедура арбитража мастеров, реализация передачи без третьего состояния и более широкие конфигурации шины данных (64/128 бит). Мостовое соединение между этим более высоким уровнем шины и текущим ASB / APB может быть эффективно выполнено, чтобы обеспечить простую интеграцию любых существующих конструкций. Широко применяется в разработках, основанных на ARM7, ARM9 и ARM Cortex-M.

Продолжение таблицы 2.1

Стандарт	Шина	Описание
AMBA3, AMBA4	Advanced Extensible Interface (AXI3, AXI4)	Протокол AXI поддерживает высокопроизводительные высокочастотные системы. Протокол AXI подходит для широкополосных сетей и систем с низкой задержкой, обеспечивает высокочастотную работу без использования сложных мостов, отвечает требованиям интерфейса широкого диапазона компонентов, подходит для контроллеров памяти с высокой начальной задержкой доступа, обеспечивает гибкость в реализации архитектур межсоединений, имеет обратную совместимость с существующими интерфейсами AHB и APB. Основные характеристики протокола AXI: отдельный адрес / управление и данные фазы, поддержка передачи данных без выравнивания с использованием байтовых стробов, использует пакетные транзакции только с выданным начальным адресом, отдельные каналы чтения и записи данных, которые могут обеспечить недорогой прямой доступ к памяти (DMA),поддержка выдачи нескольких ожидающих адресов, поддержка завершения транзакции вне порядка. Широко применяется в процессорах ARM Cortex-A, включая Cortex-A9.
AMBA3, AMBA5	Advanced High-performance Bus Lite (AHB-Lite)	AHB-Lite отвечает требованиям высокопроизводительных синтезируемых конструкций. Это интерфейс шины, который поддерживает одного мастера шины и обеспечивает работу в широкой полосе пропускания, отсутствует арбитраж. AHB-Lite реализует функции, необходимые для высокопроизводительных систем с высокой тактовой частотой, в том числе: пакетные передачи, реализация без синхронизации, широкие конфигурации шины данных, 64, 128, 256, 512 и 1024 бит . Наиболее распространенными ведомыми устройствами AHB-Lite являются устройства внутренней памяти, интерфейсы внешней памяти и периферийные устройства с высокой пропускной способностью. Хотя периферийные устройства с низкой пропускной способностью могут быть включены в качестве ведомых устройств AHB-Lite, по соображениям производительности системы они обычно находятся на APB. Мостовое соединение между этим более высоким уровнем шины и APB выполняется с использованием ведомого устройства AHB-Lite, известного как мост APB.
AMBA4	AXI Coherency Extensions (ACE)	Протокол ACE расширяет протокол AXI4 дополнительным средством оповещения передач широкой когерентности и обеспечивает поддержку аппаратно-согласованных кэшей. Это средство когерентности позволяет множеству процессоров разделять память. Протокол ACE обеспечивает: Барьерные транзакции, которые гарантируют упорядочение транзакций в системе, функциональность распределенной виртуальной памяти для управления виртуальной памятью.

Продолжение таблицы 2.1

Стандарт	Шина	Описание
AMBA4	AXI Coherency Extensions Lite (ACE-Lite)	Интерфейс ACE-Lite является подмножеством полного интерфейса ACE. ACE-Lite используется ведущими компонентами, которые не имеют аппаратных когерентных кэшей. Lite состоит из интерфейса AXI4 с дополнительными сигналами на канале чтения адреса и канале записи адреса. ACE-Lite не включает в себя: адресный канал snoop, канал ответа snoop , сигнал подтверждения чтения, сигнал подтверждения записи, любые дополнительные биты ответа чтения.
AMBA4	Advanced Extensible Interface 4 Lite (AXI4-Lite)	Протокол AXI включает в себя спецификацию AXI4-Lite, подмножество AXI4 для связи с более простыми интерфейсами управления внутри компонентов, которые не требуют полной функциональности AXI4. Основные функциональные возможности AXI4-Lite: все транзакции имеют длину пакета 1, все обращения к данным используют полную ширину шины данных – AXI4-Lite поддерживает ширина шины данных 32-битная или 64-битная, все обращения не изменяются, не буферизируются, исключительные обращения не поддерживаются.
AMBA4	Advanced Extensible Interface 4 Stream (AXI4-Stream v1.0)	Протокол AXI4-Stream используется в качестве стандартного интерфейса для подключения компонентов, которые хотят обмениваться данными. Интерфейс может использоваться для подключения одного ведущего устройства, которое генерирует данные, к одному подчиненному устройству, которое получает данные. Протокол также можно использовать при подключении большего количества основных и подчиненных компонентов. Протокол поддерживает несколько потоков данных, используя один и тот же набор общих проводов, что позволяет создать общее межсоединение, которое может выполнять операции увеличения, уменьшения размера и маршрутизации. Интерфейс AXI4-Stream также поддерживает широкий спектр различных типов потоков. Потоковый протокол определяет связь между передачами и пакетами.

Окончание таблицы

Сегодня эти протоколы являются де-факто стандартными для встраиваемых процессоров, поскольку они хорошо описаны и могут применяться без отчислений.

Еще одним из популярных на текущий день стандартов является **Wishbone**. Шина Wishbone – параллельная шина для объединения модулей в системе на кристалле. Шина описана в открытой спецификации, и широко используется в проектах цифровых систем с открытым исходным кодом на сайте OpenCores.org. Стандарт допускает присутствие нескольких ведущих устройств в системе, а также различные топологии соединения модулей.

Общие характеристики:

- ширина шин адреса и данных: 8, 16, 32, 64 бит;
- тип шины: параллельная;
- внутренняя шина, используется только для соединения модулей на кристалле.

2.2 Адресное пространство системы на кристалле

При подключении различных периферийных устройств к системной шине возникает вопрос, каким образом различать эти контроллеры между собой? Решением данной проблемы является присвоение индивидуального диапазона адресов каждому из контроллеров. Таким образом, взаимодействие процессора с конкретным контроллером происходит посредством обращения по адресам заданного диапазона.

Весь доступный диапазон адресов называется **адресным пространством** (Memory Map) системы на кристалле, а диапазон адресов, выделенный для отдельно взятого контроллера, – адресным пространством контроллера. **Базовым адресом контроллера** называется адрес начала выделенного диапазона. Структурная схема адресного пространства системы на кристалле приведена на рисунке 3.8.

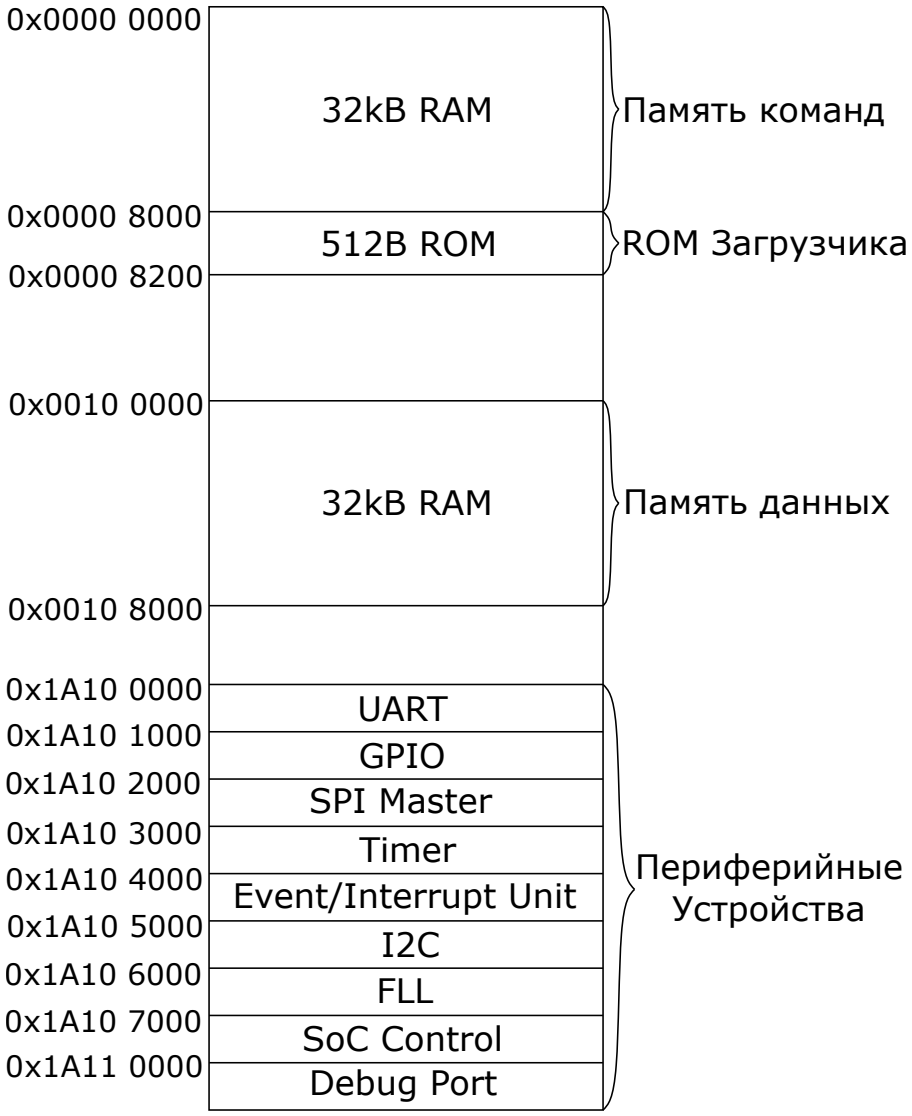

Рис. 2.8: Структурная схема адресного пространства.

Адресное пространство, выделенное под каждое периферийное устройство, определяется его стартовым (базовым) адресом и размером адресного пространства. Для обмена данными в периферийном устройстве (контроллере или вычислительном блоке) существует набор регистров, каждый из которых имеет уникальный адрес. Регистры адресуются относительно стартового адреса. Абсолютный адрес регистра в адресном пространстве системы на кристалле определяется как сумма базового адреса периферийного устройства и смещения (относитель-

ного адреса).

Так как архитектура RISC-V является load-store архитектурой, то для обращения к ячейке адресного пространства используются инструкции load и store для загрузки данных в регистры процессора и записи данных из регистра в ячейку памяти соответственно.

Следует отметить, что в RISC-V используется побайтовая адресация. Так как шина данных является 32-разрядной, то при обращении по адресу ноль происходит считывание или запись первых четырех байт (0,1,2,3). Этот факт необходимо учитывать при проектировании вычислительных блоков и при назначении регистрам адресов.

2.3 Системная шина APB

В нашей системе на кристалле мы используем системную шину **APB (Advanced Peripheral Bus)**.

Спецификация: `https://web.eecs.umich.edu/~prabal/teaching/eecs373-f12/readings/ARM_AMBA3_APB.pdf`

Шина APB – это часть семейства шин AMBA 3 фирмы ARM. Она представляет собой универсальный интерфейс для подключения периферийных устройств.

На рисунке 2.9 представлена диаграмма системной шины APB. Декодер управляется ведущим устройством, которое выставляет на системную шину адрес одного из ведомых устройств. По адресу декодер определяет, какое из устройств выбрано для обмена данными и формирует сигнал *PSEL*. Декодер формирует сигнал *Select* для мультиплексора, который указывает, данные какого из ведомых устройств должны быть выставлены на системную шину.

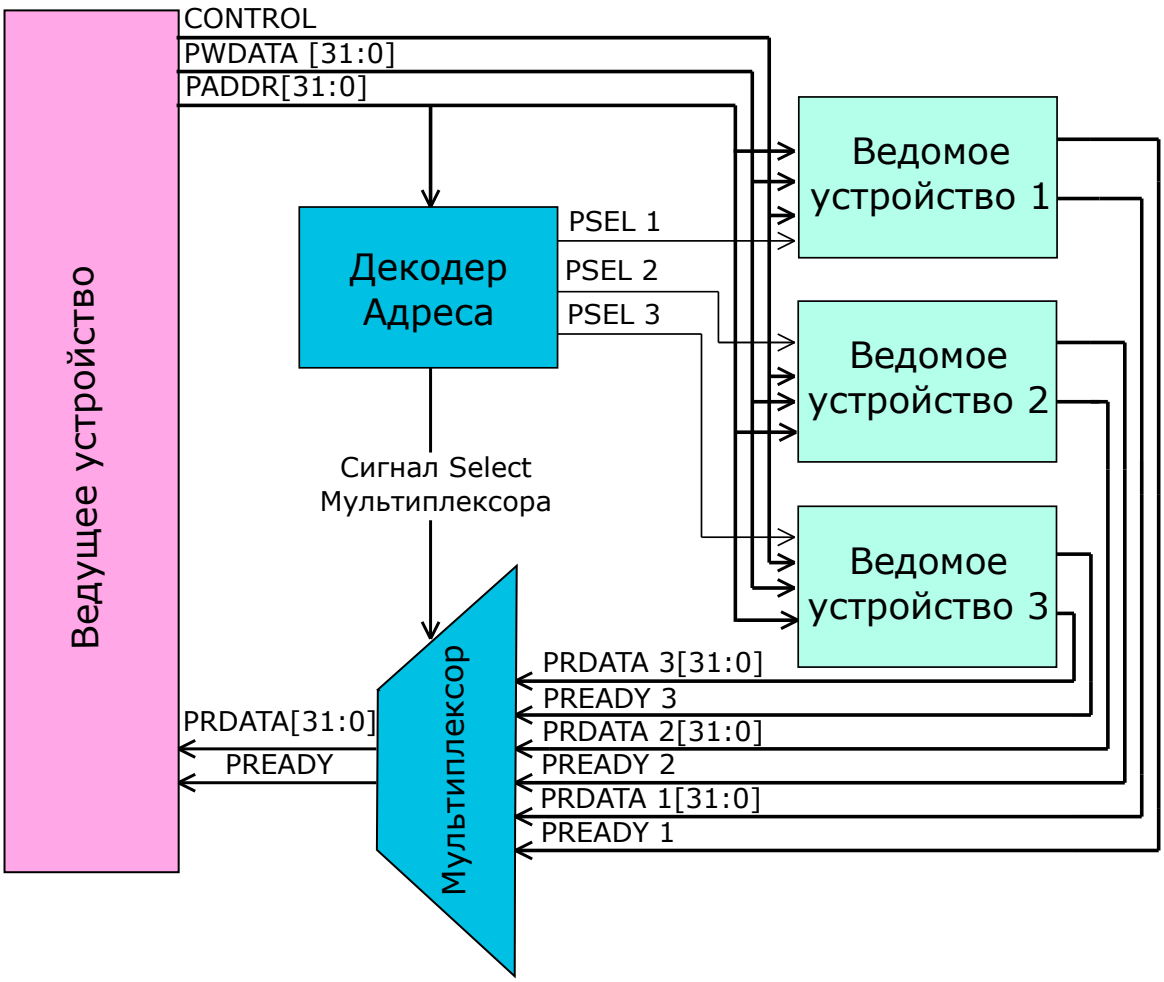

Рис. 2.9: Диаграмма системной шины APB.

Передача данных по шине APB состоит из двух фаз: фазы адреса и фазы данных. Фаза адреса всегда занимает один такт шины, а фаза данных может содержать состояния ожидания и длиться несколько тактов. Простейший цикл записи без состояний ожидания происходит следующим образом (рис. 2.10). В фазе адреса по нарастающему фронту тактового сигнала *PCLK* ведущее устройство устанавливает следующие сигналы:

- адрес ведомого устройства *PADDR*;
- сигнал записи *PWRITE* (активный уровень – высокий);
- сигнал выбора устройства *PSEL* (активный уровень – высокий);
- данные для записи *PWDATA*.

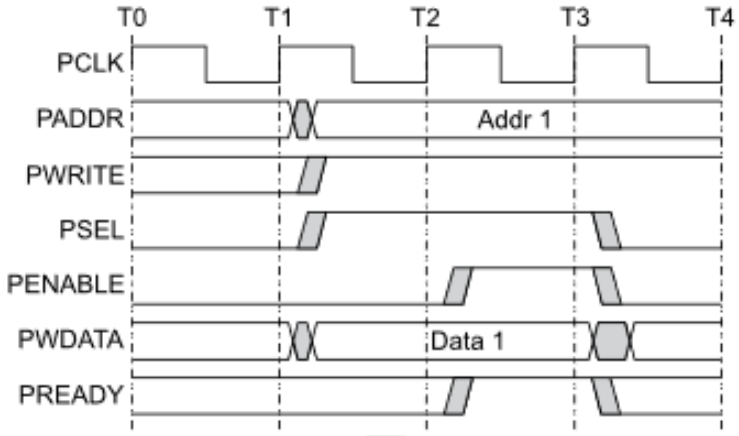

Рис. 2.10: Цикл записи по шине APB без состояния ожидания.

Состояния этих сигналов сохраняются и в фазе данных. По второму фронту тактового сигнала устанавливается сигнал *PENABLE* (активный уровень – высокий). Это означает начало фазы записи данных. До следующего такта ведомое устройство должно установить сигнал *PREADY* (активный уровень – высокий) и принять передаваемые данные. Получив сигнал *PREADY*, ведущее устройство по третьему такту снимает сигнал *PENABLE*. Сигнал выбора *PSEL* при этом также снимается, даже если следующее обращение будет происходить к тому же самому устройству. На этом цикл записи заканчивается.

Периферийное устройство может задержать окончание цикла записи (рис. 2.11). Для этого оно должно при активном сигнале *PENABLE* не устанавливать сигнал *PREADY* до тех пор, пока не закончит прием данных. В таком случае цикл записи закончится по первому фронту тактового сигнала, на котором будет обнаружен активный уровень *PREADY*.

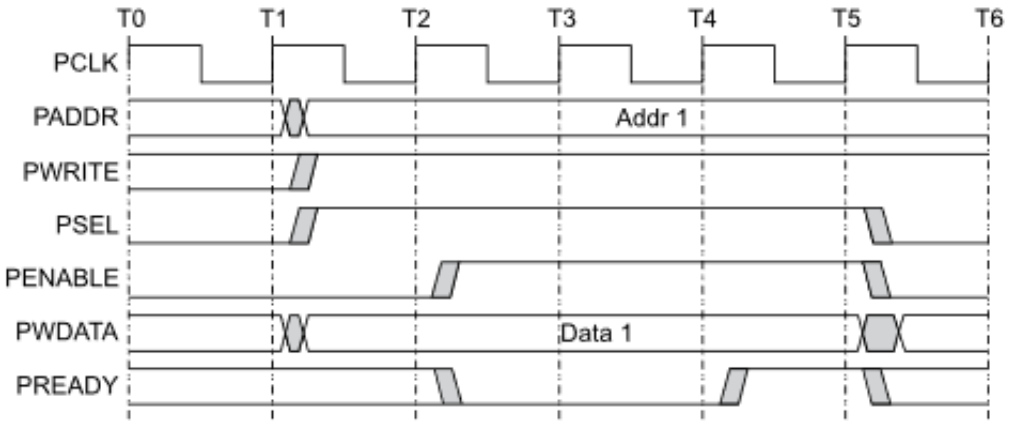

Рис. 2.11: Цикл записи по шине APB с состоянием ожидания.

Временные диаграммы циклов чтения без состояний ожидания (рис. 2.12) и с ними (рис. 2.13) выглядят похоже. Первое отличие состоит в том, что в фазе адреса сигнал *PWRITE* имеет низкий уровень. В этом случае при активном уровне сигнала *PENABLE* ведомое устройство должно выставить данные на линиях чтения *PRDATA*, сопровождая их активным уровнем сигнала *PREADY*. Если периферийное устройство хочет задержать цикл чтения, оно должно снять сигнал *PREADY* при высоком уровне сигнала *PENABLE*. Тогда ведущее устройство перейдет в состояние ожидания до тех пор, пока не получит активного уровня сигнала *PREADY*.

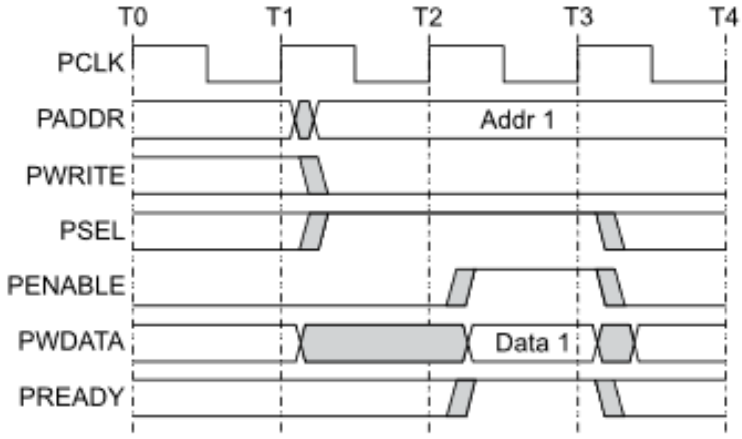

Рис. 2.12: Цикл чтения по шине APB без состояния ожидания.

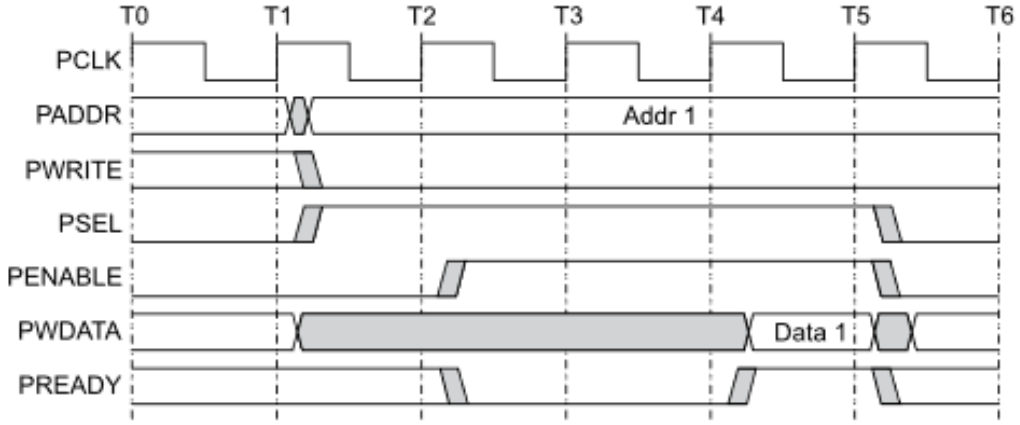

Рис. 2.13: Цикл чтения по шине APB с состоянием ожидания.

В таблице 2.2 представлены сигналы системной шины APB:

Таблица 2.2: Сигналы шины APB.

Имя сигнала	Описание
PCLK	Тактовая частота. Все действия по шине APB происходят по переднему фронту сигнала PCLK.
PRESETn	Сброс. Активный уровень сигнала – низкий.
PADDR	Шина адреса. Может иметь разрядность до 32 бит. Поступает от ведущего устройства.
PSEL	Выбор. Ведущее устройство вырабатывает этот сигнал отдельно для каждого периферийного обустройства. Этот сигнал показывает, что ведомое устройство выбрано для обмена данными.
PENABLE	Разрешение. Этот сигнал указывает второй и следующие такты передачи по APB(фаза данных).
PWRITE	Направление. Высокий уровень сигнала указывает на цикл записи данных в периферийное устройство, а низкий – цикл чтения из периферийного устройства.
PWDATA	Шина данных для записи. Может иметь разрядность до 32 бит.
PREADY	Сигнал готовности периферийного устройства.
PRDATA	Шина данных для чтения. Может иметь разрядность до 32 бит.
PSLVERR	Ошибка передачи. Периферийные устройства не обязаны его поддерживать. Если сигнал не используется, то не соответствующий вход ведущего устройства подается низкий уровень.

2.4 Проектирование аппаратного вычислителя

Постановка задачи: спроектировать аппаратный вычислитель с подключением по системной шине APB. Для примера рассмотрим проектирование вычислителя контрольной суммы CRC8. Для каких целей может потребоваться такой блок? Например, если устройство, в состав которого входит вычислитель, должно в режиме реального времени принимать массивы данных и проверять их целостность. Тогда полученный массив данных отправляется на блок CRC8, в котором происходит вычисление и далее через механизм прямого доступа к памяти записывается в память. Таким образом, можно проверять целостность данных, не затрачивая при этом процессорное время, как при использовании программных реализаций алгоритма.

Вычислитель должен принимать на вход данные по системной шине, от которых он вычисляет контрольную сумму и по сигналу чтения выдавать на выходную шину данных значение CRC8. Обобщенная структурная схема вычислителя CRC8, подключенного по шине APB приведена на рис. 2.14.

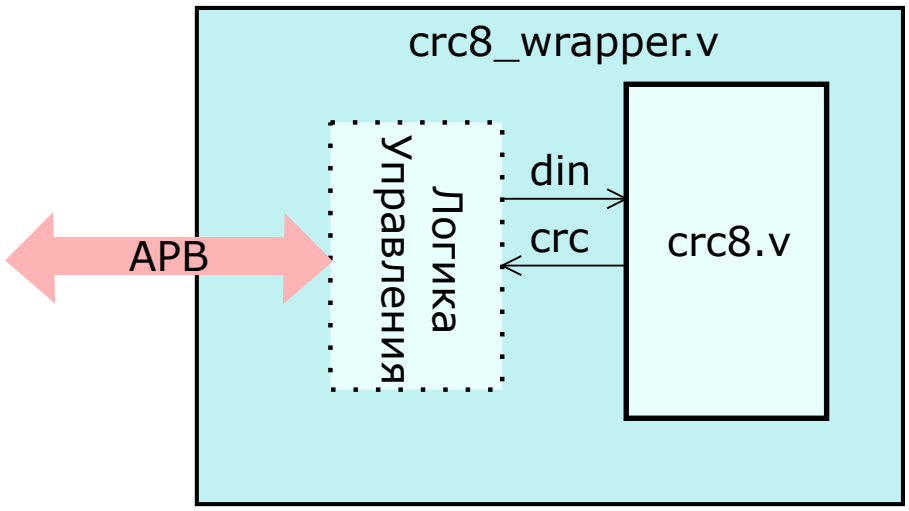

Рис. 2.14: Структурная схема вычислителя CRC8.

Вычислитель состоит из двух частей: модуля, реализующего непосредственно сам алгоритм CRC8 и модуля-оболочки, в который подключается модуль-вычислитель контрольной суммы. Также модуль-оболочка содержит в себе логику управления системной шиной.

Рассмотрим подробнее механизм управления обмена информацией по системной шине, который реализует модуль-оболочка. Как было сказано ранее, каждому из периферийных устройств выделено свое адресное пространство с базовым адресом. При подключении вычислителя в проект его будет нужно указать, а пока необходимо разобраться с адресацией внутри адресного пространства вычислителя. Ранее было рассказано про смещения, относительно базового адреса, которые определяют абсолютные адреса регистров.

$$Addr_{reg} = Addr_{base} + Offset_{reg},$$

где $Addr_{base}$ – базовый адрес контролера, а $Offset_{reg}$ – смещение относительно базового адреса.

Регистры вычислительного блока хранят в себе данные, которые потребуются для считывания или записанные данные. Какие регистры могут потребоваться для данного вычислителя? В самом базовом случае это будет регистр для записи данных для вычисления и регистр для считывания значения CRC. Таким образом получается два регистра, а значит два адреса, каждый из которых будет иметь свое уникальное смещение. В системе используется 32-разрядная системная шина и побайтовая адресация, поэтому необходимо использовать адреса кратные 4, чтобы покрывать пословно адресное пространство.

2.5 Реализация вычислителя CRC8

Для примера рассмотрим проектирование вычислителя контрольной суммы CRC8. В данном вычислительном блоке будет модуль, реализующий алгоритм CRC8, а также модуль-оболочка.

Начнем с проектирования модуля, реализующего алгоритм CRC8 (Заданный порождающий полином: $g(x) = x^8 + x^5 + x^4 + 1$).

На вход модуля поступают:

- сигнал тактирования *clk_i*;
- сигнал сброса *rst_i*;
- сигнал валидности данных *data_valid_i*;

- входная шина данных *din_i*;
- сигнал запроса на чтение *crc_rd* вычисленного значения CRC8 по шине *crc_o*. Разрядность данных была выбрана равной 8 бит.

Работа модуля построена по принципу работы конечного автомата с 3 состояниями (рис. 2.15):

Таблица 2.3: Состояния автомата модуля-вычислителя.

Состояние	Описание
IDLE	Состояние бездействия. Ожидание сигнала валидности данных или запроса на считывание CRC8.
BUSY	Состояние вычисления CRC8.
READ	Состояние считывания значения CRC8.

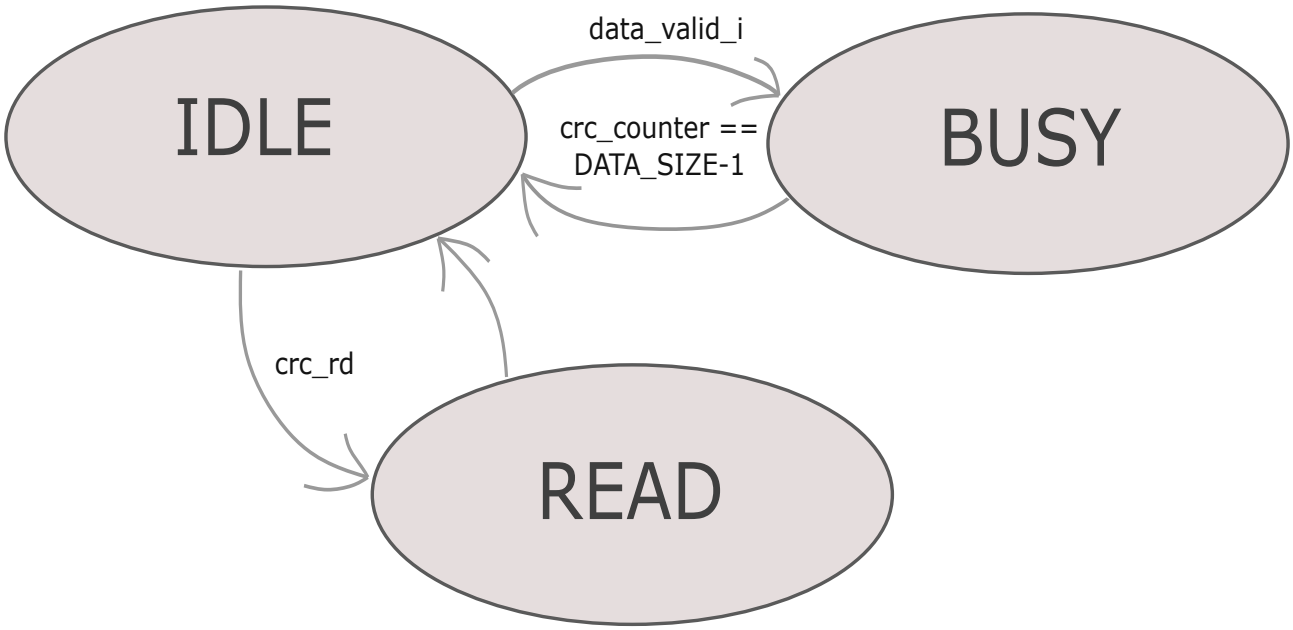

Рис. 2.15: Автомат состояний модуля-вычислителя.

**crc_counter* - счетчик для расчёта количества бит входного байта данных, обработанных алгоритмом вычисления контрольной суммы CRC8.

Код модуля-вычислителя с комментариями приведен в Листинге 2.1.

```
1  module crc8 (
2              input            clk_i,
3              input            rst_i,
4              input      [7:0] din_i,
5              input            data_valid_i,
6              input            crc_rd,
7              output reg [7:0] crc_o);
8
9
10 // Параметры для состояний автомата
11 localparam IDLE = 2'b00;
```

```
12 localparam BUSY = 2'b01;
13 localparam READ = 2'b10;
14
15 reg  [2:0] state;         // Регистр состояний
16 reg  [7:0] data_current;  // Текущие данные (сдвиговый регистр)
17 reg  [3:0] crc_counter;   // Регистр счетчик обработанных бит входного байта
   ↪  данных для состояния вычисления
18
19 always @(posedge clk_i)
20 begin
21     if (rst_i) // Сигнал сброса - обнуляем все регистры
22     begin
23         state        <= IDLE;
24         data_current <= 8'b0;
25         crc_o        <= 8'b0;
26         crc_counter   <= 4'd0;
27     end
28     else
29     begin
30         case (state)
31         IDLE:
32             begin
33                 crc_counter <= 4'b0;
34                 if (data_valid_i) // Если пришли новые данные - переходим
35                                   // в состояние вычисления
36                 begin
37                     state        <= BUSY;
38                     data_current <= din_i;
39                 end
40                 else if (crc_rd)state <= READ; // Если пришел запрос на чтение
                    ↪  - переходим в состояние чтения
41             end
42         BUSY:
43             begin
44                 crc_o[7] <= crc_o[0]^data_current[0];
45                 crc_o[6] <= crc_o[7];
46                 crc_o[5] <= crc_o[6];
47                 crc_o[4] <= crc_o[5];
48                 crc_o[3] <= (crc_o[0] ^ data_current[0])^ crc_o[4];
49                 crc_o[2] <= (crc_o[0] ^ data_current[0])^ crc_o[3];
50                 crc_o[1] <= crc_o[2];
51
52                 crc_o[0] <= crc_o[1];
53                 data_current <= {1'b0,data_current[7:1]};
54                 crc_counter   <= crc_counter+ 1'b1;
55                 if(crc_counter == 4'b0111) state <= IDLE;
56             end
57         READ:
58             begin
59                 crc_o <= 8'b0;
```

```
60                  state <= IDLE;
61              end
62          endcase
63        end
64      end
65  endmodule
```

Листинг кода 2.1: Модуль-вычислитель контрольной суммы CRC8.

Следующим шагом является написания модуля-оболочки. Так как реализуется контроллер, совершающий обмен данными по интерфейсу APB с 32-разрядной шиной в заголовке необходимо указать все сигналы интерфейса. Фрагмент кода, содержащий заголовок модуля оболочки приведен в Листинге 2.2.

```
1  module wrapper_crc8(
2                      input             p_clk_i,
3                      input             p_rst_i,
4                      input      [31:0] p_dat_i,
5                      output reg [31:0] p_dat_o,
6                      input             p_sel_i,
7                      input             p_enable_i,
8                      input             p_we_i,
9                      input      [31:0] p_adr_i,
10                     output reg        p_ready
11                     );
```

Листинг кода 2.2: Заголовок модуля-оболочки.

Для подключения модуля-вычислителя CRC необходимо объявить wire, которые будут подключаться к выводам модуля. Фрагмент кода приведен в Листинге 2.3.

```
1  wire [7:0] din_i; // Объявляем провода, которые будут подключаться к сигналам
   ↪  модуля
2  wire [7:0] crc_o;
3  wire       crc_rd;
4  wire       data_valid_i;
5
6  crc8 crc8(.clk_i(p_clk_i),  // При подключении модуля указываем имя и название
   ↪  модуля name module_name (...
7            .rst_i(!p_rst_i), // Подключаем к каждому сигналу модуля провод
             ↪  .signal_name(wire_name), ...
8            .din_i(din_i),
9            .data_valid_i(data_valid_i),
10           .crc_rd(crc_rd),
11           .crc_o(crc_o));
```

Листинг кода 2.3: Подключение модуля-вычислителя в модуль-оболочку.

Ранее было определено, что в базовой реализации будут использоваться два адреса, для обмена данными по системной шине – запись данных и чтение с адресами смещений 0 и 4 соответственно. Также необходимо генерировать строб *cs*, который будет определять фазу данных транзакции для того, чтобы считать данные с шины либо выставить данные на нее. Фрагмент кода приведен в Листинге 2.4.

```
reg cs_1;
reg cs_2;
// Формирование строба cs цикла чтения или записи по системной шине
always @ (posedge p_clk_i)
begin
    cs_1 <= p_enable_i & p_sel_i;
    cs_2 <= cs_1;
end
wire cs = cs_1 & (~cs_2);
```

Листинг кода 2.4: Генерация строба cs.

```
// Формирование выходных данных системной шины
always @ (*)
begin //Для чтения crc используем адрес 1
  if (cs & (~p_we_i) & p_adr_i[3:0] == 4'd4 )p_dat_o <= {24'd0, crc_o};
end

// Формирование сигналов на модуль-вычислитель

//Для записи данных для расчета crc используем адрес 0
assign data_valid_i = (cs &  p_we_i & p_adr_i[3:0] == 4'd0);

//Для записи данных для расчета crc используем адрес 0
assign din_i = (cs & p_we_i & p_adr_i[3:0] == 4'd0);

//Для чтения crc используем адрес 4
assign crc_rd = (cs & ~p_we_i & p_adr_i[3:0] == 4'd4);
```

Листинг кода 2.5: Формирование выходных данных системной шины и данных на модуль-вычислитель.

```
reg cs_ack1;
reg cs_ack2;
// Формирование сигнала готовности системной шины p_ready
always @ (posedge p_clk_i)
  begin
   cs_ack1 <= cs_2;
   cs_ack2 <= cs_ack1;
 end

always @ (posedge p_clk_i)
  begin
    p_ready <= (cs_ack1 & (~cs_ack2));
  end
```

Листинг кода 2.6: Формирование сигнала p_ready.

Как можно видеть в Листинге 2.5, на выходную шину данных значение crc выставляется только тогда, когда пришел запрос на обмен данными (строб *cs*), сигнал операции чтения (*p_we_i*) и совпадает адрес (*p_adr_i == 32'd4*). Аналогично выполняется операция записи, с той лишь разницей, что сигнал *p_we_i* должен быть равен единице (рис. 2.16 и рис. 2.17).

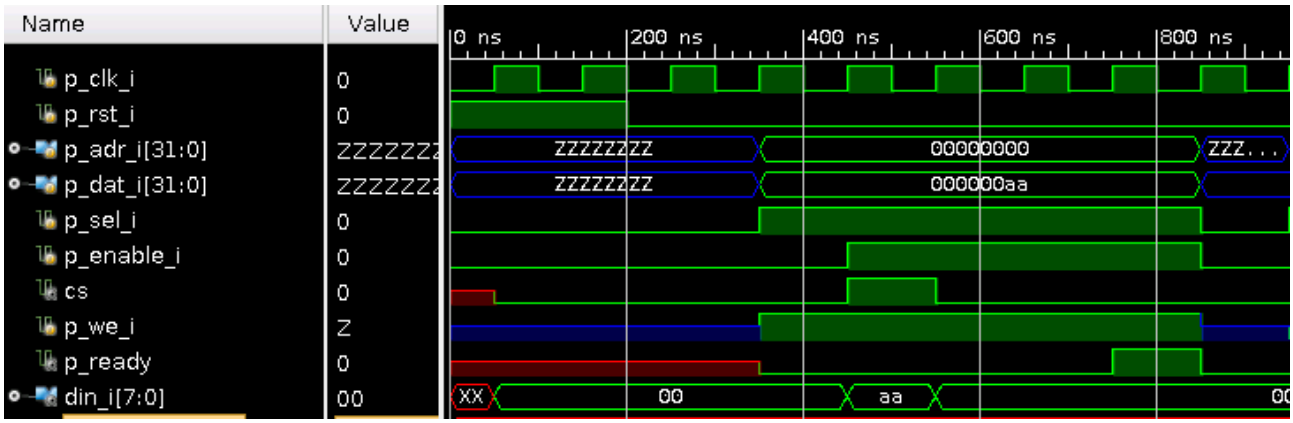

Рис. 2.16: Запись по адресу 32'd0.

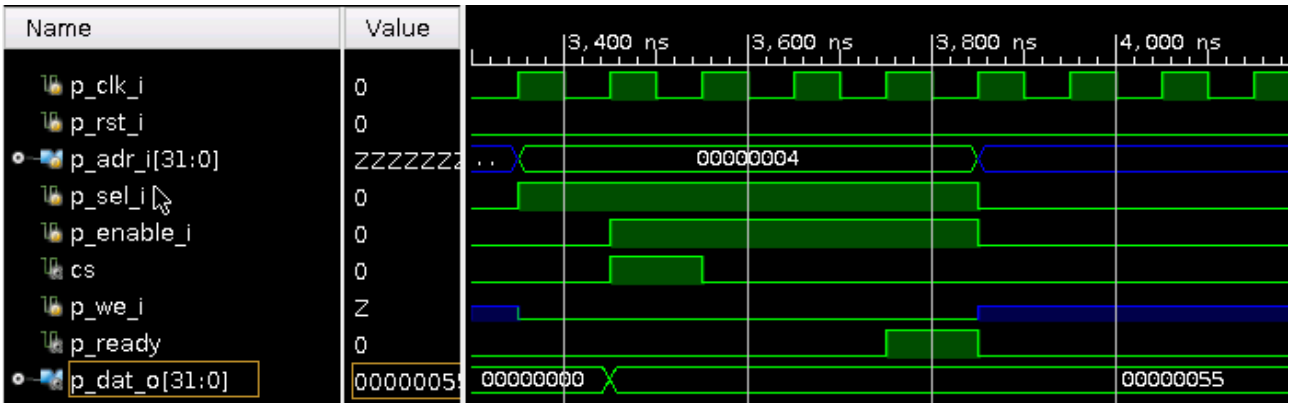

Рис. 2.17: Чтение по адресу 32'd4.

Полный исходный код модуля wrapper_crc8 показан в листинге 2.7.

```
`timescale 1ns / 1ps

module wrapper_crc8(
                    input             p_clk_i,
                    input             p_rst_i,
                    input      [31:0] p_dat_i,
                    output reg [31:0] p_dat_o,
                    input             p_sel_i,
                    input             p_enable_i,
                    input             p_we_i,
                    input      [31:0] p_adr_i,
                    output reg        p_ready,
                    output            p_slverr
                    );

wire [7:0] din_i;
wire [7:0] crc_o;
wire [1:0] state;
wire       crc_rd;
wire       data_valid_i;
assign p_slverr = 1'b0;

 crc8 crc8(.clk_i(p_clk_i),
           .rst_i(!p_rst_i),
           .din_i(din_i),
           .data_valid_i(data_valid_i),
           .crc_rd(crc_rd),
           .crc_o(crc_o),
           .state(state));

reg cs_1;
reg cs_2;

reg cs_ack1;
reg cs_ack2;

always @ (posedge p_clk_i)
begin
    cs_1 <= p_enable_i & p_sel_i;
    cs_2 <= cs_1;
end


wire cs = cs_1 & (~cs_2);

always @ (posedge p_clk_i)
  begin
   cs_ack1 <= cs_2;
   cs_ack2 <= cs_ack1;
```

```
50  end
51
52    // Generating acknowledge signal
53    always @ (posedge p_clk_i)
54    begin
55      p_ready <= (cs_ack1 & (~cs_ack2));
56    end
57
58  always @ (*)
59  begin
60    if (cs & (~p_we_i)& p_adr_i[3:0] == 4'd4)p_dat_o <= {24'd0,crc_o};
61    else if (cs & (~p_we_i)& p_adr_i[3:0] == 4'd8)p_dat_o <= {24'd0,state};
62  end
63
64
65  assign data_valid_i = (cs & p_we_i & p_adr_i[3:0]  == 4'd0);
66  assign din_i        = (cs & p_we_i & p_adr_i[3:0]  == 4'd0) ? p_dat_i[7:0]:
    ↪  8'd0;
67  assign crc_rd       = (cs & ~p_we_i & p_adr_i[3:0] == 4'd4);
68
69
70
71  endmodule
```

Листинг кода 2.7: Полный исходный код модуля wrapper_crc8.

2.6 Моделирование работы блока CRC8

Следующая задача состоит в том, чтобы проверить написанный нами вычислитель. Одним из инструментов верификации является моделирование. Для моделирования работы вычислителя или любого отдельного модуля и даже всей системы существуют специальные модули – **testbench**. Testbench имитирует входные воздействия и подает их на проверяемый модуль.

Для проверки блока сымитируем две транзакции записи данных по системной шине и одну транзакцию чтения значения CRC8.

Для проверки блока необходимо подключить его в testbench (аналогичным образом, как и модуль-вычислитель в оболочку). Фрагмент кода приведен в Листинге 2.8.

```
reg         p_clk_i;
reg         p_rst_i;
reg  [31:0] p_dat_i;
wire [31:0] p_dat_o;
reg         p_enable_i;
reg         p_sel_i;
reg         p_we_i;
reg  [31:0] p_adr_i;
wire        p_ready;

wrapper_crc8 wrapper_crc8
      ( .p_clk_i    (p_clk_i),
        .p_rst_i    (p_rst_i),
        .p_dat_i    (p_dat_i),
        .p_dat_o    (p_dat_o),
        .p_enable_i (p_enable_i),
        .p_sel_i    (p_sel_i),
        .p_we_i     (p_we_i),
        .p_adr_i    (p_adr_i),
        .p_ready    (p_ready)
        );
```

Листинг кода 2.8: Подключение вычислителя в testbench.

Следует обратить внимание, что в testbench при подключении проверяемого модуля используются регистры для входных сигналов и провода для выходных. Таким образом можно генерировать входные воздействия.

Одной из самых простых конструкций testbench является *“initial”*-блок, который определяет, какие действия должны быть сделаны при старте программы. Этот блок не является синтезируемым. С помощью такого блока зададим начальные значения сигналов. Фрагмент кода приведен в Листинге 2.9.

```
initial
begin
    p_dat_i    = 'hz;
    p_enable_i = 0;
    p_sel_i    = 0;
    p_we_i     = 'hz;
    p_adr_i    = 'hz;
    p_rst_i    = 1;
    #200
    p_rst_i    = 0; // Запись #200 обозначает что смена значения сигнала
     ↪  сброса произойдет через 200нс.
end
```

Листинг кода 2.9: initial-блок генерации начальных значений сигналов.

Еще одна конструкция, которую часто используют при моделировании – *“forever”*-блок,

который обеспечивает циклическое исполнение куска кода. Такой блок очень полезен для генерации сигнала тактовой частоты. Фрагмент кода приведен в Листинге 2.10.

```
initial
begin
    p_clk_i=0;
    forever #50 p_clk_i = ~p_clk_i; // Сигнал инвертируется каждые 50нс
end
```

Листинг кода 2.10: Генерация сигнала тактовой частоты.

Следующей задачей при написании testbench контроллера является имитация обмена данными по системной шине – циклов чтения и записи. Такие конструкции могут быть использованы при тестировании очень много раз. Например, для нашего случая это может быть посылка 20 байт для вычисления контрольной суммы с последующим считыванием результата. Использовать *“initial”*-блок в таких случаях просто нерационально, так как он будет содержать в себе большое число однотипных действий. Удобно использовать *“task”*-блоки, которые содержат в себе эти последовательности действий, например последовательность действий при цикле записи по системной шине. А далее вызывать эти *“task”*-блоки в *“initial”*-блоке с необходимыми нам параметрами. Также удобно использовать вывод информации на консоль при помощи *display*. Пример *“task”*-блока для записи по APB приведен в Листинге 2.11.

```
task write_register; // Название task
    input [31:0] reg_addr; // Параметры передаваемые в task, в нашем случае
    ↪  адрес и данные
    input [31:0] reg_data;

    begin
        @ (posedge p_clk_i); // Ожидаем один такт

        // Формируем посылку согласно документации на APB
        p_adr_i    = reg_addr; // Выставляем значения на шины адреса и данных
        p_dat_i    = reg_data;
        p_enable_i = 0;
        p_sel_i    = 1;
        p_we_i     = 1;

        @ (posedge p_clk_i); // Ожидаем один такт

        p_enable_i = 1;

        wait (p_ready); // Ожидаем появление сигнала p_ready

        // Вывод информации о совершенной операции
        $display("(%0t) Writing register [%0d] = 0x%0x", $time, p_adr_i,
        ↪  reg_data);
        @ (posedge p_clk_i);

        // Возвращаем сигналы в исходное состояние
        p_adr_i    = 'hz;
        p_dat_i    = 'hz;
        p_enable_i = 0;
        p_sel_i    = 0;
        p_we_i     = 'hz;
    end
endtask
```

Листинг кода 2.11: task для записи по системной шине APB.

Аналогичным образом написан *task* для чтения по APB. Фрагмент кода приведен в Листинге 2.12.

```
task read_register;
    input [31:0] reg_addr;
    begin
        @ (posedge p_clk_i);

        p_adr_i    = reg_addr;
        p_enable_i = 0;
        p_sel_i    = 1;
        p_we_i     = 0;

        @ (posedge p_clk_i);

        p_enable_i = 1;

        wait (p_ready);

        $display("(%0t) Reading register [%0d] = 0x%0x", $time, p_adr_i,
        ↪  p_dat_o);

        @ (posedge p_clk_i);

        p_adr_i    = 'hz;
        p_enable_i = 0;
        p_sel_i    = 0;
        p_we_i     = 'hz;
    end
endtask
```

Листинг кода 2.12: task для чтения по системной шине APB.

В блоке *initial* вызван два раза *task* для записи по системной шине и один раз для чтения. Фрагмент кода приведен в Листинге 2.13.

```
initial
begin
    write_register(32'd0, 32'hAA);
    #1200 write_register(32'd0, 32'h33);
    #1200 read_register(32'd4);
end
```

Листинг кода 2.13: Вызов блоков task в блоке initial.

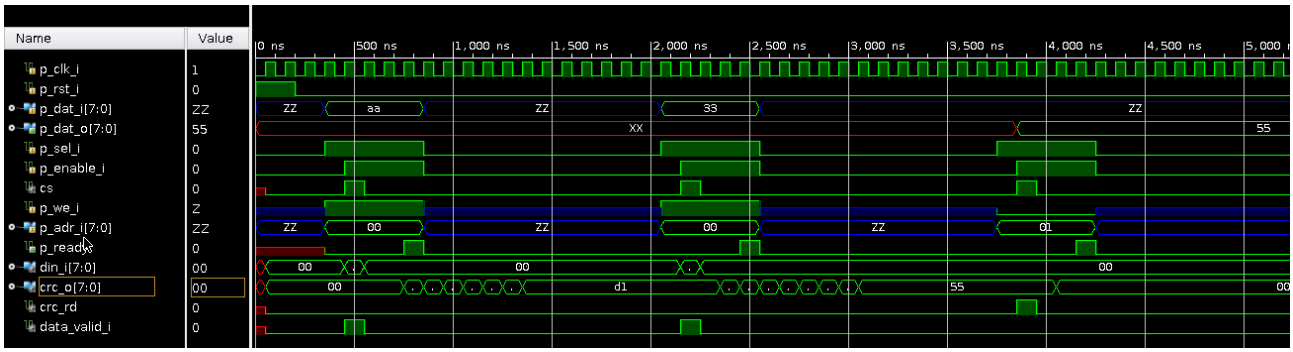

Рис. 2.18: Результат моделирования.

На временной диаграмме, изображенной на Рисунке 2.18, видно 2 цикла записи (данные 32'hAA и 32'h33), и один цикл чтения результата вычисления CRC.

2.7 Контрольные вопросы

1. Дайте определение системной шины. Назначение системной шины.
2. Классификация системных шин.
3. Каким образом различаются контроллеры в адресном пространстве? Что такое базовый адрес контроллера?
4. Системная шина APB. Каково назначение каждого из сигналов системной шины?
5. Системная шина APB. Изобразить цикл записи с задержкой.
6. Системная шина APB. Изобразить цикл чтения без задержки.

Лабораторная работа 3

Набор инструментов разработчика GNU Toolchain, Makefile, ELF

3.1 Введение

Часто, во время изучения языков программирования, студенту предлагается установить какую-либо программу: Code::Blocks или Microsoft Visual Studio для языков C/C++/C# или IntelliJ IDEA для java.

Данные программы называются IDE (*Integrated development environment* – интегрированная среда разработки) и являются инструментом, позволяющим разрабатывать огромные проекты наподобие многофункциональных графических редакторов или 3D-игр. Несмотря на то, что проекты, реализуемые при изучении языков программирования, весьма просты, и данные программы в общем-то не нужны, их предлагают установить, т.к. они позволяют с лёгкостью собирать и отлаживать проект, оставив на пользователя лишь заботу о написании кода. Это очень упрощает изучение языка на начальных этапах, однако оказывает медвежью услугу, поскольку будущий разработчик программ не понимает процессов, стоящих за нажатием кнопки “Run”.

Зачем вообще может понадобиться знание об этих процессах? Например, для кросс-компиляции. Разрабатывая программу для микроконтроллера или системы на кристалле, необходимо компилировать исходный код на рабочем компьютере, однако собранный код будет запускаться на устройстве с другой архитектурой. Код, скомпилированный под одну архитектуру не сможет работать на другой. Компиляция исходного кода на компьютере с одной архитектурой для запуска на компьютере с другой архитектурой и называется кросс-компиляцией. Распространённые IDE не используются в кросс-компиляции, необходимо использовать среды, написанные под какие-то конкретные семейства микроконтроллеров, в связи с чем, нет особого выбора. Для новичка, поиск материалов по настройке и реализации проекта не на микроконтроллерах Arduino и STM32 может вылиться в проблему, не говоря о том, что большинство материалов могут оказаться устаревшими или не совсем подходящими. В такие моменты можно прийти к выводу, что скомпилировать проект, используя один лишь кросс-компилятор будет проще. Кроме того, такой подход позволит работать на разных операционных системах и аппаратных платформах, используя инструменты сборки с открытыми исходными кодами (например, gcc).

Кроме того, в связи со своим мощным функционалом, популярные IDE, как правило, ресурсоёмки и не походят для запуска, компиляции и отладки на низкопроизводительных компьютерах или ноутбуках. Код можно редактировать и в легковесных редакторах, а сборку и отладку можно выполнять из терминала.

Таким образом, хорошему разработчику нужно уметь не только писать код, но и уметь его собирать.

3.2 Hello World

Традиционно, любой курс начинается с примера “Hello World”. В листинге 3.1 приведена простейшая программа, которую можно написать в любом текстовом редакторе, и собрать её из терминала.

```c
#include <stdio.h>

int main() {
    printf("Hello, world!\n");
    return 0;
}
```

Листинг кода 3.1: Файл hello.c.

Для её сборки, необходимо открыть терминал, указав рабочей директорией папку с данным файлом, после чего выполнить следующую команду:

```
gcc hello.c
```

Результатом её выполнения в операционной системе Windows станет исполняемый файл a.exe. При запуске файл выведет строку “Hello, world!” в окно текстового терминала. Процедура вышла весьма простой, однако лишь потому, что это упрощённый пример, и в реальных проектах процесс сборки сложнее.

Что же произошло во время выполнения этой команды?

В первую очередь, надо разобраться с тем, что такое gcc.

3.3 Что такое GCC

Изначально, GNU C Compiler, был разработан Ричардом Столлманом, основателем GNU Project. С течением времени, GCC развивался и стал поддерживать множество языков: C (gcc), C++ (g++), Objective-C, Objective-C++, Java (gcj), Fortran (gfortran), Ada (gnat), Go (gccgo), OpenMP, Cilk Plus, и OpenAcc, поэтому теперь GCC расшифровывается как GNU Compiler Collection.

GCC – ключевой компонент так называемого GNU Toolchain, предназначенного для разработки приложений и операционных систем. Он включает:

1. GNU Compiler Collection (GCC) – набор компиляторов, поддерживающий множество языков, например: C/C++ and Objective-C/C++;
2. GNU Make – средство автоматизации для компиляции и сборки приложений;
3. GNU Binutils – набор утилит, включающий компоновщик и транслятор;
4. GNU Debugger (GDB) – средство отладки программ;
5. GNU Autotools – система сборки, включающая: Autoconf, Autoheader, Automake и Libtool;
6. GNU Bison: генератор синтаксических анализаторов (похож на lex и yacc).

Прямо сейчас интерес представляет сам GCC, а также Binutils, в дальнейшем, будут разобраны Make и GDB.

3.4 Поэтапный разбор процесса компиляции

За употребляемым в среде программистов словом “компиляция” обычно стоят четыре последовательных этапа, показанные на рисунке 3.1:

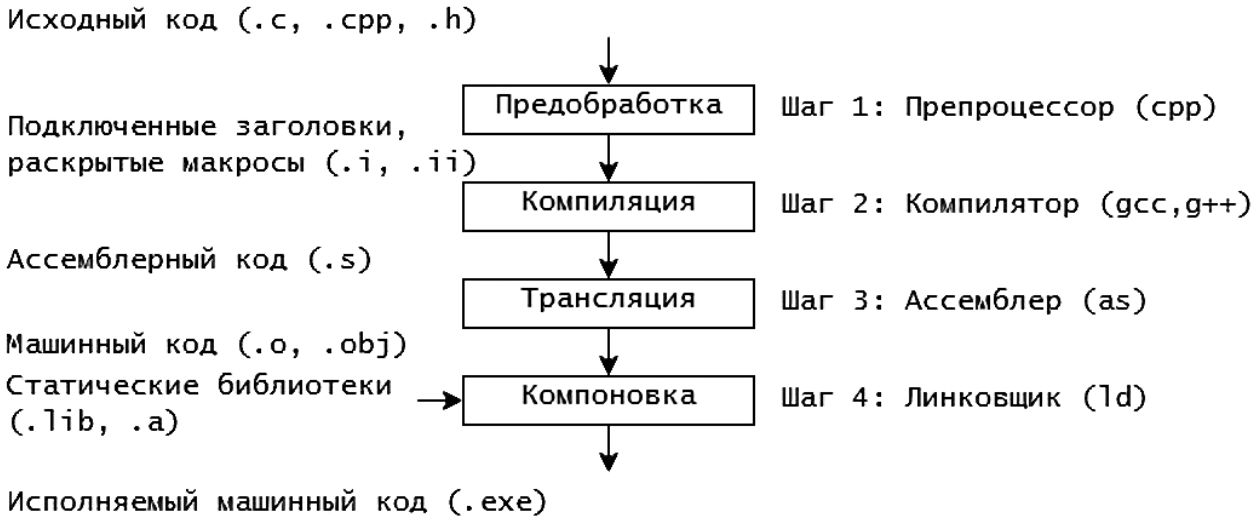

Рис. 3.1: Этапы компиляции программы.

3.4.1 Предобработка

Предобработка исходного кода выполняется препроцессором. Его работа заключается во вставке содержимого файлов, подключённых с помощью директивы #include, а также подстановке значений в макросах, определённых директивой #define. Результатом его работы является изначальный исходный код, но включающий все вставки и замены.

3.4.2 Компиляция

Компиляция переводит исходный код в набор машинных инструкций на языке ассемблера. Набор таких инструкций различается между архитектурами и именно это является причиной того, что код, скомпилированный под одну архитектуру, нельзя запустить на другой. Результатом работы компилятора является код на языке ассемблера.

3.4.3 Трансляция

Ассемблер (транслятор) переводит код на языке ассемблера в машинный – набор нулей и единиц, который, однако, ещё не получится запустить – для этого необходимо разрешить связи между функциями и переменными (или символами) программы.

3.4.4 Компоновка

Линковщик (компоновщик) устанавливает связи между вызовами функций в основном коде программы и реализацией этих функций в подключаемых библиотеках. Далее будет подробно рассмотрен процесс сборки примера “Hello world”. Сперва запускается препроцессор (исполняемый файл cpp.exe), который подключает заголовочные файлы и раскрывает макросы. Результат сохраняется в промежуточный файл .i в виде раскрытого исходного кода:

```
cpp hello.c > hello.i
```

В данной команде, > означает перенаправление потока вывода в файл hello.i. Оно необходимо, поскольку по умолчанию, результат работы препроцессора будет направлен в стандартный поток вывода (в большинстве случаев, терминал, из которого запущена программа). Т.е. будет лишь выведен результат работы, но он не будет сохранен в файл. Того же самого можно добиться, заменив '>' на '-o' (подробнее опция -o будет рассмотрена далее). Затем, происходит компиляция исходного кода в код на языке ассемблера, специфичный для конкретного процессора. За это отвечает программа gcc.exe:

```
gcc -S hello.i
```

опция -S говорит остановиться после выполнения этапа компиляции, поскольку этой же программой можно выполнить все четыре этапа сразу, как было показано в первом примере.

Далее, происходит трансляция этого кода в машинный код, который помещается в объектный файл hello.o:

```
as -o hello.o hello.s
```

Опция -o указывает, какое имя файла должно получиться на выходе. Если она не указана, именем исполняемого/объектного файла будет a.exe/a.out, а промежуточные файлы других этапов будут названы так же, как и поданный на вход файл, с изменённым суффиксом. Наконец, компоновщик (ld.exe) связывает объектный код с кодом библиотек, производя исполняемый файл hello.exe.

```
ld -o hello.exe hello.o ...libraries...
```

Полная версия команды не указана, т.к. вместо libraries необходимо указать весь список библиотек, который необходим для работы программы. Он различается между компиляторами и системами, к примеру на компьютере, где пишется этот документ, вручную скомпоновать пример можно командой:

```
ld -m i386pep -Bdynamic
 ↪  D:/Qt/Tools/mingw730_64/bin/../lib/gcc/x86_64-w64-mingw32/7.3.0/../../../../x86_64-w64-mingw
D:/Qt/Tools/mingw730_64/bin/../lib/gcc/x86_64-w64-mingw32/7.3.0/crtbegin.o
-LD:/Qt/Tools/mingw730_64/bin/../lib/gcc/x86_64-w64-mingw32/7.3.0
 ↪  -LD:/Qt/Tools/mingw730_64/bin/../lib/gcc
 ↪  -LD:/Qt/Tools/mingw730_64/bin/../lib/gcc/x86_64-w64-mingw32/7.3.0/../../../../x86_64-w64-min
 ↪  -LD:/Qt/Tools/mingw730_64/bin/../lib/gcc/x86_64-w64-mingw32/7.3.0/../../../../lib
 ↪  -LD:/Qt/Tools/mingw730_64/bin/../lib/gcc/x86_64-w64-mingw32/7.3.0/../../../../x86_64-w64-min
 ↪  -LD:/Qt/Tools/mingw730_64/bin/../lib/gcc/x86_64-w64-mingw32/7.3.0/../../.. main.o
 ↪  -lmingw32 -lgcc -lgcc_eh -lmoldname -lmingwex -lmsvcrt -lpthread -ladvapi32
 ↪  -lshell32 -luser32 -lkernel32 -liconv -lmingw32 -lgcc -lgcc_eh -lmoldname
 ↪  -lmingwex -lmsvcrt
 ↪  D:/Qt/Tools/mingw730_64/bin/../lib/gcc/x86_64-w64-mingw32/7.3.0/crtend.o
```

Чтобы все-таки собрать проект, проще воспользоваться самим gcc, который затем автоматически вызовет компоновщик со всеми необходимыми параметрами:

```
gcc -o hello.exe hello.o
```

Если всё же возникла необходимость произвести компоновку самостоятельно, можно воспользоваться следующим способом для получения полного списка библиотек: К команде вызова gcc необходимо добавить:

```
2> log.txt
```

Данная конструкция направит поток вывода в файл log.txt, внутри которого можно будет увидеть команды, которые выполнял компилятор. Надо найти строку:

```
COLLECT_GCC_OPTIONS
```

Под этой строкой необходимо найти место, где закончатся перечисления плагинов и пойдут перечисления объектных файлов и ключи -L – это и есть необходимые для компоновщика опции.

Если передать компилятору опцию -save-temps, тот сохранит промежуточные файлы каждого из выполненных этапов. Для интереса, можете посмотреть на результат работы препроцессора и компилятора (файлы .i/.ii, .s соответственно).

3.5 Компиляция кода из нескольких файлов

В реальных проектах, код не пишется в одном единственном файле, классы и функции выносятся в отдельные файлы, которые потом могут собираться в библиотеки.

Ниже приведён пример проекта, использующего собственную функцию. Обычно, прототип функции описывается в файле .h, а её реализация – в файле с расширением .c. Файлы с расширением .h (заголовочные файлы) , как правило, поставляются с прекомпилированными библиотеками и как бы представляют её функционал и интерфейс, но не реализацию.

Часто заголовочные файлы и файлы исходного кода разделяются по папкам inc и src соответственно. Таким образом, создадим проект со следующей структурой:

```
Hello_project
  inc
    hello.h
  src
    hello.c
    main.c
```

В папке с проектом находятся папки inc и src, в которых находятся hello.h и hello.c, main.c соответственно. Содержимое этих файлов показано в листингах 3.2-3.4.

```
#include <stdio.h>
void hello_world (void);
```

Листинг кода 3.2: Файл hello.h.

```
#include "hello.h"
void hello_world(void)
{
    printf("Hello World!");
}
```

Листинг кода 3.3: Файл hello.c.

```
#include "hello.h"

int main() {
    hello_world();
    return 0;
}
```

Листинг кода 3.4: Файл main.c.

Из этих трёх файлов скомпилировать надо два (заголовочный файл будет вставлен в остальные препроцессором). Сделать это можно из папки проекта командой:

```
gcc -c src/hello.c src/main.c -Iinc
```

Важно, что при вызове компилятора необходимо указывать полный путь до компилируемых файлов.

Ранее уже было сказано, что gcc последовательно вызывает препроцессор, компилятор, транслятор и компоновщик. Можно сообщить ему, на каком из этапов нужно остановиться с помощью специальных опций(ключей), переданных в команде:

- -E – остановиться после этапа предобработки;
- -S – остановиться после этапа компиляции;
- -c – остановиться после этапа трансляции.

Ключ -I (сокращение от Include) указывает компилятору, где искать подключаемые файлы. В переменных среды есть стандартные пути, поиска, поэтому нам не нужно искать, где находится заголовочный файл stdio.h. Кроме того, стоит заметить, что директории, указанные через данный ключ, обладают более высоким приоритетом, т.е. компилятор будет искать подключаемые файлы сначала в них, и если там будет заголовочный файл, одноименный с stdio.h, подключится

именно он. Если необходимо указать несколько папок, каждую из них нужно передавать со своим ключом. Поиск не производится рекурсивно, если нужно добавить вложенные папки, каждую из них так же надо передать через ключ -I.

После получения объектных файлов, их необходимо скомпоновать в исполняемый файл. Для этого, опять же, необходимо воспользоваться gcc (который автоматически вызовет компоновщик):

```
gcc -o hello.exe hello.o main.o
```

Может возникнуть вопрос: зачем два раза вызывать gcc, если можно было просто убрать ключ -c при первом вызове. Дело в том, что при изменении любого файла, необходимо пересобрать весь проект целиком. Пересборка больших проектов (скажем, ядра Linux) – это очень долгий процесс. Разделив сборку на две части, появляется возможность скомпилировать только изменённый файл, а затем, скомпоновать заново весь проект целиком.

В данный момент вручную указывается полный путь до компилируемых файлов и все папки с заголовочными файлами. Это делается для лучшего понимания процесса. Способы упрощения этой процедуры будут описаны позднее.

3.5.1 Сборка библиотеки

Пусть существует код, который будет использоваться во многих проектах. Или объём кода довольно большой, дальнейшие изменения в нём не планируются, а компиляция уже занимает весьма много времени. Или необходимо получить из вашего кода бинарный файл, чтобы, передав его человеку, не раскрывать ему реализацию (сохранить свою интеллектуальную собственность). Для всего этого можно скомпилировать код в библиотеку.

Библиотеки делятся на два типа: статические и динамические.

3.5.2 Статическая библиотека

Статическая библиотека (или “архив”) состоит из подпрограмм, которые непосредственно компилируются и линкуются с программой в исполняемый файл. При компиляции программы, которая использует статическую библиотеку, весь функционал статической библиотеки (тот, что использует программа) становится частью вашего исполняемого файла. В Windows статические библиотеки имеют расширение .lib (от “library”), тогда как в ОС семейства GNU \Linux статические библиотеки имеют расширение .a (от “archive”). Примечание автора: в дальнейшем, дабы уйти от громоздкой записи “ОС семейства GNU \Linux”, будет использоваться слово “Linux” в качестве её замены. В случаях, когда в тексте будет идти речь о ядре Linux, об этом будет сказано явно.

Одним из преимуществ статических библиотек является то, что нужно распространять всего лишь один файл (исполняемый файл), дабы пользователи могли запустить и использовать программу. Поскольку статические библиотеки становятся частью программы, то можно использовать их подобно функционалу этой же самой программы. С другой стороны, поскольку копия библиотеки становится частью каждого исполняемого файла, то это может привести к увеличению его размера. Также, для того чтобы обновить статическую библиотеку, необходимо перекомпилировать каждый исполняемый файл, который её использует.

3.5.3 Динамическая библиотека

Динамическая библиотека (или “общая (разделяемая) библиотека”) состоит из подпрограмм, которые подгружаются в программу во время её выполнения. При компиляции про-

граммы, которая использует динамическую библиотеку, эта библиотека не становится частью исполняемого файла — она так и остаётся отдельным модулем. В Windows динамические библиотеки имеют расширение .dll (от “dynamic link library” = “библиотека динамической компоновки”), тогда как в Linux динамические библиотеки имеют расширение .so (от “shared object” = “общий объект”). Одним из преимуществ динамических библиотек является то, что разные программы могут совместно использовать одну копию динамической библиотеки, что значительно экономит используемое пространство. Ещё одним преимуществом динамической библиотеки является то, что её можно обновлять до более новой версии без перекомпиляции всех исполняемых файлов, которые её используют.

Поскольку динамические библиотеки не линкуются непосредственно с программой, то программы, использующие динамические библиотеки, должны явно подключать и взаимодействовать с динамической библиотекой. Этот механизм не всегда может быть понятен для начинающих программистов, что может затруднить взаимодействие с динамической библиотекой. Для упрощения этого процесса используют библиотеки импорта.

Библиотека импорта (англ. “import library”) — это библиотека, которая автоматизирует процесс подключения и использования динамической библиотеки. В Windows это обычно делается через небольшую статическую библиотеку (.lib) с тем же именем, что и динамическая библиотека (.dll). Статическая библиотека линкуется с программой во время компиляции, после чего функционал динамической библиотеки может эффективно использоваться в программе, как если бы это была обычная статическая библиотека. В Linux общий объектный файл (с расширением .so) дублируется сразу как динамическая библиотека и библиотека импорта. Большинство компоновщиков при создании динамической библиотеки автоматически создают к ней библиотеку импорта.

3.5.4 Сборка и использование статических библиотек

Ниже будет рассмотрен использовавшийся в прошлом примере проект и описан процесс компилирования файлов hello.c и hello.h статическую библиотеку:

В первую очередь, необходимо получить объектный файл файла hello.c:

```
gcc -c src/hello.c -Iinc -o hello.o
```

После этого, необходимо создать статическую библиотеку, при помощи утилиты ar (ранее уже говорилось, что в Linux статические библиотеки называются архивами, а название утилиты получилось от сокращения “archiver” – архиватор):

```
ar rcs hello.lib hello.o
```

rcs – это опции команды, r – означает вставку с заменой, c – создать новый архив, s – записать index

Данная команда соберёт статическую библиотеку hello.lib из объектного файла hello.o.

Для того, чтобы использовать эту библиотеку при компиляции проекта, сначала необходимо скомпилировать файл main.c:

```
gcc -c src/main.c -Iinc -o main.o
```

После чего, произвести компоновку с данной библиотекой:

```
gcc main.o -L./ -lhello -o hello.exe
```

Ключ -L указывает папку, в которой необходимо искать библиотеки (“./” является синонимом для “текущая папка”), через ключ -l указывается имя подключаемой библиотеки (стоит заметить, что все библиотеки в Linux собираются с префиксом lib: libhello, libworld и т.п., поэтому в саму опцию нужно передавать имя без этого префикса).

3.5.5 Сборка и использование динамических библиотек

Объектные файлы, скомпилированные для компоновки в динамическую библиотеку, должны представлять собой “позиционно-независимый код”.

Позиционно-независимый код (англ. position-independent code) – программа, которая может быть размещена в любой области памяти, так как все ссылки на ячейки памяти в ней относительные (например, относительно счётчика команд). Такую программу можно переместить в другую область памяти в любой момент, в отличие от перемещаемой программы, которая хотя и может быть загружена в любую область памяти, но после загрузки должна оставаться на том же месте.

Вообще, тема динамических библиотек поистине обширна и заслуживает отдельной серии занятий. Здесь будут приводиться лишь некоторые крупицы информации со ссылками на более углублённые статьи.

Для того, чтобы скомпилировать объектный файл в виде позиционно-независимого кода, необходимо добавить компилятору одну из опций: -fPIC или -fpic. PIC – это, как нетрудно догадаться и запомнить, аббревиатура от position-independent code. -fPIC работает всегда, но даёт больший объём кода, чем -fpic (Запомнить это свойство можно с помощью простого мненонического правила: PIC “крупнее” pic). Применение -fpic обычно приводит к генерации меньшего и более быстрого кода, но он будет иметь некоторые платформенно-зависимые ограничения, например, количество глобально видимых символов или размер всего кода.

```
gcc -c -fpic src/hello.c -Iinc -o hello.o
```

Команда выше уже должна стать привычной, за исключением описанного ранее ключа тут не появилось ничего нового.

Далее, необходимо скомпоновать позиционно-независимый код в динамическую библиотеку:

```
gcc -shared hello.o -o hello.dll
```

Собрать итоговый проект можно так же, как и при использовании статической библиотеки:

```
gcc main.o -L./ -lhello -o hello.exe
```

3.6 Отладка программы

Помимо препроцессора, компилятора, транслятора и компоновщика, GNU Toolchain предоставляет средства для отладки, в частности, утилиту gdb (GNU Debugger).

С помощью GDB можно:

- запустить программу с определёнными аргументами;
- запустить программу в пошаговом режиме;
- установить точки останова (breakpoint);
- установить условие останова программы;
- получить информацию о состоянии программы перед тем, как она прекратила работу из-за ошибки;
- получить информацию о текущем состоянии стека;
- узнать значение переменной;
- изменить значение переменной.

Это далеко не весь список, но даже этот функционал открывает широкий спектр возможностей.

Для отладки программы необходимо, чтобы она была собрана с отладочной информацией. Для этого, при компиляции исходных кодов программы, необходимо добавить ключ -g. Необходимо иметь в виду, что при компиляции с этим ключом, размер программы значительно возрастает.

Рассмотрим программу, исходный код которой приведён в листинге 3.5. Программа должна вывести первые n символов латинского алфавита.

```c
#include <stdio.h>
#include <stdlib.h>
void print_chars(int);
int main(int argc, char* argv[]){
    int a;
    if(argc == 1)
    {
      printf("Enter the number of letter that you want to see:");
      scanf("%d",&a);
    }
            else
            {
              a = *(argv[1]) - 58; // мы считаем, что ASCII-код символа ‘0’
                                   // равен 58, а не 48, и не знаем,
                                   // что этого можно было избежать, написав ‘0’.
            }

            print_chars(a);
            return 0;
        }

void print_chars(int num_of_chars)
{
    char *str = NULL;
    str = (char *)malloc((num_of_chars + 1) * sizeof(char));
    for (int i = 0; i < num_of_chars; ++i)
    {
      str[i] = 'a' + i;
    }
    str[num_of_chars] = '\0';
    printf("%s", str);
    free(str);
}
```

Листинг кода 3.5: Пример программы, содержащий ошибки.

Произведён запуск программы, чтобы она вывела 5 латинских символов:

```
./main 5
```

Результатом выполнения будет ошибка сегментации:

```
Segmentation fault
```

Ошибка сегментации явно указывает на попытку обращения к недоступным для записи или чтения участкам памяти. Для обнаружения и исправления ошибки необходимо выполнить отладку программы.

Для начала, необходимо собрать данный файл с отладочной информацией:

```
gcc main.c -output main -g
```

После, запустить его в отладчике

```
gdb main
```

В терминале выведется длинное сообщение, сообщающее о лицензии использования gdb, а так же об отказе от ответственности (поскольку пользователь имеет доступ к редактированию переменных прямо во время работы программы, авторы gdb не могут гарантировать, что это не приведёт к негативным последствиям).

Для запуска отладчика с пропуском стартового сообщения используйте ключ -q (от англ.: quiet – тихий).

Поскольку нет никаких предположений о том, где могла возникнуть ошибка сегментации, стоит произвести тестовый запуск программы в отладчике идентичный сделанному до этого в терминале. Для запуска программы используется команда run с указанием опций, передаваемых программе:

```
run 5
```

на что gdb выведет содержимое, приблизительно соответствующее тексту ниже:

```
Starting program: main.exe 5
[New Thread 20488.0x51ac]
[New Thread 20488.0x4e74]

Program received signal SIGSEGV, Segmentation fault.
0x00000000004015b5 in print_chars (num_of_chars=-5) at src/main.c:29
29            str[num_of_chars] = '\0';
```

В результате анализа сообщений отладчика, можно сделать следующие выводы:

- ошибка произошла в функции print_chars, в частности, в строке 29;
- выведенный стек вызовов говорит, что переданное в эту функцию значение для переменной num_of_chars равно -5, что отлично от ожидаемого поведения.

Прежде чем разбираться с тем, почему в функцию передаётся неверное значение, стоит изучить функцию print_chars. Для этого, необходимо вывести исходный код данной функции:

```
list print_chars
```

Данная команда выводит исходный код, отцентрованный относительно точки входа в функцию. Функция не уместилась за один вывод, поэтому необходимо вызвать list ещё раз. Будучи переданным без аргументов, он продолжит вывод исходного кода функции из предыдущего вызова list.

```
(gdb) list print_chars
17              print_chars(a);
18              return 0;
19          }
20
21          void print_chars(int num_of_chars)
22          {
23              char *str = NULL;
24              str = (char *)malloc((num_of_chars + 1) * sizeof(char));
25              for (int i = 0; i < num_of_chars; ++i)
26              {
(gdb) list
27                  str[i] = 'a' + i;
28              }
29              str[num_of_chars] = '\0';
30              printf("%s", str);
31              free(str);
32          }
33
```

Предположим, что надо установить три точки останова:

- на входе в функцию;
- внутри цикла for, когда итератор i больше двух;
- на строке, вызвавшей падение программы.

Для создания точки останова, используется команда break (или сокращение b):

```
break print_chars
b 27 if i>2
b 29
```

Перезапуск программы с тем же аргументом

```
run 5
```

Будет сказано, что программа уже выполняется и нужно ли запустить её с начала. Вам нужно ответить вводом символа y(yes/да) или n(no/нет).

Будет произведён следующий вывод:

```
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: main.exe 5
[New Thread 12036.0x44c0]
[New Thread 12036.0x57ac]
```

```
Breakpoint 1, print_chars (num_of_chars=-5) at src/main.c:23
23              char *str = NULL;
```

Для продолжения выполнения программы после точки останова используется три команды:

- next (или сокращение n) – выполнение кода программы до следующей строчки исходного кода;
- step (или сокращение s) – выполняет вход внутрь функции, расположенной на текущей строке. Выполнив step на строке 17, вы попадёте внутрь функции print_chars. Выполнив step на строке 24, вы попадёте внутрь функции malloc.
- continue (или сокращение c) – выполнение кода программы до следующей точки останова, или, если таковые не встретятся, до её завершения.

Каждая из команд выше может иметь аргументы, расширяющие её функционал. Все опущенные здесь подробности можно найти в руководстве по gdb.

Выполнив две строчки нашей программы, можно убедиться, что вызов функции malloc с отрицательным размером памяти не привёл к должной инициализации строки str:

n 2 – выполнить не одну, а две строчки программы

print str – вывести значение переменной str (вместо print можно использовать сокращение p)

Результатом этих команд будет:

```
23              char *str = NULL;
(gdb) n 2
25              for (int i = 0; i < num_of_chars; ++i)
(gdb) p str
$1 = 0x0
```

Продолжить выполнение программы до следующей точки останова:

```
c
```

Как можно увидеть, программа не остановилась внутри цикла for, вместо этого, остановившись на 29-ой строке. Это произошло, поскольку итератор i, инициализированный нулём сразу оказался больше отрицательного аргумента num_of_char, в связи с чем, исполнение программы не было передано внутрь цикла.

```
(gdb) c
Continuing.

Breakpoint 2, print_chars (num_of_chars=-5) at src/main.c:29
29              str[num_of_chars] = '\0';
```

Теперь необходимо поставить точку останова с вызовом функции print_char и перезапустить программу:

```
(gdb) b 17
Breakpoint 3 at 0x401542: file src/main.c, line 17.
(gdb) r 5
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: C:\Users\voult\Desktop\1\hello.exe 5
[New Thread 21776.0xf54]
[New Thread 21776.0x36f4]

Breakpoint 3, main (argc=2, argv=0x7214b0) at src/main.c:17
17            print_chars(a);
```

После, нужно изменить значение переменной a на ожидаемое и вновь зайти внутрь функции print_chars

```
(gdb) p a=5
$3 = 5
(gdb) step

Breakpoint 1, print_chars (num_of_chars=5) at src/main.c:23
23            char *str = NULL;
```

Если остановиться на входе в функцию, можно увидеть, что внутрь попало изменённое значение. Считается, что теперь все пойдёт по плану, поэтому можно удалить все точки останова внутри функции print_chars. Удаление конкретной точки останова аналогично её установке, только вместо команды break используется команда clear. Для удаления всех точек останова, используется команда delete, без передачи в неё параметров. Если продолжить исполнение кода программы можно убедиться, что он выполнится без новых ошибок:

```
(gdb) delete
Delete all breakpoints? (y or n) y
(gdb) c
Continuing.
abcde[Thread 21776.0x36f4 exited with code 0]
[Inferior 1 (process 21776) exited normally]
```

Функция print_chars работает корректно. Проблема заключается в неверном присвоении значения переменной a.

Значение переменной присваивается в строке 9, если в программу не были переданы аргументы и в строке 13, если программа была запущенна с аргументами. Необходимо исправить строчку 13 на верную:

```
13:         a = *(argv[1]) - '0';
```

Далее, необходимо сохранить исходный код, скомпилировать программу без отладочных символов.

3.7 Автоматизация сборки проекта

3.7.1 Краткое введение в make

Ранее обращалось внимание на то, что указание полного пути до каждого компилируемого файла – утомительное занятие. Проект может содержать сотни файлов, ручное указание всех этих файлов становится уже нереалистичным. Для автоматизации этого процесса в GNU Toolchain есть программа make.

Директивы утилиты make служат для определения зависимостей между файлами проекта и находятся в файле по имени Makefile, расположенном в каталоге сборки.

Общий формат make-файла выглядит так:

```
цель1: зависимости
    правила
#...
цельN: зависимости
    правила
```

Цель — это метка для некоторой последовательности команд, или результирующий файл, который нужно “построить”, например, скомпилировать или скомпоновать.

Цели должны отделяться друг от друга хотя бы одной пустой строкой. Зависимости – это перечень файлов или других целей, которые нужны для достижения данной цели; он может быть и пустым.

Правила — это последовательность команд, которую нужно выполнить для достижения цели. Правила должны отделяться от начала строки символом табуляции, иначе make завершится ошибкой “missing separator” (нет разделителя).

В make могут использоваться так называемые фиктивные цели – это цели, результатом которых не будет одноименный файл. Часто используемые фиктивные цели в make, это:

- all – цель по умолчанию. Её зависимостями ставятся те цели, которые вы хотите обработать, при регулярном запуске make.
- clean – очистка промежуточных файлов. В процессе сборки исполняемого файла, в директории остаются объектные файлы. Цель clean может использоваться для их очистки.
- clean_all: полная очистка директории от продуктов работы других целей. Если цель clean удалит только промежуточные файлы, clean_all удалит всё, что было могло быть создано в процессе исполнения остальных целей из этого файла, включая исполняемые.

Для того, чтобы указать make, что подобная цель – фиктивная, необходимо указать эту цель в качестве зависимости для специальной предопределённой цели .PHONY: (от англ. phony – фальшивый):

```
.PHONY: all clean
```

Зачем необходимы фиктивные цели? Предположим, в каталоге есть файл с именем clean. В таком случае, make будет обрабатывать его так же, как и остальные файлы-цели. Если дата последнего изменения файла clean указана более поздней, чем дата изменения его зависимостей, обработка прекращается. Если цель clean не зависит при этом ни от чего, или зависит лишь от

других фиктивных целей, make будет считать, что цель не нуждается в обработке и прекратит работу. Цель .PHONY позволяет избежать подобного поведения программы.

Make-файл может содержать однострочные комментарии — они начинаются символом #.

make ищет файлы в следующем порядке: GNUmakefile, makefile и только потом Makefile. Однако, общей практикой принято использовать именно Makefile.

Запуск make происходит следующим образом:

```
make имя_цели
```

Если цель не указана, make произведет попытку сборки первой встреченной цели. По этой причине, первой целью часто ставят цель all.

Если файл назван произвольным именем, для его обработки make необходимо запускать в виде:

```
make -f имя_файла имя_цели
```

3.7.2 Пример простого Makefile

Ниже приведён пример файла Makefile для сборки исходного кода главы “Компиляция кода из нескольких файлов”. Для того, чтобы собрать программу, как уже было описано ранее, необходимо два этапа: сборка бинарного файла из объектных, сборка объектных файлов из исходных.

Необходимо в корне проекта создать файл с именем “Makefile” и содержимым, приведённым в листинге 3.6.

```
1 hello.o: src/hello.c
2     gcc -c -Iinc ./src/hello.c
3
4 main.o: src/main.c
5     gcc -c -Iinc ./src/main.c
6
7 hello: main.o hello.o
8     gcc -o hello main.o hello.o
```

Листинг кода 3.6: Пример файла Makefile.

"hello.o”, “main.o” и “hello”, указанные в началах строк – это цели.

Итоговая цель – сборка исполняемого файла, это цель “hello”. Данная цель зависит от целей “main.o”, “hello.o” – нельзя собрать исполняемый файл, пока не были собраны объектные. Сборка выполняется рекурсивно: make сначала выполняет все цели, от которых зависит текущая цель. Если зависимость представляет собой файл, то make сравнивает время его последней модификации со временем целевого файла: если целевой файл был изменён раньше или отсутствует, то будет выполнена указанная последовательность команд. Если целевой файл был изменён позже, то текущая цель считается достигнутой.

Далее, нужно запустить автоматическую сборку командой:

```
make hello
```

В результате, должны собраться объектные файлы hello.o, main.o, а так же исполняемый файл hello.exe.

Make может использоваться и для более сложных вещей, в нём можно вызывать функции и команды терминала, есть несколько видов переменных, шаблонные символы и много чего другого. В следующих параграфах будет рассмотрена часть из этого функционала.

3.7.3 Переменные в make

Переменные в make могут использоваться для многих целей: они могут определять путь исполнения самого make-файла, могут использоваться для хранения промежуточных вычислений, но часто они будут использоваться для хранения повторно используемой информации.

Переменная может представлять собой что угодно, например, список файлов, набор передаваемых компилятору опций, имя запускаемой программы, список каталогов с исходными файлами, директорию для выходных файлов и так далее.

Именем переменной может быть любая последовательность символов, не содержащая символы ':', '#', '=' и начальных или конечных пробелов. Однако, рекомендуется избегать использования имён переменных, содержащих символы, отличные от букв, цифр и символа подчёркивания.

Имена переменных чувствительны к регистру. Таким образом, имена foo, FOO, и Foo будут ссылаться на разные переменные.

Исторически, имена переменных записывались с использованием букв верхнего регистра. Однако, рекомендуется пользоваться нижним регистром для всех переменных, используемых внутри make-файла. Верхний же регистр рекомендуется оставить для переменных, влияющих на работу неявных правил или содержащих параметры, которые пользователь может переопределять.

Для подстановки значения переменной, необходимо использовать знак доллара, за которым следует имя переменной, заключённое в круглые или фигурные скобки: обе записи $(foo) и $foo представляют собой ссылку на переменную foo. Из-за подобного специального значения символа $, для получения знака доллара в имени файла или команде нужно использовать запись $$.

Ссылка на переменную может быть использована практически в любом контексте: в качестве цели, зависимости, команды. Она может использоваться в большинстве директив, а также выступать в качестве нового значения другой переменной.

Если за знаком доллара следует символ, отличный от доллара, открывающейся круглой скобки или открывающейся фигурной скобки, то этот одиночный символ рассматривается как имя переменной. Таким образом, вы можете обратиться к переменной x, используя запись $x. Однако, такая практика крайне нежелательна, за исключением случаев обращения к автоматическим переменным, о которых будет чуть позднее.

3.7.4 Установка значения переменной

В GNU make есть два способа, с помощью которых переменные могут получить своё значение. Две разновидности отличаются тем, каким образом переменная была определена и что происходит при вычислении её значения.

Первая разновидность – это рекурсивно вычисляемые переменные. Такие переменные определяются с помощью символа '='. Значение этой переменной запоминается точно в том виде, как вы его указали; если оно содержит ссылки на другие переменные, то эти ссылки будут вычислены (заменены своими текстовыми значениями) только в момент вычисления значения

самой переменной (когда будет вычисляться какая-то другая строка, где использована эта переменная). Этот процесс называется рекурсивным вычислением.

Например, make-файл, описанный в листинге 3.7:

```
foo = $(bar)
bar = $(ugh)
ugh = Huh?

all:;echo $(foo)
```

Листинг кода 3.7: Пример использования переменных в Makefile.

выведет на экран

```
Huh?
```

при вычислении ссылки $(foo), она будет заменена на $(bar), которая, свою очередь, заменена на $(ugh), которая, наконец, будет расширена в Huh?.

Эта разновидность переменных - единственная, поддерживаемая всеми версиями make. Она имеется свои достоинства и недостатки. Её преимущество заключается в том, что, например, следующий фрагмент:

```
CFLAGS = $(include_dirs) -O
include_dirs = -Ifoo -Ibar
```

будет работать так, как и ожидалось: ссылки на переменную ‘CFLAGS’ будут “раскрываться” в текст

```
-Ifoo -Ibar -O
```

С другой стороны, серьёзный недостаток заключается в том, что вы не можете ничего “добавить” к переменной, наподобие:

```
CFLAGS = $(CFLAGS) -O
```

поскольку это вызовет бесконечный цикл при попытке вычислить её значение. (На самом деле, make распознаёт ситуацию зацикливания и сообщает об ошибке. Кроме того, для подобной ситуации можно использовать оператор +=.)

Другой недостаток рекурсивно вычисляемых переменных состоит в том, что все функции, на которые они ссылаются будут вычисляться заново при каждой “подстановке” этой переменной. Работа make при этом, замедляется.

Подобных недостатков лишена другая разновидность переменных - упрощённо вычисляемые переменные.

Упрощённо вычисляемые переменные определяются с помощью ':='. Значение такой переменной вычисляется (с расширением всех ссылок на другие переменные и вычислением функций) только в момент присваивания ей нового значения. После определения переменной, её значение представляет собой обычный текст, уже не содержащий ссылок на другие переменные. Таким образом,

```
x := foo
y := $(x) bar
x := later
```

эквивалентно

```
y := foo bar
x := later
```

При ссылке на упрощено вычисляемую переменную делается простая подстановка её значения (без каких-либо дополнительных вычислений).

3.7.5 Шаблонные символы

При использовании шаблонных символов, с помощью одного имени можно задать целую группу файлов. В make шаблонными символами являются '*', '?' и '[...]' (как в оболочке Bourne Shell). Например, шаблон '*.c' будет соответствовать всем файлам, оканчивающихся на '.c', находящимся в текущей директории. '?' обозначает совпадение с любым одиночным символом, а '[. . .]' – совпадение с любым символом, указанным в квадратных скобках. В квадратных скобках можно указывать диапазон символов (например, [a-zA-Z0-9]).

Раскрытие шаблонных имён (замена их конкретным списком файлов, удовлетворяющих шаблону) автоматически производится в именах целей, именах зависимостей и командах (в командах этим занимается интерпретатор командной строки). В других случаях, раскрытие шаблона производится только при явном запросе с помощью функции wildcard.

Специальное значение шаблонных символов может быть "отключено" с помощью предшествующего им символа '\'. Таким образом, строка 'foo*bar' будет ссылаться на довольно странное имя, состоящее из семи символов: начального 'foo', звёздочки и 'bar'.

3.7.6 Примеры шаблонных имён

Шаблонные имена могут быть использованы в командах, которые содержатся в правилах. Такие имена будут "раскрыты" интерпретатором командной строки. Пример правила для удаления всех объектных файлов из текущей директории:

```
clean:
  rm -f *.o
```

3.7.7 Проблемы при использовании шаблонных имён

Вот простой пример некорректного использования шаблонного имени, результат которого совершенно отличен от ожидаемого. Предположим, составляя make-файл, разработчик хотел сказать сказать, что исполняемый файл ‘foo’ собирается из всех объектных файлов, находящихся в текущем каталоге, и записали это следующим образом:

```
objects = *.o
foo : $(objects)
  cc -o foo $(CFLAGS) $(objects)
```

При такой записи, переменная objects получит значение ’*.o’. Расширение шаблона ’*.o’ будет произведено только при обработке правила с ’foo’, и зависимостями этой цели станут все существующие в данный момент файлы ’.o’. При необходимости, эти объектные файлы будут перекомпилированы.

Но что будет, если разработчик все файлы с расширением ’.o’? Когда шаблону не соответствует ни один файл, этот шаблон остаётся в “первоначальном” виде. И, таким образом, получится что цель ’foo’ будет зависеть от файла с именем ’*.o’. Поскольку, такого файла на самом деле не существует, make аварийно завершит работу, выдав сообщение, что она не знает как построить файл ’*.o’.

3.8 Кроссплатформенная компиляция

Для кроссплатформенной компиляции используется кроссплатформенный компилятор – программа, собранная для запуска на одной архитектуре, но производящая сборку программ для другой. В этом и заключается отличие от обычной компиляции – сборка программы производится под другую архитектуру.

Собранную таким образом программу не получится запустить на вычислительной машине, с которой была произведена сборка (во всяком случае, без использования эмуляторов).

Во всём же остальном процессы совпадают, у кроссплатформенной сборки будут происходить такие же процессы: предобработка, компиляция, трансляция и компоновка. Разве что, кросскомпилятору необходимо будет передать дополнительные ключи, связанные с архитектурой.

Пример простого make-файла для кросскомпиляции в листинге 3.8

```
CC        := g++
TOOLCHAIN := arm-linux-gnueabihf-
PT        :=
CFL       := -Wextra -std=c++11
TPATH     :=
 ↪  ~/toolchain/gcc-linaro-5.3.1-2016.05-x86_64_arm-linux-gnueabihf/bin/
LPATH     :=
 ↪  ~/toolchain/sysroot-glibc-linaro-2.21-2016.05-arm-linux-gnueabihf/
ARCH      := -march=armv7-a -mcpu=cortex-a5 --sysroot=$(LPATH)

native: slc.cpp
        $(CC) $(CFL) -o eval slc.cpp

cross: slc.cpp
        $(TPATH)$(TOOLCHAIN)$(CC) $(CFL) $(ARCH) slc.cpp -o acalc -static

clear:
        rm -f *.o
        rm -f eval
```

Листинг кода 3.8: Пример простого make-файла для кросскомпиляции.

Цель “native” используется для простой компиляции, “cross” – для кросскомпиляции. Обратите внимание на их различие:

- происходит вызов arm-linux-gnueabihf-gcc вместо gcc
- используются дополнительные ключи, лежащие в переменной ARCH

Стоит, однако, заметить, что собранная таким образом программа сможет запуститься только внутри операционной системы, которая загрузит содержимое файла в необходимые участки памяти, как и в какие участки памяти необходимо загрузить данный файл. Чтобы программу можно было использовать встраиваемого ПО для системы на кристалле, необходимо скомпоновать эту программу со специальным кодом, который произведёт инициализацию устройства, векторов прерываний, сброс регистров, загрузку остальной части программы и передачу управления на точку входа в main. Подобный начальный код называется startup-кодом и обычно предоставляется вместе с вычислительным устройством. Полученный бинарный файл сможет загрузиться и запуститься на устройстве, он будет соответствовать формату исполняемых и компонуемых модулей.

Живой пример: вместе с микроконтроллером STM32 поставляется библиотека CMSIS, а также готовый startup-файл. CMSIS предоставляет последовательные и простые интерфейсы для ядра, его периферии.

3.9 ELF – Executable and Linkable Format

ELF (англ. Executable and Linkable Format — формат исполнимых и компонуемых файлов) — формат двоичных файлов, используемый во многих современных UNIX-подобных операционных системах, таких как FreeBSD, Linux, Solaris и др. ELF-файлы состоят из трёх основных частей:

- Заголовок файла;
- Секции
- Сегменты

Каждая из этих частей играет свою роль в этапах компоновки/загрузки ELF-файлов.

3.9.1 Заголовок файла

В начале файла (со смещения 0 от начала) расположен заголовок ELF-файла, описываемый структурой, приведенной в листинге 3.9.

```
typedef struct
{
    unsigned char e_ident[16];
    uint16_t      e_type;
    uint16_t      e_machine;
    uint32_t      e_version;
    uint32_t      e_entry;
    uint32_t      e_phoff;
    uint32_t      e_shoff;
    uint32_t      e_flags;
    uint16_t      e_ehsize;
    uint16_t      e_phentsize;
    uint16_t      e_phnum;
    uint16_t      e_shentsize;
    uint16_t      e_shnum;
    uint16_t      e_shstrndx;
} Elf32_Ehdr;
```

Листинг кода 3.9: заголовок ELF-файла.

Структура определена таким образом, что поля структуры выровнены по естественным для данной архитектуры правилам выравнивания (то есть 16-битные поля располагаются по чётным адресам, а 32-битные - по адресам кратным 4), а полный размер структуры кратен 4 байтам. 16- и 32-битные значения представлены в порядке байт, естественном для соответствующей архитектуры.

- Поле e_ident содержит идентификационную информацию о файле. Поле представляет собой массив байт для того, чтобы иметь одинаковое представление на архитектурах с разным размером слова и разным порядком байт в слове. Значения элементов массива описаны в таблице 3.1:

Таблица 3.1: Поле e_ident.

Элемент	Значение	Описание
e_ident[0]	’\x7f’	“Магическое” значение (константа)
e_ident[1]	’E’	“Магическое” значение (константа)
e_ident[2]	’L’	“Магическое” значение (константа)
e_ident[3]	’F’	“Магическое” значение (константа)
e_ident[4]	1	Размер слова в битах: 0 - неизвестно, 1 - 32, 2 - 64
e_ident[5]	1	Порядок байт: 0 - неизвестно, 1 - little-endian, 2 - big-endian
e_ident[6]	1	Версия формата ELF: 0 - неизвестно, 1 - текущая версия
e_ident[7]	0	ОС и бинарный интерфейс, для Linux - 0
e_ident[8]	0	Версия бинарного интерфейса, для Linux - 0
e_ident[9-15]	0	Зарезервировано

- Поле e_type идентифицирует тип файла: 0 (неизвестно), 1 (объектный файл), 2 (исполняемый файл), 3 (разделяемая библиотека), 4 (core-файл).
- Поле e_machine идентифицирует тип процессора: 0 (неизвестно), 3 (Intel 80386 и совместимые).
- Поле e_version идентифицирует версию файла: 0 (недопустимая версия), 1 (текущая версия).
- Поле e_entry определяет виртуальный адрес точки входа в программу. После загрузки программы в память управление передаётся на этот адрес.
- Поле e_phoff задаёт смещение от начала файла до начала таблицы заголовков программы. Обычно, она идёт непосредственно за заголовком файла, поэтому значение этого поля часто соответствует значению поля e_ehsize. Информация о таблице заголовков программы будет приведена ниже.
- Поле e_shoff задаёт смещение от начала файла до начала таблицы заголовков секций. Информация о таблице заголовков секций будет дана ниже.
- Поле e_flags задаёт дополнительные, специфичные для процессора, флаги.
- Поле e_ehsize хранит размер заголовка ELF-файла. Обычно, этот размер равен пятидесяти двум байтам для 32-битных систем и 64 байта для 64-битных систем.
- Поле e_phentsize хранит размер одной записи в таблице заголовков программы.
- Поле e_phnum хранит количество записей в таблице заголовков программы.
- Поле e_shentsize хранит размер одной записи в таблице заголовков секций.
- Поле e_shnum хранит количество записей в таблице заголовков секций.
- Поле e_shstrndx хранит индекс заголовка секции, которая хранит имена всех секций (см. ниже).

3.9.2 Таблица заголовков секций

Информация, хранящаяся в ELF-файле, организована в секции. Каждая секция имеет своё уникальное имя. Некоторые секции хранят служебную информацию ELF-файла (например, таблицы строк), другие секции хранят отладочную информацию, третьи секции хранят код или данные программы. Таблица заголовков секций представляет собой массив структур Elf32_Shdr. Количество элементов массива определяется полем e_shnum заголовка ELF-файла. Массив находится по смещению, хранящемуся в поле e_shoff. Элемент массива 0 зарезервирован и не используется для описания секций. Таким образом, описания секций находятся в элементах массива с индексами от 1 и до e_shnum - 1. Определение структуры Elf32_Shdr показано в листинге 3.10.

```
typedef struct
{
    uint32_t   sh_name;
    uint32_t   sh_type;
    uint32_t   sh_flags;
    uint32_t   sh_addr;
    uint32_t   sh_offset;
    uint32_t   sh_size;
    uint32_t   sh_link;
    uint32_t   sh_info;
    uint32_t   sh_addralign;
    uint32_t   sh_entsize;
} Elf32_Shdr;
```

Листинг кода 3.10: Определение структуры Elf32_Shdr.

- Поле sh_name хранит индекс имени секции. Индекс имени – это смещение в данных секции, индекс которой задаётся в поле e_shstrndx заголовка ELF-файла. По этому смещению размещается строка, завершающаяся нулевым байтом, являющаяся именем секции.

Таким образом, чтобы получить имя секции необходимо выполнить следующие действия:

1. Загрузить заголовок секции, индекс которой хранится в поле e_shstrndx заголовка ELF-файла.
2. Загрузить тело соответствующей секции.
3. По смещению, заданному в поле sh_name относительно начала области памяти, в которую загружена секция, находится искомая строка имени секции.

- Поле sh_type хранит тип секции. Возможные значения поля перечислены в таблице .

Таблица 3.2: Поле sh_type.

Значение	Символьное имя	Описание
0	SHT_NULL	Пустой заголовок секции. Значения всех прочих полей заголовка секции неопределены.
1	SHT_PROGBITS	Секции программы (код, данные или что-либо ещё).
2	SHT_SYMTAB	Таблица символов (для объектных файлов или динамических библиотек).
3	SHT_STRTAB	Таблица строк.
4	SHT_RELA	Записи о перемещаемых адресах (relocations).
5	SHT_HASH	Хеш-таблица имён для динамического связывания.
6	SHT_DYNAMIC	Информация для динамического связывания.
7	SHT_NOTE	Произвольная дополнительная информация.
8	SHT_NOBITS	Секция не занимает место в файле, но занимает место в адресном пространстве процесса.
9	SHT_REL	Записи о перемещаемых адресах.

- Поле sh_flags хранит битовые флаги, описывающие дополнительные атрибуты.

Таблица 3.3: Поле sh_flags.

Значение	Символьная константа	Описание
1	SHF_WRITE	Содержимое секции должно быть доступно на запись в адресном пространстве процесса.
2	SHF_ALLOC	Для содержимого секции выделяется память в адресном пространстве процесса.
4	SHF_EXECINSTR	Секция содержит инструкции процессора.

Флаги могут комбинироваться с помощью операции побитового ИЛИ.

- Поле sh_addr хранит адрес в виртуальном адресном пространстве процесса в случае, если секция загружается в виртуальное адресное пространство процесса.
- Поле sh_offset хранит смещение от начала файла, по которому размещаются данные секции.
- Поле sh_size хранит размер секции в байтах.
- Поле sh_link хранит индекс другой секции (в некоторых специальных случаях).
- Поле sh_info хранит дополнительную информацию о секции.
- Поле sh_addralign хранит требование по выравниванию адреса начала секции в памяти. Значения 0 или 1 означают отсутствие требования по выравниванию. В противном случае значением поля должна быть степень двойки. Например, секции, загружаемые в виртуальное адресное пространство процесса, как правило, выровнены по размеру страницы процессора (4096).
- Поле sh_entsize хранит размер одной записи, если секция хранит таблицу из записей фиксированного размера.

3.9.3 Таблица заголовков программы (сегментов)

Таблица заголовков программы содержит информацию, необходимую для загрузки программы на выполнение. Таблица заголовков программы представляет собой массив структур Elf32_Phdr. Массив размещается по смещению от начала файла, которое хранится в поле e_phoff заголовка ELF-файла, а количество элементов массива хранится в поле e_phnum заголовка ELF-файла. Определение структуры Elf32_Phdr показано в листинге 3.11.

```
typedef struct
{
    uint32_t   p_type;
    Elf32_Off  p_offset;
    Elf32_Addr p_vaddr;
    Elf32_Addr p_paddr;
    uint32_t   p_filesz;
    uint32_t   p_memsz;
    uint32_t   p_flags;
    uint32_t   p_align;
} Elf32_Phdr;
```

Листинг кода 3.11: Определение структуры Elf32_Phdr.

- Поле p_type хранит тип заголовка. Некоторые возможные значения типа заголовка приведены в таблице .

Таблица 3.4: Поле p_type.

Значение	Символьная константа	Описание
0	PT_NULL	Обозначает не используемую запись
1	PT_LOAD	Сегмент программы, загружаемый в память
2	PT_DYNAMIC	Информация для динамического связывания
3	PT_INTERP	Загрузчик программ
4	PT_NOTE	Дополнительная информация
6	PT_PHDR	Информация о самой таблице заголовков программы
7	PT_TLS	Локальная память потоков

- Поле p_offset хранит смещение от начала файла, по которому располагается данный сегмент.
- Поле p_vaddr хранит виртуальный адрес начала сегмента в памяти.
- Поле p_paddr зарезервировано физический адрес сегментов для систем, использующих физические адреса.
- Поле p_filesz хранит размер сегмента в файле (может быть 0).
- Поле p_memsz хранит размер сегмента в памяти (может быть 0).
- Поле p_flags хранит флаги доступа к сегменту в памяти (могут объединяться с помощью побитового “или”).

Таблица 3.5: Поле p_flags.

Значение	Символьная константа	Описание
1	PT_X	Сегмент доступен на выполнение
2	PT_W	Сегмент доступен на запись
4	PT_R	Сегмент доступен на чтение

Для понимания отношения между Секциями и Сегментами, мы можем представить сегменты как инструмент, упрощающий жизнь загрузчику Linux, поскольку они группируют секции по атрибутам в один общий сегмент, чтобы сделать процесс загрузки исполняемого файла более эффективным, вместо того чтобы каждая отдельная секция загружалась в память. Рисунок 3.2 ниже иллюстрирует эту концепцию:

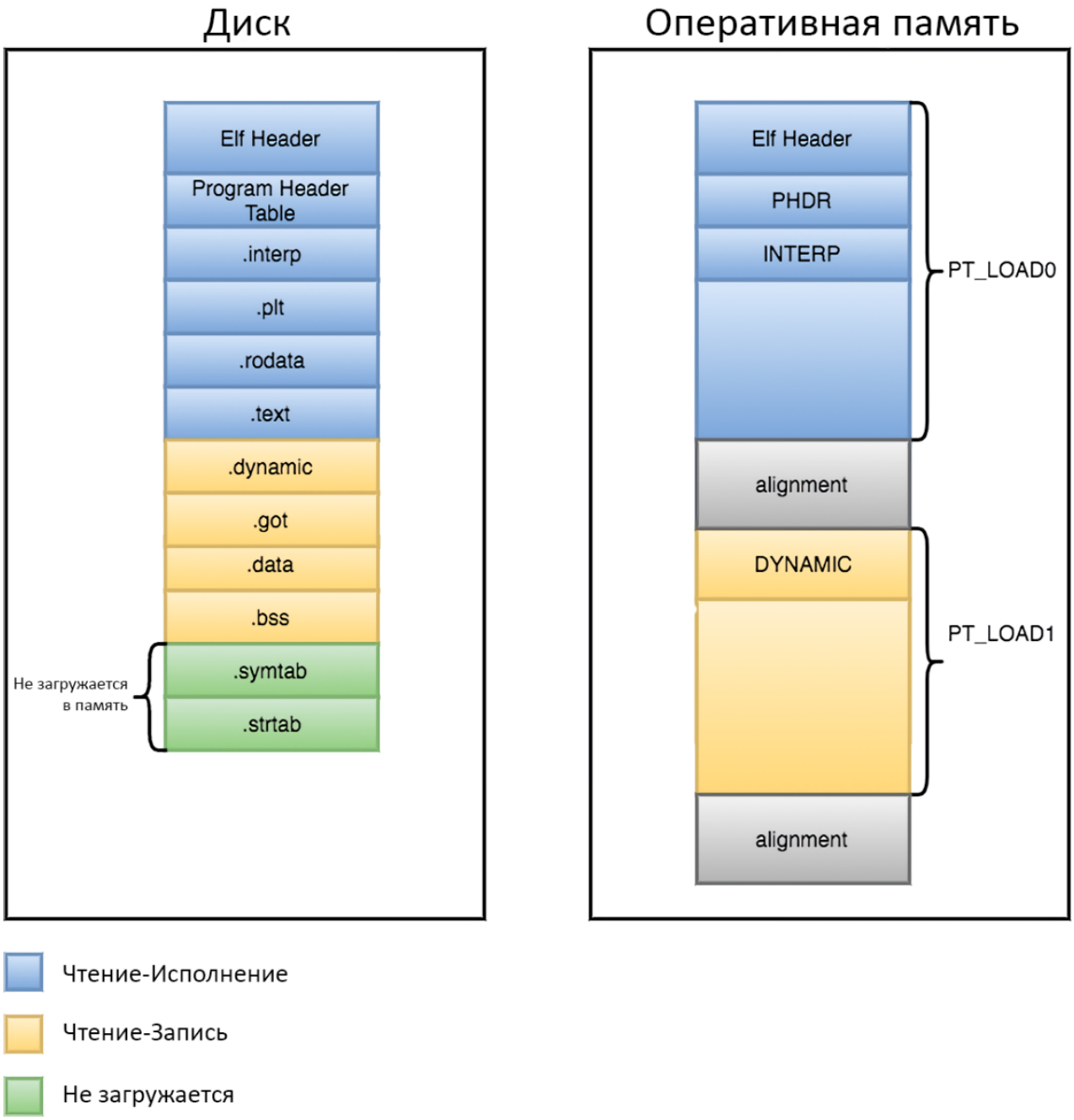

Рис. 3.2: Секции и Сегменты.

Другой важный аспект сегментов заключается в том, что их смещения и виртуальные адреса

должны равны размеру страницы, а их поле p_align должно быть кратно системному размеру страницы. Причина для подобного выравнивания заключается в предотвращении загрузки двух разных сегментов в одну страницу памяти. Это необходимо в связи с тем, что различные сегменты обычно имеют различные атрибуты доступа, что невозможно, если два сегмента загружены в одну страницу памяти.

3.9.4 core-файлы

core-файл – это ELF-файл, у которого значение поля e_type заголовка равно ET_CORE (4). В core-файле таблица заголовков секций пуста, а таблица заголовков программ состоит из записей типа PT_LOAD, хранящих содержимое адресного пространства процесса на момент завершения работы процесса, и записи типа PT_NOTE, хранящей состояние процесса на момент завершения работы процесса. Для тестового файла sample.core содержимое таблицы заголовков программ имеет следующий вид.

Таблица 3.6: Пример содержимого таблицы заголовков программ.

p_type	p_flags	p_offset	p_vaddr	p_filesz	p_memsz	p_align
PT_NOTE	—	0x00000234	0x00000000	1216	0	0
PT_LOAD	r-x	0x00001000	0x08048000	4096	8192	4096
PT_LOAD	rw-	0x00002000	0x0804a000	4096	4096	4096
PT_LOAD	rw-	0x00003000	0x08542000	135168	135168	4096
PT_LOAD	r-x	0x00024000	0x4f2d0000	4096	126976	4096
PT_LOAD	r–	0x00025000	0x4f2ef000	4096	4096	4096
PT_LOAD	rw-	0x00026000	0x4f2f0000	4096	4096	4096
PT_LOAD	r-x	0x00027000	0x4f2f7000	4096	1748992	4096
PT_LOAD	—	0x00028000	0x4f4a2000	0	4096	4096
PT_LOAD	r–	0x00028000	0x4f4a3000	8192	8192	4096
PT_LOAD	rw-	0x0002a000	0x4f4a5000	4096	4096	4096
PT_LOAD	rw-	0x0002b000	0x4f4a6000	12288	12288	4096
PT_LOAD	rw-	0x0002e000	0xb778a000	4096	4096	4096
PT_LOAD	rw-	0x0002f000	0xb77a1000	8192	8192	4096
PT_LOAD	r-x	0x00031000	0xb77a3000	4096	4096	4096
PT_LOAD	rw-	0x00032000	0xbfe6d000	139264	139264	4096

Первый сегмент (сегмент PT_NOTE) содержит информацию о состоянии процесса на момент создания core-файла.

Для core-файлов возможны следующие значения поля n_type (в таблице ниже перечислены не все возможные значения).

Таблица 3.7: Возможные значения поля n_type.

Значение	Символьная константа	Описание
1	NT_PRSTATUS	Информационная часть записи имеет тип prstatus_t
2	NT_FPREGSET	Информационная часть записи имеет тип prfpregset_t
3	NT_PRPSINFO	Информационная часть записи имеет тип prpsinfo_t

Типы структур, используемые в информационных частях записей, определены в заголовочном файле:

```
#include <sys/procfs.h>
```

Определение структуры prstatus_t показано в листинге 3.12.

```
typedef struct elf_prstatus
{
    struct elf_siginfo pr_info;                 // информация о сигналах
    short int pr_cursig;                       // текущий сигнал
    unsigned long int pr_sigpend;               // множество сигналов, ожидающих
    ↪ доставки
    unsigned long int pr_sighold;               // множество удерживаемых сигналов
    pid_t pr_pid;                              // pid процесса
    pid_t pr_ppid;                             // pid родителя
    pid_t pr_pgrp;                             // группа процессов
    pid_t pr_sid;                              // идентификатор сессии
    struct timeval pr_utime;                    // пользовательское время
    struct timeval pr_stime;                    // системное время
    struct timeval pr_cutime;                   // накопленное пользовательское
    ↪ время
    struct timeval pr_cstime;                   // накопленное системное время
    elf_gregset_t pr_reg;                      // регистры общего назначения
    int pr_fpvalid;                            // true, если использовались
    ↪ регистры FPU
}
```

Листинг кода 3.12: Определение структуры prstatus_t.

Тип elf_gregset_t определён следующим образом:

```
typedef unsigned long elf_gregset_t[ELF_NGREG];
```

то есть представляет собой массив, в котором каждый регистр общего назначения находится по определённому индексу.

Таблица 3.8: elf_gregset_t для архитектуры x86.

индекс	регистр
0	ebx
1	ecx
2	edx
3	esi
4	edi
5	ebp
6	eax
7	ds
8	es
9	fs
10	gs
11	orig_eax
12	eip
13	cs
14	eflags
15	sp
16	ss

3.9.5 Формат отладочной информации stabs

При компиляции в исполняемый файл может добавляться отладочная информация, которую отладчик использует для отображения хода исполнения программы в терминах языка высокого уровня. Существует несколько форматов отладочной информации (STABS, DWARF), далее описывается формат STABS как самый простой.

Для компиляции программы с добавлением отладочной информации в формате STABS используется опция gcc -gstabs, например

```
gcc -gstabs -std=gnu11 sample.c -o sample
```

В формате STABS отладочная информация хранится в секциях .stab и .stabstr ELF-файла. Секция .stab содержит массив структур, описанных в листинге 3.13.

```
1 struct Stab
2 {
3     uint32_t n_strx;    // позиция начала строки в секции .strstab
4     uint8_t n_type;     // тип отладочного символа
5     uint8_t n_other;    // прочая информация
6     uint16_t n_desc;    // описание отладочного символа
7     uintptr_t n_value; // значение отладочного символа
8 };
```

Листинг кода 3.13: Содержимое секции .stab.

Секция .stabstr хранит символьные строки, завершающиеся байтом 0, которые используются в записях в секции .stab. Поле n_type хранит тип записи. Возможные типы записей находятся в stab.h, в таблице приведены только наиболее значимые из них.

Таблица 3.9: Поле n_type.

Симв. имя	Значение	Описание
N_SO	0x64	Информация о единице компиляции: n_desc – язык исходного кода; n_strx – индекс в секции stabstr строки имени основного файла единицы компиляции; n_value – адрес первой инструкции.
N_SOL	0x84	Имя файла, устанавливаемое с помощью директивы #line или имя файла, включаемого с помощью #include n_strx – индекс в секции .stabstr строки имени файла; Будем предполагать, что действие имени файла, устанавливаемого в записи N_SOL, начинается со следующей записи.
N_FUN	0x24	Имя функции n_value – адрес первой инструкции функции; n_strx – индекс в секции .stabstr строки имени функции Имя состоит непосредственно из названия и после ':' следуют типы параметров, если таковые есть.
N_SLINE	0x44	Номер строки исполняемого кода n_value – смещение первой инструкции строки относительно начала функции n_desc – номер строки в исходном тексте

Каждой единице компиляции соответствуют две записи N_SO. Первая запись находится в начале описания единицы компиляции и содержит её имя в поле n_strx и адрес первой инструкции в поле n_value. Вторая запись находится в конце описания единицы компиляции и содержит нулевое значение (пустая строка) в поле n_strx и адрес, непосредственно следующий за концом кода данной единицы компиляции в поле n_value.

Записи N_SLINE упорядочены по смещениям внутри функции.

Первая запись в таблице .stab является служебной и имеет тип N_UNDF. Индекс этой записи полагается равным 0.

Формат STABS не позволяет однозначно установить адрес, на котором заканчивается тело функции. Можно использовать следующую эвристику: функция заканчивается либо с началом следующей функции, тогда адрес конца функции - это адрес начала следующей функции, либо с концом единицы компиляции, тогда адрес конца функции - это адрес, хранящийся в поле n_value записи N_SO в конце единицы компиляции.

Как обычно, предполагается, что все диапазоны адресов и значений включают в себя нижнее значение, но не включают в себя верхнее значение, то есть имеют вид [low;high).

Имя файла, в котором располагается функция, может находиться в записи N_SOL после записи N_FUN самой функции, но до первой записи N_SLINE. Следует полагать, что имя файла, в котором определена функция совпадает с именем файла, установленному до первой записи N_SLINE в данной функции.

Пример: файл file1.h в листинге 3.14.

```
1 #include <stdio.h>
2 void func2(int val)
3 {
4   printf("%d\n", val);
5 }
```

Листинг кода 3.14: Файл file1.h.

Файл file1.c в листинге 3.15

```c
#include "file1.h"
int val;
int func1(void)
        {
            scanf("%d", &val);
            return val;
        }

int main()
        {
            func1();
            func2(val);
            return 0;
        }
```

Листинг кода 3.15: Файл file1.c.

Записи STABS:

Таблица 3.10: Записи STABS.

SO	0	2	08048430	1	file1.c
FUN	0	0	08048430	3880	func2:F(0,18)
...					
SOL	0	0	08048430	668	file1.h
SLINE	0	3	00000000	0	
SLINE	0	4	00000006	0	
SLINE	0	5	00000019	0	
FUN	0	0	0804844b	3905	func1:F(0,1)
SOL	0	0	0804844b	1	file1.c
SLINE	0	4	00000000	0	
SLINE	0	5	00000006	0	
SLINE	0	6	0000001a	0	
SLINE	0	7	0000001f	0	
FUN	0	0	0804846c	3918	main:F(0,1)
SLINE	0	10	00000000	0	
SLINE	0	11	00000009	0	
SLINE	0	12	0000000e	0	
SLINE	0	13	0000001b	0	
SLINE	0	14	00000020	0	
SO	0	0	0804848e	0	

3.10 Контрольные вопросы

1. Опишите этапы сборки кода.
2. В чём различие между статической и динамической библиотеками?

3. Ваша программа использует прекомпилированную библиотеку mylib. Вызовом какой команды вы скомпонуете эту библиотеку с объектными файлами вашей программы:
   (a) если библиотека статическая;
   (b) если библиотека динамическая.
4. Опишите последовательность команд, выполняющую следующие действия:
   (a) Запуск вашей программы my_program в отладчике;
   (b) Установка точек останова на:
      i. Строку 28;
      ii. Начало функции foo;
      iii. Строку 35, если ваша локальная переменная bar лежит в диапазоне [0:9];
   (c) Вывод исходного кода функции foo;
   (d) Выполнение кода программы до очередной точки останова;
   (e) Выполнение следующего шага программы без перехода во вложенную функцию;
   (f) Вывод значения переменной bar и последующая установка значения этой переменной в 8;
   (g) Удаление всех точек останова;
   (h) Выполнение программы вплоть до её завершения.
5. Модифицируйте пример make-файла из раздела “Пример простого Makefile” таким образом, чтобы он не зависел от имён файлов вашей программы.

3.11 Ссылки

1. Теория по GCC: `https://www3.ntu.edu.sg/home/ehchua/programming/cpp/gcc_make.html`
2. Совет по ручной компоновке: `https://stackoverflow.com/questions/44361841/manually-link-standard-library-in-mingw`
3. Теория по библиотекам: `https://ravesli.com/staticheskie-i-dinamicheskie-biblioteki/`
4. Практикум по библиотекам: `https://renenyffenegger.ch/notes/development/languages/C-C-plus-plus/GCC/create-libraries/index`
5. Отладка программ: `https://it.wikireading.ru/14060`
6. Make: `https://it.wikireading.ru/14051`
   `http://rus-linux.net/nlib.php?name=/MyLDP/algol/gnu_make/gnu_make_3-79_russian_manual.html`
7. Кроссплатформенная компиляция: `https://habr.com/ru/post/319736/`
8. ELF: `http://www.intezer.com/executable-linkable-format-101-part1-sections-segments/`
   `https://ejudge.ru/study/3sem/elf.html`

Лабораторная работа 4

Система PULPino
Программирование и отладка СнК

4.1 Система PULPino

На данный момент существует несколько крупных платформ для проектирования и разработки систем на кристалле, основанных на архитектуре RISC-V. В данном курсе мы остановимся на платформе PULP (Parallel Ultra-Low Power), предоставляющей несколько вариаций процессоров, различные периферийные контроллеры и, самое главное, готовые решения систем. В частности, к таким готовым решениям относится PULPino.

PULPino – это открытая система на базе 32-битного одноядерного RISC-V микропроцессора. Одной из ключевых ее особенностей является простота, поэтому в целях упрощения системы и достижения легкости работы с ней в PULPino намерено не реализована часть функционала. Например, в системе не используется кэш-память и возможность прямого доступа к памяти (DMA). В то же время в системе имеются основные необходимые функциональные блоки, рассмотреть которые можно на Рисунке 4.1. Исходные файлы аппаратной части PULPino на SystemVerilog находятся в архиве pulpino.zip.

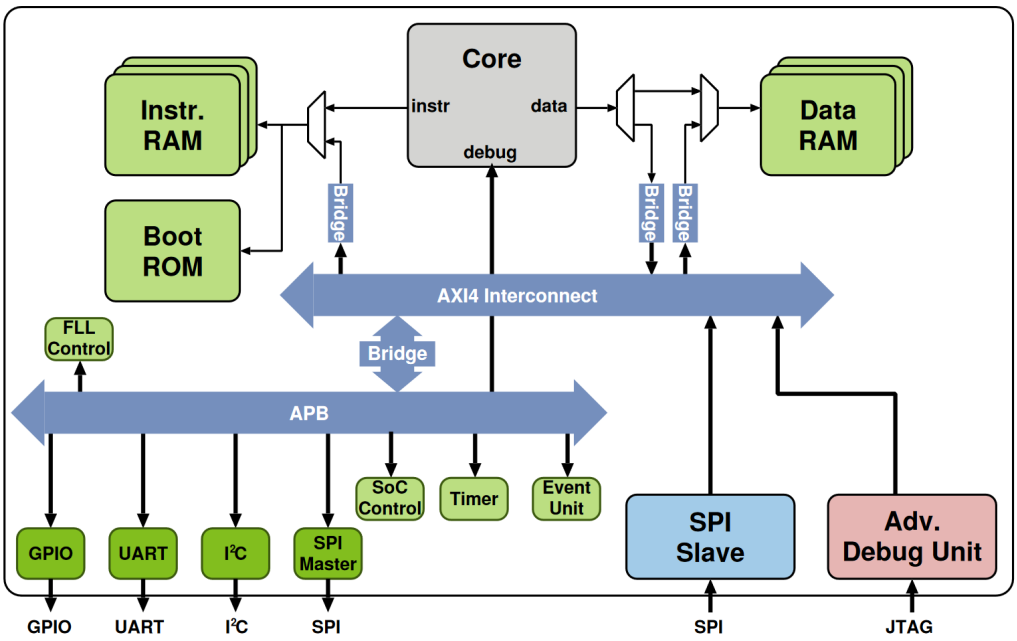

Рис. 4.1: Структурная схема системы PULPino.

В зависимости от конфигурации в качестве ядра могут использоваться ядра zero-riscy или

RI5CY. В zero-riscy применяется вычислительный конвейер с двумя стадиями, ядро полностью поддерживает работу с наборами инструкций RV32I и RV32C (обозначается как RV32IC). Оно также может быть сконфигурировано для использования RV32M и ограниченного количества инструкций RV32E. Основными преимуществами zero-riscy, помимо простоты, являются сниженное энергопотребление, а также малая занимаемая площадь на кристалле. Ядро RI5CY, в свою очередь, основано на четырехстадийном конвейере. Ядро может исполнять инструкции RV32IMC и, опционально, RV32F. Оно также обладает сниженными энергозатратами, однако обеспечивает большую функциональность и производительность, чем zero-riscy. В связи с этим в данном курсе мы используем систему PULPino с ядром RI5CY.

Как можно видеть из структурной схемы, в системе разделены память инструкций и данных. Также присутствует 512-байтовое ПЗУ, содержащее в себе инструкции загрузчика прошивки, который загружает программу из внешнего запоминающего устройства через интерфейс SPI (модуль SPI Master на схеме). При этом и ОЗУ с инструкциями и данными, и ПЗУ с загрузчиком находятся в едином адресном пространстве.

К шине APB подключены контроллеры интерфейсов, среди которых UART, I2C и ведущая сторона SPI, а также простой интерфейс GPIO, поддерживающий до 32 входных и выходных сигналов.

FLL Control (Frequency-locked loop control) – модуль управления автоподстройкой частоты, необходимый для поддержания тактовой частоты устройства на постоянном уровне при помощи цепи отрицательной обратной связи. В текущей реализации PULPino фактически не используется.

Помимо перечисленного, к шине подключен модуль обработчика событий и прерываний (Event unit). Модуль поддерживает векторные прерывания, количество как прерываний, так и событий ограничено 32 линиями.

Также на линии APB есть модуль счетчика времени (Timer). Количество таймеров внутри модуля определяется параметром при инициализации и по умолчанию равно двум. При переполнении или равенстве некоторому устанавливаемому значению, счетчик вызывает соответствующее прерывание.

Наконец, модуль SoC Control обеспечивает вспомогательный функционал для управления работой некоторых из блоков. К его возможностям относятся:

- установка начального адреса загрузчика прошивки;
- управление подачей тактового сигнала на контроллеры GPIO, UART, I2C, SPI, FLL, таймер и обработчик событий;
- чтение информации о версии системы, размере ОЗУ и ПЗУ и наличии кэша инструкций и данных;
- запись и чтение во вспомогательный статусный регистр, который может хранить результат верификации;
- мультиплексирование выходных сигналов, иначе говоря, использование одних и тех же выходных контактов схемы для передачи сигналов разного назначения.

Можно заметить, что на Рисунке 4.1, шина APB связана с портом отладки процессора. Отдельный модуль конвертирует транзакции по шине APB в команды отладочного интерфейса RI5CY. На схеме распределения адресов в памяти (memory map) отмечены зарезервированные адреса Debug Port – они и обеспечивают доступ к управлению процессором. В частности, таким образом можно изменить счетчик команд на необходимое значение.

Из функциональных блоков на схеме системы мы пока не затронули только один: Advanced Debug Unit (“расширенный модуль отладки”). Однако перед тем, как подробно с ним познакомиться, рассмотрим вопрос тестирования и отладки систем на кристалле в целом.

4.2 Интерфейс JTAG

В процессе написания программного обеспечения для систем на кристалле возникает необходимость его тестирования. Первоначально происходит проверка корректности работы отдельных функций и затем ПО в целом. На этом этапе используются инструменты отладчика: точки останова, трассировка, отслеживание состояния переменных и другие.

Проверка ошибок компиляции или адресации требует уже интегрирования программы в систему и исследования ее выполнения на другом уровне. Для этого необходим инструмент, который может обеспечивать как загрузку данных и управление текущим состоянием системы, так и обратное считывание состояния внутренних регистров. Широкое распространение в подобном применении получил интерфейс JTAG.

JTAG – последовательный протокол, используемый для тестирования интегральных схем, внутрисхемного программирования, инициализации внешней энергонезависимой памяти. Под JTAG обычно понимают интерфейс, заданный стандартом IEEE 1149.1. Реализация JTAG включает в себя сигнальные линии TAP (Test Access Port), а также TAP контроллер, содержащий в себе конечный автомат, регистры инструкций и данных и шину отладки для связи с ядром. Сигналы, входящие в TAP, перечислены в Таблице 4.1.

Таблица 4.1: Сигнальные линии Test Access Port.

Сигнал	Описание
TCK	Test Clock -– Входной тактовый сигнал.
TRSTN	Test Reset -– Входной сигнал сброса.
TDI Test	Data Input -– Входной порт для приема данных устройством.
TDO Test	Data Output -– Выходной порт для передачи данных устройством.
TMS	Test Mode Select -– Вход, определяющий состояние TAP контроллера.

Линия TCK используется как источник сигнала синхронизации. По положительному фронту считывается значение с линий TMS и TDI, а по отрицательному устанавливается логический уровень на TDO.

На вход TDI передается поток входных данных, назначение которых зависит от состояния конечного автомата TAP контроллера. Линии TDI и TMS имеют подтяжку к логической единице, поэтому в момент отсутствия сигнала на входе, на них установлен высокий уровень. Линия TDO, с которой считываются выходные данные, в свою очередь, находится в состоянии высокого импеданса (Z), если на нее не подаются значения из устройства.

Сигнал TRSTN не является обязательным для использования в JTAG, однако он используется в системе PULPino. В данном случае это асинхронный сигнал сброса активным низким логическим уровнем.

4.3 TAP контроллер

TAP контроллер представляет собой конечный автомат, состояния которого задаются сигналами TCK и TMS. Переход от одного состояния к другому определяется логическим уровнем TMS на переднем фронте TCK. В зависимости от состояния, контроллер может производить операции с регистрами данных или инструкций, либо же находиться в режиме ожидания или сброса. Представление конечного автомата изображено на Рисунке 4.2.

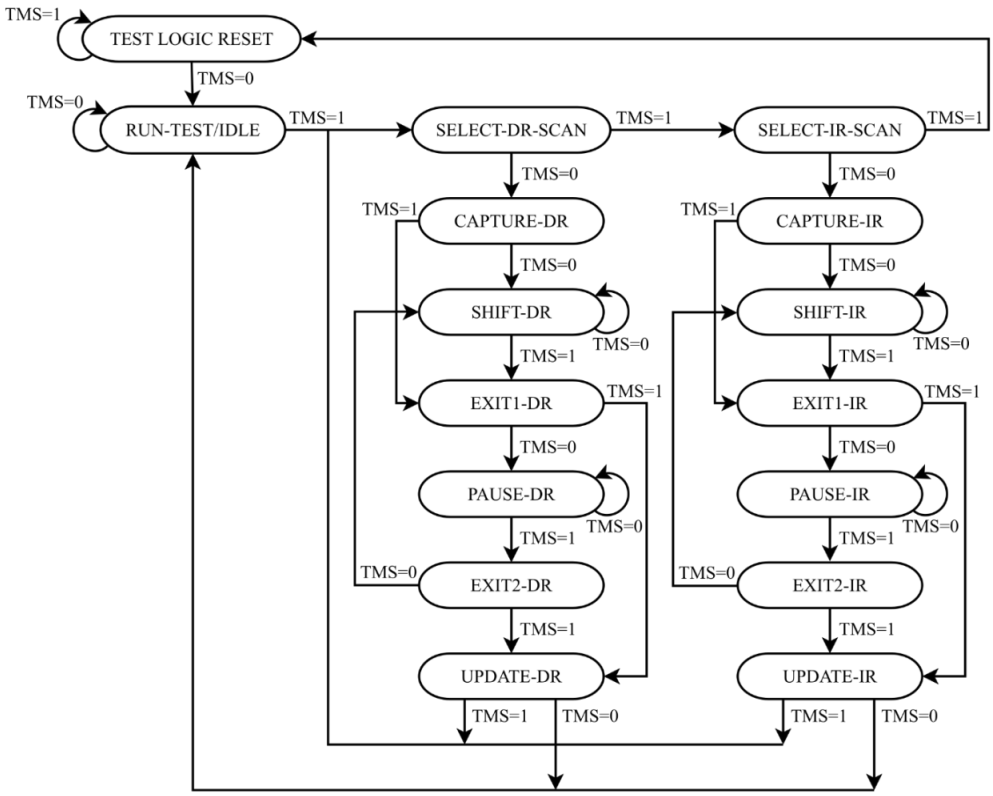

Рис. 4.2: Схема конечного автомата TAP контроллера.

Доступ к отладочному функционалу осуществляется через последовательный сдвиг команды в регистр инструкций. Команда определяет операцию, совершаемую над соответствующим регистром данных: чтение или запись. К регистрам данных в соответствии со стандартом относятся:

- регистр периферийного сканирования (Boundary Scan Register, BSR),
- регистр обхода (Bypass Register),
- регистр идентификации устройства (Device Identification Register),
- прочие регистры, зависящие от конкретной реализации.

Периферийное сканирование, или Boundary Scan, представляет собой механизм проверки цепей устройства на целостность электрических соединений. Также Boundary Scan используется для передачи тестовых импульсов на входные контакты модуля и считывания значений с выходных. Для реализации механизма используются ячейки периферийного сканирования (Boundary Scan Cells). Ячейки последовательно соединены друг с другом, тем самым образуя регистр периферийного сканирования. В нормальном режиме работы ячейки связывают контакты устройства с его внутренними входными и выходными сигналами. В режиме периферийного сканирования логические входы и выходы определяются содержимым BSR. BSR реализован как сдвиговый регистр, на вход которого поступают данные линии TDI, а выдвигаемые данные передаются на выход TDO. Принцип работы периферийного сканирования показан на рисунке 4.3.

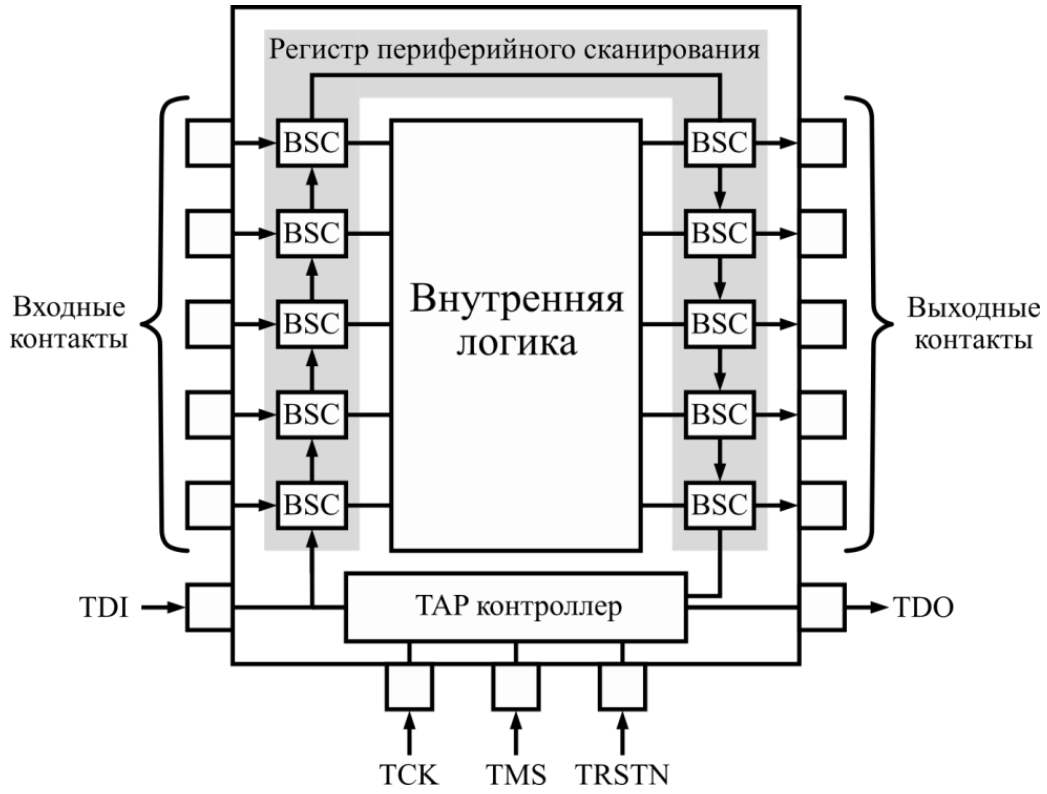

Рис. 4.3: Схема применения периферийного сканирования в устройстве.

Регистр обхода содержит в себе всего один бит и служит для установки прямой связи между линиями TDI и TDO. После передачи команды BYPASS на выходе TDO появляются значения с линии TDI с задержкой в один такт сигнала синхронизации TCK. Это полезно в тех случаях, когда несколько аппаратных модулей имеют собственные JTAG порты и последовательно соединяются в отладочную цепь. Тогда некоторые из модулей возможно изолировать из цепи прохождения данных, не прерывая при этом их работы. Такой подход находит применение в поиске некорректно работающих частей системы, а также в выборочном тестировании.

Доступ к значению регистра идентификации устройства осуществляется при помощи команды IDCODE. В 32 бита регистра входит информация о версии, номере и производителе устройства. В то время как использование регистра является не обязательным, в стандарте JTAG зафиксирована команда для доступа к нему.

TAP контроллер в системе puplino определяет шесть 4-битных команд для записи в регистр инструкций. Команды и их коды приведены в Таблице 4.2. При этом аппаратная поддержка существует только для трех из них: IDCODE, BYPASS и DEBUG. Если первые две команды известны и описаны в стандарте JTAG, то DEBUG является пользовательской командой. Использование команды DEBUG позволяет нам работать с модулем отладки – Advanced Debug Unit, который уже упоминался в ранее. Рассмотрим его работу подробнее.

Таблица 4.2: Коды инструкций TAP контроллера.

Код команды	Команда
0x0	EXTEST
0x1	SAMPLE_PRELOAD
0x2	IDCODE
0x8	DEBUG
0x9	MBIST
0xF	BYPASS

4.4 Advanced Debug Interface

В системе PULPino интерфейс JTAG используется в аппаратном модуле отладки, основанном на Advanced Debug Interface (ADI – расширенный интерфейс отладки). ADI – это интерфейс между портом JTAG, системной шиной AXI4 и отладочным интерфейсом ядра. Хотя в текущей реализации недоступен режим периферийного сканирования, модуль отладки предоставляет возможности для тестирования и программирования устройства.

При получении инструкции DEBUG на линии TDO устанавливается выходное значение из внешнего модуля отладки. Помимо этого, в модуле отладки биты TDI последовательно вдвигаются в 64-битный регистр. Содержимое регистра передается как в модуль связи с системной шиной AXI4, так и в модуль обмена данными с CPU. Старший бит сдвигового регистра используется для установки источника сигнала TDO. Если он равен единице и TAP контроллер находится в состоянии UPDATE_DR, происходит проверка следующих 5 бит регистра. На выход передаются данные с подмодуля AXI4, если значение равно 0x0, с подмодуля CPU – если 0x1. Схема модуля отладки представлена на Рисунке 4.4.

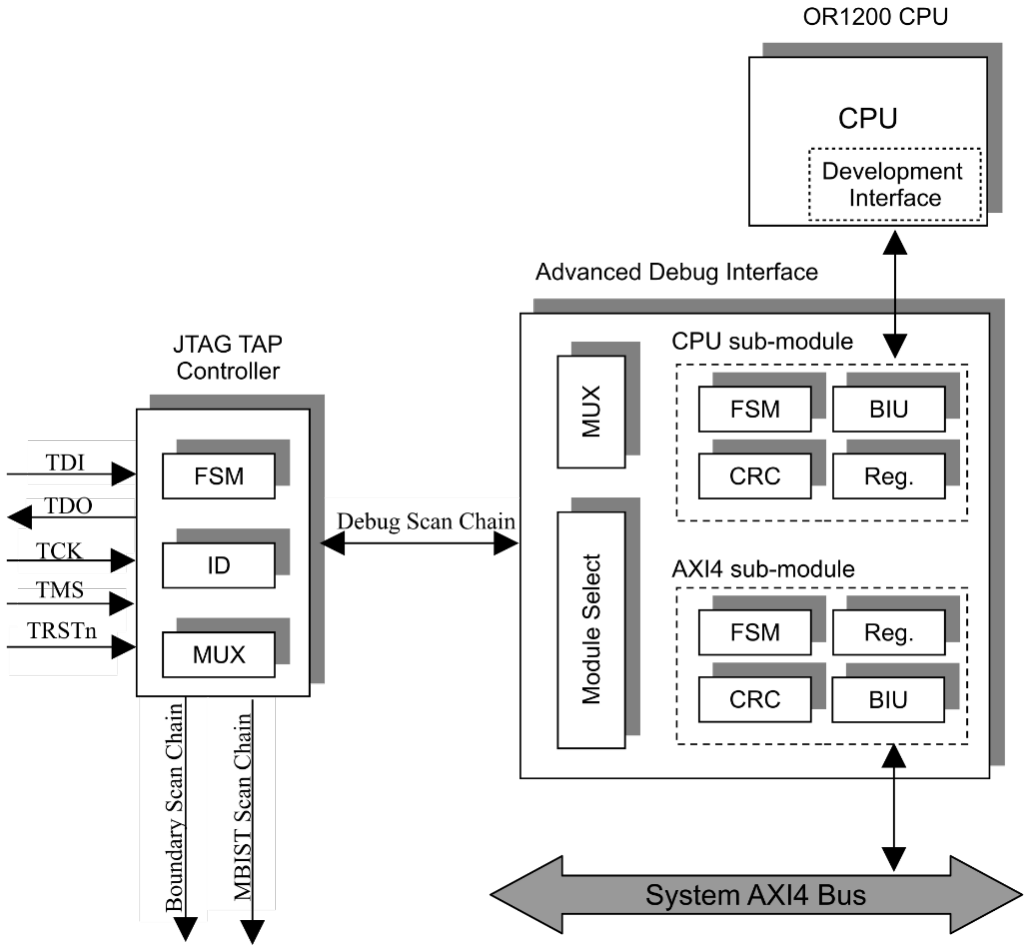

Рис. 4.4: Структурная схема модуля отладки Advanced Debug Interface.

Основной функцией подмодуля CPU является возможность доступа к статусному регистру процессора. Статусный регистр имеет длину равную числу ядер и содержит информацию об их режиме работы. Если установить значение “1” в соответствующий ядру бит, ядро перейдет в режим останова. В режиме останова оно не исполняет инструкции, однако сохраняет способность восстановить свою работу с сохраненного состояния. Установка значения “0” возобновляет работу ядра. Аналогичным образом изменяются значения статусного регистра во время отладки при попадании в точку останова. Так, отслеживая значения регистра, можно узнать о

возникновении условия останова и продолжить работу процессора после завершения отладки.

Инструкция, передаваемая на подмодуль CPU, состоит из 57 старших бит сдвигового регистра ADI. Единственный старший бит, как указывалось выше, используется для управления линией TDO. Поэтому в момент, когда он равен логической “1” и производится установка источника, подмодуль CPU игнорирует текущую команду. Следующие 4 бита определяют код выполняемой операции. Оставшиеся биты инструкции обрабатываются в зависимости от операции.

Для работы со статусным регистром используются три команды: команда выбора регистра, записи в регистр и команда NOP (“no operation” – не выполнять операцию) для чтения из регистра. Коды данных операций указаны ниже в Таблице 4.3.

Таблица 4.3: Коды операций подмодуля CPU.

Код команды	Операция
0x0	Инструкция NOP
0x9	Запись в регистр
0xD	Выбор регистра

Подмодуль AXI4 предназначен для предоставления доступа к системной памяти. Он позволяет загружать прошивку в память инструкций и данных, считывать данные из памяти, а также задавать точки останова для тестирования.

Все 64 бита сдвигового регистра ADI передаются на подмодуль AXI4. Назначение старших 5 бит аналогично подмодулю CPU: 1 бит управления линией TDO, 4 бита кода команды.

К командам, необходимым для работы с системной памятью, относятся команды чтения и записи. Они работают в серийном (“burst”) режиме: после отправки инструкции чтение и запись производятся единым блоком данных без прерывания на повторную отправку команды. Как запись, так и чтение могут производиться для 8-, 16-, 32- или 64-битных слов. После каждой оправки слова текущий адрес для доступа к памяти инкрементируется. Коды команд приведены в Таблице 4.4.

Таблица 4.4: Коды операций подмодуля AXI4.

Код команды	Операция
0x0	Инструкция NOP
0x1	Серийная запись 8-битных слов
0x2	Серийная запись 16-битных слов
0x3	Серийная запись 32-битных слов
0x4	Серийная запись 64-битных слов
0x5	Серийное чтение 8-битных слов
0x6	Серийное чтение 16-битных слов
0x7	Серийное чтение 32-битных слов
0x8	Серийное чтение 64-битных слов

4.5 Секции и адресация исполняемого ELF файла

Как было сказано ранее, интерфейс JTAG и Advanced Debug Interface системы PULPino позволяют производить внутрисхемное программирование СнК, иначе говоря, записывать прошивку в память инструкций и данных. Компилятор GCC генерирует прошивку в виде исполняемого файла формата ELF. Как нам уже известно, файл помимо данных для записи в память содержит в себе различные метаданные о целевой машине: архитектуру, порядок байт, названия секций и

другое. Исходя из этого, встает задача генерации из ELF двух файлов: один для записи в память инструкций, другой – в память данных.

В Таблице 4.5 перечислен список секций, содержимое которых требуется распределить в соответствующую память.

Таблица 4.5: Секции ELF файла с указанием памяти для записи.

Память инструкций	.vectors, .text
Память данных	.preinit_array, .init_array, .fini_array, .rodata, .shbss, .data, .bss, .stack, .stab, .stabstr

Считать содержимое секций ELF файла можно при помощи различных утилит, например, readelf, objcopy, objdump. Приведем пример скрипта, который генерирует два файла, содержащие 32-битные значения, каждое из которых соответствует своему адресу в памяти инструкций или данных:

```
filename=$(echo $(basename $1) | sed -e 's/\.[^.]*$//')
objcopy -I elf32-little -O binary -j .vectors -j .text $1 "$filename"_text.bin
hexdump -ve '"%08x\n"' "$filename"_text.bin > "$filename"_text.dat
rm "$filename"_text.bin

objcopy -I elf32-little -O binary -j .preinit_array -j .init_array -j
 ↪  .fini_array \
    -j .rodata -j .shbss -j .data -j .bss -j .stack -j .stab -j .stabstr $1
     ↪  "$filename"_data.bin
hexdump -ve '"%08x\n"' "$filename"_data.bin > "$filename"_data.dat
rm "$filename"_data.bin

if [ $# -eq 2 ]
    then
        cp "$filename"_text.dat "$filename"_data.dat "$2"
fi
```

При запуске скрипта с названием исполняемого файла в качестве первого аргумента, в текущей директории должны появиться DAT файлы с именем ELF файла, дополненного постфиксом “_data” или “_text”. Постфикс указывает на назначение содержимого: “_data” для записи в память данных, “_text” – в память инструкций. Помимо этого, можно указать вторым аргументом путь до директории, в которую необходимо скопировать результат. Это бывает удобно в случаях, когда САПР требует расположения файлов данных в корневой папке проекта или же если, например, в тесте указан относительный путь до исходных данных.

4.6 Организация загрузки данных в память

Итак, на данном этапе рассмотрено, как устроена система PULPino, и теперь для проверки работы в ней программного обеспечения необходимо провести моделирование. Мы знаем в теории, что записать в память можно 32-битный набор значений через модуль отладки. А доступ к модулю отладки можем получить через сигналы TAP. При этом нужные значения уже сгенерированы в виде двух DAT файлов.

Рассмотрим, как производится загрузка полученных значений в память с помощью интерфейсов JTAG и ADI. Для обмена данными с модулем отладки можем использовать готовые классы JTAG_reg и adv_dbg_if_t, которые можно найти в репозитории training_soc в файле tb/tb_jtag_pkg.sv. Ниже приведена структура классов с кратким пояснением существующих методов:
[H]

```
class JTAG_reg  #(int unsigned size = 32);

    // поля
    virtual jtag_i jtag_if // объект интерфейса JTAG
    logic [`JTAG_INSTR_WIDTH-1:0] instr; // инструкция TAP контроллера

    // конструктор
    function new (virtual jtag_i j, logic [`JTAG_INSTR_WIDTH-1:0] i = 'h0);

    // методы
    task jtag_softreset(); // переход в IDLE конечного автомата JTAG
    task jtag_reset(); // переход в RESET конечного автомата JTAG
    task idle(); // переход из EXIT в IDLE
    task update_and_goto_shift(); // переход из CAPTURE/SHIFT/PAUSE в IDLE
    task jtag_goto_SHIFT_IR(); // переход из IDLE в SHIFT_IR
    task jtag_goto_SHIFT_DR(); // переход из IDLE в SHIFT_DR
    task jtag_shift_SHIFT_IR(); // запись поля instr в регистр инструкций и
     ↪  переход из SHIFT_IR в EXIT_IR
    task jtag_shift_NBITS_SHIFT_DR(input int unsigned numbits, input
     ↪  logic[size-1:0] datain,output logic[size-1:0] dataout); // запись
     ↪  numbits значений data_in в регистр данных, переход из SHIFT_DR в
     ↪  EXIT_DR
    task shift_nbits_noex(input int unsigned numbits, input logic[size-1:0]
     ↪  datain,output logic[size-1:0] dataout); // сдвиг numbits значений
     ↪  data_in по линии TDI в состоянии IDLE
    // методы с указателем на объект класса, являются «обертками» описанных
     ↪  выше методов
    task start_shift();
    task shift_nbits(input int unsigned numbits, input logic[size-1:0]
     ↪  datain,output logic[size-1:0] dataout);
    task setIR();
    task shift(input logic[size-1:0] datain,output logic[size-1:0] dataout);
    // методы синхронизации JTAG через линию TCK
    local task jtag_clock(input int cycles);
    local task jtag_wait_halfperiod(input int cycles);

class adv_dbg_if_t;

    // поля
    JTAG_reg #(.size(256)) jtag_cluster_dbg; // объект класса JTAG_reg
    virtual jtag_i jtag_if; // объект интерфейса JTAG

    // конструктор
```

```
36      function new (virtual jtag_i j);
37
38      // методы
39      task jtag_reset(); // jtag_reset() для jtag_cluster_dbg
40      task jtag_softreset(); // jtag_softreset() для jtag_cluster_dbg
41      task init(); // setIR() для jtag_cluster_dbg
42      task axi4_nop(); // AXI4 NOP инструкция
43      // AXI4 серийная запись 8-, 16-, 32- и 64-битных слов
44      task axi4_write8(input logic[31:0] addr, input int nwords, input logic
        ↪  [255:0][7:0] data);
45      task axi4_write16(input logic[31:0] addr, input int nwords, input logic
        ↪  [255:0][15:0] data);
46      task axi4_write32(input logic[31:0] addr, input int nwords, input logic
        ↪  [255:0][31:0] data);
47      task axi4_write64(input logic[31:0] addr, input int nwords, input logic
        ↪  [255:0][63:0] data);
48      local task axi_write(input [4:0] write_size, input logic[31:0] addr, input
        ↪  int nwords, input logic [255:0][31:0] data);
49      // AXI4 серийное чтение 8-, 16-, 32- и 64-битных слов
50      task axi4_read8(input logic[31:0] addr, input int nwords, output logic
        ↪  [255:0][7:0] data);
51      task axi4_read16(input logic[31:0] addr, input int nwords, output logic
        ↪  [255:0][15:0] data);
52      task axi4_read32(input logic[31:0] addr, input int nwords, output logic
        ↪  [255:0][31:0] data);
53      task axi4_read64(input logic[31:0] addr, input int nwords, output logic
        ↪  [255:0][63:0] data);
54      // запись в регистр CPU
55      task cpu_write(input logic [3:0] cpu_id, input logic[31:0] addr, input int
        ↪  nwords, input logic [255:0][31:0] data);
56      // чтение из регистра CPU
57      task cpu_read(input logic [3:0] cpu_id, input logic[31:0] addr, input int
        ↪  nwords, output logic [255:0][31:0] data);
58      // ожидание останова CPU
59      task cpu_wait_for_stall();
60      // вызов останова CPU
61      task cpu_stall(input logic [15:0] cpu_mask);
62      // возобновление работы CPU
63      task cpu_reset();
64      // чтение данных по debug шине CPU
65      task cpu_read_gpr(input logic [3:0] cpu_id, input logic [4:0] addr, output
        ↪  logic [31:0] data);
66      // запись данных по debug шине CPU
67          task cpu_write_gpr(input logic [3:0] cpu_id, input logic [4:0] addr,
            ↪  input logic [31:0] data);
```

Когда инструменты для работы определены, разберем принцип работы части перечисленных функций и сценариев. Сначала рассмотрим объявление классов их конструкторов:
[H]

```
`define JTAG_CLUSTER_DEBUG  4'b1000
`define JTAG_INSTR_WIDTH    4

interface jtag_i;
   logic tck   = 1'b0;
   logic trstn = 1'b0;
   logic tms   = 1'b0;
   logic tdi   = 1'b0;
   logic tdo;
endinterface

class JTAG_reg #(int unsigned size = 32);

  virtual jtag_i jtag_if;
  logic [`JTAG_INSTR_WIDTH-1:0] instr;

  function new (virtual jtag_i j, logic [`JTAG_INSTR_WIDTH-1:0] i = 'h0);
    jtag_if = j;
    instr = i;
  endfunction

  ...

class adv_dbg_if_t;

  JTAG_reg #(.size(256)) jtag_cluster_dbg;
  virtual jtag_i jtag_if;

  function new (virtual jtag_i j);
    jtag_if = j;
    jtag_cluster_dbg = new(jtag_if, `JTAG_CLUSTER_DEBUG);
  endfunction
```

Здесь можем заметить, что при создании объекта класса adv_dbg_if_t, мы также создаем объект JTAG_reg. При этом изначально передаем в его поле instr значение, соответствующее коду команды DEBUG. В этом можно убедиться, сверившись с Таблицей **??**. Теперь необходимо передать инструкцию в TAP контроллер.

Предположим, что мы не знаем, в каком состоянии в текущий момент времени находится конечный автомат контроллера. Однако он спроектирован таким образом, что установка сигнала TMS в логическую единицу на протяжении по крайней мере пяти тактов TCK гарантирует его нахождение в состоянии RESET. Если затем опустить TMS в “0”, то можно утверждать, что текущее состояние конечного автомата – IDLE. Убедитесь в описанном выше по схеме конечного автомата. Зная же текущее состояние и ориентируясь на схему, можем сформировать на входах TCK и TMS необходимые логические уровни. Подобным образом и работают события jtag_softreset() и setIR():
[H]

```
task jtag_softreset();
    jtag_if.tms   <= 1'b1;
```

```
    jtag_if.trstn <= 1'b1;
    jtag_if.tdi    <= 1'b0;
    this.jtag_clock(5); //enter RST
    jtag_if.tms   <= 1'b0;
    this.jtag_clock(1); // back to IDLE
  endtask

  task setIR();
    this.jtag_goto_SHIFT_IR();
    this.jtag_shift_SHIFT_IR();
    this.idle();
  endtask

  task jtag_goto_SHIFT_IR();
    jtag_if.trstn <= 1'b1;
    jtag_if.tdi    <= 1'b0;
    // from IDLE to SHIFT_IR : tms sequence 1100
    jtag_if.tms   <= 1'b1;
    this.jtag_clock(2);
    jtag_if.tms   <= 1'b0;
    this.jtag_clock(2);
  endtask

   task jtag_shift_SHIFT_IR();
      jtag_if.trstn <= 1'b1;
      jtag_if.tms   <= 1'b0;
      for(int i=0; i < `JTAG_INSTR_WIDTH; i=i+1) begin
         if (i == `JTAG_INSTR_WIDTH-1)
              jtag_if.tms   <= 1'b1;
         jtag_if.tdi <= instr[i];
         this.jtag_clock(1);
      end
   endtask

   task idle();
      jtag_if.trstn <= 1'b1;
      // from SHIFT_DR to RUN_TEST : tms sequence 10
      jtag_if.tms   <= 1'b1;
      jtag_if.tdi    <= 1'b0;
      this.jtag_clock(1);
      jtag_if.tms   <= 1'b0;
      this.jtag_clock(1);
   endtask
```

В результате вызова событий на выходе TDO установлено значение из модуля отладки и началась запись данных в командный сдвиговый регистр внутри модуля.

Когда подготовка закончена, можем начинать полноценно работать с командами ADI. Сначала, во избежание ошибок в функционировании процессора и некорректной записи данных в память следует остановить выполнение инструкций процессором. Так как работа теперь ведется с модулем отладки, последовательно вдвигаем по линии TDI необходимые команды. Сначала

передаем команду о том, что выходной сигнал TDO устанавливается из подмодуля CPU. После используем команду записи в статусный регистр для установки в нем значения “1”. Пронаблюдать данную последовательность действий можно в событии cpu_stall:
[H]

```
`define ADV_DBG_AXI4_MODULE 6'b100000
`define ADV_DBG_CPU_MODULE  6'b100001

`define ADV_DBG_CPU_REG_STATUS 3'b000

  task cpu_stall(input logic [15:0] cpu_mask);
    logic [255:0] dataout;
    jtag_cluster_dbg.start_shift();
    jtag_cluster_dbg.shift_nbits(6, `ADV_DBG_CPU_MODULE, dataout);
    jtag_cluster_dbg.update_and_goto_shift();
    jtag_cluster_dbg.shift_nbits(24,{`ADV_DBG_CPU_WREG,
     ↪  `ADV_DBG_CPU_REG_STATUS, cpu_mask}, dataout);
    jtag_cluster_dbg.idle();
    $display("[adv_dbg_if] CPU STALL command.");
  endtask
```

После этих действий можем начать серийную запись в память через подмодуль AXI4. Аналогично предыдущим шагам переключаем источник линии TDO и начинаем вдвигать данные, считанные из DAT файлов по нужным адресам:
[H]

```
`define ADV_DBG_AXI4_WRITE32 5'h3

  task axi4_write32(input logic[31:0] addr, input int nwords, input logic
   ↪  [255:0][31:0] data);
    this.axi_write(`ADV_DBG_AXI4_WRITE32, addr, nwords, data);
  endtask

  local task axi_write(input [4:0] write_size, input logic[31:0] addr, input
   ↪  int nwords, input logic [255:0][31:0] data);
    logic [255:0] dataout;
    int bit_size = (write_size == `ADV_DBG_AXI4_WRITE8) ? 8 : (write_size ==
     ↪  `ADV_DBG_AXI4_WRITE16) ? 16 : (write_size == `ADV_DBG_AXI4_WRITE32) ?
     ↪  32 : 64;

    jtag_cluster_dbg.start_shift();
    jtag_cluster_dbg.shift_nbits(6, `ADV_DBG_AXI4_MODULE, dataout);
    jtag_cluster_dbg.update_and_goto_shift();
    jtag_cluster_dbg.shift_nbits(53,{write_size, addr, nwords[15:0]},
     ↪  dataout);
    jtag_cluster_dbg.update_and_goto_shift();
    jtag_cluster_dbg.shift_nbits_noex(bit_size + 1, {data[0], 1'b1}, dataout);
    for(int i=1; i<nwords; i++)
      jtag_cluster_dbg.shift_nbits_noex(bit_size, data[i], dataout);
```

```
19      jtag_cluster_dbg.shift_nbits(34, {2'b0, 32'h11111111}, dataout); // for
        ↪  now we completely ignore CRC
20      jtag_cluster_dbg.idle();
21      $display("[adv_dbg_if] AXI4 WRITE%d burst @%h for %d bytes.", bit_size,
        ↪  addr, nwords*4);
22    endtask
```

Для получения адресов, по которым производится запись можно обратиться к схеме адресации памяти на Рисунке 4.5.

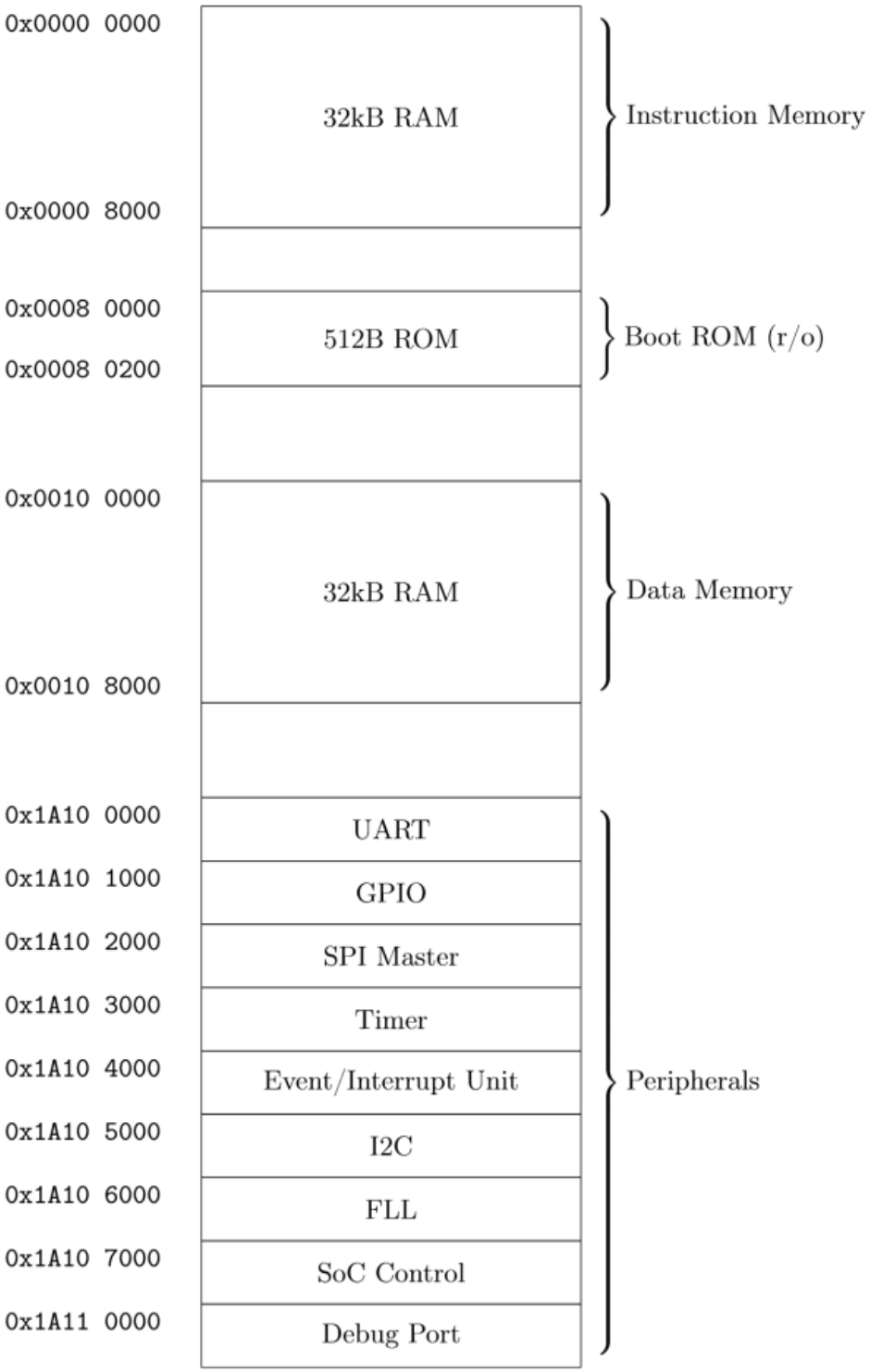

Рис. 4.5: Структура адресного пространства системы PULPino.

После завершения записи в память стоит вспомнить, что до останова процессор выполнял инструкции. Соответственно встречаемся с необходимостью перезаписывания счетчика команд (PC – program counter). Иначе после возобновления работы процессора исполнение инструкций начнется с последнего сохраненного адреса, что может привести к ошибкам и перезаписи данных в памяти. Однако ранее уже упоминалось, что существует возможность изменения счетчика при помощи записи в память. Воспользуемся этим и запишем с помощью подмодуля AXI4 требуемое значение PC по адресу 0x1A112000. Старшая часть адреса в данном случае

взята из схемы адресного пространства, а младшие 14 бит являются командой для внутреннего отладчика RI5CY.

Наконец, когда запись завершена, возобновляем работу процессора при помощи события cpu_reset():
[H]

```
task cpu_reset();
  logic [255:0] dataout;
  jtag_cluster_dbg.start_shift();
  jtag_cluster_dbg.shift_nbits(6, `ADV_DBG_CPU_MODULE, dataout);
  jtag_cluster_dbg.update_and_goto_shift();
  jtag_cluster_dbg.shift_nbits(24, {`ADV_DBG_CPU_WREG,
  ↪  `ADV_DBG_CPU_REG_STATUS, 16'b0}, dataout);
  jtag_cluster_dbg.idle();
  $display("[adv_dbg_if] CPU RESET command.");
endtask
```

В конце, повторим общую последовательность действий для достижения конечной цели – записи прошивки в память:

1. Записываем в регистр инструкций TAP контроллера инструкцию DEBUG
2. Переходим в состояние IDLE TAP контроллера
3. Переводим CPU в режим останова
4. Записываем данные в память инструкций
5. Записываем данные в память данных
6. Устанавливаем счетчик команд по адресу 0x80
7. Возобновляем работу CPU

4.7 Контрольные вопросы

1. Расскажите об основных периферийных блоках системы PULPino. Каковы их функции? Какие интерфейсы обмена данными с внешними устройствами поддерживает система?
2. Из каких функциональных элементов состоит интерфейс JTAG? Какие сигнальные линии он использует? Кратко поясните их назначение.
3. Что представляет из себя TAP контроллер? Какие функции он выполняет?
4. Приведите пример последовательности действий для чтения данных из регистра идентификации устройства через интерфейс JTAG.
5. Поясните структуру расширенного модуля отладки. Как происходит взаимодействие с ним через интерфейс JTAG?
6. Перечислите последовательность действий, необходимую для считывания содержимого из памяти данных. Как определить, какой объем данных необходимо считать?

Лабораторная работа 5

Интеграция контроллера в СнК Разработка программного драйвера

5.1 Введение

В рамках прошлых лабораторных работ было рассказано про такие понятия как архитектура микропроцессора, СнК, системная шина, показано как использовать Ассемблер, реализовывать вычислительные блоки, совершающие обмен данными по системной шине, писать Makefile. Также было рассказано про систему на кристалле PULPino (рис. 5.1), которую предлагается использовать для выполнения лабораторных работ.

В рамках данного курса будет рассмотрена та часть системы, которая совершает обмен данными по шине APB. Все периферийные устройства подключены к APB, за исключением контроллера SPI Slave, так как данный интерфейс может совершать обмен данными без участия процессорного ядра. Его целью является функционирование в качестве внешнего отладочного интерфейса, через который пользователь может получить доступ к внутренней памяти извне. Этот механизм может использоваться для предварительной загрузки программ в память, запуска системы, ожидания подтверждения завершения работы программы и проверки результатов.

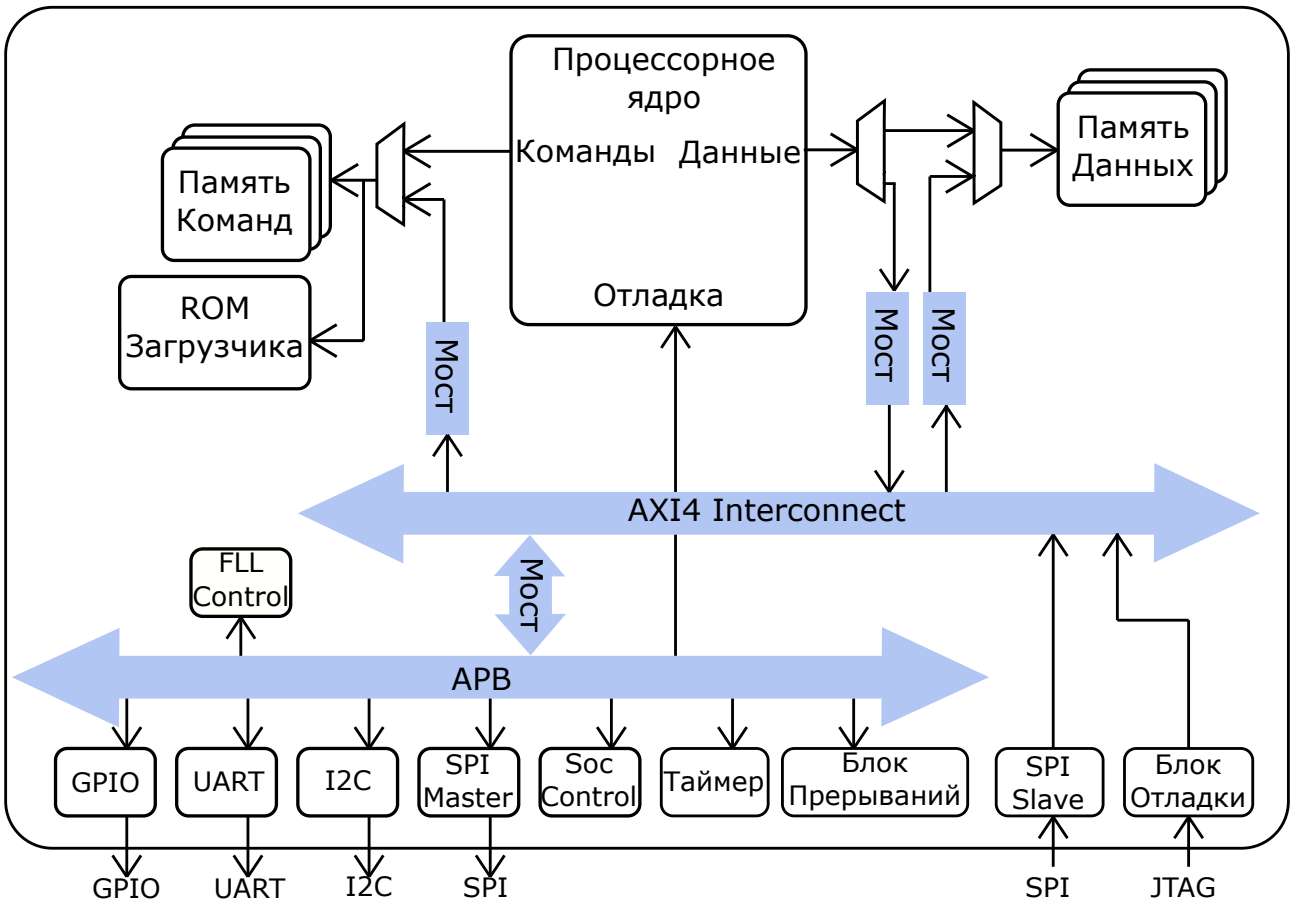

Рис. 5.1: Блок-схема системы на кристалле PULPino.

После того, как был написан аппаратный вычислитель CRC или любой другой контроллер встает вопрос: Каким образом управлять вычислительным блоком, и как связаны аппаратная и программная часть проекта? По ходу данной лабораторной работы будут даны ответы на эти важные вопросы, а также рассказано о том, как писать драйвера периферийных устройств, тестовое программное обеспечение и проводить моделирование работы системы с участием ПО.

5.2 Clock gating

Для подключения контроллера или вычислительного блока к системе PULPino необходимо разобраться с механизмом clock gating, который в ней используется. Clock gating — это методика отключения тактирования для определенного блока, когда он не нужен, и используется сегодня большинством конструкций SoC как эффективный метод для уменьшения энергопотребления.

Механизм используется для периферийных устройств, которые не будут использоваться постоянно. Вычислитель CRC как раз является таким из них, поэтому для экономии следует использовать этот механизм при подключении блока, а не подавать сигнал тактирования непосредственно на него.

Механизм clock gating реализован в архитектуре SoC и является частью функциональности RTL. Он останавливает тактирование отдельных блоков, когда эти блоки неактивны, эффективно отключая все функции этих блоков. Поскольку большая часть блоков логики в проекте может не переключаться в течение многих циклов, это значительно экономит энергопотребление. Самая простая и наиболее распространенная форма clock gating — это когда логическая функция “И” используется для выборочного отключения тактирования для отдельных блоков с помощью управляющего сигнала, как показано на рисунке 5.2.

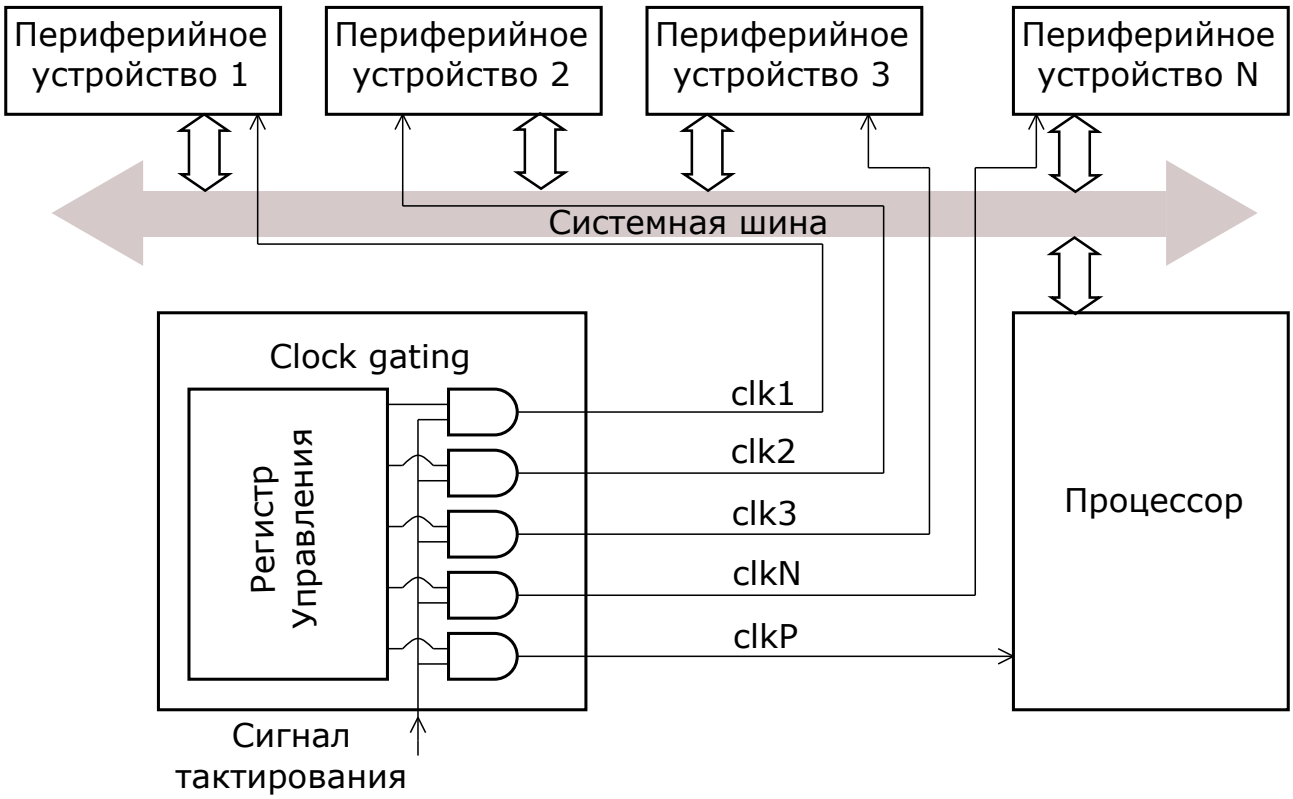

Рис. 5.2: RTL Clock gating.

Рассмотрим реализацию механизма clock gating в PULPino. В файле cluster_clock_gating.sv описан механизм работы: на вход модуля поступают сигналы тактирования и разрешения на тактирование, а на выход формируется сигнал тактирования для периферийных устройств. Фрагмент кода приведен в Листинге 5.1.

```
module cluster_clock_gating
(
    input  logic clk_i, //Входной сигнал тактирования
    input  logic en_i, //Разрешение на тактирование
    input  logic test_en_i,
    output logic clk_o //Выходной сигнал тактирования
);
`ifdef PULP_FPGA_EMUL //Если не используем механизм clock gating
// no clock gates in FPGA flow
assign clk_o = clk_i;
`else
logic clk_en;

always_latch
  begin
    if (clk_i == 1'b0)
      clk_en <= en_i | test_en_i;
  end
assign clk_o = clk_i & clk_en; //Если сигнал разрешения есть - на выход подаем
 ↪  сигнал тактирования
`endif
endmodule
```

Листинг кода 5.1: Реализация механизма clock gating в PUPLino.

Подключение модулей, реализующий механизм clock gating происходит в peripherals.sv. Подключается число экземпляров модуля clock gating равное количеству периферийных устройств. К каждому экземпляру подключается свой сигнал разрешения и на выход идет свой сигнал тактирования, который затем подключается к нужному периферийному устройству. Фрагмент кода приведен в Листинге 5.2.

```
generate
  genvar i;
    for (i = 0; i < APB_NUM_SLAVES; i = i + 1) begin //APB_NUM_SLAVES -
     ↪  количество периферийных устройств
      cluster_clock_gating core_clock_gate
      (
        .clk_o ( clk_int[i] ), //Выходной сигнал тактирования
        .en_i ( peripheral_clock_gate_ctrl[i] ),  //Разрешение на тактирование
        .test_en_i ( testmode_i ),
        .clk_i ( clk_i ) //Входной сигнал тактирования
      );
    end
endgenerate
```

Листинг кода 5.2: Подключение модулей, реализующих механизм clock gating.

Управление механизмом происходит аналогичным образом, как и управление периферий-

ными устройствами – при помощи ПО и системной шины в модуле apb_pulpino.sv. Фрагмент кода приведен в Листинге 5.3.

```
// Address offset: bit [4:2]
`define REG_PAD_MUX      4'b0000
`define REG_CLK_GATE     4'b0001
 ...
// register write logic
    always_comb
    begin
        ...
        clk_gate_n = clk_gate_q;
        ...
        if (PSEL && PENABLE && PWRITE)
        begin
            case (register_adr)
                `REG_PAD_MUX:
                    pad_mux_n     = PWDATA;
                `REG_CLK_GATE:
                    clk_gate_n    = PWDATA;
```

Листинг кода 5.3: Запись данных в регистры по системной шине для механизма clock gating.

Включение тактирования модулей происходит в функциях инициализации драйверов, которые будут рассмотрены в одном из следующих параграфов.

5.3 Подключение вычислительного блока CRC в проект

Для того чтобы подключить вычислитель CRC в проект необходимо выполнить ряд инструкций. Обратимся к рисунку 5.3, на котором показана схема организации адресного пространства системы на кристалле PULPino. Под периферийные устройства, находящиеся на шине APB выделено адресное пространство с адресами от 0x1A100000 до 0x1A11FFFF.

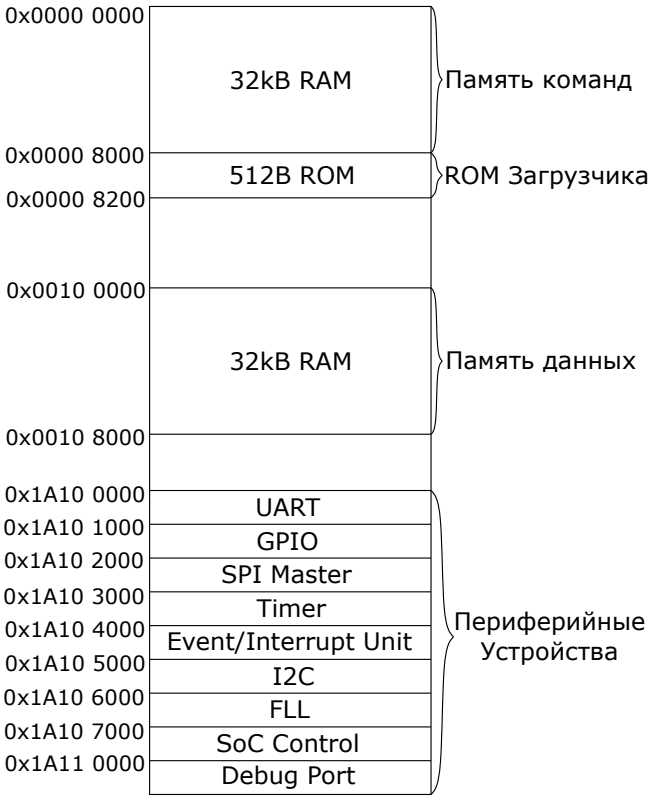

Рис. 5.3: Структурная схема адресного пространства PULPino.

Для подключения нового периферийного устройства в систему необходимо выделить ему адресное пространство – диапазон адресов, который принадлежит диапазону, выделенному для контроллеров на APB, а также не перекрывает адрес других контроллеров, подключенных к системной шине. Для этого в файле rtl/includes/apb_bus.sv нужно объявить константы на адреса начала и конца диапазона адресного пространства, выделяемого под данное периферийное устройство и увеличить на единицу константу количества периферийных устройств. Фрагмент кода приведен в Листинге 5.4.

```
1 // SOC PERIPHERALS APB BUS PARAMETRES
2 `define NB_MASTER  10
3 ...
4 // MASTER PORT TO CRC
5 `define CRC_START_ADDR        32'h1A10_8000
6 `define CRC_END_ADDR          32'h1A10_8FFF
7 ...
```

Листинг кода 5.4: define на количество периферийных устройств и define на адреса вычислителя CRC.

Обычно количество адресов диапазона равно числу регистров для записи и чтения данных контроллера. Однако можно брать диапазон больше, с запасом, для дальнейшего расширения функциональности блока вместе с сохранением программной совместимости. Далее необходимо объявить интерфейс APB для ведомого устройства – вычислителя CRC и объявить его адреса начала конца диапазона в файле rtl/periph_bus_wrap.sv. Фрагмент кода приведен в Листинге 5.5.

```
...
APB_BUS.Master    debug_master,
APB_BUS.Master    crc_master
...
`APB_ASSIGN_MASTER(s_masters[8], debug_master);
assign s_start_addr[8] = `DEBUG_START_ADDR;
assign s_end_addr[8]   = `DEBUG_END_ADDR;
`APB_ASSIGN_MASTER(s_masters[9], crc_master);
assign s_start_addr[9] = `CRC_START_ADDR;
assign s_end_addr[9]   = `CRC_END_ADDR;
...
```

Листинг кода 5.5: Объявление интерфейса APB для вычислителя CRC и его адресов начала и конца диапазона адресного пространства.

Следующий шаг - объявление экземпляра структуры интерфейса для CRC вычислителя и подключение самого вычислителя. Про механизм интерфейсов можно почитать в спецификации на язык SystemVerilog [1]. Также для механизма clock gating указать, что число периферийных устройств увеличилось на 1 в файле rtl/peripherals.sv. Фрагмент кода приведен в Листинге 5.6.

```
...
localparam APB_NUM_SLAVES  = 9;
...
 APB_BUS s_soc_ctrl_bus();
 APB_BUS s_debug_bus();
 APB_BUS s_crc_bus();
...
wrapper_crc8
wrapper_crc8_i
(
 .p_clk_i    ( clk_int[8]    ), //На вычислитель CRC подается тактирование с 9
  ↪  gate
 .p_rst_i    ( rst_n         ),
 .p_adr_i    ( s_crc_bus.paddr[31:0] ),
 .p_dat_i    ( s_crc_bus.pwdata      ),
 .p_we_i     ( s_crc_bus.pwrite      ),
 .p_sel_i    ( s_crc_bus.psel        ),
 .p_enable_i ( s_crc_bus.penable     ),
 .p_dat_o    ( s_crc_bus.prdata      ),
 .p_ready    ( s_crc_bus.pready      ),
 .p_slverr   ( s_crc_bus.pslverr     )
);
```

Листинг кода 5.6: Объявление экземпляра структуры интерфейса для CRC и подключение вычислителя.

5.4 Связь аппаратной части проекта и программного обеспечения

В данном параграфе описано, каким образом связаны аппаратные контроллеры и вычислительные блоки с программным обеспечением, которое исполняется процессором. После подключения нового блока в систему, необходимо чтобы ПО, которое управляет контроллером, знало об этом.

В файле pulpino/rtl/includes/apb_bus.sv прописаны адрес начала и конца диапазона адресов, выделенного каждому периферийному устройству, а также указано количество периферийных устройств на шине APB. Фрагмент кода приведен в Листинге 5.7.

```
1  // SOC PERIPHERALS APB BUS PARAMETRES
2  `define NB_MASTER 9 //Количество периферийных устройств
3
4  // MASTER PORT TO CVP
5  `define UART_START_ADDR 32'h1A10_0000 //Адрес начала диапазона
6  `define UART_END_ADDR 32'h1A10_0FFF //Адрес конца диапазона
7
8  // MASTER PORT TO GPIO
9  `define GPIO_START_ADDR 32'h1A10_1000
10 `define GPIO_END_ADDR 32'h1A10_1FFF
11
12 // MASTER PORT TO SPI MASTER
13 `define SPI_START_ADDR 32'h1A10_2000
14 `define SPI_END_ADDR 32'h1A10_2FFF
15
16 // MASTER PORT TO TIMER
17 `define TIMER_START_ADDR 32'h1A10_3000
```

Листинг кода 5.7: Объявление define на адреса начала и конца диапазонов для каждого периферийного устройства.

Кроме того базовые (начальные) адреса указаны в библиотечном файле системы PULPino pulpino/sw/libs/sys_lib/inc/pulpino.h . Фрагмент когда приведен в Листинге 5.8.

```
#define PULPINO_BASE_ADDR             0x10000000 //Базовый адрес системы

/** SOC PERIPHERALS */
#define SOC_PERIPHERALS_BASE_ADDR ( PULPINO_BASE_ADDR + 0xA100000 ) //Базовый
 ↪  адрес контроллеров на APB

#define UART_BASE_ADDR ( SOC_PERIPHERALS_BASE_ADDR + 0x0000 ) //Базовый адрес
 ↪  контроллера UART
#define GPIO_BASE_ADDR ( SOC_PERIPHERALS_BASE_ADDR + 0x1000 )
#define SPI_BASE_ADDR ( SOC_PERIPHERALS_BASE_ADDR + 0x2000 )
#define TIMER_BASE_ADDR ( SOC_PERIPHERALS_BASE_ADDR + 0x3000 )
#define EVENT_UNIT_BASE_ADDR ( SOC_PERIPHERALS_BASE_ADDR + 0x4000 )
#define I2C_BASE_ADDR ( SOC_PERIPHERALS_BASE_ADDR + 0x5000 )
#define FLL_BASE_ADDR ( SOC_PERIPHERALS_BASE_ADDR + 0x6000 )
#define SOC_CTRL_BASE_ADDR ( SOC_PERIPHERALS_BASE_ADDR + 0x7000 )
```

Листинг кода 5.8: define на базовые адреса системы, контроллеров.

Адреса смещений указаны в специальных библиотеках, в которых описаны функции для чтения и записи в регистры контроллеров, такие библиотека называются - драйверами. Фрагмент кода с описание адресов регистров контроллера UART приведен в Листинге 5.9.

```
#include "pulpino.h" //Подключение библиотечного файла pulpino.h
#include <stdint.h>
// UART_BASE_ADDR - базовый адрес UART, 0x00 -адрес смещения регистра
 ↪  UART_REG_RBR
#define UART_REG_RBR ( UART_BASE_ADDR + 0x00) // Receiver Buffer Register
 ↪  (Read Only)
#define UART_REG_DLL ( UART_BASE_ADDR + 0x00) // Divisor Latch (LS)
#define UART_REG_THR ( UART_BASE_ADDR + 0x00) // Transmitter Holding Register
 ↪  (Write Only)
#define UART_REG_DLM ( UART_BASE_ADDR + 0x04) // Divisor Latch (MS)
#define UART_REG_IER ( UART_BASE_ADDR + 0x04) // Interrupt Enable Register
#define UART_REG_IIR ( UART_BASE_ADDR + 0x08) // Interrupt Identity Register
 ↪  (Read Only)
#define UART_REG_FCR ( UART_BASE_ADDR + 0x08) // FIFO Control Register (Write
 ↪  Only)
#define UART_REG_LCR ( UART_BASE_ADDR + 0x0C) // Line Control Register
#define UART_REG_MCR ( UART_BASE_ADDR + 0x10) // MODEM Control Register
#define UART_REG_LSR ( UART_BASE_ADDR + 0x14) // Line Status Register
#define UART_REG_MSR ( UART_BASE_ADDR + 0x18) // MODEM Status Register
#define UART_REG_SCR ( UART_BASE_ADDR + 0x1C) // Scratch Register
```

Листинг кода 5.9: Адреса регистров контроллера UART.

Таким образом, при вызове функций драйвера происходит обмен данными с контроллером конкретного периферийного устройства по системной шине APB. На линии адреса будет выставлен адрес, находящийся в выделенном данному контроллеру адресном пространстве.

5.5 Реализация драйвера для вычислительного блока CRC

Драйвер – это программная библиотека, функции которой управляют контроллерами периферийных устройств. Существует несколько подходов к написанию драйверов. Предлагается использовать подход, основанный на структурах. Данных метод позволяет использовать один драйвер для множества одинаковых устройств (например, в системе есть несколько контроллеров интерфейса UART).

Первое, что необходимо сделать – добавить в заголовочный файл pulpino/sw/libs/sys_lib/inc/pulpino.h define на базовый адрес вычислителя CRC:

```
#define CRC_BASE_ADDR       ( SOC_PERIPHERALS_BASE_ADDR + 0x8000 ) //Базовый
↪  адрес вычислителя CRC
```

Написание заголовочного файла драйвера. В заголовочный файл необходимо подключить h-файл с базовыми адресами периферийных устройств:

```
#include <pulpino.h> //Подключение библиотечного файла pulpino.h
```

Далее приступим к описанию самой структуры в заголовочном файле драйвера. Структура содержит поля, соответствующие регистрам контроллера. В случае контроллера CRC из лабораторной работы 3, содержащего регистры данных, crc и статуса в структуре будет соответственно 3 поля. Фрагмент кода приведен в Листинге 5.10.

```
1 __attribute__ ((packed)) struct CRC_APB {
2     uint32_t CRC_REG_WRITE_DATA; //Поле данных
3     uint32_t CRC_REG_READ_CRC; //Поле CRC
4     uint32_t CRC_REG_READ_STATUS; //Поле статуса
5 };
```

Листинг кода 5.10: Структура для вычислителя CRC.

Также часто требуется прописывать define на константы, которые могут использоваться, например, в функциях драйвера. Использование таких констант удобно тем, что при изменении аппаратной части необходимо исправить только значения констант, объявленных в этом файле. Фрагмент кода приведен в Листинге 5.11.

```
1 #define CRC_STATUS_IDLE  0x00 //Состояние бездействия
2 #define CRC_STATUS_BUSY 0x01  //Состояние вычисления
3 #define CRC_STATUS_READ 0x02 //Состояние чтения CRC
```

Листинг кода 5.11: define на адреса регистров вычислителя CRC.

Далее прописываем заголовки функций драйвера. Функции драйвера делятся на 3 типа – функция инициализации структуры, функция чтения данных из регистра, функция записи данных в регистр. Фрагмент кода приведен в Листинге 5.12.

```
void crc_init(void); //Инициализация структуры
void crc_write_data(int data); //Запись данных
int crc_read_status(void); //Чтение регистра состояния
int crc_read_idle_status(void); //Чтение состояния бездействия
int crc_read_busy_status(void); //Чтение состояния вычисления
int crc_read_read_status(void); //Чтение состояния чтения
int crc_read_crc(void); //Чтения значения CRC
```

Листинг кода 5.12: Заголовки функций для вычислителя CRC.

Приступим к написанию с-файла - подключаем заголовочный файл:

```
#include <crc.h>
```

Объявляем экземпляр структуры CRC_APB:

```
volatile struct CRC_APB *crc8;
```

Функция инициализации структуры присваивает экземпляру структуры указатель на базовый адрес вычислителя CRC. Фрагмент кода приведен в Листинге 5.13.

```
void crc_init(){
    crc8 = (volatile struct CRC_APB *)CRC_BASE_ADDR;
}
```

Листинг кода 5.13: Функция инициализация структуры для вычислителя CRC.

В качестве примера функции записи данных в регистр в Листинге 5.14 приведена функция записи данных.

```
void crc_write_data(int data){
    crc8-> CRC_REG_WRITE_DATA = data;
}
```

Листинг кода 5.14: Функция записи данных для вычислителя CRC.

Функция чтения статуса приведена в Листинге 5.15.

```
int  crc_read_status(){
    return crc8->CRC_REG_READ_STATUS;
}
```

Листинг кода 5.15: Функция чтения статуса для вычислителя CRC.

Некоторые функции драйвера часто используют другие функции драйвера. Например, в данном случае это может быть функция чтения состояния “Бездействие”. Фрагмент кода приведен в Листинге 5.16.

```
int crc_read_idle_status(){
    volatile int status = crc_read_status();
    if(status == CRC_STATUS_IDLE) return 1;
    else return 0;
}
```

Листинг кода 5.16: Функция чтения состояния “Бездействие” для вычислителя CRC.

После того, как написаны оба файла драйвер можно считать завершенным и приступать к следующему действию – реализации тестового ПО и проверке результатов при помощи моделирования.

5.6 Проектирование тестового ПО. Проверка результатов на моделировании

Написание тестового программного обеспечения очень важный этап, при проверке вычислительного блока и драйвера. Такое ПО должно покрывать как можно больше случаев и сценариев, чтобы убедиться в том, что все работает верно, и не допустить возникновения ошибок в драйвере, ПО и вычислителе. Первым шагом, при написании тестового ПО является разработка алгоритма. Для вычислителя контрольной суммы из прошлой лабораторной работы предлагается следующий алгоритм:

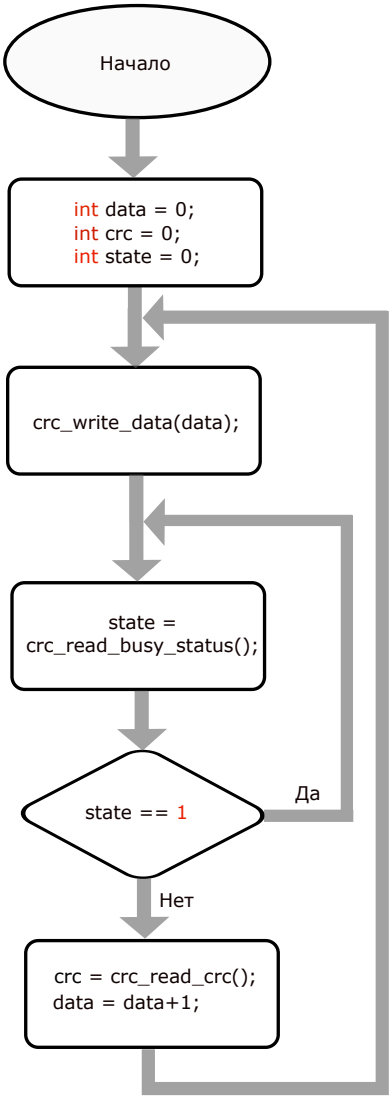

Рис. 5.4: Алгоритм проверки вычислительного блока CRC.

Алгоритм представляет собой вычисление контрольной суммы для одного байта данных – происходит запись одного байта с последующим считыванием результата. После получения результата вычисления контрольной суммы происходит запись следующего байта данных, на единицу большего предыдущего. Первый записываемый байт равен нулю. Таким образом, проверяются все возможные входные данные при вычисления CRC от одного байта. Кроме того, необходимо не забывать про механизм clock gating, используемый в системе. Помимо управление вычислительным блоком при помощи ПО, необходимо так же управлять механизмом clock gating. При подключении вычислителя в проект мы указывали, с какого gate будет подаваться тактирование на вычислитель. Для того, чтобы управлять механизмом следует указать номер этого gate в заголовочном файле pulpino/sw/libs/sys_lib/inc/pulpino.h :

```
#define CGCRC      0x08 //Константа для механизма clock gate вычислителя CRC
```

Управление clock gating происходит путем записи в регистр CGREG единиц в разряды, равные номерам модулей, который мы хотим тактировать. Фрагмент кода приведен в Листинге 5.17.

```
#include <crc.h>

void main(){
    CGREG |= (1 << CGCRC); //Разрешение на тактирование вычислителя clock gate
    ↪  - запись 1 в 8 разряд
    int data = 0;
    int crc  = 0;
    int state = 0;
    crc_init();//Инициализация структуры
    while(1){
        crc_write_data(int data);//Запись данных
        do{
            state = crc_read_busy_status();//Чтение статуса
        }while (state == 1);
        crc = crc_read_crc();//Чтение CRC8
        data = data + 1;
    }
}
```

Листинг кода 5.17: Тестовое ПО для проверки вычислителя CRC.

После того, как написано ПО нужно выполнить компиляцию и линковку. В системе PULPino управление сборкой происходит посредством Makefile. В Makefile указывается список используемых библиотек, архитектура используемого процессора, имя собираемого файла, путь до настроек компоновщика, имена выходных файлов и указания к сборке для определенных target. Подробнее про Makefile было рассказано в лабораторной работе 4. Когда dat-файлы с прошивкой скомпилированы можно приступать к моделированию вычислителя совместно с написанным ПО. Для этого необходимо добавить полученные файлы в проект и сымитировать загрузку по JTAG, как было описано в рамках лабораторной работы 5. Имитация транзакций по системной шине APB в таком моделирование не нужны, есть необходимость генерации сигналов сброса и тактирования. На рисунке 5.5 приведена временная диаграмма записи данных (32'h4), которая соответствует вызову функции crc_write_data(int data);

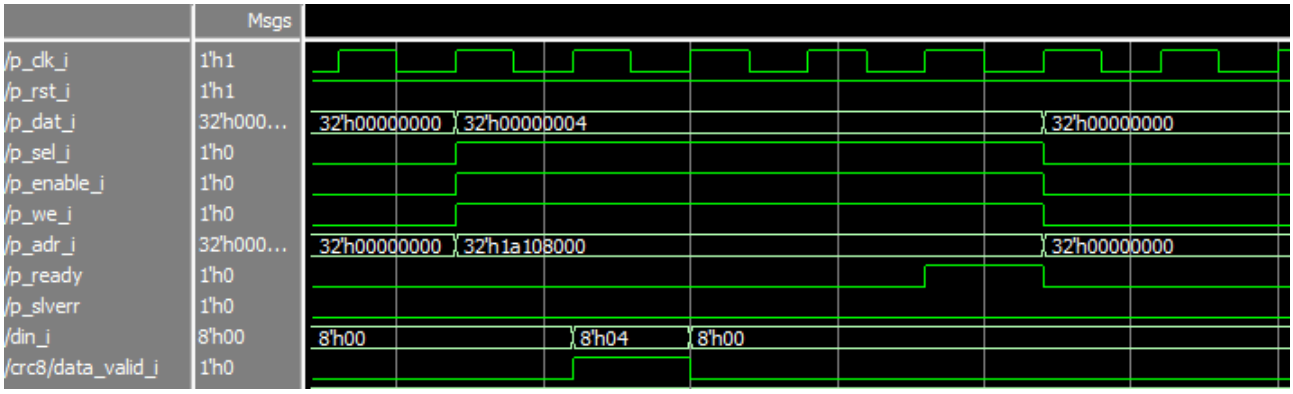

Рис. 5.5: Запись данных.

На рисунке 5.6 происходит считывание регистра состояния(32'h0), которое соответствует вызову функции crc_read_busy_status();

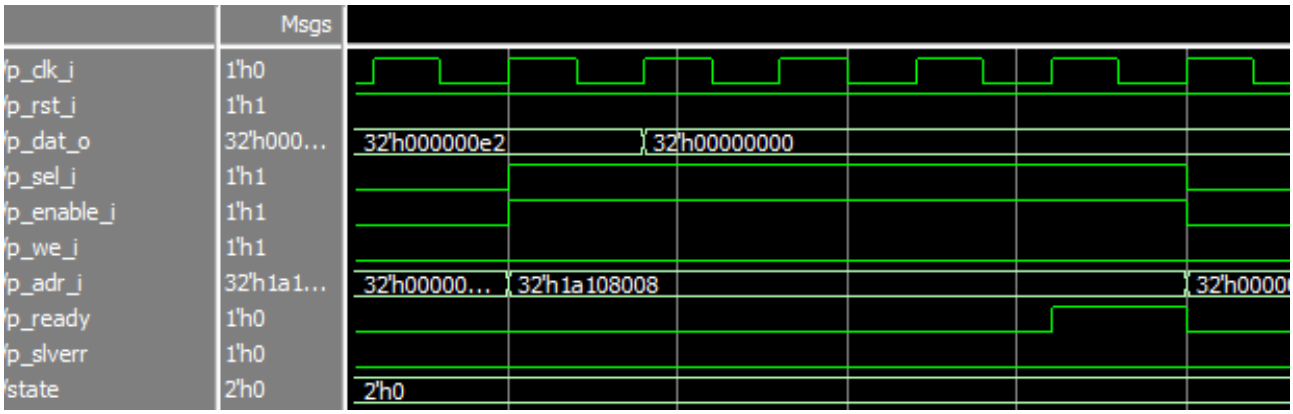

Рис. 5.6: Чтение состояния конечного автомата.

На рисунке 5.7 считываем результат контрольной суммы CRC8, которое соответствует вызову функции crc_read_crc();

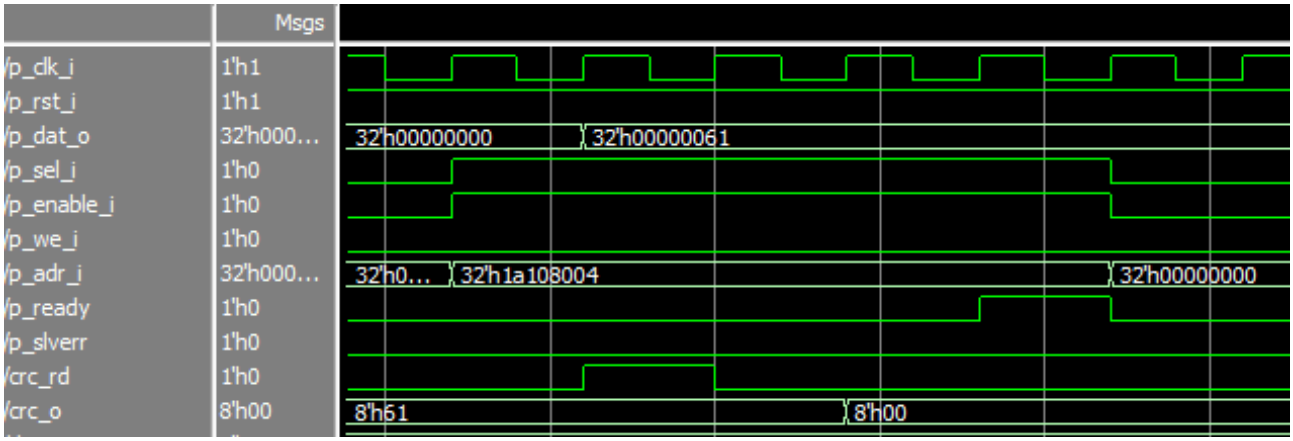

Рис. 5.7: Чтение значения CRC.

5.7 Контрольные вопросы

1. Что такое драйвер? Назначение драйверов.
2. Почему при написании драйвера удобно использовать структуры?
3. Механизм clock gating. Назначение.
4. Как реализован clock gating в систему PULPino?
5. Как связаны аппаратная и программная часть проекта?
6. Какие блоки подключаются к шине APB, а какие к AXI4?

5.8 Список литературы

1. Спецификация SystemVerilog. URL: `http://www.ece.uah.edu/~gaede/cpe526/2012%20System%20Verilog%20Language%20Reference%20Manual.pdf`

Лабораторная работа 6

Прерывания в системах на кристалле

6.1 Прерывания, типы прерываний

Наверняка вы уже сталкивались с прерываниями в других курсах, связанных с операционными системами или микроконтроллерами. Из них известно, что прерывания – не только удобный инструмент для программиста, но и необходимость для современных микропроцессорных систем. Тем не менее, полезно будет вспомнить и еще раз формализовать что такое прерывания, зачем они используются в цифровой аппаратуре и, главное, как их будем использовать мы в нашей системе на кристалле.

Прерыванием называется сигнал для процессора, указывающий на появление некоторого события и изменяющий последовательность выполнения инструкций процессором. К таким событиям могут относиться получение данных от внешнего устройства, возникновение ошибки при исполнении инструкции, запрос от программного обеспечения и другое.

В зависимости от того, что за событие произошло, прерывания разделяют по типам. В общем выделяют три типа прерываний: аппаратные, программные и исключения.

Часто прерывания связаны с работой периферийных устройств. При появлении новых данных на линии связи или при ошибках обмена может быть предусмотрен вызов аппаратных прерываний. Их также называют внешними, указывая на то, что источником события является внешнее по отношению к системе устройство. Говорят, что аппаратные прерывания являются асинхронными. Под этим подразумевается то, что появление прерывания не зависит от текущего состояния процессора или инструкции, которую он выполняет.

Исключения, наоборот, напрямую связаны с исполняемыми на данном этапе инструкциями. Исключения называют внутренними прерываниями, так как их инициатором служит сам процессор. Причиной для вызова являются ошибки исполнения программы. К таким ошибкам могут относиться вызов несуществующей инструкции, ошибки выполнения программы (например, деление на ноль), неверный запрос на доступ в память.

Программные прерывания часто рассматривают в связке с исключениями, однако у них есть свои отличительные особенности. В то время как их также можно отнести к синхронным и внутренним прерываниям, в отличии от исключений, они намеренно вызываются программным обеспечением. Наиболее распространено использование программных прерываний для совершения системных вызовов. В тех случаях, когда ПО необходимо использовать сервисы ядра (например, осуществить ввод-вывод данных, выделить память), производится системный вызов, передающий управление ядру операционной системы. В системе команд RISC-V для этого используются инструкции ECALL и EBREAK.

6.2 Механизм работы прерываний

После того как мы поняли, что представляют из себя прерывания, рассмотрим механизм их работы.

Когда прерывание возникает, процессор должен выполнить в ответ на него определенные инструкции или, иначе говоря, обработать прерывание. Требуемые инструкции уже лежат в системной памяти, причем для каждого источника прерывания они разные. Возникает вопрос: как процессор определяет, что именно является источником прерывания и как это прерывание обработать?

Здесь существует два основных способа для реализации системы оповещения о прерывании. Один из них основывается на опрашиваемых (polled) прерываниях. Если процессор, например, получает сигнал прерывания от устройства ввода-вывода, ему необходимо идентифицировать это устройство. Для этого он вызывает все доступные обработчики прерывания, пока не обнаружит тот из них, который подтвердит вызов прерывания. Такой подход редко оправдывает свою неэффективность, поэтому куда чаще используется реализация с векторными прерываниями.

Подход состоит в том, что на вход процессорного ядра поступает не только сам сигнал прерывания, “единица” или “ноль”, но и идентификатор прерывания. В зависимости от идентификатора при возникновении прерывания происходит переход по определенному адресу в памяти инструкций. Для того чтобы получить необходимый адрес, используется таблица векторов прерываний (IVT – Interrupt Vector Table), которая хранится там же, в памяти. В системе PULPino значения таблицы записаны начиная с адреса 0x0000005C, увидеть это можно из Таблицы 6.1.

Таблица 6.1: Адреса таблицы векторов прерываний системы PULPino.

Описание	**Адрес**
Зарезервированы – не используются	0x00000000 - 0x00000058
Разряд 23: Прерывание I2C	0x0000005C
Разряд 24: Прерывание UART	0x00000060
Разряд 25: Прерывание GPIO	0x00000064
Разряд 26: Прерывание SPI Master 0	0x00000068
Разряд 27: Прерывание SPI Master 1	0x0000006C
Разряд 28: Таймер A – переполнение	0x00000070
Разряд 29: Таймер A – окончание подсчета	0x00000074
Разряд 30: Таймер B – переполнение	0x00000078
Разряд 31: Таймер B – окончание подсчета	0x0000007C
Сигнал сброса	0x00000080
Исключение – некорректная инструкция	0x00000084
Инструкция ECALL – системный вызов	0x00000088
Некорректное обращение к памяти	0x0000008C

Стоит сказать, что существует два основных подхода к организации IVT. Первый, интуитивно понятный, заключается в том, что в таблице хранятся адреса обработчиков прерывания. Когда прерывание вызывается, содержимое нужной ячейки таблицы загружается в счетчик команд и таким образом происходит переход в другую область памяти. Второй подход предлагает хранить в ячейках не адреса, а инструкции. В случае системы команд RISC-V в них хранится инструкция JAL, выполняющая безусловный переход и сохраняющая адрес PC+4 в регистр для возврата после выполнения. Система PULPino использует второй подход.

В одной из предыдущих лабораторных работ мы уже рассматривали, как производится запись исполняемого файла в системную память. При этом в память инструкций мы записывали

содержимое секций .vectors и .text из ELF файла. Секция .vectors как раз содержит в себе данные для записи в таблицу векторов прерываний. Если открыть сгенерированный тогда DAT файл, можно обратить внимание, что первые его строки заполнены кодами инструкции NOP (no operation) – 0x00000013, и соответствуют первым 23 зарезервированным адресам в памяти. За ними и следуют интересующие нас инструкции безусловного перехода по адресам обработчиков прерываний.

Разберемся теперь, какие операции выполняет процессор, когда получает сигнал прерывания. В первую очередь ему необходимо сохранить содержимое регистров общего назначения, для возврата системы в положение, на котором возникло прерывание. При этом значения регистров x0-x2 сохранять не требуется, как и регистров, чье состояние гарантированно не изменится после возврата из подпрограммы (“non-volatile” регистры):

```
store_regs:
  // Сохранение значений из регистров общего назначения
  sw  x3, 0x00(x2)  // gp
  sw  x4, 0x04(x2)  // tp
  sw  x5, 0x08(x2)  // t0
  sw  x6, 0x0c(x2)  // t1
  sw  x7, 0x10(x2)  // t2
  sw x10, 0x14(x2)  // a0
  sw x11, 0x18(x2)  // a1
  sw x12, 0x1c(x2)  // a2
  sw x13, 0x20(x2)  // a3
  sw x14, 0x24(x2)  // a4
  sw x15, 0x28(x2)  // a5
  sw x16, 0x2c(x2)  // a6
  sw x17, 0x30(x2)  // a7
  sw x28, 0x34(x2)  // t3
  sw x29, 0x38(x2)  // t4
  sw x30, 0x3c(x2)  // t5
  sw x31, 0x40(x2)  // t6
  // Сохранение данных ``аппаратных циклов'' -- одной из особенностей RI5CY
  csrr x28, 0x7B0
  csrr x29, 0x7B1
  csrr x30, 0x7B2
  sw x28, 0x44(x2)  // lpstart[0]
  sw x29, 0x48(x2)  // lpend[0]
  sw x30, 0x4C(x2)  // lpcount[0]
  csrr x28, 0x7B4
  csrr x29, 0x7B5
  csrr x30, 0x7B6
  sw x28, 0x50(x2)  // lpstart[1]
  sw x29, 0x54(x2)  // lpend[1]
  sw x30, 0x58(x2)  // lpcount[1]
  // Возврат по адресу из x1
  jalr x0, x1
```

Теперь с помощью идентификатора прерывания и таблицы векторов прерываний, определяется адрес, с которого начинаются инструкции обработчика. После чего производится переход в

требуемую область памяти и выполнение программы. Когда исполнение инструкций обработчика завершается, процессор восстанавливает сохраненное состояние и продолжает работу с того места, на котором остановился перед вызовом. При этом восстановление состояния производится аналогично его сохранению через последовательную выгрузку значений регистров из стека. Отсюда можем оценить, насколько затратно по времени выполнять вызов прерываний. Для количественной оценки этого времени используют величину задержки прерывания (Interrupt latency), которая равна количеству тактов между появлением запроса на прерывание и началом исполнения первой инструкции из программы обработчика. Графическое представление определения задержки прерывания показано на Рисунке 6.1. Для примера, задержка прерывания в процессорных ядрах Cortex-M3 и Cortex-M4 составляет 12 тактов. Важно понимать, что применение прерываний не всегда является эффективным с точки зрения производительности, а их возникновение полезно уметь контролировать.

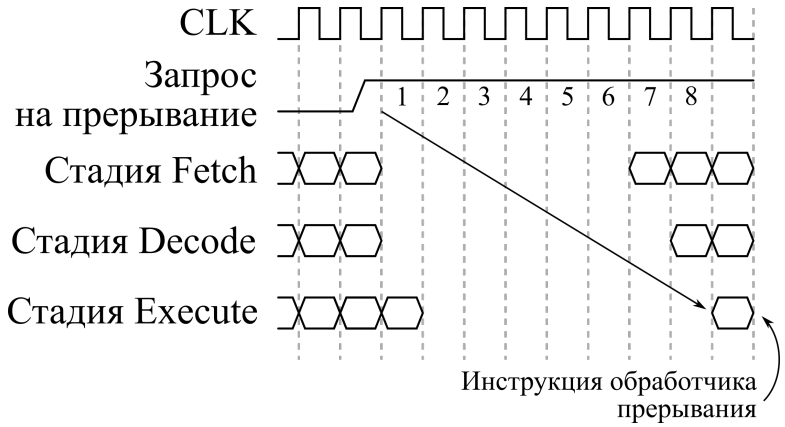

Рис. 6.1: Задержка прерывания после получение запроса.

В системе PULPino для генерации сигнала прерывания для процессора используется модуль управления событиями и прерываниями (apb_event_unit). Мы упоминали этот модуль ранее, когда говорили о структуре системы. Блок подключается к шине APB и может быть сконфигурирован через нее. На вход модуля, помимо прочего, поступают сигналы запроса прерывания от периферийных модулей, объединенные в 32-битную шину irq_i, как показано на Рисунке 6.2.

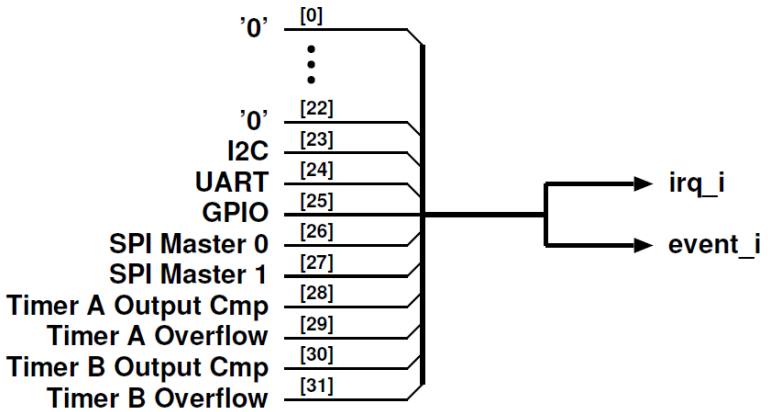

Рис. 6.2: Линии шины irq_i.

Одной из возможных конфигураций модуля является разрешение или запрет на вызов прерывания от конкретного источника. Иначе говоря, по шине APB может быть передана маска, которая накладывается на irq_i. В адресном пространстве к регистру маскирования можно обратиться по адресу 0x1A10_4000. После снятия сигнала сброса маска по умолчанию равна

нулю, из чего следует что ни одно прерывание от периферийных устройств не будет передано на процессор, пока маска не будет перезаписана. Помимо регистра маскирования полезными могут оказаться регистры вызова и снятия запросов на прерывание. Описание некоторых регистров модуля управления прерываниями приведено в Таблице 6.2. Перечисленные регистры имеют разрядность 32 бита, при этом каждый бит управляет соответствующей ему линией irq_i. Функции регистров IPR и ISR аналогичны друг другу.

Таблица 6.2: Регистры модуля apb_event_unit.

Регистр	**Адрес**	**Значение после сброса**
Маскирование прерываний (IER – Interrupt Enable)	0x1A10_4000	0x0000_0000
Запрос прерывания (IPR – Interrupt Pending)	0x1A10_4004	0x0000_0000
Установка запроса на прерывание (ISR – Interrupt Set Pending)	0x1A10_4008	0x0000_0000
Снятие запроса на прерывание (ICR – Interrupt Clear Pending)	0x1A10_400C	0x0000_0000

Также может возникнуть ситуация, при которой несколько периферийных прерываний поступят на модуль управления в течение одного тактового цикла. В таком случае выбор прерывания, которое будет обработано первым, зависит от его приоритета. Приоритет определяется позицией запроса прерывания на шине irq_i, где старшие биты являются наиболее приоритетными. То есть в соответствии с Рисунком 6.2, приоритет запроса от модуля SPI выше, чем от модуля I2C. Выходным сигналом модуля является 32-битный сигнал irq_o, содержащий либо все “нули”, если запросов на прерывание нет, либо одну “единицу” на позиции наиболее приоритетного незамаскированного прерывания. Пример формирования irq_o показан на Рисунке 6.3.

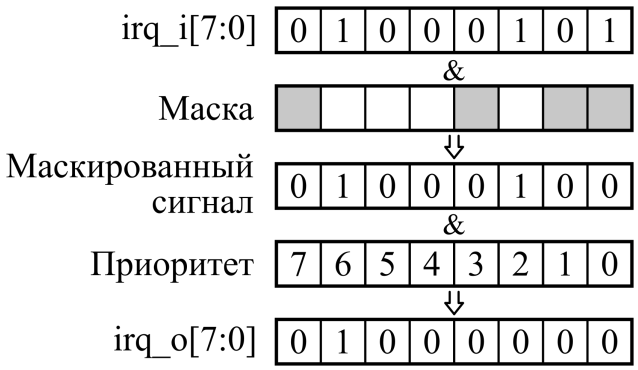

Рис. 6.3: Формирование 8-битного приоритетного запроса на прерывание.

6.3 Пользовательские прерывания

Теперь, когда мы имеем общее представление о том, как работают прерывания в системе PULPino, необходимо разобраться, как мы можем их применять. В данном случае нашей задачей является добавление в систему пользовательского прерывания и использование его в программном обеспечении.

Ранее вы уже подключали аппаратный блок контроллера CRC к шине APB в peripherals.sv. Если блок уже формирует на своем выходе сигнал запроса на прерывание, то подключить его к модулю apb_event_unit можно в том же файле. Обратим внимание на создание экземпляра apb_event_unit:

```
apb_event_unit
apb_event_unit_i
(
  ...

  .irq_i   ( {timer_irq, s_spim_event, s_gpio_event,
              s_uart_event, i2c_event, 23'b0} ),
  .event_i ( {timer_irq, s_spim_event, s_gpio_event,
              s_uart_event, i2c_event, 23'b0} ),
  .irq_o   ( irq_o),

  ...
);
```

Видно, что сигналы запросов на прерывания из разных модулей собираются в шину, изображенную на Рисунке 6.2. Добавив в нее выходной сигнал из контроллера CRC, можем обеспечить передачу в модуль своего низкоприоритетного запроса на прерывание. В текущей реализации PULPino шины irq_i и event_i дублируют друг друга, поэтому формируем запрос на обеих из них. Общая схема следования запроса на прерывание из контроллера CRC в процессор изображена на Рисунке 6.4.

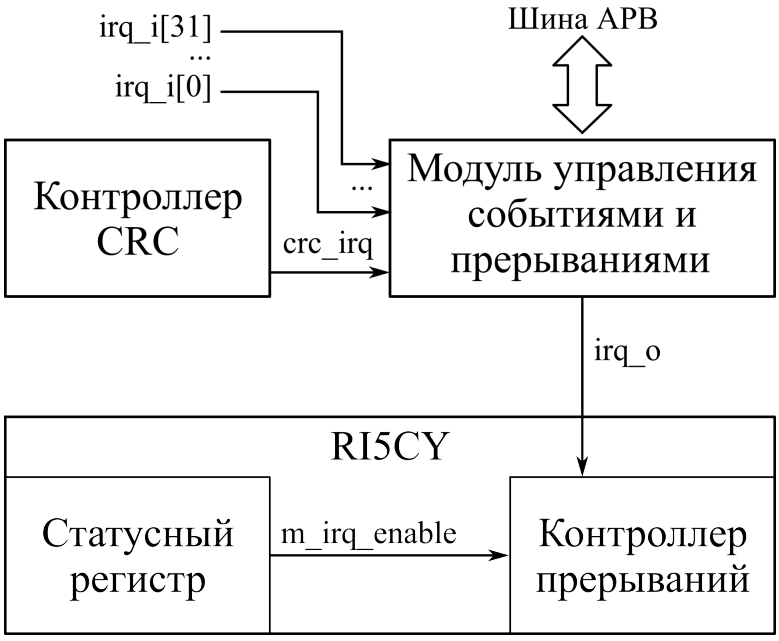

Рис. 6.4: Аппаратный запрос прерывания контроллером CRC.

Из рисунка выше видно, что в процессоре функционирует контроллер прерываний, принимающий выходной сигнал из apb_event_unit. При этом для его работы необходимо глобально разрешить обработку прерываний процессором. Реализуется это через запись в статусный регистр Machine Status (MSTATUS) ядра RI5CY. В программном обеспечении для этого используются следующие функции из pulpino_hwlib/include/int.h:

```
1 static inline void int_disable(void) {
2 #ifdef __riscv__
```

```
  // read-modify-write
  int mstatus;
  asm volatile ("csrr %0, mstatus": "=r" (mstatus));
  mstatus &= 0xFFFFFFF7;
  asm volatile ("csrw mstatus, %0" : /* no output */ : "r" (mstatus));
  asm("csrw 0x300, %0" : : "r" (0x0) );
#else
  mtspr(SPR_SR, mfspr(SPR_SR) & (~SPR_SR_IEE));
#endif
}

static inline void int_enable(void) {
#ifdef __riscv__
  // read-modify-write
  int mstatus;
  asm volatile ("csrr %0, mstatus": "=r" (mstatus));
  mstatus |= 0x08;
  asm volatile ("csrw mstatus, %0" : /* no output */ : "r" (mstatus));
#else
  mtspr(SPR_SR, mfspr(SPR_SR) | (SPR_SR_IEE));
#endif
}
```

Обе функции изменяют значение бита, ответственного за прерывания, в MSTATUS и соответственно устанавливают значение сигнала m_irq_enable.

Там же можно обратить внимание на объявление функций обработчиков прерываний, а в файле pulpino_hwlib/int.c добавить их описание. По аналогии с обработчиками других модулей объявим в int.h:

```
void ISR_CRC (void);    // 22: crc_controller
```

В int.c описывается содержание программы обработчика. Обратите внимание, что в конце выполнения программы необходимо снять прерывание в apb_event_unit, иначе оно снова будет вызвано по завершении работы обработчика:

```
// 22: crc_controller
__attribute__ ((weak))
void ISR_CRC (void){
  // ...
}
```

Помимо этого, требуется дополнить таблицу векторов адресом нашего обработчика прерываний. Как уже упоминалось, для этого нужно правильно скомпоновать секцию .vectors при компиляции. В таких применениях часто пишется платформенно зависимый код на языке ассемблера. Секция .vectors определяется как раз в таком виде в файле pulpino_hwlib/startup/crt0.riscv.S:

```
.section .vectors, "ax"
.option norvc;

// external interrupts are handled by the same callback
// until compiler supports IRQ routines
.org 0x00
.rept 22
nop                         // unused
.endr

jal x0, ISR_CRC_ASM         // 22: crc
jal x0, ISR_I2C_ASM         // 23: i2c
jal x0, ISR_UART_ASM        // 24: uart
jal x0, ISR_GPIO_ASM        // 25: gpio
jal x0, ISR_SPIM0_ASM       // 26: spim        end of transmission
jal x0, ISR_SPIM1_ASM       // 27: spim R/T finished
jal x0, ISR_TA_OVF_ASM      // 28: timer A overflow
jal x0, ISR_TA_CMP_ASM      // 29: timer A compare
jal x0, ISR_TB_OVF_ASM      // 30: timer B overflow
jal x0, ISR_TB_CMP_ASM      // 31: timer B compare


// reset vector
.org 0x80
jal x0, reset_handler

// illegal instruction exception
.org 0x84
jal x0, illegal_insn_handler

// ecall handler
.org 0x88
jal x0, ecall_insn_handler
```

В этом же файле описываем содержимое ISR_CRC_ASM для управления состоянием регистров общего назначения и использования ранее написанных функций:

```
ISR_CRC_ASM:
  addi x2, x2, -EXCEPTION_STACK_SIZE
  sw x1, 0x5C(x2)
  jal x1, store_regs
  la x1, end_except
  jal x0, ISR_CRC
```

6.4 Контрольные вопросы

1. Перечислите типы прерываний в зависимости от их источника. Какие функции они могут выполнять в системе?

2. Какие способы определения источника прерывания вы знаете? Что представляет из себя таблица векторов прерываний?

3. Какое содержимое может иметь таблица векторов прерываний? Как она формируется в системе PULPino?

4. Опишите общую последовательность действий, выполняемых процессором при фиксировании сигнала прерывания.

5. Периферийный контроллер формирует запрос на прерывание, однако спустя десять тактов процессор не приступает к выполнению программы обработчика. Какие у этого поведения могут быть причины?

Лабораторная работа 7

Загрузчик ПО во встраиваемых системах

При включении электронного устройства процессор СнК начинает выполнение программы, начиная с инструкции, расположенной по первому адресу. Основная пользовательская программа начинается с первой страницы памяти. Учитывая тот факт, что версия основной программы может обновляться множество раз в процессе тестирования устройств, то обновление ее с помощью специального программатора (например JTAG-программатора) зачастую является не оптимальной по времени задачей. Более того, в случае появления новой версии ПО для устройств гражданского назначения многомилионного тиража обновление с помощью такого стороннего программатора является, вообще говоря, невозможным. Введу вышеуказанных причин практически любое микропроцессорное электронное устройство оснащено специальным программным приложением – загрузчиком ПО (в англоязычной литературе boot-loader).

Загрузчик ПО – это специальная программа, которая располагается в памяти микропроцессора и может самостоятельно перепрограммировать его (новая версия может быть загружена по стандартному интерфейсу (UART, USB и т. д.) через компьютер). Таким образом при использовании загрузчика основная программа (ПО приложения) записывается уже не с первого адреса, а располагается начиная, например, с адреса 0x0000A000. А область памяти (0x00000000 – 0x0000A000) целиком отдается загрузчику.

У загрузчика ПО есть две основные функции: первая — это обновление ПО (запись новой версии в ПЗУ), вторая – загрузка ПО приложения из ПЗУ в ОЗУ для последующего выполнения.

В системе, упомянутой во всех предыдущих лабораторных работах для загрузчика так же выделен отдельный блок памяти, который на рисунке 7.1 обозначен как Boot ROM. Для хранения ПО приложения подразумевается использование внешней микросхемы flash-памяти с интерфейсом SPI/QPI (для чего в систему встроен контроллер интерфейса). А для загрузки новых версий – использование интерфейса UART.

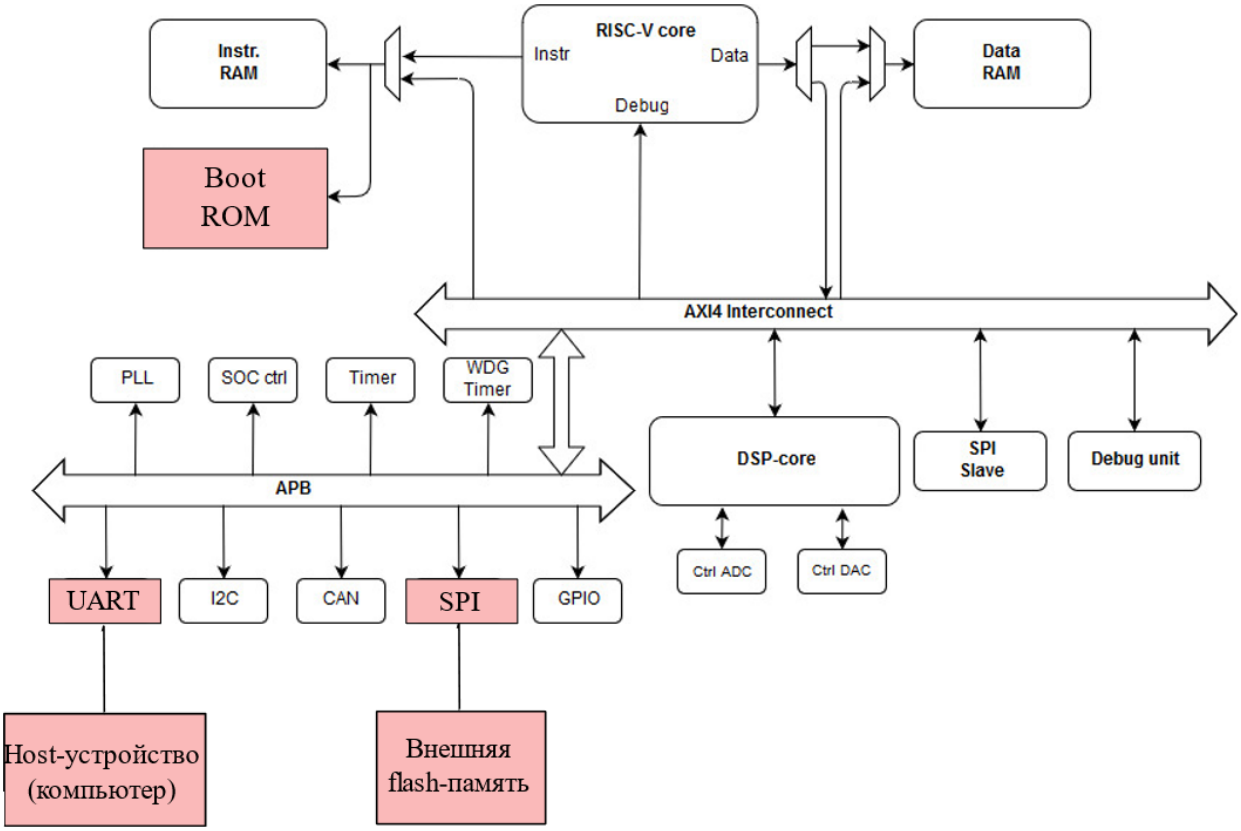

Рис. 7.1: Структурная схема СнК с подключенной flash-памятью.

В зависимости от конкретного изделия реализация загрузчика ПО может быть абсолютно разной (требования к интерфейсу загрузки, интерфейс используемой для хранения ПО приложения флэш-памяти, команды для управления ею и т. д.). Однако в целом принцип работы системы с загрузчиком является стандартным и делится на 3 блока: блок ветвления и инициализации периферийных устройств, ПО приложения и загрузчика (рис. 7.2).

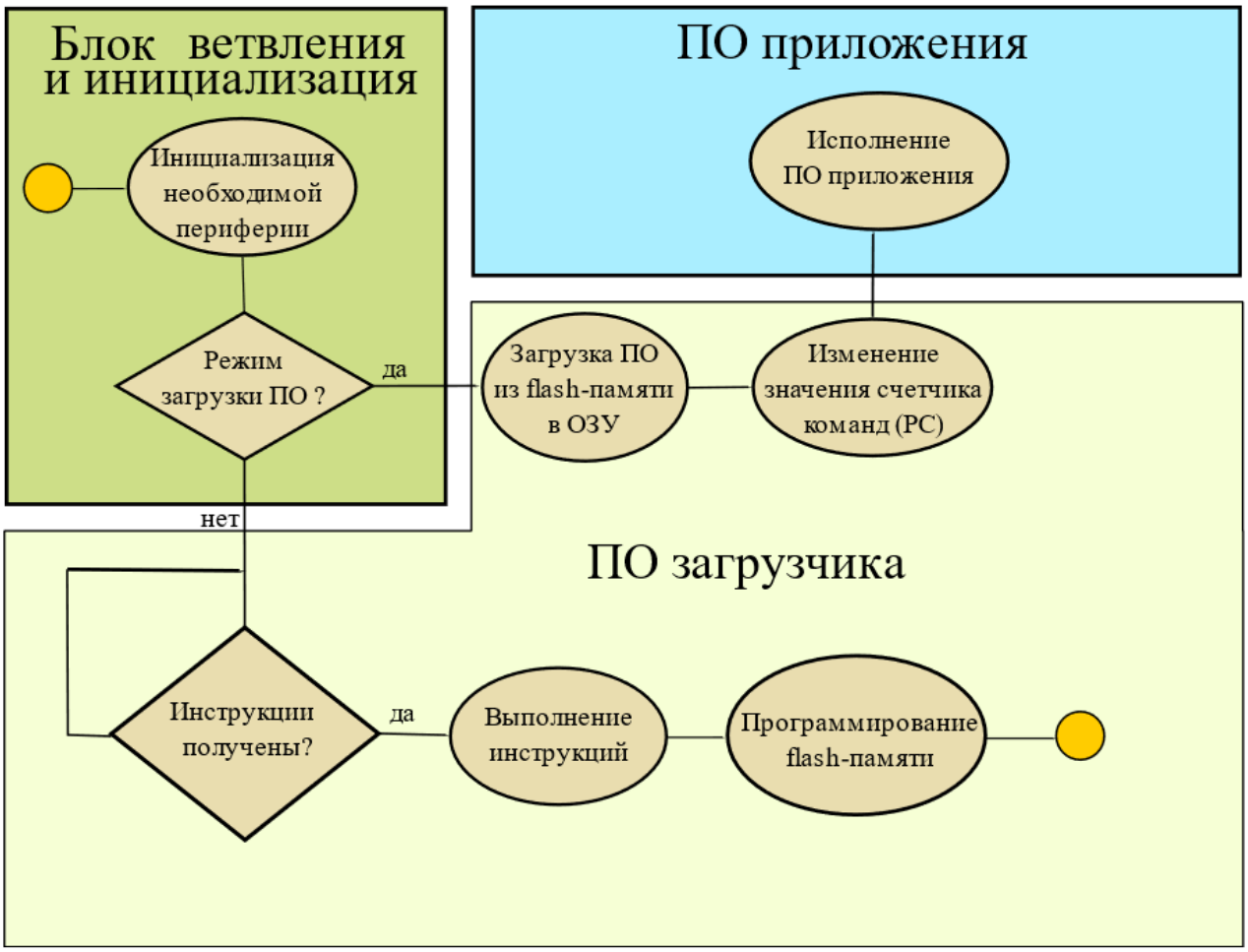

Рис. 7.2: Структурная схема работы системы с загрузчиком.

На начальном этапе после включения устройства определяется, необходимо ли произвести загрузку новой версии ПО или же выгрузить из флэш-памяти текущую и начать ее выполнение. Обычно в простых системах режим работы задается посредством проверки вывода GPIO.

Код приложения начинает выполняться только после того, как на 1 этапе было принято решение, что ПО не нуждается в обновлении и текущая версия была загружена из flash-памяти. ПО приложения должно быть спроектировано, с учетом того, что даже во время выполнения процессор может получить запрос на обновление. Когда приложение получает такой запрос, выполняется сброс системы.

Если на этапе ветвления был подтвержден запрос на обновление ПО, то процессор начинает исполнять часть кода загрузчика, отвечающую за обновлении версий. Как и в любом другом приложении, одной из первых задач загрузчика является инициализация минимального количество периферийных устройств, необходимых для выполнения функций загрузки (обычно это системный таймер, контроллеры интерфейсов для взаимодействия с flash-памятью, источником загрузки новой версии ПО и т. д.). Лучше ограничить использование периферийных устройств как можно меньшим количеством, чтобы попытаться максимально увеличить объем памяти, доступный для кода и данных приложения.

В большинстве случаев процесс загрузки новой версии не происходит автоматически, вместо этого загрузчик находится в состоянии ожидания и ждет инструкций от внешнего источника, которым обычно является программное приложение на host (компьютере).

Очевидно, что в различных реализациях загрузчика количество поддерживаемых им операций будет различным в зависимости от специфики поставленной задачи, однако, существует ряд операций, которые должны быть реализованы в любом загрузчике ПО:

- Стирание чипа флэш-памяти для удаления образа ПО устаревшего приложения;
- Запись по флэш-память для загрузки образа ПО новой версии приложения;
- Перезагрузка (программный сброс) для возможности выгрузки из флэш-памяти и исполнения новой версии приложения.

При проектировании более сложных загрузчиков могут быть реализованы и другие функции, например такие как расчет и проверка контрольной суммы для мониторинга целостности пакетов данных.

7.1 Загрузчик ПО с внешним интерфейсом UART

Рассмотрим пример простейшего загрузчика ПО для встраиваемых систем с внешним интерфейсом UART. В качестве хранилища ПО приложения для конкретики примера возьмем микросхему флэш-памяти IS25WP016D [1] вендора ISSI объемом 16Мбит с интерфейсом SPI/QPI. Команды и инструкции должны храниться отдельно, как в памяти процессора, так и во флэш-памяти. ПО загрузчика делится на две части (режимы): загрузка приложения по внешнему интерфейсу через процессор во флэш-память (program part) и выгрузку текущей версии из флэш-памяти с последующим запуском (boot part). Рассмотрим более детально program part. Program part загрузчика включает в себя следующее: начальную инициализацию минимального количества периферийных устройств, необходимых для загрузки ПО приложения, загрузка ПО по интерфейсу UART с последующей загрузкой во внешнюю флэш-память.

7.1.1 Инициализация и блок ветвления

Как было упомянуто ранее, изначально необходимо инициализировать все используемые загрузчиком периферийные контроллеры, а также саму периферию, для данной СнК это контроллеры SPI/QPI, UART и GPIO и внешняя flash-память. Приняв во внимание, что при включении устройства СнК не может взаимодействовать с flash-памятью до тех пор, пока напряжение на цепи Vcc (цепи питания) не достигнет нужного уровня, то необходимо выдержать временную задержку после подачи питания на печатную плату, прежде чем проводить инициализацию. Для этой цели могут быть использованы функции драйвера системного таймера, или же ассемблерная вставка (подробнее ассемблерные вставки будут рассмотрены в следующей главе лабораторной работы), выполняющая необходимое количество nop инструкций (листинг 7.1).

```
1 // Время задержки     = 300мкс
2 // Системная частота = 25МГц
3 for (int i = 0; i < 7500; i++) {
4     __asm__ volatile("nop");
5   }
```

Листинг кода 7.1: .

После того, как время задержки было выдержанно, необходимо выполнить инициализацию минимального количества периферийных модулей. Для данной реализации это контроллеры интерфейсов UART, SPI/QPI и контроллер портов ввода-вывода GPIO. Для инициализации и настройки необходимо сделать запись в определенные регистры соответствующего контроллера.

Для инициализации контроллера UART ему необходимо сообщить информацию о формате кадра (бите честности, количестве стоповых бит), делителе частоты для настройки скорости передачи данных по интерфейсу, разрешенный размер внутреннего FIFO контроллера, а так же при необходимости разрешить обработку событий, вызывающих прерывания (например, ошибка в кадре UART).

Контроллером интерфейса SPI в свою очередь должна быть получена информация о количестве ведомых устройств, подключенных к SPI-мастеру (контроллеру), а также как и для контроллера UART значение делителя синхросигнала и опционально разрешения на обработку прерываний.

Порты ввода-вывода будут использоваться в блоке ветвления (рис. 7.1) для выбора последующего режима работы. Таким образом, понадобится два входных порта: один для выбора режима (program part – загрузка новой версии или boot part – запуск существующей версии), второй для подтверждения выбора и соответственно перехода в один из двух режимов. Так как в данной системе для гибкости работы вывод микросхемы может быть сконфигурирован, как порт ввода-вывода, так и как сигнальный вывод какого-либо интерфейса, то необходимо определить функцию вывода (в данном случае GPIO). Также оба вывода определяются как входные порты микросхемы.

```
// Инициализация UART контроллера
uart_set_line_cfg(PARITY_CNT, PARITY_EN, PARITY_EVEN, STOP_NUMBER,
 ↪  BIT_NUMBER);
uart_set_counter(UART_CLK_CNT);
uart_set_fifo_en(FIFO64_EN, TRIG_CODE);

// Инициализация SPI контроллера
spi_set_clk_div(SPI_CLK_DIV);

// Инициализация портов GPIO
set_pin_function(BOOT_PIN, FUNC_GPIO);
set_gpio_pin_direction(BOOT_PIN, DIR_IN);
mode = get_gpio_pin_value(BOOT_PIN);
```

Листинг кода 7.2: .

Вслед за инициализацией контроллеров периферийных устройств, необходимо произвести настройку внешней SPI-flash. В данной системе – это микросхема flash-памяти IS25WP016D. Как и контроллеры, микросхема обладает рядом управляющих регистров, диктующих режимы и правила ее работы. По умолчанию микросхема работает как SPI-слейв устройство. Чтобы перейти в QPI-режим необходимо выполнять ряд следующих транзакций.

Для разрешения работы микросхемы в QPI-режиме необходимо произвести запись в соответствующее поле статус-регистра с помощью инструкции Write Status Register – WRSR (стоит отметить, что запись в статус-регистр, как и ряд других инструкций, требует предварительного разрешения – команда Write Enable (WREN)). Для отслеживания состояния готовности микросхемы к следующим операциям используется инструкция Read Status Register (RDSR). Инструкция RDSR обеспечивает доступ к регистру состояния. Во время выполнения операции записи или стирания, команда RDSR может использоваться для проверки хода выполнения или завершения операции путем считывания бита Write in Progress (WIP) статус-регистра.

После того, как переход в QPI-режим стал разрешенным нужно выполнить инструкцию Enter Quad Peripheral Interface (QPIEN), включающую flash-устройство для работы с шиной

QPI (рис. 7.3). После ее выполнения все последующие инструкции будут работать с 4-битными мультиплексированными входами / выходами до тех пор, пока на устройство не будет отправлена команда выключения.

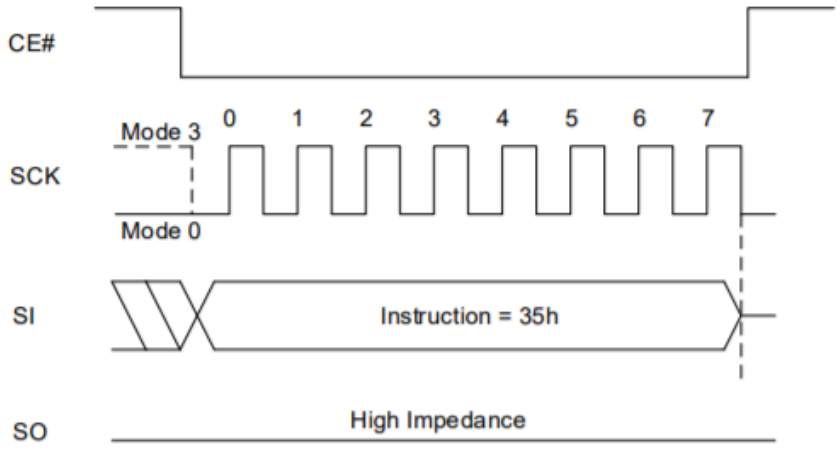

Рис. 7.3: Транзакция QPIEN.

Стоит отметить, что для совершения любой транзакции по шине SPI контроллеру необходимо сообщить команду, адрес и данные, их длинны, а также количество так называемых dummy cycles (циклы задержки между выставлением адреса и данных на шину).

```
// Разрешение QPI
atomic_spi_wr(WREN,SPI_CMD_WR);

spi_setup_cmd_addr(WRSR, 8, 0x42, 8);                // Запись команды и адреса
 ↪  и их длин в битах
spi_start_transaction(SPI_CMD_WR, SPI_CSN0);         // Начало транзакции
 ↪  записи в SPI-режиме
while ((spi_get_status() & 0x3F) != 1);              // Ожидание окончания
 ↪  транзакции

// Ожидание окончания записи -
// Отслеживание значение бита WIP в статус-регистре
do{
  spi_setup_cmd_addr(RDSR, 8, 0, 0);
  spi_set_datalen(8);
  spi_start_transaction(SPI_CMD_RD, SPI_CSN0);
  while ((spi_get_status() & 0x3F) != 1);
  spi_read_fifo(status_reg, 8);
}while (((*status_reg) & 0x01) == 1);

atomic_spi_wr(QPIEN, SPI_CMD_WR);

// Блок ветвления
while (1) {
  mode = get_gpio_pin_value(BOOT_PIN);
  switch (mode) {
    case PROG_MODE : program_mode();
      break;
    case BOOT_MODE : boot_mode();
      break;
    default        : uart_sendchar(0x01);
      break;
  }
}
```

Листинг кода 7.3: .

7.1.2 ПО загрузчика Program part

Прежде чем выполнять загрузку новой версии, необходимо отчистить флэш-память от старой. Инструкция Chip Erase (CER) стирает весь массив памяти (рис. 7.4). Однако перед выполнением инструкции CER (также как и WRSR) необходимо установить Write Enable Latch (WEL). WEL автоматически сбрасывается после завершения операции удаления данных с чипа. Для отслеживания состояния также используется инструкция RDSR (листинг 7.4).

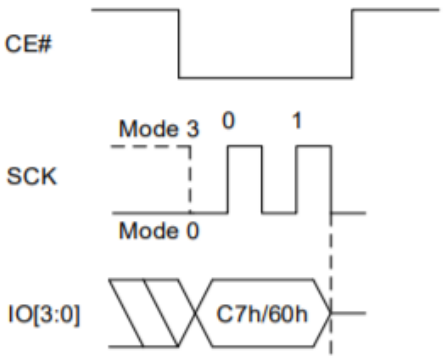

Рис. 7.4: Транзакция CER.

```
// Стирание чипа
// Отправить команду write enable
atomic_spi_wr(WREN,SPI_CMD_QWR);

// Отправить команду chip erase
atomic_spi_wr(CER, SPI_CMD_QWR);

// Ожидание завершения процесса стирания чипа
uint8_t *status_reg;
do {
  spi_setup_cmd_addr(RDSR, 8, 0, 0);
  spi_set_datalen(8);
  spi_start_transaction(SPI_CMD_QRD, SPI_CSN0);
  spi_read_fifo(status_reg, 8);
} while ((*status_reg & 0x01) == 1);
```

Листинг кода 7.4: .

Для загрузки ПО приложения с компьютера используется интерфейс UART. Для согласованности передачи данных используется механизм обмена контрольными сообщениями между компьютером и процессором. Передача массива данных и инструкций предваряется получением процессором информации о размерах в байтах обоих массивов соответственно. Размеры обоих массивов являются первыми единицами информации, которые будут записаны во флэш-память с помощью инструкции Page Program (PP). Однако, учитывая тот факт, что запись данных во флэш-память с помощью PP ограничена величиной 256 байт, то дополнительно вычисляется контрольная информации о количестве будущих транзакций записи, а также о количестве секторов, которые будут заняты данными и инструкциями соответственно (каждый сектор размером по 4 Кбайта).

```
uint32_t instr_addr    = INSTR_START_ADDR;
uint32_t instr_size    = 0x0; //Размер массива инструкций в байтах
uint32_t instr_nmb     = 0x0; // Количество итераций PP для записи всего
 ↪  массива инструкций
uint32_t instr_sector  = 0x0; // Количество секторов для массива инструкций

uint32_t data_addr     = DATA_START_ADDR;
uint32_t data_size     = 0x0; // Размер массива данных в байтах
uint32_t data_nmb      = 0x0; // Количество итераций PP для записи всего
 ↪  массива данных
uint32_t data_sector   = 0x0; // Количество секторов для массива данных

// Получение контрольной информации
instr_size   = load_control( 0xA0,   instr_size );
instr_nmb    = load_control( 0xA1,    instr_nmb );
instr_sector = load_control( 0xA2, instr_sector );
data_size    = load_control( 0xA3,    data_size );
data_nmb     = load_control( 0xA4,     data_nmb );
data_sector  = load_control( 0xA5,  data_sector );

// Запись контрольной информации об инструкциях и данных во флэш-память
atomic_spi_wr(WREN,SPI_CMD_QWR);

spi_setup_cmd_addr(PP, 8, (0x000000 & 0xFFFFFF), 24);
spi_set_datalen(6 * 32);
spi_write_fifo(&instr_addr,       32);
spi_write_fifo(&instr_sector,     32);
spi_write_fifo(&instr_size,       32);
spi_write_fifo(&data_addr,        32);
spi_write_fifo(&data_sector,      32);
spi_write_fifo(&data_size,        32);
while ((spi_get_status() & 0xF000000) != 6<<24);
spi_start_transaction(SPI_CMD_QWR, SPI_CSN0);
while ((spi_get_status() & 0x3F) != 1);

// Ожидание завершения процесса записи контрольной информации
do {
  spi_setup_cmd_addr(RDSR, 8, 0, 0);
  spi_set_datalen(8);
  spi_start_transaction(SPI_CMD_QRD, SPI_CSN0);
  spi_read_fifo(status_reg, 8);
} while ((*status_reg & 0x01) == 1);
```

Листинг кода 7.5: .

Перед загрузкой данных и инструкций новой версии ПО необходимо разделить все адресное пространство флэш-памяти на 3 части: часть для хранения контрольной информации (обычно это младшие адреса флэш как в примере выше), часть для хранения данных и для хранения инструкций. Занимающая 24 байта, контрольная информация располагается в первых 24 ячей-

ках флэш-памяти соответственно. Было принято решение разделить оставшееся пространство памяти на 2 части следующим образом:

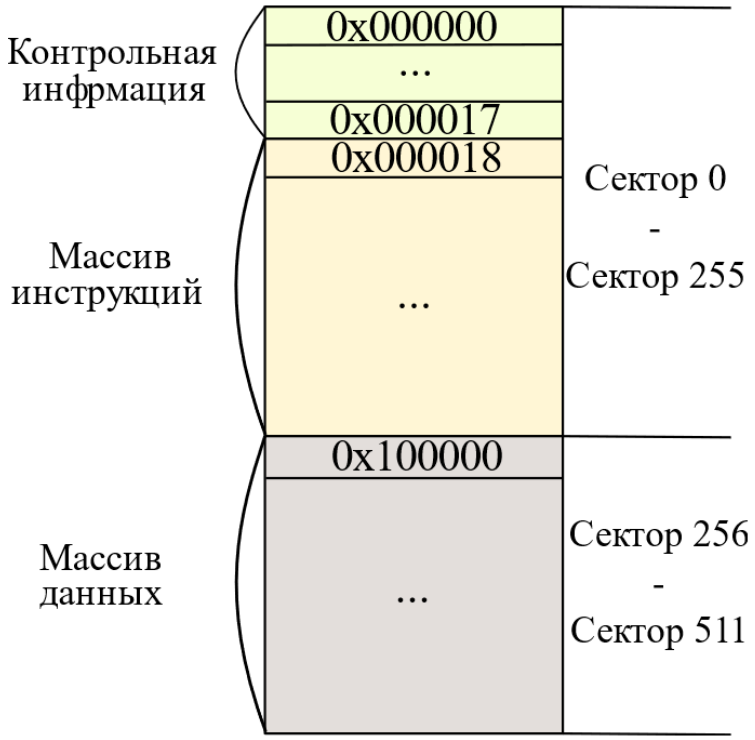

Рис. 7.5: Фрагментация флэш-памяти.

Последним этапом program part является чтение по UART, а затем запись инструкций и данных во флэш-память пакетами по 256 байт данных в каждом с помощью инструкции PP. Данные транзакции идентичны транзакциям записи контрольной информации. Для записи очередного пакета каждый раз вычисляется его начальный адрес в общем массиве памяти по следующей формуле:

$$Addr = Addr_{base} + 256 * N,$$

где $Addr_{base}$ – начальный адрес всего массива инструкций или данных, N – номер пакета. Так же после отправки пакета, производится расчет количества байт в следующем пакете, в случае если объем незаписанных во flash-память данных (инструкций) больше 256 байт, то данная величина равна 256, иначе следующий пакет является последним и количество единиц информации в байтах в нем рассчитывается следующим образом:

$$V = V_{com} - 256 * N_{last},$$

где V_{com} – общий объем массива в байтах, N_{last} – номер последнего пакета (счет начинается с 0). Для оптимизации кода удобно использовать одну функцию, как для записи инструкций, так и для записи данных. Входными параметрами функции являются количество пакетов, стартовый адрес во flash-памяти, а также объем в байтах (листинг 7.6).

```
void receive_from_uart(uint32_t nmb, uint32_t start_addr, uint32_t size) {
  uint32_t len = PP_NUMB;
  uint8_t  mass [256];

  if (size < PP_NUMB)len = size;

  for (uint32_t j = 0; j<nmb; j++) {

    //Получение очередного пакета информации по UART
    while(uart_getchar() != 'S'){};
    for (int i = 0; i<len; i++) {
      mass[i] = (uint8_t)(uart_getchar());
    }

    uart_send_with_chk(0x00);

    // Запись очередного пакета информации во flash-память
    atomic_spi_wr(WREN,SPI_CMD_QWR);

    spi_setup_cmd_addr(PP, 8, ((start_addr + j*PP_NUMB) & 0xFFFFFF), 24);
    spi_set_datalen((len<<3)); //in bits
    spi_start_transaction(SPI_CMD_QWR, SPI_CSN0);
    spi_write_fifo(&mass, (len<<3));
    while ((spi_get_status() & 0x3F) != 1);

    // Подсчет количества байт для следующей транзакции
    len = ((size-PP_NUMB*(j+1)) > PP_NUMB) ? PP_NUMB : (size-PP_NUMB*(j+1));

    uart_send_with_chk(j);
  }
}
```

Листинг кода 7.6: .

7.2 Ассемблерные вставки

После выгрузки ПО из flash-памяти необходимо изменить значение счетчика команд (PC) на адрес первой команды ПО приложения. Очевидно, что сделать это с помощью языка C не представляется возможным. Для выполнения данной операции можно использовать ассемблерные вставки.

Ассемблерные вставки используются для явного размещения в C-программе кода на языке ассемблер. В коде загрузчика ассемблерные вставки могут использованы во первых, для минимизации количества подключаемых драйверов (например, как было упомянуто во 2 главе лабораторной работы, вместе отсчета времени с помощью системного таймера, можно использовать nop инструкции), а во-вторых, для выполнения команд безусловного перехода для начала выполнения кода приложения (boot part).

Для компилятора GCC [2] синтаксис оператора ассемблерной вставки выглядит следующим образом: начинается с ключевого слова __asm__, в круглых скобках располагается сама вставка,

представляющая собой строковую константу, которая может содержать константу с ассемблерными инструкциями, выходные операнды, входные операнды, а также разрушаемые регистры (листинг 7.7).

```
1 __asm__ (вставка : список_выходных_операндов : список_входных_операндов :
  ↪  список_разрушаемых_регистров );
```

Листинг кода 7.7: .

Списки входных и выходных операндов, перечисляемые через запятую, используются для связи между переменными С-программы и ассемблерными инструкциями внутри вставки. Обращение к каждому операнду осуществляется по номеру с префиксом %, нумерация начинается с 0. Операнд описывается следующим образом: сначала указывается ограничение типа, затем в скобках имя переменной – переменная, объявленная в С-коде, значение которой будет использовано во вставке. Ограничение типа – константа, описывающая допустимый тип операнда, указывающий, компилятору, куда нужно поместить значение переменной. Для того чтобы поместить значение в один из РОН используется ограничение r. Для ограничения типа выходного операнда используется префикс =.

Ниже рассмотрен пример ассемблерной вставки для изменения значения счетчика команд на адрес первой инструкции ПО приложения (листинг 7.8).

```
1 __asm__(    "jalr x0, %0\n" : // Вставка
2                              : // Список выходных операндов
3     "r" (instr_start_addr) ); // Список входных операндов
```

Листинг кода 7.8: .

7.3 Контрольные вопросы

Что такое загрузчик ПО? Преимущество систем с загрузчиком ПО. Обобщенная структурная схема системы с загрузчиком ПО. Стандартный алгоритм работы. Основные функции загрузчика ПО. Привести примеры возможных дополнительных функций. Синтаксис ассемблерных вставок для gcc компилятора. Привести пример ассемблерной вставки. Для чего могут быть использованы ассемблерные вставки в коде загрузчика ПО?

7.4 Приложение

Таблица 7.1: Инструкции, требующие предварительную WREN инструкцию.

Имя инструкции	hex-код	Описание
PP	0x02	Последовательная запись
PPQ	0x32/ 0x38	Запись с использованием 4 линий IO для передачи данных
SER	0xD7/ 0x20	Стирание сектора (4 КБайта)
BER32	0x52	Стирание блока (32 КБайта)
BER64	0xD8	Стирание блока (64 КБайта)
CER	0xC7/ 0x60	Стирание чипа
WRSR	0x01	Запись в статус-регистр
WRFR	0x42	Запись в функциональный регистр
SRPNV	0x65	Запись в read регистр
SERPNV	0x85	Расширенная запись в read регистр
IRER	0x64	Стирание данных из массива Information Row
IRP	0x62	Запись данных в массив Information Row
WRABR	0x15	Запись в auto-boot регистр

```
#include <spi_boot.h>
#include <gpio_boot.h>
#include <uart_boot.h>
#include <utils.h>
#include <gecko.h>

#define CALC_DIV(clk_sys, clk_out) (clk_sys/clk_out-1)
#define CALC_DIV_UART(clk_sys, clk_out) (clk_sys/(clk_out*16)-1)

#define INSTR_START_ADDR 0x000018 // сектор 1   блок 0
#define DATA_START_ADDR  0x100000 // сектор 256 блок 16
#define SECTOR_SIZE      0x001000

// Настройки для контроллера GPIO
#define BOOT_PIN   0x1
#define PROG_MODE  0x0
#define BOOT_MODE  0x1

// Настройки для контроллера UART
#define BIT_NUMBER   0x8
#define STOP_NUMBER  0x2
#define PARITY_EN    0x1
#define PARITY_CNT   0x1
#define PARITY_EVEN  0x1
#define TRIG_CODE    0x2
#define FIFO64_EN    0x1
#define UART_CLK_Hz  9600
#define UART_CLK_CNT CALC_DIV_UART(1000000* SYS_CLK_MHz,UART_CLK_Hz) //
 ↪ Вычисление делителя

```

```
30 // Настройки для контроллера SPI
31 #define SPI_NUM_CS   0x01
32 #define SPI_CLK_MHz  0x01
33 #define SPI_CLK_DIV  CALC_DIV(SYS_CLK_MHz, SPI_CLK_MHz) // Вычисление
    ↪  делителя
34
35 // IS25LP016D / IS25WP016D Команды
36 #define RDUID 0x4B // Read Unique ID
37 #define FRQIO 0xEB // Fast Read Quad I/O
38 #define IRRD  0x68 // Read Information Row
39 #define QPIEN 0x35 // Enter QPI mode
40 #define WREN  0x06 // Write Enable
41 #define PP    0x02 // Page Program
42 #define CER   0x60 // Chip Erase
43 #define RDSR  0x05 // Read Status Register
44 #define WRSR  0x01 // Write Status Register
45
46 #define PP_NUMB 256 // Максимальное количество байт данных для PP
47
48 void boot_mode();
49 void app_mode();
50 void receive_from_uart(uint32_t nmb, uint32_t start_addr, uint32_t size);
51 int *receive_from_flash(int addr, int blocks, int size);
52
53 // Функция для отправки по UART с ожиданием
54 void uart_send_with_chk(uint8_t data) {
55   uart_sendchar(data);
56   uart_wait_tx_done();
57 }
58
59 // Функция для записи во flash с проверкой (только для операция содержащих
    ↪  одну команду)
60 void atomic_spi_wr(uint8_t cmd, uint8_t mode) {
61   spi_setup_cmd_addr(cmd, 8, 0x0, 0);
62   spi_start_transaction(mode, SPI_CSN0);
63   while ((spi_get_status() & 0x3F) != 1);
64 }
65
66 // Функция для получения контрольной информации по UART
67 uint32_t load_control( uint8_t msg, uint32_t ctrl_inf ) {
68   for (int i=0;i<4;i++) ctrl_inf   = (ctrl_inf<<8)   |
     ↪  (uint32_t)(uart_getchar());
69   uart_send_with_chk(msg);
70   return ctrl_inf;
71 }
72
73 int main() {
74   int      mode;
75   uint8_t *status_reg;
76   uint8_t *id;
```

```
77
78  for (int i = 0; i < 7500; i++) {
79    __asm__ volatile("nop");
80  }
81
82
83  uart_set_line_cfg(PARITY_CNT, PARITY_EN, PARITY_EVEN, STOP_NUMBER,
    ↪  BIT_NUMBER);
84  uart_set_counter(UART_CLK_CNT);
85  uart_set_fifo_en(FIFO64_EN, TRIG_CODE);
86  uart_send_with_chk(0x01);
87
88  spi_set_clk_div(SPI_CLK_DIV);
89  uart_send_with_chk(0x02);
90
91  set_pin_function(BOOT_PIN, FUNC_GPIO);
92  set_gpio_pin_direction(BOOT_PIN, DIR_IN);
93  mode = get_gpio_pin_value(BOOT_PIN);
94  uart_send_with_chk(0x03);
95
96
97  atomic_spi_wr(WREN,SPI_CMD_WR);
98
99  spi_setup_cmd_addr(WRSR, 8, 0x42, 8);
100 spi_start_transaction(SPI_CMD_WR, SPI_CSN0);
101 while ((spi_get_status() & 0x3F) != 1);
102
103 do{
104   spi_setup_cmd_addr(RDSR, 8, 0, 0);
105   spi_set_datalen(8);
106   spi_start_transaction(SPI_CMD_RD, SPI_CSN0);
107   while ((spi_get_status() & 0x3F) != 1);
108   spi_read_fifo(status_reg, 8);
109   uart_send_with_chk(*status_reg);
110 }while (((*status_reg) & 0x01) == 1);
111
112 atomic_spi_wr(QPIEN, SPI_CMD_WR);
113
114
115 while (1) {
116   mode = get_gpio_pin_value(BOOT_PIN);
117   switch (mode) {
118     case PROG_MODE : program_mode();
119       break;
120     case BOOT_MODE : boot_mode();
121       break;
122     default        : uart_sendchar(0x01);
123       break;
124   }
125 }
```

```
126
127   return 0;
128 }
129
130 void program_mode() {
131
132   uart_send_with_chk(0x05);
133
134   uint32_t instr_addr    = INSTR_START_ADDR;
135   uint32_t instr_size    = 0x0;
136   uint32_t instr_nmb     = 0x0;
137   uint32_t instr_sector  = 0x0;
138
139   uint32_t data_addr     = DATA_START_ADDR;
140   uint32_t data_size     = 0x0;
141   uint32_t data_nmb      = 0x0;
142   uint32_t data_sector   = 0x0;
143
144   instr_size   = load_control( 0xA0,   instr_size );
145   instr_nmb    = load_control( 0xA1,    instr_nmb );
146   instr_sector = load_control( 0xA2, instr_sector );
147   data_size    = load_control( 0xA3,    data_size );
148   data_nmb     = load_control( 0xA4,     data_nmb );
149   data_sector  = load_control( 0xA5,  data_sector );
150
151
152   atomic_spi_wr(WREN,SPI_CMD_QWR);
153   atomic_spi_wr(CER, SPI_CMD_QWR);
154
155   uint8_t *status_reg;
156   do {
157     spi_setup_cmd_addr(RDSR, 8, 0, 0);
158     spi_set_datalen(8);
159     spi_start_transaction(SPI_CMD_QRD, SPI_CSN0);
160     spi_read_fifo(status_reg, 8);
161   } while ((*status_reg & 0x01) == 1);
162
163   uart_send_with_chk(0x06);
164
165
166   atomic_spi_wr(WREN,SPI_CMD_QWR);
167
168   spi_setup_cmd_addr(PP, 8, (0x000000 & 0xFFFFFF), 24);
169   spi_set_datalen(6 * 32);
170   spi_write_fifo(&instr_addr,       32);
171   spi_write_fifo(&instr_sector,     32);
172   spi_write_fifo(&instr_size,       32);
173   spi_write_fifo(&data_addr,        32);
174   spi_write_fifo(&data_sector,      32);
175   spi_write_fifo(&data_size,        32);
```

```
176   while ((spi_get_status() & 0xF000000) != 6<<24);
177   spi_start_transaction(SPI_CMD_QWR, SPI_CSN0);
178   while ((spi_get_status() & 0x3F) != 1);
179
180   do {
181     spi_setup_cmd_addr(RDSR, 8, 0, 0);
182     spi_set_datalen(8);
183     spi_start_transaction(SPI_CMD_QRD, SPI_CSN0);
184     spi_read_fifo(status_reg, 8);
185   } while ((*status_reg & 0x01) == 1);
186
187   uart_send_with_chk(0x07);
188   receive_from_uart(instr_nmb, INSTR_START_ADDR, instr_size);
189
190   uart_send_with_chk(0x08);
191   receive_from_uart(data_nmb, DATA_START_ADDR, data_size);
192 }
193
194 void receive_from_uart(uint32_t nmb, uint32_t start_addr, uint32_t size) {
195   uint32_t len = PP_NUMB;
196   uint8_t  mass [256];
197
198   if (size < PP_NUMB)len = size;
199
200   for (uint32_t j = 0; j<nmb; j++) {
201
202     while(uart_getchar() != 'S'){};
203     for (int i = 0; i<len; i++) {
204       mass[i] = (uint8_t)(uart_getchar());
205     }
206
207     uart_send_with_chk(0x00);
208
209     atomic_spi_wr(WREN,SPI_CMD_QWR);
210
211     spi_setup_cmd_addr(PP, 8, ((start_addr + j*PP_NUMB) & 0xFFFFFF), 24);
212     spi_set_datalen((len<<3)); //in bits
213     spi_start_transaction(SPI_CMD_QWR, SPI_CSN0);
214     spi_write_fifo(&mass, (len<<3));
215     while ((spi_get_status() & 0x3F) != 1);
216
217     len = ((size-PP_NUMB*(j+1)) > PP_NUMB) ? PP_NUMB : (size-PP_NUMB*(j+1));
218
219     uart_send_with_chk(j);
220   }
221 }
222
223 void boot_mode() {
224
225   // Ваш код для boot part
```

```
226 }
```

Листинг кода 7.9: Код загрузчика.

7.5 Полезные ссылки

1. IS25LP016D Datasheet `http://www.issi.com/WW/pdf/25LP-WP016D.pdf`
2. GCC online documentation `https://gcc.gnu.org/onlinedocs/`