Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Факультет компьютерных систем и сетей

Кафедра встраиваемых вычислительных систем

М. В. Качинский

СТРУКТУРНАЯ И ФУНКЦИОНАЛЬНАЯ ОРГАНИЗАЦИЯ ВЫЧИСЛИТЕЛЬНЫХ МАШИН. ЛАБОРАТОРНЫЙ ПРАКТИКУМ

Рекомендовано УМО по образованию в области информатики и радиоэлектроники в качестве пособия для специальности 6-05-0611-05 «Компьютерная инженерия»

Минск БГУИР 2026

УДК 004.2(076.5)
ББК 32.971.32-02я73
К30

Рецензенты:

кафедра «Технология и методика преподавания»
Белорусского национального технического университета
(протокол № 12 от 22.04.2025);

начальник отдела ОАО «КБ Радар» –
управляющей компании холдинга «Системы радиолокации»
кандидат технических наук А. В. Шарамет

Качинский, М. В.

К30 Структурная и функциональная организация вычислительных машин. Лабораторный практикум : пособие / М. В. Качинский. – Минск : БГУИР, 2026. – 112 с. : ил.

ISBN 978-985-543-884-8.

Содержит описание лабораторных работ по дисциплине «Структурная и функциональная организация вычислительных машин». Лабораторная работа № 1 имеет целью закрепление практических навыков по созданию принципиальной схемы цифрового устройства в схемотехническом редакторе Xilinx ISE и ее функциональному моделированию. Лабораторные работы № 2–6 посвящены проектированию арифметического устройства как структурного блока процессора вычислительной машины. В лабораторных работах № 3–5 фактически осуществляется проверка правильности выполненного в лабораторной работе № 2 функционального проектирования заданного арифметического устройства путем построения проекта устройства и его функционального моделирования в системе проектирования Xilinx ISE. В лабораторной работе № 6 реализуется окончательная проверка проекта путем выполнения прототипирования разработанного устройства с помощью отладочной платы Spartan-3 Starter Kit. Лабораторные работы № 7–10 посвящены изучению организации RISC-процессора с архитектурой MIPS, основ языка ассемблера MIPS и особенностей программирования на нем, а также созданию и отладке программ на языке ассемблера MIPS с помощью симулятора.

УДК 004.2(076.5)
ББК 32.971.32-02я73

ISBN 978-985-543-884-8

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Пособие включает методические указания по выполнению лабораторных работ по дисциплине «Структурная и функциональная организация вычислительных машин». Приведенные в лабораторном практикуме лабораторные работы ориентированы на освоение студентами профилизации «Встраиваемые системы» специальности 6-05-0611-05 «Компьютерная инженерия».

Лабораторный практикум включает описания десяти лабораторных работ, которые условно можно разделить на два блока. Первый блок образуют лабораторные работы № 2–6, которые посвящены проектированию арифметического устройства как структурного блока процессора вычислительной машины. В качестве операции, реализуемой арифметическим устройством, выбрана операция умножения целых чисел со знаком. Целью лабораторной работы № 1 является закрепление практических навыков по созданию принципиальной схемы цифрового устройства в схемотехническом редакторе Xilinx ISE и ее функциональному моделированию для выявления ошибок и проверки работоспособности. Данные темы подробно изучались и отрабатывались при выполнении курсовой работы по дисциплине «Основы цифровой схемотехники». Основной целью лабораторной работы № 2 является изучение методики описания структуры арифметического устройства средствами языка операционных схем и получение практических навыков по разработке функциональной схемы операционной и управляющей частей арифметического устройства. В лабораторных работах № 3–5 фактически осуществляется проверка правильности выполненного в лабораторной работе № 2 функционального проектирования заданного арифметического устройства (правильности разработанной функциональной схемы устройства) путем построения проектов операционной и управляющей частей арифметического устройства, а также устройства в целом, и их функционального моделирования в системе проектирования Xilinx ISE. Окончательная проверка реализуется путем выполнения прототипирования разработанного устройства с помощью отладочной платы Spartan-3 Starter Kit, включающей ПЛИС фирмы Xilinx Spartan-3 (микросхема XC3S200).

Вторая часть лабораторного практикума (лабораторные работы № 7–10) посвящена изучению организации RISC-процессора с архитектурой MIPS, основ языка ассемблера MIPS и особенностей программирования на нем, в том числе с использованием механизма процедур и стека, а также получению практических навыков создания и отладки программ на языке ассемблера MIPS с помощью симулятора.

ЛАБОРАТОРНАЯ РАБОТА № 1.
ИЗУЧЕНИЕ ИНТЕГРИРОВАННОЙ СРЕДЫ РАЗРАБОТКИ ЦИФРОВЫХ УСТРОЙСТВ НА ОСНОВЕ ПЛИС XILINX ISE

Цель работы – изучение методики проектирования цифровых устройств на основе ПЛИС с использованием интегрированной среды разработки серии Xilinx ISE.

1.1. Общие сведения о ПЛИС

Программируемые логические интегральные схемы (ПЛИС; программируемые пользователем логические интегральные схемы) относятся к микросхемам с программируемой структурой и представляют собой одно из самых перспективных и быстроразвивающихся направлений современной цифровой микроэлектроники. За последние десятилетия произошел быстрый рост рынка этих устройств и существенное улучшение их характеристик. Это привело к широкому использованию ПЛИС для проектирования цифровых устройств различного функционального назначения.

С появлением ПЛИС стало возможным проектирование и изготовление небольшой партии уникальных цифровых устройств в условиях небольшого предприятия, исследовательской или учебной лаборатории. Промышленно выпускаемые программируемые микросхемы с электрическим программированием и автоматизированным процессом настройки микросхемы на схему пользователя делают проектирование новых цифровых устройств сравнимым с разработкой программного обеспечения.

Различают два основных типа ПЛИС:

– FPGA (Field Programmable Gate Array);

– CPLD (Complex Programmable Logic Device).

В зависимости от сложности решаемых задач выбирается тот или иной тип ПЛИС. Если требуется небольшое число триггеров и эквивалентных логических элементов, то подходит CPLD или младшие серии FPGA. Для более крупных проектов выбираются кристаллы FPGA большей емкости по числу триггеров и эквивалентных логических элементов, при этом иногда требуется использовать несколько кристаллов в проекте.

FPGA – программируемые пользователем вентильные матрицы – наиболее обширный класс программируемых схем, обладающих максимальными функциональными возможностями.

Основными областями применения FPGA являются следующие две:

– отработка прототипов цифровых устройств и систем при их проектировании;

– создание быстрыми и эффективными способами конечной продукции в случае, когда требуется выпустить относительно небольшое количество устройств.

В FPGA базовой архитектуры по строкам и столбцам размещаются идентичные функциональные блоки КЛБ (конфигурируемые логические блоки), между которыми проходят трассы межсоединений. На периферии кристалла располагаются блоки ввода/вывода. Кроме того, в состав FPGA могут входить дополнительные функциональные ресурсы, такие как встроенные блоки памяти и умножители, а также специализированные средства для автоподстройки задержек в системе тактирования (PLL, DLL, DCM), средства для поддержки интерфейса JTAG, для реализации высокопроизводительных шин, для генерации тактовых сигналов и т. д. В качестве логических преобразователей КЛБ для FPGA с триггерной памятью конфигурации наиболее часто используются LUT-блоки (LUT – Look-Up Table) – программируемые запоминающие устройства (ЗУ). Например, логический блок FPGA семейства Spartan фирмы Xilinx содержит три LUT-блока: G, F и H. Преобразователи G и F представляют собой программируемые ЗУ с организацией 16 × 1, способные воспроизводить любые функции четырех переменных, преобразователь H – программируемое ЗУ с организацией 8 × 1, обеспечивающее воспроизведение функции трех переменных. Выходы блоков G и F могут подаваться на входы блока H для образования функции, зависящей от более чем четырех аргументов. В результате ресурсы логической части FPGA позволяют воспроизводить следующие функции:

– две функции с числом аргументов до четырех плюс функцию с числом аргументов до трех;

– одну функцию пяти аргументов;

– любую функцию четырех аргументов плюс некоторые функции шести аргументов, некоторые функции с числом аргументов до девяти.

1.2. Этапы проектирования в системе Xilinx ISE

Проектирование цифровых устройств на основе ПЛИС заключается в выполнении следующих этапов в системе проектирования:

1) создание принципиальной схемы проектируемого устройства в схемотехническом редакторе или описание проектируемого устройства на языке описания аппаратуры, например VHDL;

2) предварительное функциональное моделирование (Behavioral Simulation) для выявления ошибок и проверки работоспособности проектируемого устройства в целом или отдельных его частей;

3) привязка выводов проекта к входам/выходам кристалла, выбор выходных уровней, критичных цепей и т. д.;

4) размещение проекта в кристалле и анализ генерируемых отчетов для выявления возможных предупреждений и ошибок (Implement Design);

5) верификация проекта, т. е. окончательное временное моделирование (Post-Route Simulation) после размещения проекта в кристалле при всех реальных задержках распространения сигналов внутри микросхемы ПЛИС;

6) конфигурирование кристалла ПЛИС с помощью конфигурационной последовательности (файла программирования).

Загрузка конфигурационной последовательности осуществляется через специально выделенные конфигурационные выводы с использованием различных способов и режимов загрузки ПЛИС. После загрузки выполняется проверка и отладка проекта. В случае неправильной работы или необходимости модификации (доработки) проекта все этапы повторяются до полного завершения проекта в целом.

В данной лабораторной работе предполагается выполнение первых двух этапов: создание принципиальной схемы проектируемого устройства в схемотехническом редакторе среды проектирования Xilinx ISE и выполнение функционального моделирования.

1.3. Схемотехническое моделирование в среде проектирования Xilinx ISE

1.3.1. Создание проекта

В качестве демонстрационного проекта будет реализована комбинационная схема, описываемая следующей логической функцией:

$$f = \overline{x}_3\overline{x}_1 \vee x_2x_1x_0 \vee x_3\overline{x}_2x_1\overline{x}_0. \qquad (1.1)$$

Для создания нового проекта в среде Xilinx ISE необходимо воспользоваться либо пунктом меню *File → New Project*, либо соответствующей кнопкой в области быстрого доступа *New Project*. Загрузка уже созданного проекта выполняется аналогичным способом: пункт меню *File → Open Project* или кнопка *Open Project* из области быстрого доступа. При создании нового проекта открывается окно мастера пошагового создания проектов *New Project Wizard*.

В данном окне указывается название проекта и рабочая папка. Выпадающее меню *Top-level source type* в нижней части окна мастера создания проектов должно быть установлено в пункт *Schematic* – создание проекта на основе принципиальной схемы.

Следующее окно в мастере создания проектов отвечает за настройки аппаратной платформы, на базе которой будет производиться моделирование, и выбор программы симулятора для данного проекта. В этом окне необходимо указать рекомендуемые для выполнения лабораторной работы настройки: семейство кристаллов (*Family*), на базе которого будет проводиться моделирование, – Spartan 3; непосредственно кристалл (*Device*) – XC3S200; корпус кристалла – FT256; параметр быстродействия кристалла – –4; симулятор (*Simulator*) – ISim.

После выбора необходимых настроек аппаратной платформы и выбора инструмента симуляции в мастере создания проектов откроется окно с суммарной информацией о создаваемом проекте. Если все настройки, отображенные в данном окне, корректны, то после нажатия кнопки *Finish* произойдет создание пустого проекта.

Для построения схемы моделируемого устройства необходимо создать первый файл проекта с помощью мастера пошагового создания нового файла проекта. Сначала необходимо выбрать тип создаваемого исходного файла к проекту *Schematic*. Также на этом шаге указывается имя исходного файла. При этом поле с рабочей папкой проекта заполняется автоматически. После нажатия кнопки *Next* появится окно с суммарной информацией о создаваемом файле схемы. При нажатии в этом окне кнопки *Finish* будет создан новый файл схемы для текущего проекта, что отобразится в области иерархии проекта вместо надписи «*Empty Project*». Также в центральной области главного окна появится поле создания и редактирования схемы моделируемого устройства. Следующий шаг построения схемы – размещение необходимых компонентов и соединение их между собой в редакторе схемы.

В библиотеке среды проектирования Xilinx ISE имеется большое количество компонентов, которые позволяют построить схемы различного уровня сложности. Данные компоненты находятся во вкладке *Symbols* (рис. 1.1, область 1).

Все компоненты разбиты на категории (логические элементы, арифметические элементы, триггеры и т. д.). Для более удобного поиска необходимого элемента библиотеки есть строка фильтра по имени компонента (*Symbol Name Filter*, область 2 на рис. 1.1). Под строкой фильтрации имени компонентов расположен пункт выбора пространственной ориентации компонентов, где доступно четыре варианта поворота выбранного элемента (на четыре угла: 0°, 90°, 180°, 270°), а также возможность зеркального отражения с одновременным поворотом.

Для построения моделируемой схемы сначала необходимо расположить на поле создания и редактирования логические элементы. Для построения схемы, заданной формулой (1.1), необходимы элементы: 2И (*AND2*), 3И (*AND3*), 4И (*AND4*), 3ИЛИ (*OR3*) и четыре инвертора – НЕ (*INV*). Все перечисленные элементы находятся в категории *Logic*. На рис. 1.2 показано расположение логических элементов для реализации схемы рассматриваемого проекта.

Для того чтобы поменять позицию логического элемента на листе схемы, можно обратиться к контекстному меню, раздел *Symbol* – там находятся команды зеркального отражения и поворота (*Mirror* и *Rotate*, соответственно). Также в данном пункте контекстного меню находится пункт вызова справочной информации о компоненте – *Symbol Info*, в котором содержится описание, а также таблица истинности для большинства элементов.

После того как все необходимые логические элементы из библиотеки будут расположены на поле схемы, необходимо произвести редактирование схемы: соединение всех элементов, а также обозначение входных и выходных портов. Инструменты для осуществления данных действий находятся в пункте меню *Add*, а также на панели быстрого доступа, находящейся слева от поля редактирования схемы.

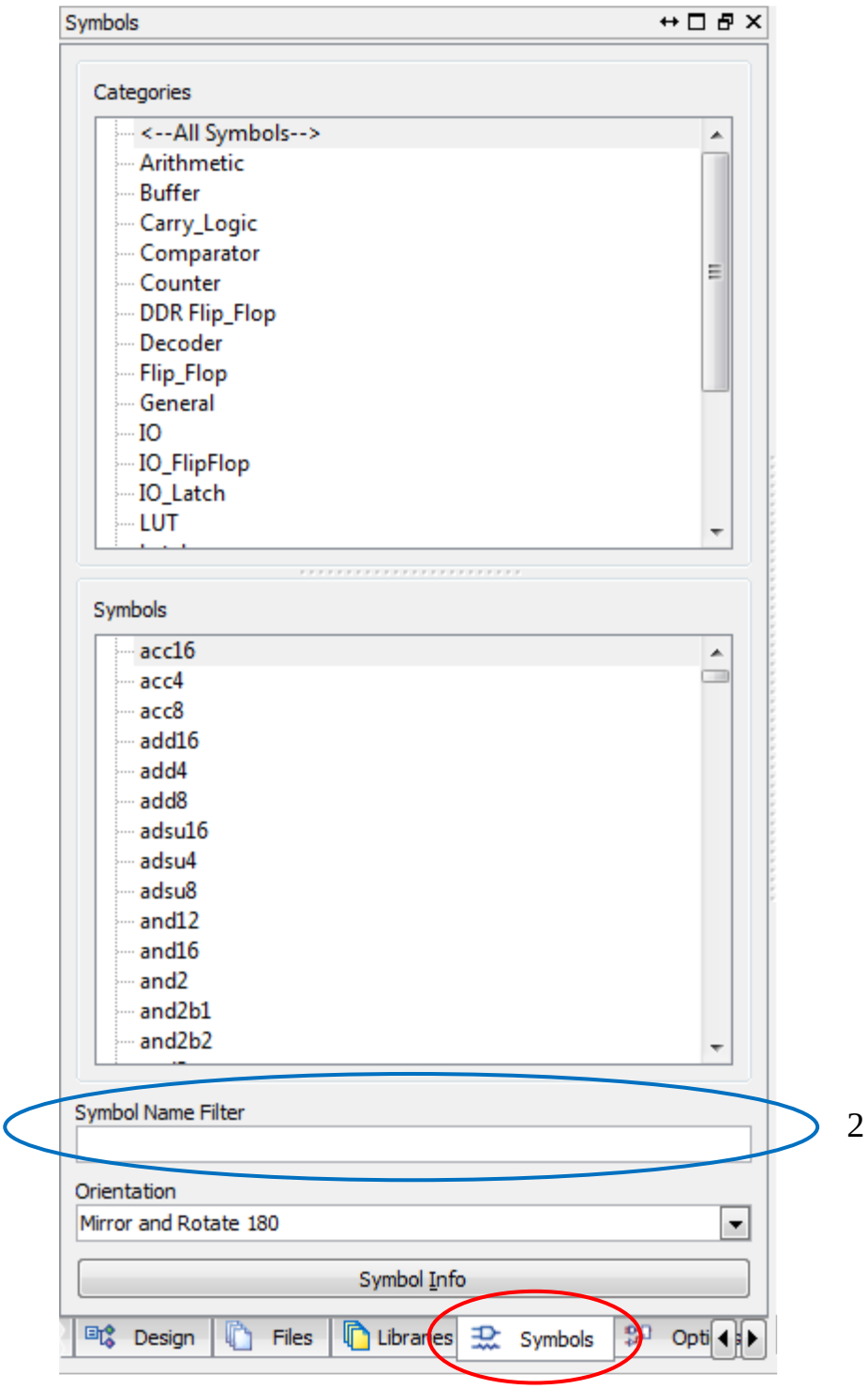

Рис. 1.1. Вкладка *Symbols*

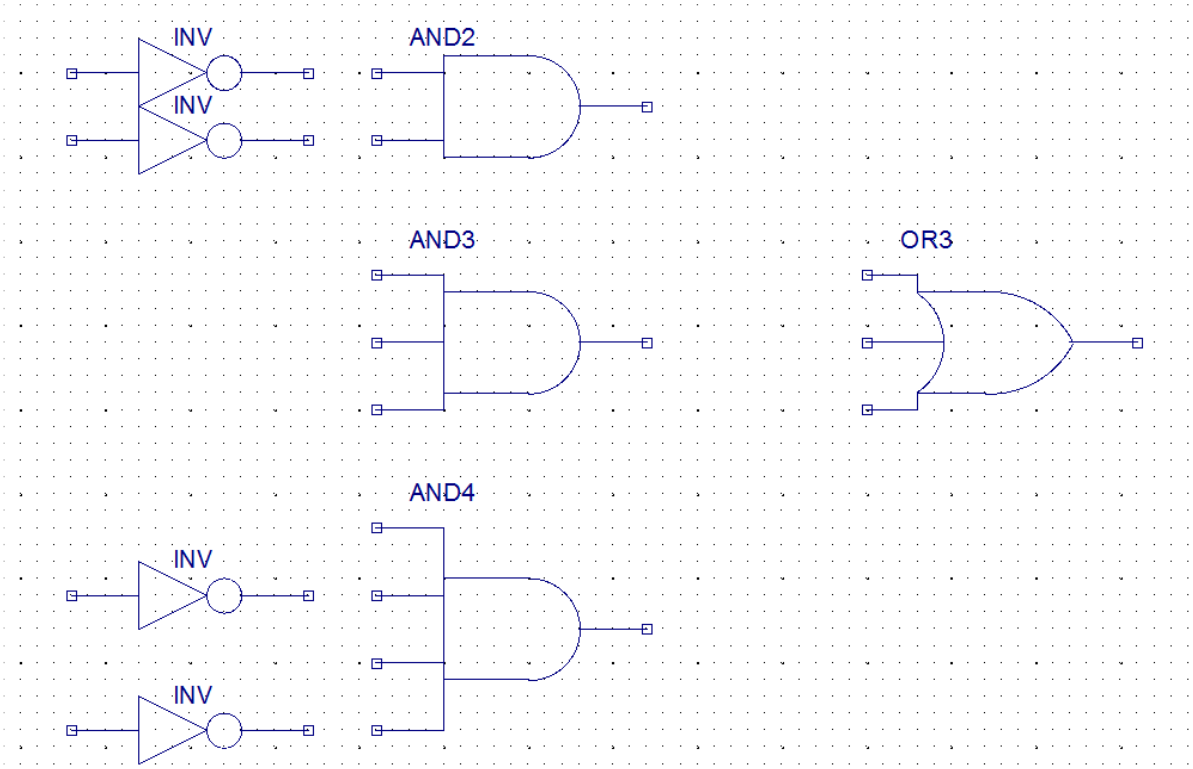

Рис. 1.2. Расположение элементов схемы

Для соединения логических элементов друг с другом нужно воспользоваться инструментом *Add Wire*. Для соединения двух элементов необходимо после выбора данного инструмента навести курсор на квадратное поле, которым заканчивается вход/выход логического элемента, и щелкнуть левой кнопкой мыши. Далее навести курсор мыши на целевой вход/выход элемента (аналогичное квадратное поле) и щелкнуть левой кнопкой мыши по нему. При этом произойдет создание связи между необходимыми элементами схемы. В случае если нужно разделить одну связь на несколько логических элементов (например, общий вход), можно щелкнуть левой кнопкой мыши в любом свободном месте необходимой линии соединения и провести связь, как это описано выше, к целевому логическому элементу. Если необходимо отредактировать связи между логическими элементами, то однократным щелчком левой кнопки мыши можно выделить нужную связь. Существует два варианта выделения линий связи, переключение между которыми находится на вкладке *Options* (см. рис. 1.1, справа от вкладки *Symbols*): выделение линии и всех ответвлений от нее или выделение только выбранного сегмента данной связи.

Для того чтобы обозначить входные и выходные порты на схеме, необходимо воспользоваться инструментом *Add I/O Marker*. Для установки маркера порта необходимо щелкнуть левой кнопкой мыши по квадратному полю входа/выхода логического элемента либо необходимой линии связи. По умолчанию среда проектирования сама расставляет имена портов входов/выходов. Для их переименования необходимо щелкнуть по маркеру правой кнопкой мыши и

выбрать пункт контекстного меню *Rename Port*. После этого появится окно со строкой ввода имени порта входа/выхода.

На рис. 1.3 показана моделируемая схема со всеми связями и обозначенными входными и выходными портами.

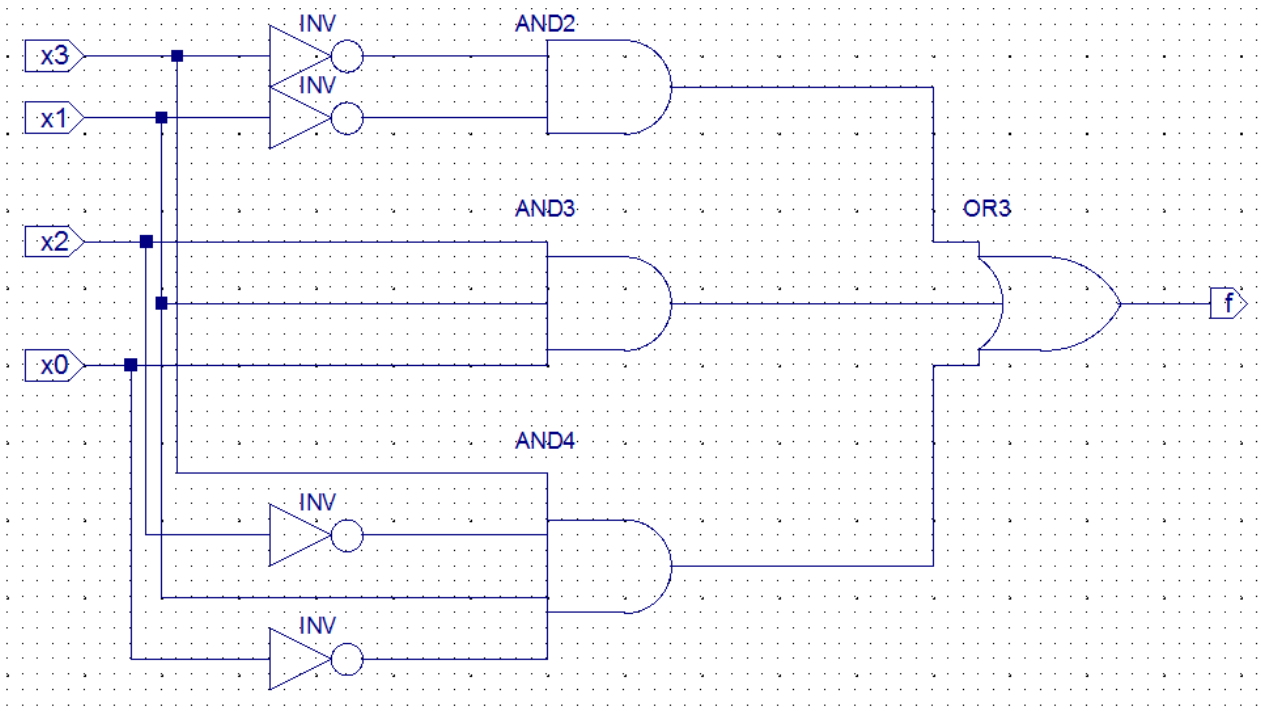

Рис. 1.3. Моделируемая схема

1.3.2. Симуляция моделируемой схемы с помощью тестового файла

После создания схемы и внесения в нее необходимых правок необходимо сохранить схему и приступить к симуляции ее работы. Для этого в левой области главного экрана среды проектирования над иерархией проекта необходимо выбрать пункт *Simulation*.

Симуляция представляет собой генерацию выходных значений моделируемой схемы как отклик на различные наборы значений входных сигналов. Для того чтобы реализовать последовательную подачу сигналов на входы схемы, необходимо создать тестовый файл, в котором будут находиться входные значения. Создание тестового файла выполняется аналогично созданию файла схемы с помощью мастера пошагового создания файла, однако теперь вместо пункта *Schematic* необходимо выбрать пункт *VHDL Test Bench*. После совершения необходимых шагов среда Xilinx ISE создаст текстовый шаблон тестового файла.

В примере 1.1 приведен вариант VHDL-модуля шаблона тестовых воздействий, создаваемого средой проектирования Xilinx ISE.

Как видно из шаблона тестового файла, среда проектирования автоматически определила сигналы, которые были обозначены на схеме как входные и выходные.

Пример 1.1. VHDL-модуль шаблона тестовых воздействий

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;
LIBRARY UNISIM;
USE UNISIM.Vcomponents.ALL;

ENTITY first_project_scheme_first_project_scheme_sch_tb IS
END first_project_scheme_first_project_scheme_sch_tb;

ARCHITECTURE behavioral OF first_project_scheme_first_project_scheme_sch_tb IS

   COMPONENT first_project_scheme
   PORT(x3        : IN  STD_LOGIC;
        x1        : IN  STD_LOGIC;
        x2        : IN  STD_LOGIC;
        x0        : IN  STD_LOGIC;
        f         : OUT STD_LOGIC);
   END COMPONENT;

   SIGNAL x3      : STD_LOGIC;
   SIGNAL x1      : STD_LOGIC;
   SIGNAL x2      : STD_LOGIC;
   SIGNAL x0      : STD_LOGIC;
   SIGNAL f       : STD_LOGIC;

BEGIN

   UUT: first_project_scheme PORT MAP(
      x3 => x3,
      x1 => x1,
      x2 => x2,
      x0 => x0,
      f  => f
   );

-- *** Test Bench - User Defined Section ***
   tb : PROCESS
   BEGIN
      WAIT; -- will wait forever
   END PROCESS;
-- *** End Test Bench - User Defined Section ***

END;
```

Задачей разработчика является инициализация последовательности подаваемых входных наборов. Для этого в области, границы которой обозначены строками

```
-- *** Test Bench – User Defined Section ***
                    …
-- *** End Test Bench – User Defined Section ***,
```

между зарезервированными словами `BEGIN` и `END PROCESS` необходимо задать все желаемые наборы значений входных сигналов и указать, в течение какого интервала времени симуляции они будут поступать на вход схемы. Так как разрабатываемая в примере схема является функцией от четырех переменных, то

для полной ее симуляции необходимо описать 16 двоичных наборов таблицы истинности. Фрагмент тестового файла, отвечающий за данную симуляцию, представлен в табл. 1.1.

Таблица 1.1

Входные наборы для демонстрационной схемы

Наборы 0000–0011	Наборы 0100–0111	Наборы 1000–1011	Наборы 1100–1111
x3 <= '0'; x2 <= '0'; x1 <= '0'; x0 <= '0'; wait for 50 ns; x3 <= '0'; x2 <= '0'; x1 <= '0'; x0 <= '1'; wait for 50 ns; x3 <= '0'; x2 <= '0'; x1 <= '1'; x0 <= '0'; wait for 50 ns; x3 <= '0'; x2 <= '0'; x1 <= '1'; x0 <= '1'; wait for 50 ns;	x3 <= '0'; x2 <= '1'; x1 <= '0'; x0 <= '0'; wait for 50 ns; x3 <= '0'; x2 <= '1'; x1 <= '0'; x0 <= '1'; wait for 50 ns; x3 <= '0'; x2 <= '1'; x1 <= '1'; x0 <= '0'; wait for 50 ns; x3 <= '0'; x2 <= '1'; x1 <= '1'; x0 <= '1'; wait for 50 ns;	x3 <= '1'; x2 <= '0'; x1 <= '0'; x0 <= '0'; wait for 50 ns; x3 <= '1'; x2 <= '0'; x1 <= '0'; x0 <= '1'; wait for 50 ns; x3 <= '1'; x2 <= '0'; x1 <= '1'; x0 <= '0'; wait for 50 ns; x3 <= '1'; x2 <= '0'; x1 <= '1'; x0 <= '1'; wait for 50 ns;	x3 <= '1'; x2 <= '1'; x1 <= '0'; x0 <= '0'; wait for 50 ns; x3 <= '1'; x2 <= '1'; x1 <= '0'; x0 <= '1'; wait for 50 ns; x3 <= '1'; x2 <= '1'; x1 <= '1'; x0 <= '0'; wait for 50 ns; x3 <= '1'; x2 <= '1'; x1 <= '1'; x0 <= '1'; wait;

Для финального этапа моделирования – построения временной диаграммы – необходимо сохранить созданный тестовый файл и в левой области главного окна программы с помощью пункта *Simulate Behavioral Model* запустить процесс симуляции работы моделируемой схемы. Результат процесса симуляции работы моделируемой схемы приведен на рис. 1.4.

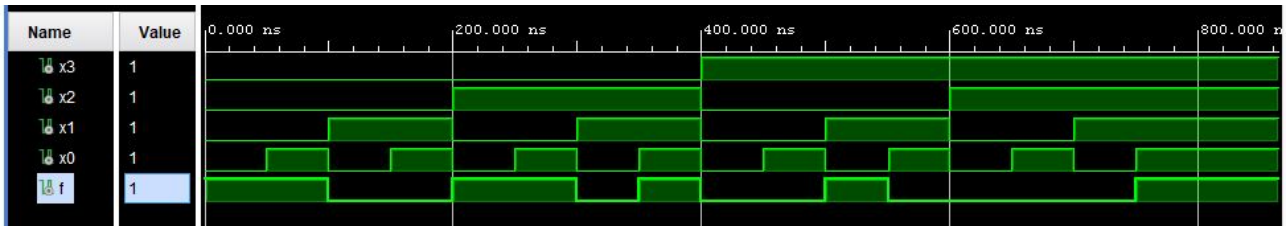

Рис. 1.4. Окно отображения временной диаграммы моделируемой схемы

Центральную область окна, показанного на рис. 1.4, занимает непосредственно временная диаграмма. Левее данной области располагаются столбец с текущими значениями входных/выходных сигналов – *Value* – и столбец *Name*, в котором отображаются имена всех входных и выходных сигналов схемы. Для проверки правильности построения и функционирования демонстрационной схемы можно поочередно подставлять входные наборы в формулу (1.1) и сверять значение функции с соответствующими значениями временной диаграммы. В

общем случае разработка устройства связана с синтезом таблицы истинности (либо в задании она указана в качестве исходных данных), и временная диаграмма является прямым отражением этой таблицы. Временную диаграмму в окне программы симуляции можно масштабировать для более детализированного изучения. Масштаб можно изменять, либо обратившись в пункт меню *View* → *Zoom* и выбрав в нем необходимые действия, либо с помощью соответствующих данному меню пиктограмм на панели инструментов, а также правее столбца с перечнем имен входных/выходных переменных. Для более удобной группировки сигналов их можно перетаскивать. Для этого необходимо щелкнуть правой кнопкой мыши по имени сигнала в области *Name* и, не отпуская ее, произвести перетаскивание объекта.

1.4. Порядок выполнения работы

1. Получить номер варианта задания у преподавателя.
2. Разработать логическую схему заданного устройства.
3. Создать проект в виде принципиальной схемы проектируемого устройства в схемотехническом редакторе среды проектирования Xilinx ISE.
4. Выполнить его моделирование в среде проектирования Xilinx ISE.
5. Оформить отчет.

1.5. Содержание отчета

1. Титульный лист.
2. Цель работы.
3. Краткие сведения по методике проектирования цифровых устройств на основе ПЛИС с использованием интегрированной среды разработки серии Xilinx ISE.
4. Описание процесса разработки логической схемы заданного цифрового устройства.
5. Описание процесса построения принципиальной схемы проектируемого устройства в схемотехническом редакторе среды проектирования Xilinx ISE.
6. Описание процесса моделирования проекта заданного цифрового устройства в среде проектирования Xilinx ISE.
7. Выводы по работе.

1.6. Варианты задания

1. Преобразователь 4-разрядного прямого кода в дополнительный.
2. Последовательный сумматор двух 4-разрядных двоичных чисел.
3. Компаратор двух 2-разрядных двоичных чисел (равно, больше, меньше).
4. Преобразователь кода 8-4-2-1 в 5-4-2-1.
5. Преобразователь 4-разрядного двоичного кода в код Грея.

6. Полный двоичный 3-разрядный дешифратор.
7. Генератор чисел 3-2-8-12 на базе двух триггеров.
8. Генератор чисел 2-5-14-1 на базе четырех триггеров.
9. Регистр сдвига 3-разрядный, имеющий последовательность состояний 0-1-2-5-3-6-4-0.
10. Преобразователь 4-разрядного дополнительного кода в прямой.
11. Мажоритарный элемент для $n = 5$.
12. Преобразователь кода 7-4-2-1 в 2-4-2-1.
13. Преобразователь 4-разрядного кода Грея в двоичный код.
14. Генератор чисел 1-0-7-11 на базе двух триггеров.
15. Генератор чисел 2-13-8-7 на базе четырех триггеров.
16. Сумматор 4-разрядный с условным переносом.
17. Счетчик 4-разрядный с собственной остановкой в состоянии N, где N – входной 4-разрядный двоичный код. Для запуска цикла счета счетчик необходимо установить в состояние 0.
18. Счетчик 3-разрядный, который при управляющем входе $M = 1$ работает как двоичный суммирующий счетчик, а при $M = 0$ – как счетчик в коде Грея.
19. Устройство 4-разрядное пересчетное на сдвиговом регистре с модулем счета 10.
20. Счетчик 4-разрядный синхронный с коэффициентом счета $K = 19 - N$, где N – входной 4-разрядный двоичный код.

ЛАБОРАТОРНАЯ РАБОТА № 2. РАЗРАБОТКА ФУНКЦИОНАЛЬНОЙ СХЕМЫ АРИФМЕТИЧЕСКОГО УСТРОЙСТВА

Цель работы – изучение методики описания структуры арифметического устройства средствами языка операционных схем; получение практических навыков по разработке функциональной схемы операционной части арифметического устройства; получение практических навыков описания порядка функционирования арифметического устройства в форме содержательной граф-схемы алгоритма (ГСА); изучение методики проектирования управляющей части арифметического устройства; получение практических навыков по синтезу микропрограммного автомата (МПА) арифметического устройства; получение практических навыков по разработке функциональной схемы управляющей части арифметического устройства.

2.1. Этапы разработки операционной части арифметического устройства

Любое вычислительное устройство состоит из двух частей: управляющего и операционного автоматов (рис. 2.1).

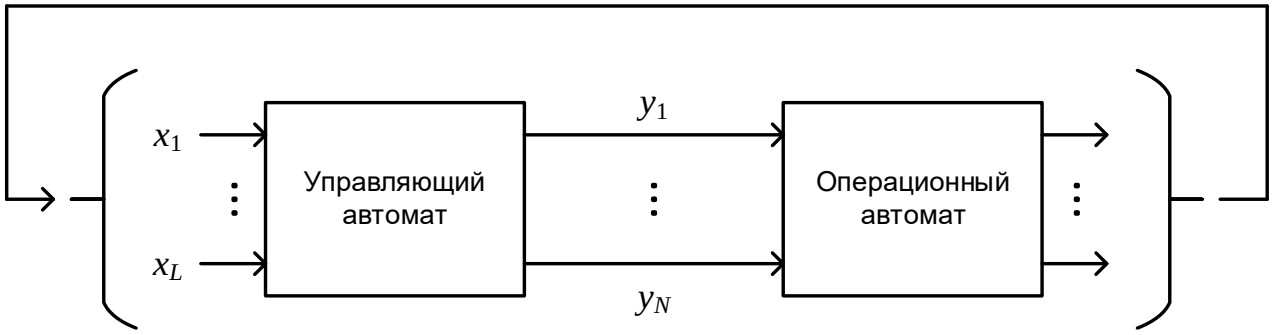

Рис. 2.1. Представление вычислительного устройства в виде композиции управляющего и операционного автоматов

Элементарное действие, выполняемое в операционном автомате, называют *микрооперацией*. Пусть $Y = \{y_1, \dots, y_N\}$ – множество микроопераций, которые выполняются под действием сигналов управляющего автомата. Обозначим эти сигналы символами $y_1, \dots, y_N$, причем для выполнения микрооперации y_n $(n = 1, \dots, N)$ должен появиться соответствующий управляющий сигнал, равный единице. Если в операционном автомате одновременно выполняются несколько микроопераций, то они образуют *микрокоманду*. Множество микрокоманд вместе с логическими условиями $x_1, \dots, x_L$ определяющими последовательность их выполнения, составляют *микропрограмму*. Логические условия соответствуют осведомительным сигналам, вырабатываемым в операционном автомате.

Проектирование арифметического устройства рассмотрим для случая, когда в устройстве выполняется одна арифметическая операция. Порядок разработки операционной части арифметического устройства состоит из следующих шагов:

1) определить исходные данные на разработку арифметического устройства;
2) составить блок-схему алгоритма выполнения заданной операции;
3) разработать структурную схему арифметического устройства;
4) построить функциональную схему операционного автомата арифметического устройства;
5) разработать микропрограмму работы арифметического устройства;
6) составить список микроопераций и логических условий;
7) отметить на функциональной схеме операционного автомата управляющие и осведомительные сигналы, соответствующие микрооперациям и логическим условиям.

В лабораторных работах будем рассматривать реализацию операции целочисленного умножения. В качестве примера на рис. 2.2 показана структурная схема устройства умножения, в котором реализуется метод выполнения операции умножения, в соответствии с которым умножение начинается с младших разрядов множителя, сумма частичных произведений (СЧП) сдвигается вправо, а множимое остается неподвижным.

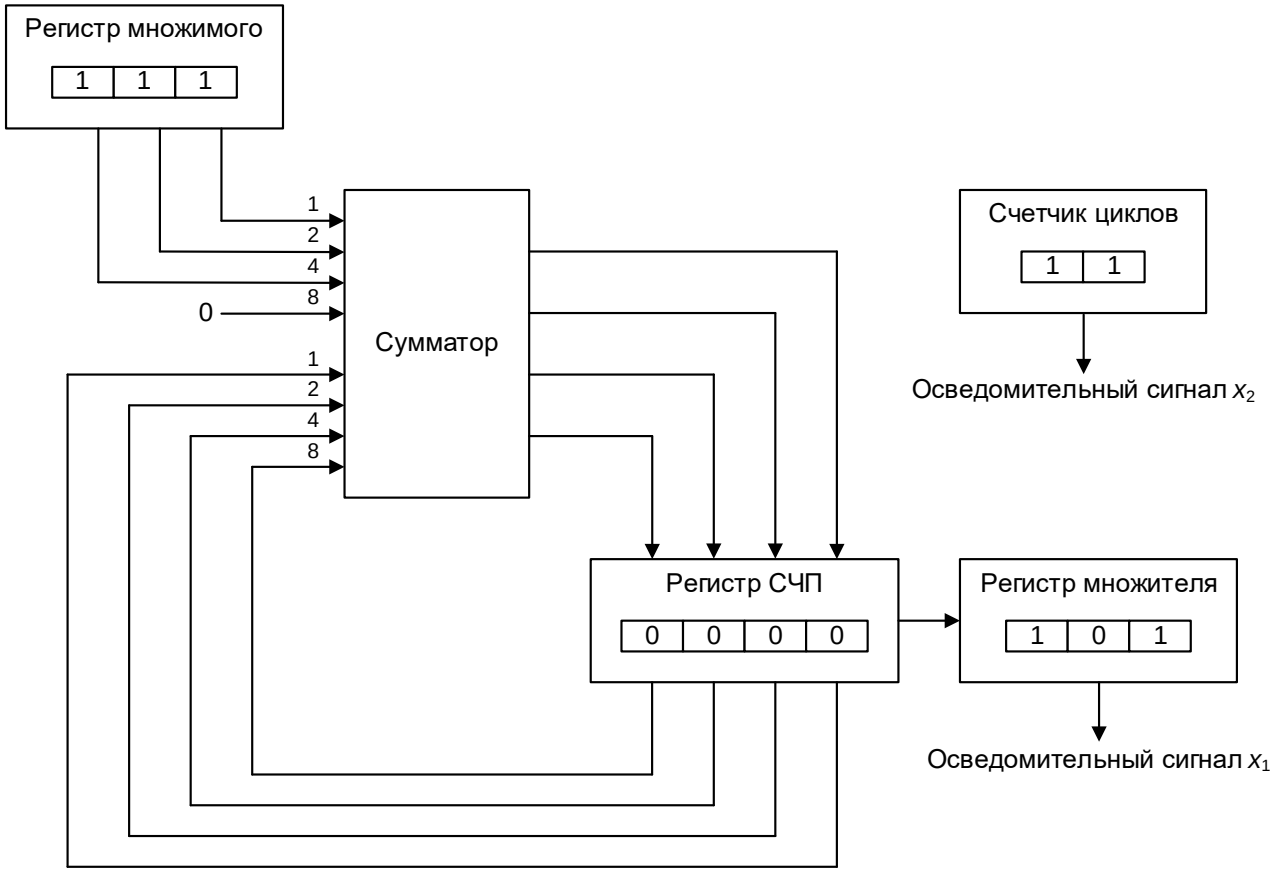

Рис. 2.2. Структурная схема устройства умножения

В рассматриваемом примере множимое и множитель представляют собой числа без знака и имеют по три двоичных разряда, в результате произведение должно получиться 6-разрядным. В исходном состоянии множимое 111 загружено в регистр, показанный в левой верхней части рис. 2.2, регистр СЧП очищен (установлен в состояние 0000) и множитель 101 загружен в регистр, показанный в нижней правой части рис. 2.2. Для уменьшения аппаратных затрат регистр СЧП и регистр множителя при выполнении сдвига вправо рассматриваются как единый регистр. Это отражено на рисунке стрелкой, соединяющей оба регистра. Кроме того, в состав умножителя входит счетчик циклов, в который в исходном состоянии загружается число циклов умножения 11 (десятичное число 3). Счетчик циклов выдает осведомительный сигнал, который устанавливается в 1, когда содержимое счетчика становится равным 00.

Используя структурную схему умножителя, рассмотрим подробно процедуру умножения. Рис. 2.3 дает поэтапную иллюстрацию процесса умножения двоичного числа 111 (множимое) на двоичное число 101 (множитель).

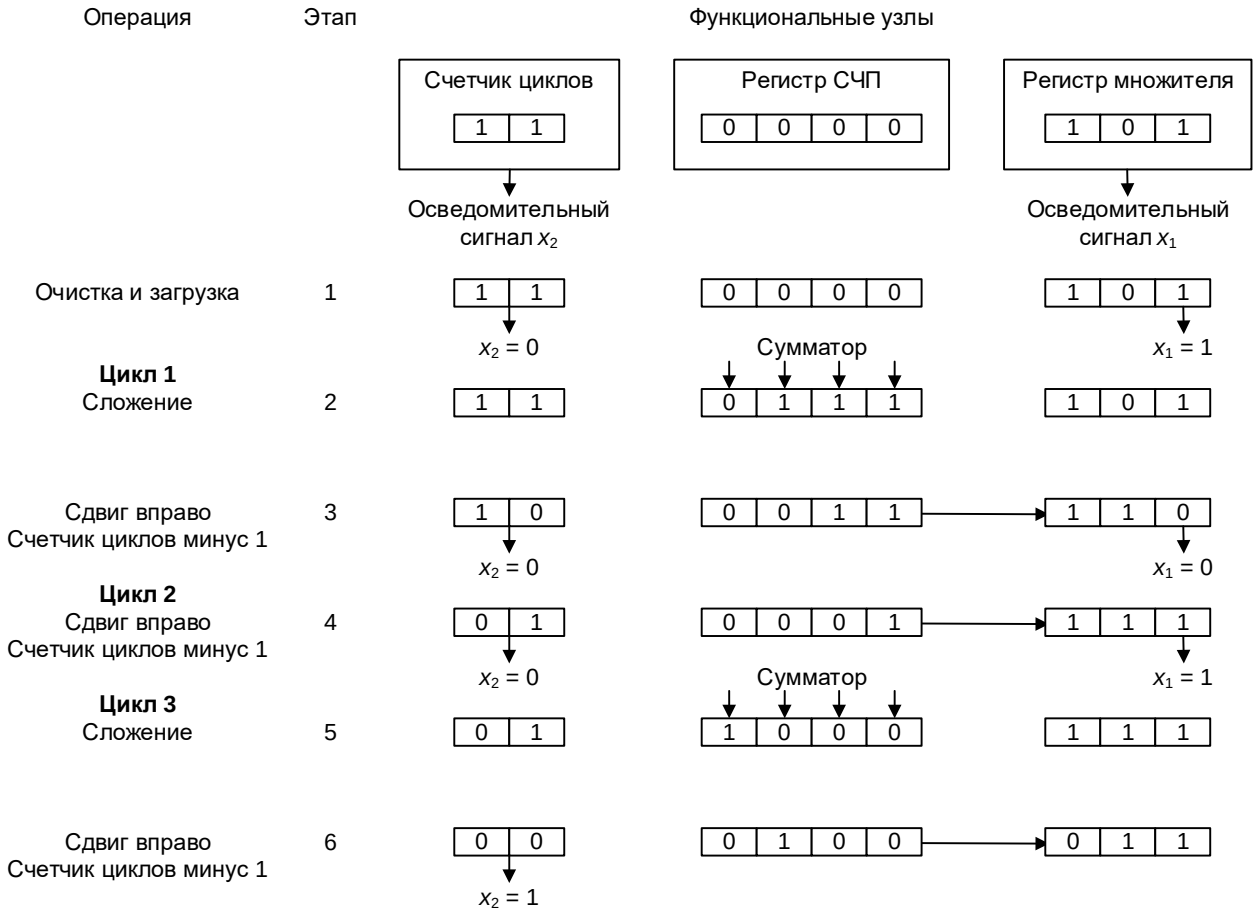

Рис. 2.3. Поэтапный процесс выполнения умножения

На этапе 1 выполняется очистка регистра СЧП, а также загрузка регистров множимого, множителя и счетчика циклов. Далее начинается *первый цикл* умножения, который включает два этапа. На этапе 2 суммируется содержимое регистра СЧП 0000 и регистра множимого, так как осведомительный сигнал x_1 (МЗР

множителя) равен единице. В результате в регистре СЧП получается первая сумма частичных произведений. На этапе 3 содержимое регистра СЧП и регистра множителя (при сдвиге они рассматриваются как единый регистр) сдвигается на один разряд вправо. При этом уходит из регистра и теряется единица крайнего правого разряда множителя. На этом этапе содержимое счетчика циклов становится равным 10, поэтому осведомительный сигнал $x_2 = 0$, и, следовательно, можно начинать следующий цикл умножения, т. е. процесс умножения еще не закончен и его надо продолжать. *Второй цикл* умножения включает только один этап, на котором содержимое регистра СЧП и регистра множителя сдвигаются на один разряд вправо, а содержимое счетчика циклов уменьшается на единицу. При этом уходит из регистра и теряется бит разряда двоек множителя. В этом цикле отсутствует операция сложения содержимого регистра СЧП и регистра множителя. Это связано с тем, что при переходе к циклу 2 осведомительный сигнал $x_1 = 0$. Это означает, что второй разряд множителя равен 0. В этом случае к содержимому регистра СЧП необходимо было бы добавить код 000. Но эта операция не изменяет содержимого регистра СЧП, поэтому ее выполнять не нужно. В этом цикле содержимое счетчика циклов становится равным 01, так что осведомительный сигнал $x_2 = 0$. Поэтому осуществляется переход к *циклу 3*, в котором выполняются операции, аналогичные первому циклу. В этом цикле умножение заканчивается, так как содержимое счетчика циклов становится равным 00 (осведомительный сигнал $x_2 = 1$). При этом младшие три разряда регистра СЧП и регистр множителя содержат полное произведение 100011 (десятичное число 35).

Процесс проектирования арифметического устройства рассмотрим на примере умножителя, выполняющего умножение двух 8-разрядных двоичных чисел со знаком в прямом коде. Множимое (*Мн*) и множитель (*Мт*) перед началом умножения могут быть представлены как в прямом, так и в дополнительном или обратном коде. Однако умножение чисел в прямом коде выполняется несколько проще, чем в дополнительном и обратном. При умножении чисел в прямом коде перемножаются модули множимого и множителя. Знак произведения определяется с помощью операции СЛОЖЕНИЕ ПО МОДУЛЮ ДВА знаковых разрядов сомножителей и приписывается произведению после завершения перемножения модулей *Мн* и *Мт*.

На рис 2.4 показан интерфейс разрабатываемого умножителя.

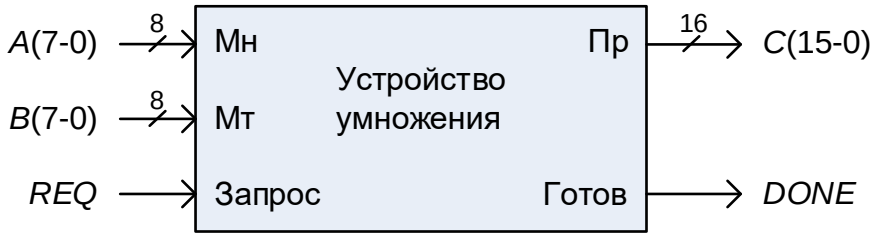

Рис. 2.4. Интерфейс умножителя

Умножитель имеет две входные 8-разрядные шины *A* и *B* для приема соответственно множимого и множителя и 16-разрядную выходную шину *C* для выдачи произведения (*Пр*). Передача множимого и множителя сопровождается сигналом запроса *REQ*, по которому осуществляется их прием в умножитель и происходит запуск процесса умножения. По окончании процесса умножения и выдачи произведения на выходную шину умножителем формируется сигнал готовности *DONE*.

На рис. 2.5 приведена функциональная схема операционного автомата разрабатываемого умножителя.

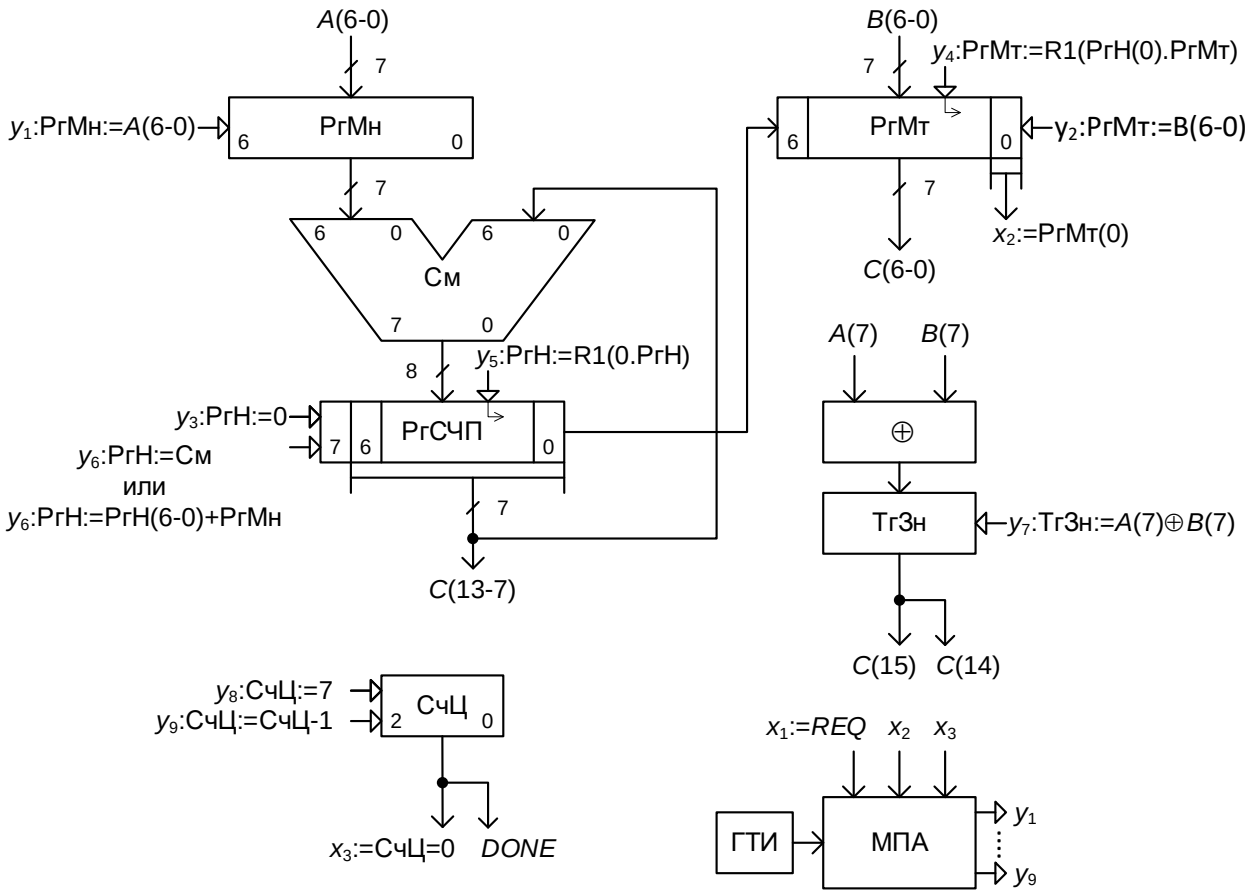

Рис. 2.5. Функциональная схема операционного автомата умножителя

Операционный автомат включает 7-разрядные регистр множимого и регистр множителя, а также 8-разрядный регистр СЧП. Старшие (знаковые) разряды множимого и множителя по линиям *A*(7) и *B*(7) поступают на входы элемента СЛОЖЕНИЕ ПО МОДУЛЮ ДВА. На выходе этого элемента формируется знак произведения, который фиксируется в специальном триггере знака (*ТгЗн*), откуда поступает на соответствующие линии выходной шины. Промежуточные СЧП формируются 7-разрядным сумматором с выходным переносом и фиксируются в регистре СЧП. Регистр СЧП и регистр множителя связаны цепью переноса и рассматриваются при сдвиге как единый регистр. Для подсчета числа циклов умножения в состав умножителя входит 3-разрядный счетчик циклов (*СчЦ*).

На рис. 2.6 приведена микропрограмма работы умножителя, разработанная в соответствии с функциональной схемой (см. рис. 2.5) и выбранным методом выполнения операции умножения.

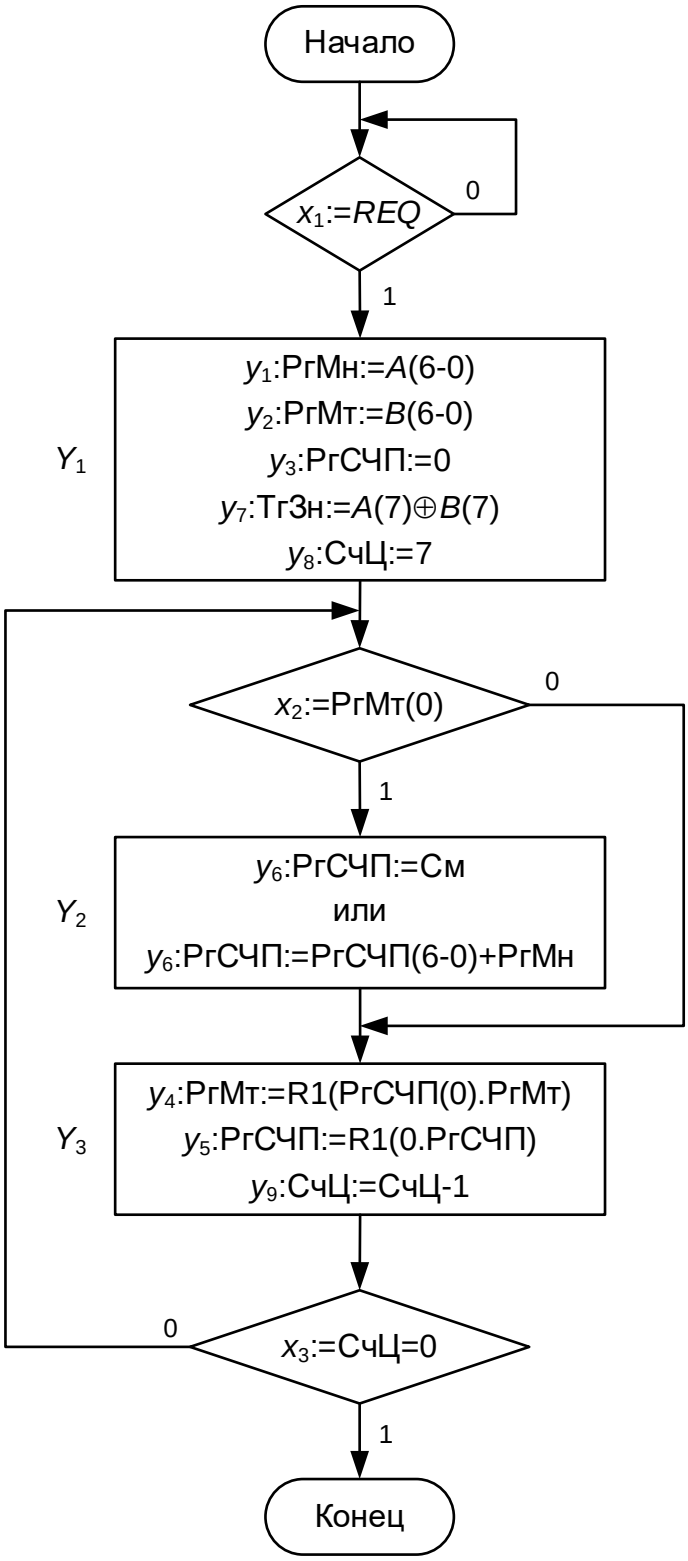

Рис. 2.6. Микропрограмма работы умножителя

2.2. Этапы разработки управляющей части арифметического устройства

Проектирование управляющей части арифметического устройства будем осуществлять каноническим методом структурного синтеза микропрограммного автомата (МПА).

На этапе структурного синтеза принято представлять МПА в виде двух частей: памяти и комбинационной схемы (рис. 2.7).

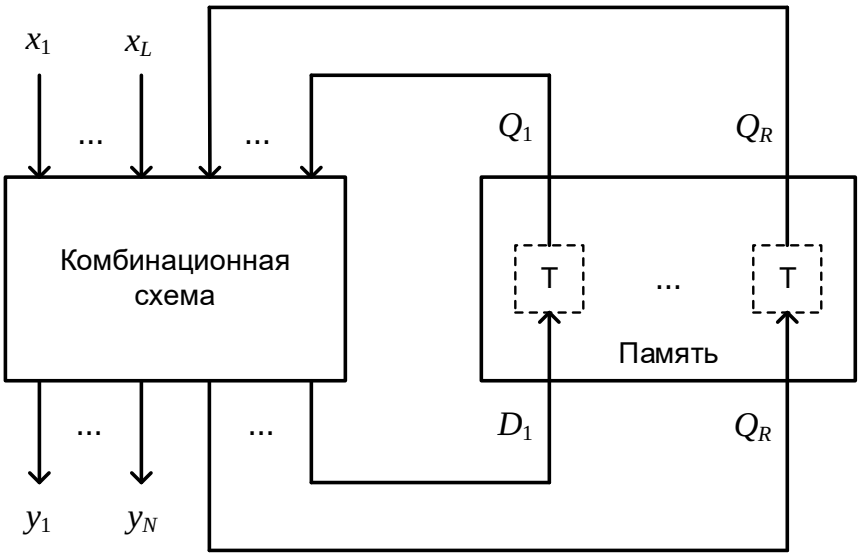

Рис. 2.7. Представление управляющей части устройства в виде композиции памяти и комбинационной схемы

Память автомата состоит из предварительно выбранных элементов памяти. В качестве элементов памяти используются различного типа триггеры.

После выбора элементов памяти (типа триггеров) синтез управляющего автомата сводится к синтезу комбинационной схемы, реализующей функции

$$y_n = y_n(Q_1, \dots, Q_R, x_1, \dots, x_L), \;\; n = 1, \dots, N,$$
$$D_r = D_r(Q_1, \dots, Q_R, x_1, \dots, x_L), \;\; r = 1, \dots, R, \qquad (2.1)$$

где $Q_1, \dots, Q_R$ – функции обратной связи от памяти автомата к комбинационной схеме; D_r – функции возбуждения элементов памяти автомата.

Порядок разработки управляющей части арифметического устройства (синтеза микропрограммного автомата) состоит из следующих шагов:

1) разработать микропрограмму работы операционного блока в виде содержательной граф-схемы алгоритма;

2) составить список микроопераций $y_n(n = 1, \dots, N)$ и логических условий $x_l(l = 1, \dots, L)$;

3) заменив микрооперации и логические условия в содержательной ГСА соответствующими символами y_n и x_l, получить ГСА;

4) выполнить разметку ГСА по следующим правилам (на примере разметки состояний для автомата Мили):

– вход вершины, следующей за начальной, а также вход конечной вершины отметить состоянием a_0;

– входы всех вершин, следующих за операторными, отметить состояниями $a_1, a_2, \dots, a_{Q-1}$;

5) составить список всех путей переходов в ГСА. Для этого в отмеченной ГСА ищутся все пути переходов из одной отметки в другую, содержащие точно одну операторную вершину. Число условных вершин в таком пути может быть произвольным, в том числе и нулевым;

6) построить таблицу переходов МПА. Таблица включает пять столбцов:

– a_m – исходное состояние;

– a_s – состояние перехода;

– $X(a_m, a_s)$ – логические условия на переходе МПА из состояния a_m в состояние a_s;

– $Y(a_m, a_s)$ – выходные сигналы (микрооперации) на переходе МПА из состояния a_m в состояние a_s;

– h – номер строки (перехода). Каждая строка таблицы переходов МПА соответствует одному пути перехода в ГСА;

7) выбрать элементы памяти и определить их количество $R = [\log_2 Q]$;

8) закодировать состояния автомата $K(a_q), q = 0, \dots, Q-1$;

9) перейти к структурной таблице МПА. Эта таблица получается из таблицы переходов МПА путем добавления трех столбцов:

– $K(a_m)$ – коды состояний a_m;

– $K(a_s)$ – коды состояний a_s;

– $F(a_m, a_s)$ – множество функций возбуждения элементов памяти на переходе МПА из состояния a_m в состояние a_s;

10) по структурной таблице МПА записать выражения для функций $F_r(r = 0, \dots, R-1)$ и $y_n(n = 0, \dots, N-1)$. Эти выражения получаются как дизъюнкции выражений вида $A_m X(a_m, a_s)$, где A_m – конъюнкция, соответствующая коду состояния a_m, взятых по всем строкам структурной таблицы МПА, в которых записаны y_n или $F_r = 1$;

11) по полученным выражениям построить комбинационную схему МПА.

Ниже приведен пример синтеза МПА для проектируемого умножителя.

1. Из содержательной граф-схемы алгоритма (см. рис. 2.6) по приведенным выше правилам получаем размеченную ГСА, которая показана на рис. 2.8. ГСА имеет четыре состояния, для которых можно сформировать восемь переходов, приведенных в таблице переходов МПА (табл. 2.1).

2. Определяем число элементов памяти $R = \log_2 M$, где M – число состояний МПА $R = \log_2 4 = 2$.

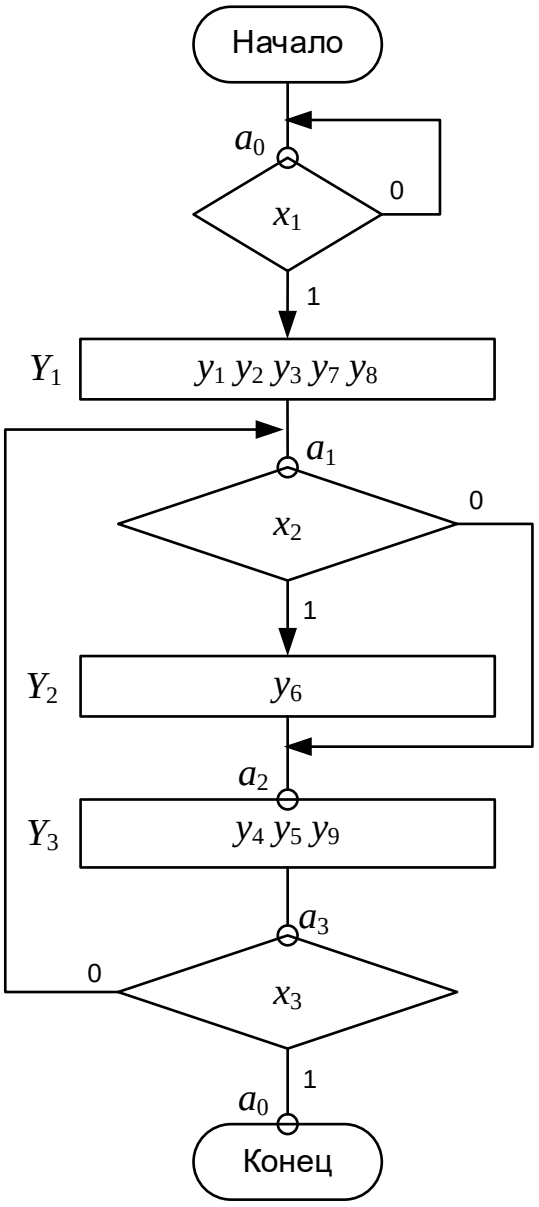

Рис. 2.8. ГСА МПА управляющей части устройства умножения

Таблица 2.1

Таблица переходов МПА

h	a_m	a_s	X_h	y_n	Y_t
1	a_0	a_0	$\overline{x}_1$	–	–
2		a_1	x_1	$y_1y_2y_3y_7y_8$	Y_1
3	a_1	a_2	x_2	y_6	Y_2
4		a_3	$\overline{x}_2$	$y_4y_5y_9$	Y_3
5	a_2	a_3	1	$y_4y_5y_9$	Y_3
6	a_3	a_0	x_3	–	–
7		a_2	$x_2\overline{x}_3$	y_6	Y_2
8		a_3	$\overline{x}_2\overline{x}_3$	$y_4y_5y_9$	Y_3

3. Кодируем состояния МПА. Поскольку ГСА имеет всего четыре состояния, удобно код состояния назначить в соответствии с его номером (табл. 2.2).

Таблица 2.2

Коды состояний МПА

Состояние	T_1	T_2
a_0	0	0
a_1	0	1
a_2	1	0
a_3	1	1

4. Выбираем тип триггеров, которые будем использовать в качестве элементов памяти МПА. Для этих целей можно использовать либо *JK*-, либо *D*-триггеры, таблица переходов которых представлена ниже (табл. 2.3).

Таблица 2.3

Объединенная таблица

переходов *JK*- и *D*-триггеров

Переход	*J*	*K*	*D*
0→0	0	*	0
0→1	1	*	1
1→0	*	1	0
1→1	*	0	1

5. Строим структурную таблицу МПА (табл. 2.4). В таблице приведены сигналы управления как для *JK*-, так и для *D*-триггеров.

Таблица 2.4

Структурная таблица МПА

h	a_m	$K(a_m)$	a_s	$K(a_s)$	X_h	y_n	Y_t	J_1	K_1	J_2	K_2	D_1	D_2
1	a_0	00	a_0	00	$\overline{x}_1$	–	–	0	*	0	*	0	0
2			a_1	01	x_1	$y_1y_2y_3y_7y_8$	Y_1	0	*	1	*	0	1
3	a_1	01	a_2	10	x_2	y_6	Y_2	1	*	*	1	1	0
4			a_3	11	$\overline{x}_2$	$y_4y_5y_9$	Y_3	1	*	*	0	1	1
5	a_2	10	a_3	11	1	$y_4y_5y_9$	Y_3	*	0	1	*	1	1
6	a_3	11	a_0	00	x_3	–	–	*	1	*	1	0	0
7			a_2	10	$x_2\overline{x}_3$	y_6	Y_2	*	0	*	1	1	0
8			a_3	11	$\overline{x}_2\overline{x}_3$	$y_4y_5y_9$	Y_3	*	0	*	0	1	1

6. По структурной таблице записываем систему канонических уравнений МПА:

1) сигналы управления функциональными блоками операционной части умножителя:

$$\begin{aligned} Y_1 &= \overline{Q}_1\overline{Q}_2x_1, \\ Y_2 &= \overline{Q}_1Q_2x_2 \vee Q_1Q_2x_2\overline{x}_3, \\ Y_3 &= \overline{Q}_1Q_2\overline{x}_2 \vee Q_1\overline{Q}_2 \vee Q_1Q_2\overline{x}_2\overline{x}_3; \end{aligned} \qquad (2.2)$$

2) сигналы управления триггерами:

– для *JK*-триггеров:

$$\begin{aligned} J_1 &= \overline{Q}_1Q_2x_2 \vee \overline{Q}_1Q_2\overline{x}_2 = \overline{Q}_1Q_2, \\ K_1 &= Q_1Q_2x_3, \\ J_2 &= \overline{Q}_1\overline{Q}_2x_1 \vee Q_1\overline{Q}_2, \\ K_2 &= \overline{Q}_1Q_2x_2 \vee Q_1Q_2x_3 \vee Q_1Q_2x_2\overline{x}_3; \end{aligned} \qquad (2.3)$$

– для *D*-триггеров:

$$\begin{aligned} D_1 &= \overline{Q}_1Q_2 \vee Q_1\overline{Q}_2 \vee Q_1Q_2\overline{x}_3, \\ D_2 &= \overline{Q}_1\overline{Q}_2x_1 \vee \overline{Q}_1Q_2\overline{x}_2 \vee Q_1\overline{Q}_2 \vee Q_1Q_2\overline{x}_2\overline{x}_3. \end{aligned} \qquad (2.4)$$

7. По полученным логическим выражениям строим функциональную схему управляющей части устройства умножения.

2.3. Порядок выполнения работы

1. Получить у преподавателя номер варианта задания.
2. Определить исходные данные на разработку устройства умножения.
3. Составить блок-схему алгоритма выполнения операции умножения.
4. Разработать структурную схему устройства умножения.
5. Построить функциональную схему операционной части устройства умножения.
6. Разработать микропрограмму работы устройства умножения.
7. Составить список микроопераций и логических условий.
8. Отметить на функциональной схеме операционной части управляющие и осведомительные сигналы, соответствующие микрооперациям и логическим условиям.
9. Построить ГСА работы устройства умножения в соответствии с разработанной операционной частью.

10. Выполнить разметку ГСА.

11. Составить список всех путей перехода в ГСА.

12. Построить таблицу переходов МПА.

13. Закодировать состояния МПА.

14. Построить структурную таблицу МПА.

15. Записать выражения для функций возбуждения элементов памяти автомата и функций выходов.

16. Построить функциональную схему управляющей части устройства умножения.

17. Оформить отчет.

2.4. Содержание отчета

1. Титульный лист.
2. Цель работы.
3. Краткие сведения по методике проектирования арифметических устройств.
4. Описание процесса разработки операционной части устройства умножения:

а) блок-схема алгоритма выполнения заданного варианта операции умножения;

б) структурная схема устройства умножения;

в) функциональная схема операционной части устройства умножения;

г) микропрограмма работы устройства умножения в виде содержательной ГСА.

5. Описание процесса разработки управляющей части устройства умножения:

а) размеченная ГСА работы устройства умножения в соответствии с разработанной операционной частью;

б) список всех путей перехода в ГСА;

в) таблица переходов МПА;

г) структурная таблица МПА;

д) система логических выражений для сигналов управления функциональными блоками операционной части умножителя и сигналов управления триггерами;

е) функциональная схема управляющей части устройства умножения.

6. Выводы по работе.

2.5. Варианты задания

Разработать операционный и управляющий автоматы умножителя, выполняющего операцию умножения двух 8-разрядных двоичных чисел со знаком способом сложений со сдвигами, в соответствии с вариантом задания, выданным преподавателем (табл. 2.5, рис. 2.9–2.14). Методы выполнения операции умножения приведены на рис. 2.15.

Таблица 2.5

Варианты задания

Номер варианта задания	Вариант выполнения операции умножения (см. рис. 2.15)	Вариант загрузки операндов и выдачи результата выполнения операции умножения (см. рис. 2.9–2.14)
1	Умножение в прямых кодах, метод 1	Вариант 1 (рис. 2.9)
2	Умножение в прямых кодах, метод 2	Вариант 2 (рис. 2.10)
3	Умножение в прямых кодах, метод 3	Вариант 3 (рис. 2.11)
4	Умножение в прямых кодах, метод 4	Вариант 4 (рис. 2.12)
5	Умножение в дополнительных кодах, метод 1	Вариант 5 (рис. 2.13)
6	Умножение в дополнительных кодах, метод 2	Вариант 6 (рис. 2.14)
7	Умножение в дополнительных кодах, метод 3	Вариант 1 (рис. 2.9)
8	Умножение в дополнительных кодах, метод 4	Вариант 2 (рис. 2.10)
9	Умножение в обратных кодах, метод 1	Вариант 3 (рис. 2.11)
10	Умножение в обратных кодах, метод 2	Вариант 4 (рис. 2.12)
11	Умножение в обратных кодах, метод 3	Вариант 5 (рис. 2.13)
12	Умножение в обратных кодах, метод 4	Вариант 6 (рис. 2.14)

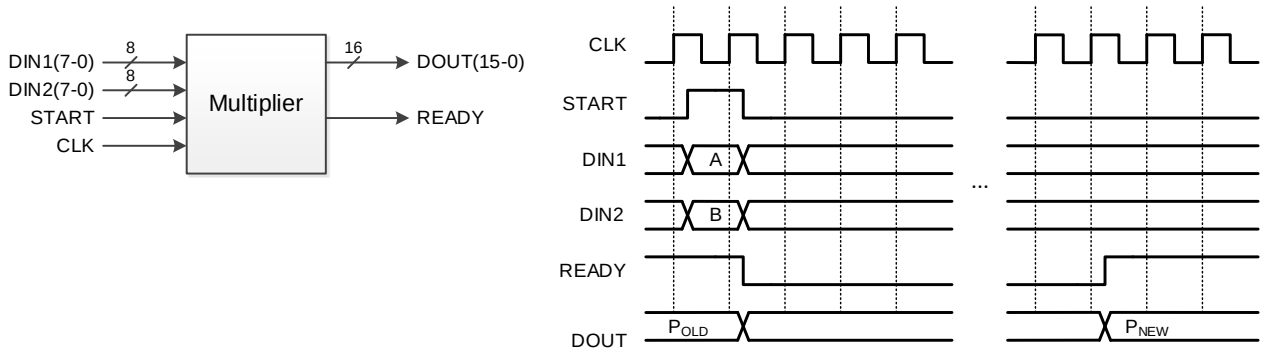

Рис. 2.9. Устройство умножения с параллельной загрузкой операндов (вариант 1)

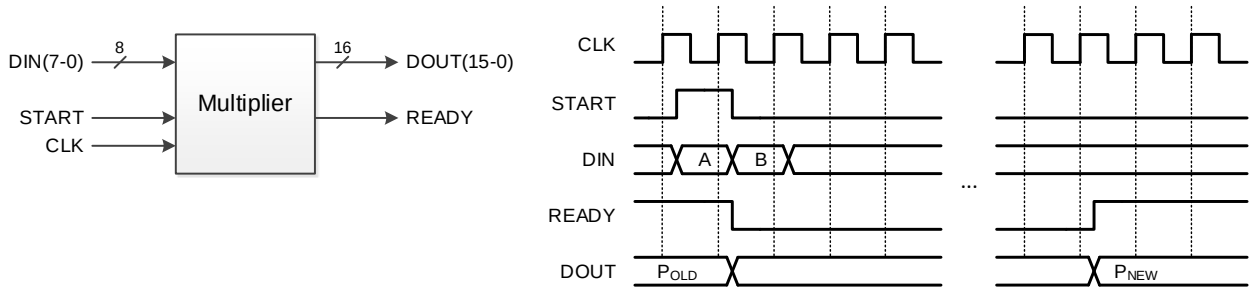

Рис. 2.10. Устройство умножения с последовательной загрузкой операндов (вариант 2)

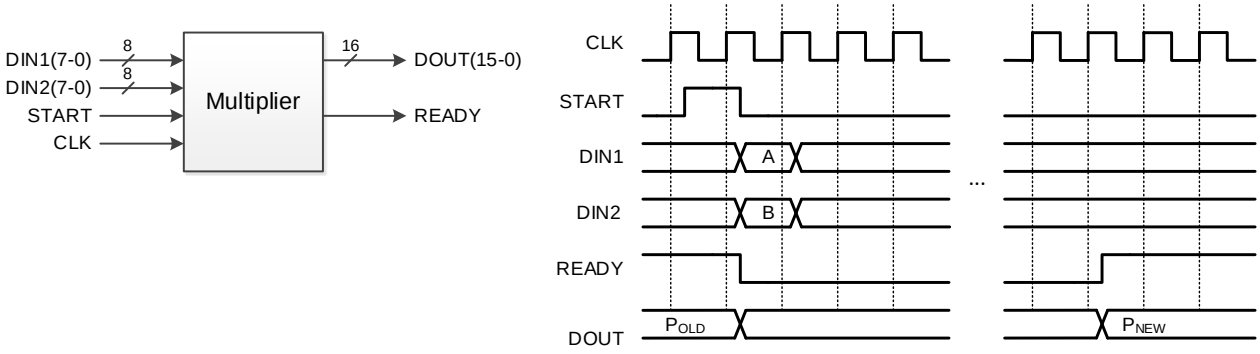

Рис. 2.11. Устройство умножения с задержанной параллельной загрузкой операндов (вариант 3)

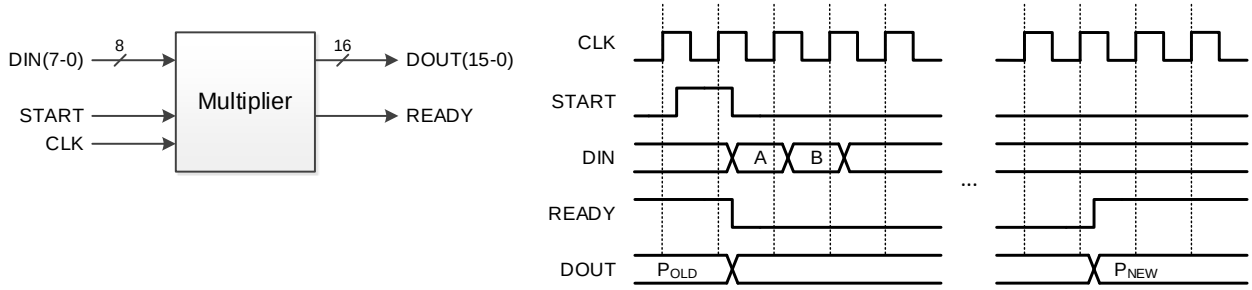

Рис. 2.12. Устройство умножения с задержанной последовательной загрузкой операндов (вариант 4)

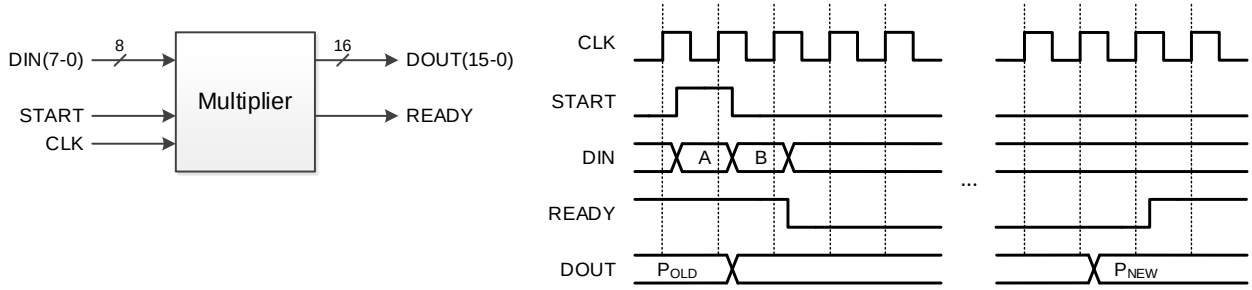

Рис. 2.13. Устройство умножения с последовательной загрузкой операндов (вариант 5)

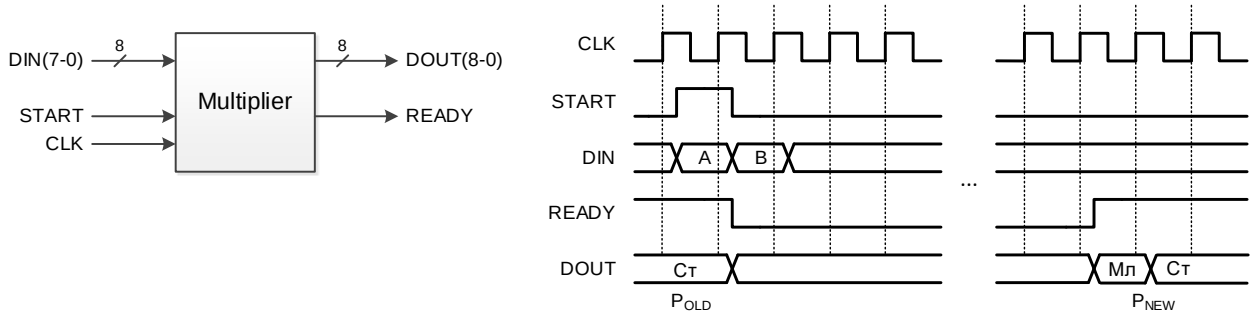

Рис. 2.14. Устройство умножения с последовательной загрузкой операндов (вариант 6)

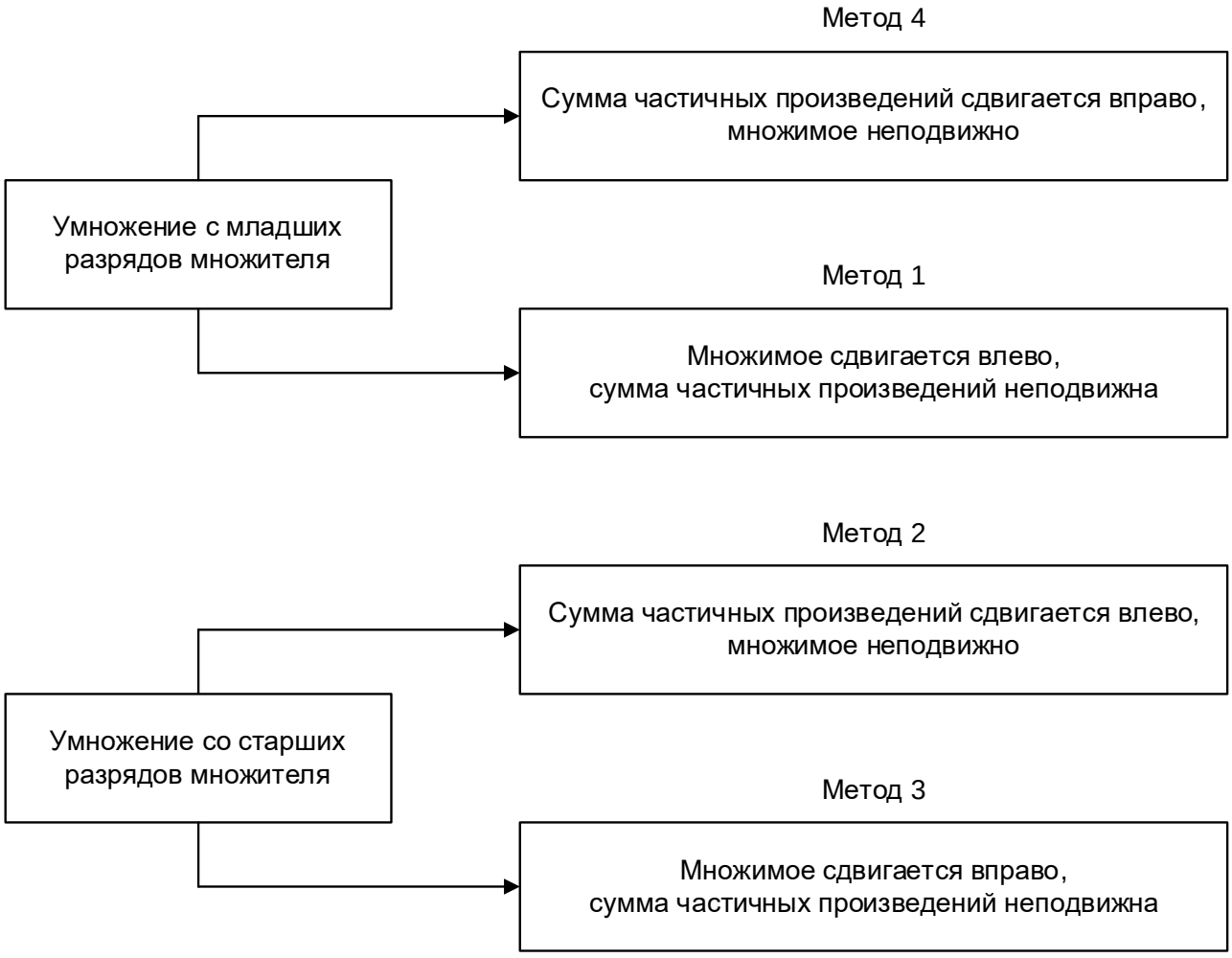

Рис. 2.15. Методы выполнения операции умножения

ЛАБОРАТОРНАЯ РАБОТА № 3.
РАЗРАБОТКА ПРОЕКТА ОПЕРАЦИОННОЙ ЧАСТИ АРИФМЕТИЧЕСКОГО УСТРОЙСТВА С ИСПОЛЬЗОВАНИЕМ ИНТЕГРИРОВАННОЙ СРЕДЫ РАЗРАБОТКИ

Цель работы – получение практических навыков по разработке проекта операционной части арифметического устройства с использованием интегрированной среды разработки Xilinx ISE; получение практических навыков моделирования проекта операционной части арифметического устройства в среде проектирования Xilinx ISE.

3.1. Создание тестового модуля проекта и функциональное моделирование проектируемого устройства

В процессе создания проекта арифметического устройства с использованием среды проектирования Xilinx ISE формируют несколько моделей разрабатываемого устройства. После создания модулей исходного описания устройства генерируется поведенческая модель, которая позволяет выполнить функциональную проверку проекта. На этой стадии проектирования отсутствует информация о значениях задержек распространения сигналов, поэтому при функциональном моделировании можно выявить только логические и синтаксические ошибки в описании разрабатываемого устройства.

Для того чтобы выполнить функциональную проверку проекта, необходимо разработать тестовый модуль. Тестовый модуль проекта представляет собой модель испытательного стенда для разрабатываемого устройства на языке описания аппаратуры HDL (Hardware Description Language), используемом в процессе проектирования. В лабораторных работах предполагается, что процесс разработки осуществляется с использованием языка описания аппаратуры VHDL. В этом случае тестовый модуль проекта имеет стандартную структуру VHDL-описания, которая включает в себя следующие элементы:

– ссылки на используемые библиотеки и пакеты;

– описание интерфейса объекта (ENTITY);

– описание архитектуры объекта;

– декларацию компонента, представляющего модуль описания верхнего уровня иерархии проектируемого устройства;

– декларацию сигналов, используемых для подачи входных тестовых воздействий и контроля выходных реакций моделируемого устройства;

– выражение создания экземпляра компонента с подключением соответствующих сигналов;

– формирование поведения входных тестовых сигналов.

Поскольку в данном случае под объектом описания понимается модель испытательного стенда в целом, то он не имеет интерфейсных сигналов (портов). Архитектура этого объекта содержит единственный компонент, представляющий разрабатываемое устройство.

В примере 3.1 приведен вариант VHDL-модуля операционной части устройства умножения, проектирование которого рассмотрено в лабораторной работе № 2, а в примере 3.2 – вариант VHDL-модуля тестовых воздействий для этого проекта.

VHDL-модуль операционной части устройства умножения содержит описания всех операционных узлов функциональной схемы операционной части умножителя (см. рис. 2.5 на с. 20), а также всех связей между ними.

Пример 3.1. VHDL-модуль операционной части устройства умножения

```
--------------------------------------------------------------------------------
-- Description:   Модуль реализует операционную часть устройства умножения
--                - умножаются два 8-разрядных двоичных числа со знаком
--                  в прямом коде, произведение 16-разрядное
--                  в модифицированном коде
--                - метод умножения: с младших разрядов множителя,
--                  сумма частичных произведений сдвигается вправо,
--                  множимое неподвижно
--                - вариант загрузки операндов: задержанная последовательная
--------------------------------------------------------------------------------
-- Structure:     top_mult
--                - mult
--                  - mult_oper
--                  - mult_control
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity mult_oper is
port (
-- входные сигналы
CLK               : in STD_LOGIC;   -- сигнал синхронизации
RESET             : in STD_LOGIC;   -- сброс (активный - высокий)
MN_IN             : in std_logic_vector (7 downto 0);       -- множимое
MT_IN             : in std_logic_vector (7 downto 0);       -- множитель
Y_1_IN            : in std_logic;   -- микрокоманда 1
Y_2_IN            : in std_logic;   -- микрокоманда 2
Y_3_IN            : in std_logic;   -- микрокоманда 3
-- выходные сигналы
MT_0_OUT          : out std_logic;  -- младший разряд множителя
CNT_STEP_0_OUT    : out std_logic;  -- флаг: счетчик циклов умножения равен 0
PR_OUT            : out std_logic_vector (15 downto 0)      -- произведение
);
end mult_oper;

architecture implementation of mult_oper is

-- internal signals and registers
signal rg_mn      : std_logic_vector (6 downto 0) := (others => '0'); -- РгМн
signal rg_mt      : std_logic_vector (6 downto 0) := (others => '0'); -- РгМт
```

Продолжение примера 3.1

```
signal rg_sum_pr  : std_logic_vector (7 downto 0) := (others => '0'); -- РгСЧП
signal sum        : std_logic_vector (7 downto 0) := (others => '0'); -- См
signal ff_sign    : std_logic := '0';     -- ТгЗн
signal cnt_step   : std_logic_vector (2 downto 0) := (others => '0'); -- СчЦ
signal cnt_step_0 : std_logic := '0';     -- флаг: СчЦ = 0

begin

-- регистр множимого (7 разрядов)
process (CLK) is
begin
      if (CLK'EVENT and CLK = '1') then        -- rising clock edge
            if Y_1_IN = '1' then
                  rg_mn <= MN_IN(6 downto 0);
            end if;
      end if;
end process;

-- регистр множителя (7 разрядов)
process (CLK) is
begin
      if (CLK'EVENT and CLK = '1') then        -- rising clock edge
            if Y_1_IN = '1' then
                  rg_mt <= MT_IN(6 downto 0);
            elsif Y_3_IN = '1' then
                  rg_mt <= rg_sum_pr(0) & rg_mt(6 downto 1);
            end if;
      end if;
end process;

MT_0_OUT <= rg_mt(0);

-- регистр суммы частичных произведений (8 разрядов)
process (CLK) is
begin
      if (CLK'EVENT and CLK = '1') then        -- rising clock edge
            if Y_1_IN = '1' then
                  rg_sum_pr <= (others => '0');
            elsif Y_2_IN = '1' then
                  rg_sum_pr <= sum;
            elsif Y_3_IN = '1' then
                  rg_sum_pr <= '0' & rg_sum_pr(7 downto 1);
            end if;
      end if;
end process;

-- сумматор
sum <= rg_sum_pr + ('0' & rg_mn);

-- триггер знака произведения
process (CLK) is
begin
      if (CLK'EVENT and CLK = '1') then        -- rising clock edge
            if Y_1_IN = '1' then
                  ff_sign <= MN_IN(7) xor MT_IN(7);
            end if;
      end if;
end process;
```

Продолжение примера 3.1

```
PR_OUT <= ff_sign & ff_sign & rg_sum_pr(6 downto 0) & rg_mt;

-- счетчик циклов алгоритма умножения (3 разряда)
process (CLK) is
begin
      if (CLK'EVENT and CLK = '1') then         -- rising clock edge
            if RESET = '1' then
                  cnt_step <= "000";
            elsif Y_1_IN = '1' then
                  cnt_step <= "111";
            elsif Y_3_IN = '1' then
                  cnt_step <= cnt_step - 1;
            end if;
      end if;
end process;

-- флаг: счетчик шагов умножения равен 0
cnt_step_0 <= '1' when cnt_step = "000" else '0';

CNT_STEP_0_OUT <= cnt_step_0;

end implementation;
```

VHDL-модуль тестовых воздействий для операционной части устройства умножения обеспечивает выдачу множимого и множителя в операционную часть, а также формирование последовательности управляющих сигналов микрокоманд в соответствии с микропрограммой (см. рис. 2.6 на с. 21) и двоичным кодом значащей части множителя.

Пример 3.2 разработан для значений множимого $10000010_{(2)} = -2_{(10)}$ и множителя $00000101_{(2)} = +5_{(10)}$.

Пример 3.2. VHDL-модуль тестовых воздействий для операционной части устройства умножения

```
--------------------------------------------------------------------------------
-- VHDL Test Bench for module: mult_oper
--------------------------------------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY test_mult_oper IS
END test_mult_oper;

ARCHITECTURE behavior OF test_mult_oper IS

-- Component Declaration for the Unit Under Test (UUT)
COMPONENT mult_oper is
port (
-- входные сигналы
CLK               : in STD_LOGIC;   -- сигнал синхронизации
RESET             : in STD_LOGIC;   -- сброс (активный - высокий)
MN_IN             : in std_logic_vector (7 downto 0);      -- множимое
MT_IN             : in std_logic_vector (7 downto 0);      -- множитель
Y_1_IN            : in std_logic;   -- микрокоманда 1
Y_2_IN            : in std_logic;   -- микрокоманда 2
Y_3_IN            : in std_logic;   -- микрокоманда 3
```

Продолжение примера 3.2

```
-- выходные сигналы
MT_0_OUT          : out std_logic;  -- младший разряд множителя
CNT_STEP_0_OUT    : out std_logic;  -- флаг: счетчик циклов умножения равен 0
PR_OUT            : out std_logic_vector (15 downto 0)      -- произведение
);
end COMPONENT;

--Inputs
signal clk              : std_logic := '0';
signal reset            : std_logic := '0';
signal mn_in            : std_logic_vector (7 downto 0) := (others => '0');
signal mt_in            : std_logic_vector (7 downto 0) := (others => '0');
signal y_1_in           : std_logic := '0';
signal y_2_in           : std_logic := '0';
signal y_3_in           : std_logic := '0';
--Outputs
signal mt_0_out         : std_logic;
signal cnt_step_0_out   : std_logic;
signal pr_out           : std_logic_vector (15 downto 0);

-- Clock period definitions
constant clk_period : time := 20 ns;

BEGIN

-- Instantiate the Unit Under Test (UUT)
uut: mult_oper
PORT MAP (
      CLK               => clk,
      RESET             => reset,
      MN_IN             => mn_in,
      MT_IN             => mt_in,
      Y_1_IN            => y_1_in,
      Y_2_IN            => y_2_in,
      Y_3_IN            => y_3_in,

      MT_0_OUT          => mt_0_out,
      CNT_STEP_0_OUT    => cnt_step_0_out,
      PR_OUT            => pr_out
);

-- Clock process definitions
clk_process :process
begin
      clk <= '1';
      wait for clk_period/2;
      clk <= '0';
      wait for clk_period/2;
end process;

-- Stimulus process
stim_proc: process
begin
-- hold reset state for 200 ns
      wait for 200 ns;
      wait for clk_period*0.05;          -- + 0.05 периода;
      reset <= '1';
      wait for clk_period;
```

Продолжение примера 3.2

```
      reset <= '0';
      wait for clk_period*2;
-- выдача множимого и множителя
      mn_in <= "10000010";    -- Мн = -2
      wait for clk_period;
      mt_in <= "00000101";    -- Мт = +5
-- формирование управляющих сигналов микрокоманд
      y_1_in <= '1';
      wait for clk_period;
      y_1_in <= '0';
-- mt(0)=1
      y_2_in <= '1';
      wait for clk_period;
      y_2_in <= '0';
      y_3_in <= '1';
      wait for clk_period;
-- mt(1)=0
      wait for clk_period;
      y_3_in <= '0';
-- mt(2)=1
      y_2_in <= '1';
      wait for clk_period;
      y_2_in <= '0';
      y_3_in <= '1';
      wait for clk_period;
-- mt(3)=0
      wait for clk_period;
-- mt(4)=0
      wait for clk_period;
-- mt(5)=0
      wait for clk_period;
-- mt(6)=0
      wait for clk_period;
      y_3_in <= '0';
      wait;
end process;

END;
```

На рис. 3.1 приведены результаты функционального моделирования операционной части устройства умножения для рассматриваемого проекта в среде проектирования для базового кристалла ПЛИС (микросхема XC3S200) отладочной платы Spartan-3 Starter Kit.

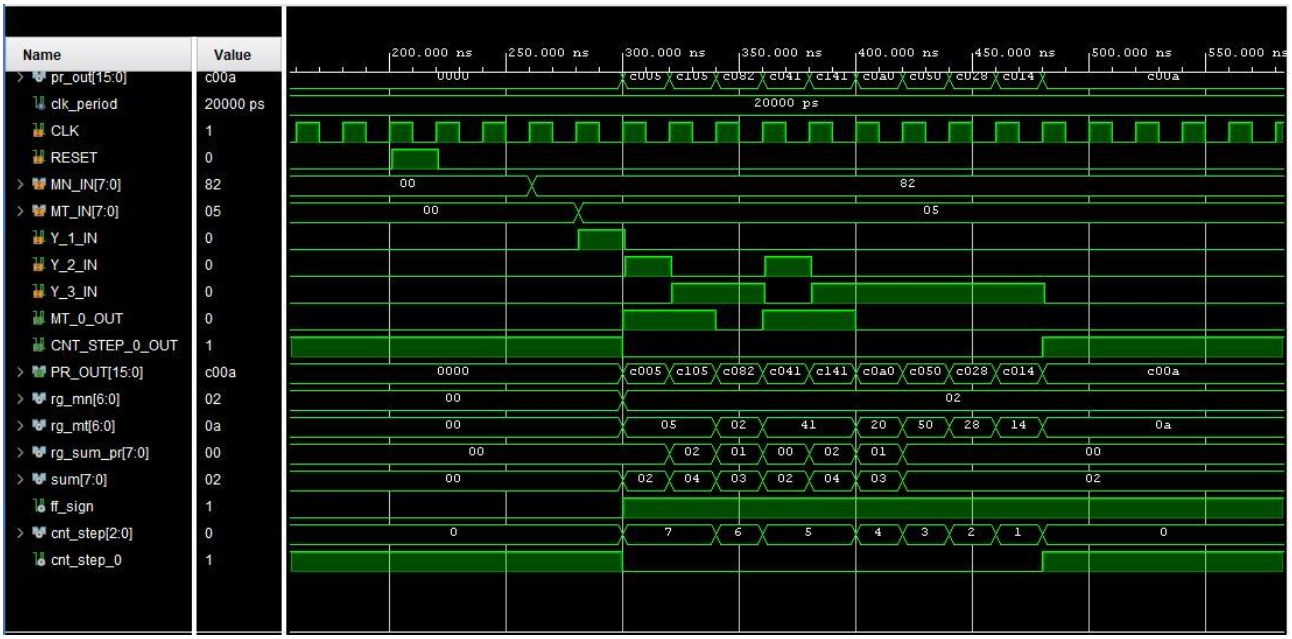

Рис. 3.1. Временная диаграмма функционального моделирования операционной части устройства умножения

3.2. Порядок выполнения работы

1. На основе функциональной схемы операционного части устройства умножения, разработанной в лабораторной работе № 2, создать проект операционной части умножителя в виде принципиальной схемы в схемотехническом редакторе или путем описания на языке VHDL.
2. Выполнить функциональное моделирование созданного проекта в среде проектирования Xilinx ISE.
3. Оформить отчет.

3.3. Содержание отчета

1. Титульный лист.
2. Цель работы.
3. Описание процесса создания проекта операционной части устройства умножения в схемотехническом редакторе или путем описания на языке VHDL.
4. Описание процесса создания VHDL-модуля тестовых воздействий для операционной части устройства умножения.
5. Описание процесса функционального моделирования проекта операционной части устройства умножения в среде проектирования Xilinx ISE.
6. Выводы по работе.

ЛАБОРАТОРНАЯ РАБОТА № 4.
РАЗРАБОТКА ПРОЕКТА УПРАВЛЯЮЩЕЙ ЧАСТИ АРИФМЕТИЧЕСКОГО УСТРОЙСТВА С ИСПОЛЬЗОВАНИЕМ ИНТЕГРИРОВАННОЙ СРЕДЫ РАЗРАБОТКИ

Цель работы – получение практических навыков по разработке проекта управляющей части арифметического устройства с использованием интегрированной среды разработки Xilinx ISE; получение практических навыков моделирования проекта управляющей части арифметического устройства в среде проектирования Xilinx ISE.

4.1. Создание тестового модуля проекта и функциональное моделирование проектируемого устройства

В примере 4.1 приведен вариант VHDL-модуля управляющей части устройства умножения, проектирование которого рассмотрено в лабораторной работе № 2, а в примере 4.2 – вариант VHDL-модуля тестовых воздействий для этого проекта.

Управляющая часть устройства умножения представляет собой конечный автомат. Типичное описание на VHDL конечного автомата включает в себя два процесса. В первом процессе определяются состояние перехода и значения выходных сигналов. Во втором процессе выполняется фиксация сигналов состояния перехода с выхода первого процесса в элементах памяти, реализованных на *JK*- или *D*-триггерах. Формирование значений сигналов в первом процессе обычно выполняется в виде оператора **`case`**, селектором которого выступают сигналы, представляющие текущее состояние автомата. Поскольку основной целью данной лабораторной работы является проверка правильности синтеза МПА, то в примере 4.1 для формирования значений сигналов состояния перехода и выходных сигналов используются операторы назначения сигналов непосредственно в соответствии с логическими выражениями, образующими систему канонических уравнений МПА.

Пример 4.1. VHDL-модуль управляющей части устройства умножения

```
----------------------------------------------------------------------------
-- Description:   Модуль реализует управляющую часть устройства умножения
--                - умножаются два 8-разрядных двоичных числа со знаком
--                  в прямом коде, произведение 16-разрядное
--                  в модифицированном коде
--                - метод умножения: с младших разрядов множителя,
--                  сумма частичных произведений сдвигается вправо,
--                  множимое неподвижно
--                - вариант загрузки операндов: задержанная последовательная
----------------------------------------------------------------------------
-- Structure:     top_mult
--                - mult
--                  - mult_oper
--                  - mult_control
```

Продолжение примера 4.1

```
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity mult_control is
port (
-- входные сигналы
CLK               : in STD_LOGIC;   -- сигнал синхронизации
RESET             : in STD_LOGIC;   -- сброс (активный - высокий)
REQ_IN            : in std_logic;   -- запрос на выполнение операции умножения
MT_0_IN           : in std_logic;   -- младший разряд множителя
CNT_STEP_0_IN     : in std_logic;   -- флаг: счетчик циклов умножения равен 0
-- выходные сигналы
Y_1_OUT           : out std_logic;  -- микрокоманда 1
Y_2_OUT           : out std_logic;  -- микрокоманда 2
Y_3_OUT           : out std_logic   -- микрокоманда 3
);
end mult_control;

architecture implementation of mult_control is

-- internal signals and registers
signal d1, d2           : std_logic := '0';          -- входы D-триггеров
signal q1, q2           : std_logic := '0';          -- выходы D-триггеров
signal y1, y2, y3       : std_logic := '0';          -- микрокоманды
begin

-- D-триггер 1
process (CLK) is
begin
      if (CLK'EVENT and CLK = '1') then        -- rising clock edge
            if RESET = '1' then
                  q1 <= '0';
            else
                  q1 <= d1;
            end if;
      end if;
end process;

d1 <= (not(q1) and q2) or (q1 and not(q2)) or
      (q1 and q2 and not(CNT_STEP_0_IN));

-- D-триггер 2
process (CLK) is
begin
      if (CLK'EVENT and CLK = '1') then        -- rising clock edge
            if RESET = '1' then
                  q2 <= '0';
            else
                  q2 <= d2;
            end if;
      end if;
end process;

d2 <= (not(q1) and not(q2) and REQ_IN) or (not(q1) and q2 and not(MT_0_IN)) or
      (q1 and not(q2)) or (q1 and q2 and not(MT_0_IN) and not(CNT_STEP_0_IN));
```

Продолжение примера 4.1

```
-- -- микрокоманды
y1 <= not(q1) and not(q2) and REQ_IN;
y2 <= (not(q1) and q2 and MT_0_IN) or (q1 and q2 and MT_0_IN and
not(CNT_STEP_0_IN));
y3 <= (not(q1) and q2 and not(MT_0_IN)) or (q1 and not(q2)) or (q1 and q2 and
not(MT_0_IN) and not(CNT_STEP_0_IN));

Y_1_OUT <= y1;
Y_2_OUT <= y2;
Y_3_OUT <= y3;

end implementation;
```

VHDL-модуль управляющей части устройства умножения содержит описания элементов памяти на *D*-триггерах и комбинационной схемы МПА в соответствии с логическими выражениями (2.2) и (2.4) из лабораторной работы № 2 (см. с. 26), а также всех связей между ними.

VHDL-модуль тестовых воздействий для управляющей части устройства умножения обеспечивает формирование последовательности осведомительных сигналов (сигналов, с помощью которых задаются значения логических условий) в соответствии с микропрограммой (см. рис. 2.6 на с. 21) и двоичным кодом значащей части множителя. Пример 4.2 разработан для значения множителя из лабораторной работы № 3 Мт = $00000101_{(2)} = +5_{(10)}$.

Пример 4.2. VHDL-модуль тестовых воздействий для управляющей части устройства умножения

```
--------------------------------------------------------------------------------
-- VHDL Test Bench for module: mult_control
--------------------------------------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY test_mult_control IS
END test_mult_control;

ARCHITECTURE behavior OF test_mult_control IS

-- Component Declaration for the Unit Under Test (UUT)
COMPONENT mult_control is
port (
-- входные сигналы
CLK               : in STD_LOGIC;   -- сигнал синхронизации
RESET             : in STD_LOGIC;   -- сброс (активный - высокий)
REQ_IN            : in std_logic;   -- запрос на выполнение операции умножения
MT_0_IN           : in std_logic;   -- младший разряд множителя
CNT_STEP_0_IN     : in std_logic;   -- флаг: счетчик циклов умножения равен 0
-- выходные сигналы
Y_1_OUT           : out std_logic;  -- микрокоманда 1
Y_2_OUT           : out std_logic;  -- микрокоманда 2
Y_3_OUT           : out std_logic   -- микрокоманда 3
);
end COMPONENT;
--Inputs
signal clk              : std_logic := '0';
```

Продолжение примера 4.2

```
signal reset            : std_logic := '0';
signal req_in           : std_logic := '0';
signal mt_0_in          : std_logic := '0';
signal cnt_step_0_in    : std_logic := '1';
--Outputs
signal y_1_out          : std_logic;
signal y_2_out          : std_logic;
signal y_3_out          : std_logic;

-- Clock period definitions
constant clk_period : time := 20 ns;

BEGIN

-- Instantiate the Unit Under Test (UUT)
uut: mult_control
PORT MAP (
      CLK               => clk,
      RESET             => reset,
      REQ_IN            => req_in,
      MT_0_IN           => mt_0_in,
      CNT_STEP_0_IN     => cnt_step_0_in,

      Y_1_OUT           => y_1_out,
      Y_2_OUT           => y_2_out,
      Y_3_OUT           => y_3_out
);

-- Clock process definitions
clk_process :process
begin
      clk <= '1';
      wait for clk_period/2;
      clk <= '0';
      wait for clk_period/2;
end process;
-- Stimulus process
stim_proc: process
begin
-- hold reset state for 200 ns
      wait for 200 ns;
      wait for clk_period*0.05;          -- + 0.05 периода;
      reset <= '1';
      wait for clk_period;
      reset <= '0';
      cnt_step_0_in <= '1';
      wait for clk_period*2;
-- формирование осведомительных сигналов в соответствии с микропрограммой
-- mt = "00000101";
      req_in <= '1';
      wait until (y_1_out = '1');
      wait until rising_edge(clk);
      cnt_step_0_in <= '0';
-- mt(0)=1
      mt_0_in <= '1';
      wait for clk_period;
      req_in <= '0';
      wait for clk_period;
```

Продолжение примера 4.2

```
-- mt(1)=0
      mt_0_in <= '0';
      wait for clk_period;
-- mt(2)=1
      mt_0_in <= '1';
      wait for clk_period*2;
-- mt(3)=0
      mt_0_in <= '0';
      wait for clk_period;
-- mt(4)=0
      wait for clk_period;
-- mt(5)=0
      wait for clk_period;
-- mt(6)=0
      wait for clk_period;
      cnt_step_0_in <= '1';
      wait;
end process;

END;
```

После перевода управляющей части устройства умножения в исходное состояние по сигналу `reset` испытательный стенд устанавливает в 1 осведомительный сигнал `cnt_step_0_in`, что соответствует окончанию выполнения предыдущей операции умножения, формирует сигнал запроса на выполнение новой операции умножения `req_in` и переходит к ожиданию начала выполнения микропрограммы. Начало выполнения микропрограммы можно определить по формированию управляющего сигнала `y_1_out`, соответствующего первой микрокоманде. Получив этот сигнал, испытательный стенд сбрасывает в 0 осведомительный сигнал `cnt_step_0_in`, что соответствует началу выполнения текущей операции умножения, выдает значение младшего разряда множителя, снимает сигнал запроса на выполнение операции умножения `req_in` и переходит к последовательной выдаче в управляющую часть умножителя в циклах умножения очередных разрядов множителя. Циклы умножения, соответствующие единичному значению разряда множителя, занимают по два такта синхронизации, а циклы умножения, соответствующие нулевому значению, – по одному такту. После выдачи значения старшего разряда значащей части множителя (разряд 6) выполняется последний, седьмой цикл умножения, о чем испытательный стенд сообщает установкой в 1 осведомительного сигнала `cnt_step_0_in`. Получив этот сигнал, управляющая часть устройства умножения переходит в исходное состояние (микропрограмма переходит в состояние a_0).

На рис. 4.1 приведены результаты функционального моделирования управляющей части устройства умножения для рассматриваемого проекта в среде проектирования для базового кристалла ПЛИС (микросхема XC3S200) отладочной платы Spartan-3 Starter Kit.

Рис. 4.1. Временная диаграмма функционального моделирования управляющей части устройства умножения

4.2. Порядок выполнения работы

1. На основе функциональной схемы управляющей части устройства умножения, разработанной в лабораторной работе № 2, создать проект управляющей части умножителя в виде принципиальной схемы в схемотехническом редакторе или путем описания на языке VHDL.
2. Выполнить функциональное моделирование созданного проекта в среде проектирования Xilinx ISE.
3. Оформить отчет.

4.3. Содержание отчета

1. Титульный лист.
2. Цель работы.
3. Описание процесса создания проекта операционной части устройства умножения в схемотехническом редакторе или путем описания на языке VHDL.
4. Описание процесса создания VHDL-модуля тестовых воздействий для операционной части устройства умножения.
5. Описание процесса функционального моделирования проекта операционной части устройства умножения в среде проектирования Xilinx ISE.
6. Выводы по работе.

ЛАБОРАТОРНАЯ РАБОТА № 5. ФУНКЦИОНАЛЬНОЕ МОДЕЛИРОВАНИЕ АРИФМЕТИЧЕСКОГО УСТРОЙСТВА С ИСПОЛЬЗОВАНИЕМ ИНТЕГРИРОВАННОЙ СРЕДЫ РАЗРАБОТКИ

Цель работы – получение практических навыков по разработке и моделированию проекта арифметического устройства в целом (объединению операционной и управляющей частей устройства) с использованием интегрированной среды разработки Xilinx ISE.

5.1. Создание тестового модуля проекта и функциональное моделирование проектируемого устройства в целом

В примере 5.1 приведен вариант VHDL-модуля устройства умножения в целом, проектирование которого рассмотрено в лабораторной работе № 2, а в примере 5.2 – вариант VHDL-модуля тестовых воздействий для этого проекта.

VHDL-модуль устройства умножения включает два компонента: `mult_oper` – модуль операционной части устройства умножения и `mult_control` – модуль управляющей часть устройства умножения. Эти компоненты связаны между собой с помощью соответствующих осведомительных и управляющих сигналов. В примере 5.1 реализовано устройство умножения с задержанной последовательной загрузкой операндов (см. с. 29, лабораторная работа № 2, вариант 4, рис. 2.12). Процесс умножения начинается по стробу запуска `START_IN` длительностью 1 такт синхронизации. Для реализации задержанной последовательной загрузки операндов по стробу запуска `START_IN` формируются два дополнительных строба `start_del1` и `start_del2`, задержанных соответственно на 1 и 2 такта относительно строба запуска. По стробу `start_del1` с входной шины `D_IN` в соответствующий регистр заносится первый операнд *A* (множимое), а по стробу `start_del2` – второй операнд *B* (множитель). Одновременно с приемом множителя формируется сигнал запроса на выполнение операции умножения `ff_req`, который поступает на соответствующий вход модуля управляющей части умножителя. По этому сигналу управляющая часть начинает процесс формирования последовательности управляющих сигналов в соответствии с микропрограммой работы умножителя (см. с. 21, лабораторная работа № 2, рис. 2.6). По окончании процесса умножения на выходной шине `PR_OUT` операционной части появляется итоговое произведение и устанавливается осведомительный сигнал `cnt_step_0` (флаг: счетчик циклов умножения равен 0). На основании этого сигнала формируется строб готовности произведения на выходе операционной части `strb_done`. В свою очередь по этому сигналу произведение заносится в выходной регистр, с выхода которого появляется на выходной шине `D_OUT` умножителя. Одновременно устанавливается сигнал готовности произведения `READY_OUT`.

Пример 5.1. VHDL-модуль верхнего уровня устройства умножения в целом

```
--------------------------------------------------------------------------------
-- Description:   Модуль реализует устройство умножения в целом
--                - умножаются два 8-разрядных двоичных числа со знаком
--                  в прямом коде, произведение 16-разрядное
--                  в модифицированном коде
--                - метод умножения: с младших разрядов множителя,
--                  сумма частичных произведений сдвигается вправо,
--                  множимое неподвижно
--                - вариант загрузки операндов: задержанная последовательная
--------------------------------------------------------------------------------
-- Structure:     top_mult
--                - mult
--                  - mult_oper
--                  - mult_control
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity mult is
port (
-- входные сигналы
CLK         : in STD_LOGIC;                      -- сигнал синхронизации
RESET       : in STD_LOGIC;                      -- сброс (активный - высокий)
D_IN        : in std_logic_vector (7 downto 0); -- шина операндов
START_IN    : in std_logic;   -- строб запуска (активный - высокий, 1 такт)
-- выходные сигналы
READY_OUT   : out std_logic;  -- готовность произведения (активный - высокий)
D_OUT       : out std_logic_vector (15 downto 0)      -- шина произведения
);
end mult;

architecture implementation of mult is
----- component mult_oper - операционная часть устройства умножения -----
component mult_oper
port (
-- входные сигналы
CLK               : in STD_LOGIC;   -- сигнал синхронизации
RESET             : in STD_LOGIC;   -- сброс (активный - высокий)
MN_IN             : in std_logic_vector (7 downto 0);       -- множимое
MT_IN             : in std_logic_vector (7 downto 0);       -- множитель
Y_1_IN            : in std_logic;   -- микрокоманда 1
Y_2_IN            : in std_logic;   -- микрокоманда 2
Y_3_IN            : in std_logic;   -- микрокоманда 3
-- выходные сигналы
MT_0_OUT          : out std_logic;  -- младший разряд множителя
CNT_STEP_0_OUT    : out std_logic;  -- флаг: счетчик циклов умножения равен 0
PR_OUT            : out std_logic_vector (15 downto 0)      -- произведение
);
end component mult_oper;
----- component mult_control - управляющая часть устройства умножения -----
component mult_control
port (
-- входные сигналы
CLK               : in STD_LOGIC;   -- сигнал синхронизации
RESET             : in STD_LOGIC;   -- сброс (активный - высокий)
REQ_IN            : in std_logic;   -- запрос на выполнение операции умножения
```

Продолжение примера 5.1

```
MT_0_IN           : in std_logic;   -- младший разряд множителя
CNT_STEP_0_IN     : in std_logic;   -- флаг: счетчик циклов умножения равен 0
-- выходные сигналы
Y_1_OUT           : out std_logic;  -- микрокоманда 1
Y_2_OUT           : out std_logic;  -- микрокоманда 2
Y_3_OUT           : out std_logic   -- микрокоманда 3
);
end component mult_control;

-- internal signals and registers
signal rg_mn      : std_logic_vector (7 downto 0) := (others => '0');  -- РгМн
signal rg_mt      : std_logic_vector (7 downto 0) := (others => '0');  -- РгМт
signal y1, y2, y3 : std_logic := '0';          -- микрокоманды
signal mt_0       : std_logic := '0';          -- младший разряд множителя
signal cnt_step_0 : std_logic := '0';          -- флаг СчЦ = 0
signal cnt_step_0_del   : std_logic := '0';    -- задержанный сигнал
signal pr         : std_logic_vector (15 downto 0) := (others => '0'); -- Пр
signal ff_req     : std_logic := '0';     -- запрос на выполнение умножения
signal start_del1 : std_logic := '0';     -- задержанный сигнал строба запуска
signal start_del2 : std_logic := '0';     -- задержанный сигнал строба запуска
signal strb_done  : std_logic := '0';     -- строб готовности произведения
signal ff_ready   : std_logic := '0';     -- сигнал готовности произведения
signal rg_pr_out  : std_logic_vector (15 downto 0) := (others => '0'); -- РгПр

begin

u1: mult_oper           -- операционная часть устройства умножения
port map    (
      CLK               => CLK,           -- сигнал синхронизации
      RESET             => RESET,         -- сброс (активный - высокий)
      MN_IN             => rg_mn,         -- множимое
      MT_IN             => rg_mt,         -- множитель
      Y_1_IN            => y1,            -- микрокоманда 1
      Y_2_IN            => y2,            -- микрокоманда 2
      Y_3_IN            => y3,            -- микрокоманда 3
      MT_0_OUT          => mt_0,          -- младший разряд множителя
      CNT_STEP_0_OUT    => cnt_step_0,    -- флаг: СчЦ = 0
      PR_OUT            => pr             -- произведение
);
u2: mult_control  -- управляющая часть устройства умножения
port map    (
      CLK               => CLK,           -- сигнал синхронизации
      RESET             => RESET,         -- сброс (активный - высокий)
      REQ_IN            => ff_req,        -- запрос на выполнение умножения
      MT_0_IN           => mt_0,          -- младший разряд множителя
      CNT_STEP_0_IN     => cnt_step_0,    -- флаг: СчЦ = 0
      Y_1_OUT           => y1,            -- микрокоманда 1
      Y_2_OUT           => y2,            -- микрокоманда 2
      Y_3_OUT           => y3             -- микрокоманда 3
);
-- задержка входного строба запуска
process (CLK) is
begin
      if (CLK'EVENT and CLK = '1') then        -- rising clock edge
            start_del1 <= START_IN;
            start_del2 <= start_del1;
      end if;
end process;
```

Продолжение примера 5.1

```
-- регистр первого операнда DA (множимого, 8 разрядов)
process (CLK) is
begin
      if (CLK'EVENT and CLK = '1') then        -- rising clock edge
            if start_del1 = '1' then
                  rg_mn <= D_IN;
            end if;
      end if;
end process;

-- регистр второго операнда DB (множителя, 8 разрядов)
process (CLK) is
begin
      if (CLK'EVENT and CLK = '1') then        -- rising clock edge
            if start_del2 = '1' then
                  rg_mt <= D_IN;
            end if;
      end if;
end process;

-- триггер запроса на выполнение операции умножения
process (CLK) is
begin
      if (CLK'EVENT and CLK = '1') then        -- rising clock edge
            if RESET = '1' then
                  ff_req <= '0';
            elsif (start_del2 = '1' and cnt_step_0 = '1') then
                  ff_req <= '1';
            elsif cnt_step_0 = '0' then
                  ff_req <= '0';
            end if;
      end if;
end process;

-- задержка флага: счетчик шагов умножения равен 0
process (CLK) is
begin
      if (CLK'EVENT and CLK = '1') then        -- rising clock edge
            cnt_step_0_del <= cnt_step_0;
      end if;
end process;

-- строб готовности произведения на выходе операционной части
strb_done <= cnt_step_0 and (not (cnt_step_0_del));

-- триггер готовности произведения
process (CLK) is
begin
      if (CLK'EVENT and CLK = '1') then        -- rising clock edge
            if RESET = '1' then
                  ff_ready <= '1';
            elsif START_IN = '1' then
                  ff_ready <= '0';
            elsif strb_done = '1' then
                  ff_ready <= '1';
            end if;
      end if;
end process;

READY_OUT <= ff_ready;
```

Продолжение примера 5.1

```
-- выходной регистр произведения
process (CLK) is
begin
      if (CLK'EVENT and CLK = '1') then        -- rising clock edge
            if RESET = '1' then
                  rg_pr_out <= (others => '0');
            elsif strb_done = '1' then
                  rg_pr_out <= pr;
            end if;
      end if;
end process;

D_OUT <= rg_pr_out;

end implementation;
```

VHDL-модуль тестовых воздействий для устройства умножения в целом обеспечивает выдачу множимого и множителя в умножитель в соответствии с вариантом 4 (задержанной последовательной загрузкой операндов, с. 29, лабораторная работа № 2, рис. 2.12). Пример 5.2 разработан для значений множимого $10000010_{(2)} = -2_{(10)}$ и множителя $00000101_{(2)} = +5_{(10)}$ из лабораторной работы № 3.

Пример 5.2. VHDL-модуль тестовых воздействий для устройства умножения в целом

```
--------------------------------------------------------------------------------
-- VHDL Test Bench for module: mult
--------------------------------------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY test_mult IS
END test_mult;
ARCHITECTURE behavior OF test_mult IS

-- Component Declaration for the Unit Under Test (UUT)
COMPONENT mult is
port (
-- входные сигналы
CLK         : in STD_LOGIC;                     -- сигнал синхронизации
RESET       : in STD_LOGIC;                     -- сброс (активный - высокий)
D_IN        : in std_logic_vector (7 downto 0); -- шина операндов
START_IN    : in std_logic;   -- строб запуска (активный - высокий, 1 такт)
-- выходные сигналы
READY_OUT   : out std_logic;  -- готовность произведения (активный - высокий)
D_OUT       : out std_logic_vector (15 downto 0)      -- шина произведения
);
end COMPONENT;

--Inputs
signal clk        : std_logic := '0';
signal reset      : std_logic := '0';
signal d_in       : std_logic_vector (7 downto 0) := (others => '0');
signal start_in   : std_logic := '0';
--Outputs
signal ready_out  : std_logic;
signal d_out      : std_logic_vector (15 downto 0);
```

Продолжение примера 5.2

```
-- Clock period definitions
constant clk_period : time := 20 ns;

BEGIN

-- Instantiate the Unit Under Test (UUT)
uut: mult
PORT MAP (
      CLK         => clk,
      RESET       => reset,
      D_IN        => d_in,
      START_IN    => start_in,

      READY_OUT   => ready_out,
      D_OUT       => d_out
);

-- Clock process definitions
clk_process :process
begin
      clk <= '1';
      wait for clk_period/2;
      clk <= '0';
      wait for clk_period/2;
end process;

-- Stimulus process
stim_proc: process
begin
-- hold reset state for 200 ns
      wait for 200 ns;
      wait for clk_period*0.05;           -- + 0.05 периода;
      reset <= '1';
      wait for clk_period;
      reset <= '0';
      wait for clk_period*10;
-- выдача множимого и множителя
      start_in <= '1';
      wait for clk_period;
      start_in <= '0';
      d_in <= "10000010";     -- Мн = -2
      wait for clk_period;
      d_in <= "00000101";     -- Мт = +5
      wait;
end process;

END;
```

После перевода устройства умножения в исходное состояние по сигналу `reset`, испытательный стенд формирует сигнал запроса на выполнение операции умножения `start_in` и последовательно за два такта выдает на входную шину операндов `d_in` сначала множимое, затем множитель. После завершения процесса умножения на выходной шине `d_out` появляется вычисленное значение произведения, которое сопровождается установкой активного уровня сигнала готовности произведения `ready_out`.

На рис. 5.1 приведены результаты функционального моделирования устройства умножения в целом для рассматриваемого проекта для базового кристалла ПЛИС (микросхема XC3S200) отладочной платы Spartan-3 Starter Kit.

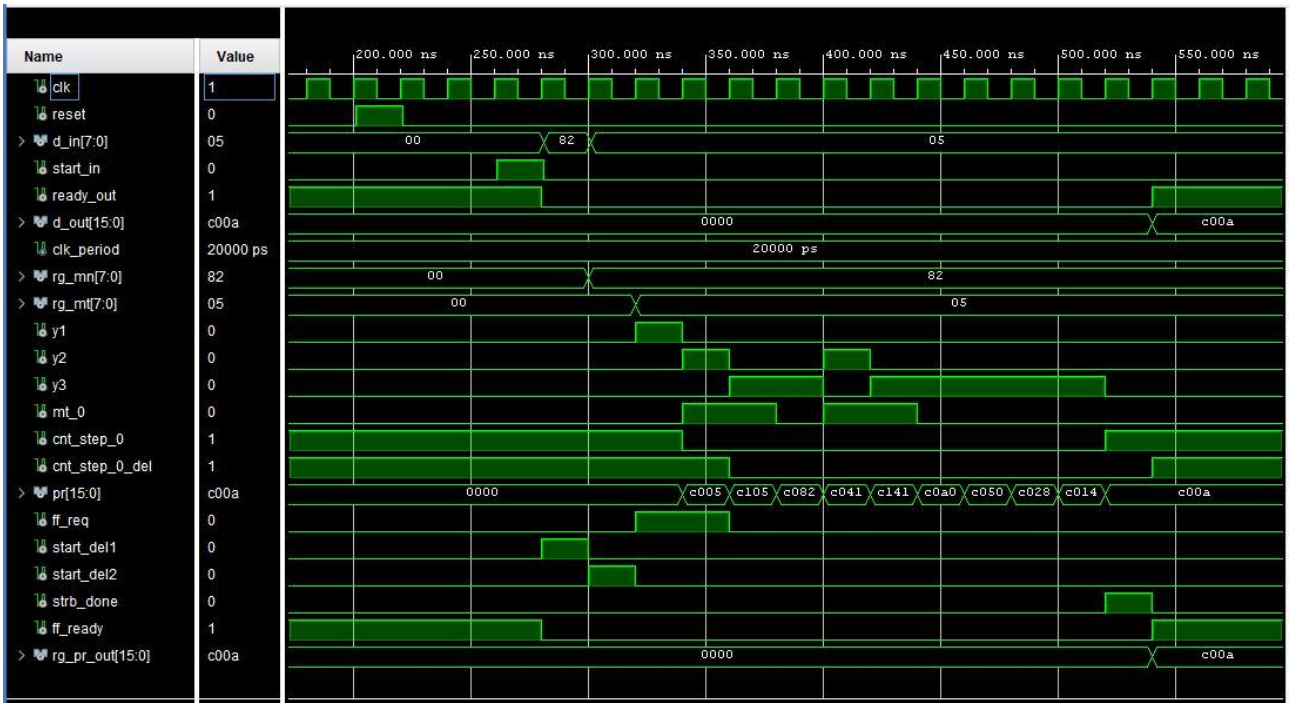

Рис. 5.1. Временная диаграмма функционального моделирования устройства умножения

5.2. Порядок выполнения работы

1. На основе модулей операционного и управляющего автоматов устройства умножения, разработанных в лабораторных работах № 3 и 4, создать проект устройства умножения в целом путем описания на языке VHDL.
2. Выполнить функциональное моделирование созданного проекта в среде проектирования Xilinx ISE.
3. Оформить отчет.

5.3. Содержание отчета

1. Титульный лист.
2. Цель работы.
3. Описание процесса создания проекта модуля верхнего уровня устройства умножения в целом путем описания на языке VHDL.
4. Описание процесса создания VHDL-модуля тестовых воздействий для устройства умножения в целом путем описания на языке VHDL.
5. Описание процесса функционального моделирования проекта устройства умножения в целом в среде проектирования Xilinx ISE.
6. Выводы по работе.

ЛАБОРАТОРНАЯ РАБОТА № 6. ТЕСТИРОВАНИЕ ПРОЕКТА АРИФМЕТИЧЕСКОГО УСТРОЙСТВА С ИСПОЛЬЗОВАНИЕМ ОТЛАДОЧНОЙ ПЛАТЫ

Цель работы – получение практических навыков по выполнению всех этапов синтеза проекта арифметического устройства, включая размещение и трассировку проекта, генерацию прошивки ПЛИС, с использованием интегрированной среды разработки серии Xilinx ISE; получение практических навыков по построению тестирующей системы арифметического устройства, обеспечивающей ввод исходных данных с устройства ввода (переключателей, клавиатуры) и вывод результата на устройство индикации (светодиоды, цифровые индикаторы); получение практических навыков по тестированию проекта арифметического устройства с использованием отладочной платы Spartan-3 Starter Kit.

6.1. Этапы синтеза, размещения и трассировки проекта арифметического устройства с использованием среды разработки серии Xilinx ISE

После создания нового проекта арифметического устройства следующий шаг процесса разработки цифровых устройств на базе ПЛИС в среде САПР Xilinx ISE предполагает выполнение этапов синтеза, размещения и трассировки проекта.

Для выполнения этапов синтеза, размещения и трассировки необходимо определить топологические и временные свойства проекта. Основным топологическим свойством проекта является распределение выводов ПЛИС между внешними портами (цепями, подключаемыми к контактам кристалла), определенными в проекте. Основными временными свойствами проекта являются ограничения на задержки распространения отдельных сигналов. Такая дополнительная информация задается либо непосредственно в модулях исходного описания проекта в форме атрибутов, либо в файле временных и топологических ограничений UCF (*User Constraints File*). Более предпочтительным является использование файла UCF, так как при этом сохраняется универсальность модулей исходного описания, которые могут применяться и в других проектах. При этом для внесения изменений в параметры ограничений, например при выборе другого типа ПЛИС, не потребуется редактировать модули исходного описания проекта, а достаточно будет только скорректировать содержимое файла UCF. Файл временных и топологических ограничений проекта представляет собой текстовый файл, каждая строка которого описывает определенное ограничение. Ограничения задаются с помощью соответствующих выражений, имеющих определенный формат. Для создания файла UCF можно использовать любой текстовый редактор (например, «Блокнот») или специальную программу Constraints Editor пакета ISE, которая в интерактивном режиме автоматически формирует соответствующие выражения для описания ограничений проекта.

Ниже приведены форматы выражений, описывающих наиболее часто используемые ограничения.

Параметр LOC позволяет назначить внешним цепям проекта требуемые выводы ПЛИС перед трассировкой. Для задания этого параметра используется следующий формат выражения:

NET <название_цепи> LOC = <номер_вывода_ПЛИС>;

Например, выражения

```
NET clk LOC = C5;
NET data1 LOC = P9;
```

указывают, что внешнюю цепь (сигнал) clk необходимо подключить к выводу C5, а цепь data1 – к выводу P9.

С помощью параметра PERIOD задается максимальное значение периода сигнала синхронизации для соответствующей цепи проекта. Общий формат выражения для этого ограничения имеет следующий вид:

NET <название_цепи> PERIOD = <длительность_периода_сигнала>
[<единицы_измерения>] [{HIGH|LOW} [<длительность_первой
фазы_периода> [<единицы_измерения>]]];

где значение HIGH или LOW указывает логический уровень сигнала в первой фазе периода.

По умолчанию установлены в качестве единиц измерения длительности наносекунды (нс) и одинаковая продолжительность состояний высокого и низкого уровня периода синхросигнала. В большинстве случаев установленные по умолчанию значения подходят для построения проекта, поэтому можно пользоваться сокращенным форматом выражения для задания параметра PERIOD:

NET <название_цепи> PERIOD = <длительность_периода>;

Например,

```
NET clk PERIOD = 20 ns;
```

В процессе синтеза из HDL-описаний проектируемого устройства формируется описание проекта на более низком логическом уровне в формате EDIF (Electronic Data Interchange Format), воспринимаемом программой трассировки. Если исходные описания проекта представлены в виде принципиальных схем в графической (схемотехнической) форме, то в процессе синтеза выполняется их преобразование в требуемый HDL-формат.

Прежде чем запускать процесс синтеза проекта, можно изменить значения его параметров, установленные по умолчанию (настроить процесс синтеза проекта). При выполнении лабораторной работы (и в целом в процессе обучения) рекомендуется не изменять значения параметров, установленные по умолчанию.

После успешного завершения этапа синтеза следует перейти к размещению и трассировке проекта в кристалле ПЛИС. Этот этап включает в себя две фазы: фазу трансляции и фазу выделения ресурсов кристалла для реализации проектируемого устройства. В процессе трансляции выполняется объединение

описания проекта в формате EDIF и информации обо всех ограничениях, определяющих топологические и временные свойства проекта (которые, в частности, содержатся в файле UCF). В результате формируется логическое описание проекта в терминах примитивов Xilinx низкого уровня с учетом временных и топологических ограничений в формате NGD (Native Generic Database). На второй фазе производится разбиение логического описания проекта на блоки в соответствии с ресурсами выбранного кристалла. При этом выполняется оптимизация проекта с целью минимизации используемых ресурсов кристалла с учетом заданных ограничений. В результате этапа размещения и трассировки создается описание использования физических ресурсов кристалла для реализации функций проектируемого устройства, которое помещается в двоичный файл.

При получении успешных результатов размещения и трассировки нужно сгенерировать для проектируемого устройства прошивку ПЛИС. Для этого результаты, полученные на этапе размещения и трассировки проекта, необходимо преобразовать в формат, используемый средствами программирования. В среде разработки серии Xilinx ISE необходимо в левой области главного окна программы с помощью пункта *Generate Programming File* запустить процесс создания конфигурационной последовательности (файла программирования).

После успешного завершения этого процесса можно запрограммировать полученную прошивку в ПЛИС отладочной платы Spartan-3 Starter Kit.

6.2. Описание доступных ресурсов отладочной платы Spartan-3 Starter Kit

На рис. 6.1и 6.2 приведен внешний вид отладочной платы Spartan-3 Starter Kit, включающей ПЛИС фирмы Xilinx Spartan-3 (микросхема XC3S200).

Отладочная плата Spartan-3 Starter Kit представляет собой автономную платформу разработки, включающую ПЛИС фирмы Xilinx Spartan-3 (микросхема XC3S200) и оснащенную встроенными устройствами ввода/вывода и двумя микросхемами статической памяти. Отладочная плата предназначена для экспериментов с любыми проектами, от простой логической схемы до встроенного ядра процессора. Плата совместима со всеми версиями систем проектирования Xilinx ISE.

На рис. 6.1 и 6.2 показано расположение компонентов на верхней и нижней сторонах платы соответственно. Ниже перечислены основные компоненты и интерфейсы, размещенные на отладочной плате, с указанием их позиций в соответствии с рис. 6.1 и 6.2:

– FPGA Xilinx Spartan-3 XC3S200FT256 (позиция 1, рис. 6.1);

– внутрисистемная конфигурационная флеш-память Xilinx XCF02S объемом 2 Мбит и перемычка, задающая режим ее использования (позиции 2 и 3, рис. 6.1);

– асинхронная статическая память SRAM (2 микросхемы 256К × 16 бит) объемом 1 Мбайт (позиция 4, рис. 6.2);

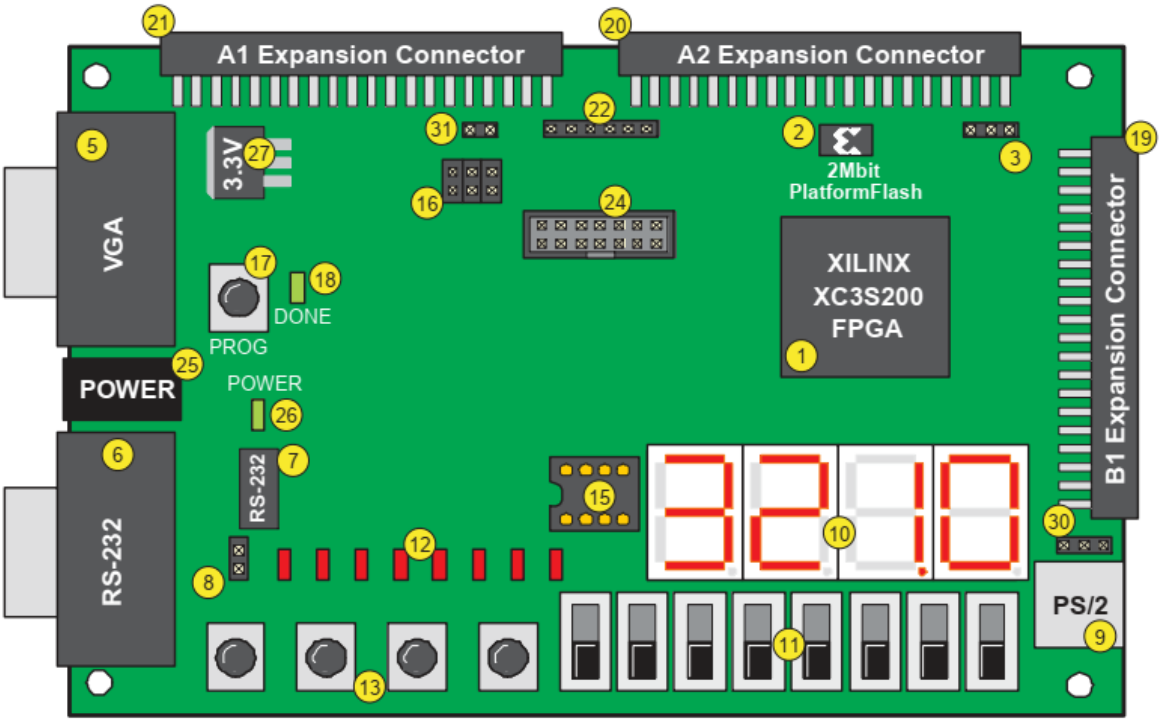

Рис. 6.1. Ресурсы отладочной платы Spartan-3 Starter Kit (верхняя сторона)

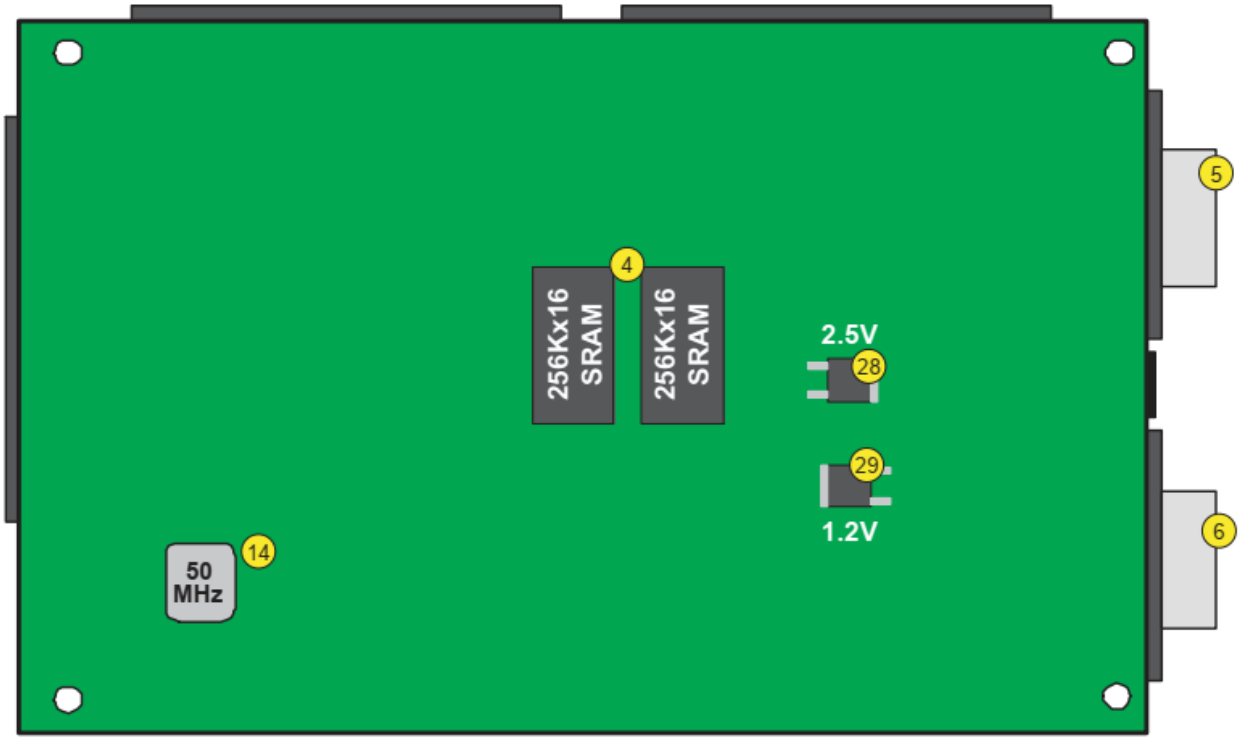

Рис. 6.2. Ресурсы отладочной платы Spartan-3 Starter Kit (нижняя сторона)

– порт VGA, последовательный порт RS232, порт мыши/клавиатуры PS/2 (позиции 5, 6, 9, рис. 6.1);

– 4-символьный 7-сегментный светодиодный дисплей (позиция 10, рис. 6.1);

– 8 переключателей (позиция 11, рис. 6.1);

– 8 отдельных светодиодов (позиция 12, рис. 6.1);

– 4 кнопочных переключателя с мгновенным срабатыванием (позиция 13, рис. 6.1);

– кварцевый резонатор генератора тактовых импульсов 50 МГц (позиция 14, рис. 6.2);

– перемычки, задающие режим конфигурирования FPGA (позиция 16, рис. 6.1);

– кнопочный переключатель для принудительного реконфигурирования FPGA (позиция 17, рис. 6.1);

– светодиод, показывающий что FPGA успешно сконфигурирована (позиция 18, рис. 6.1);

– порт загрузки/отладки JTAG, совместимый с Xilinx Parallel Cable IV (позиция 24, рис. 6.1);

– вход адаптера переменного тока для входящего в комплект источника питания +5 В (позиция 25, рис. 6.1);

– светодиодный индикатор включения питания (позиция 26, рис. 6.1).

6.3. Создание тестового модуля проекта устройства умножения для отладочной платы Spartan-3 Starter Kit

В примере 6.1 приведен вариант VHDL-описания интерфейса блока верхнего уровня (TOP-модуля) для отладочной платы.

Пример 6.1. VHDL-описание интерфейса блока верхнего уровня (TOP-модуля) для отладочной платы (top.vhd)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity top is
port (
CLK   : in std_logic;                          -- синхросигнал 50 МГц (T9)
BTN   : in std_logic_vector(3 downto 0);       -- кнопки (L14 L13 M14 M13)
SW    : in std_logic_vector(7 downto 0);       -- переключатели (K13 K14 J13
                                               -- J14 H13 H14 G12 F12)
LED   : out std_logic_vector(7 downto 0);      -- светодиоды (P11 P12 N12 P13
                                               -- N14 L12 P14 K12)
-- семисегментный светодиодный дисплей
SEG   : out std_logic_vector(7 downto 0);      -- сегменты (P16 N16 F13 R16
                                               -- P15 N15 G13 E14)
AN    : out std_logic_vector(3 downto 0);      -- индикаторы (E13 F14 G14 D14)
-- выводы интерфейса RS232
RXD   : in std_logic_vector(1 downto 0);       -- входы приемников RXD-A
                                               -- (N10), RXD (T13)
```

Продолжение примера 6.1

```
TXD   : out std_logic_vector(1 downto 0);       -- выходы передатчиков TXD-A
                                                -- (T14), TXD (R13)
-- выводы интерфейса VGA
GRB   : out std_logic_vector(2 downto 0);       -- видеосигнал зеленый,
                                                -- красный, синий (T12 R12 R11)
VH    : out std_logic_vector(1 downto 0);       -- сигналы вертикальной и
                                                -- горизонтальной синхронизации
                                                -- (T10 R9)
-- выводы интерфейса PS2
PS2   : inout std_logic_vector(1 downto 0);     -- сигналы синхронизации CLK
                                                -- (M16) и передаваемых данных
                                                -- DATA (M15)
-- выводы подключения статической памяти SRAM (256K × 16 бит)
ADDR  : out std_logic_vector(17 downto 0);      -- шина адреса A17-A0 (L3 K5 K3
                                                -- J3 J4 H4 H3 G5 E4 E3 F4 F3
                                                -- G4 L4 M3 M4 N3 L5)
DATA  : inout std_logic_vector(31 downto 0);    -- шина данных
                                                -- IO15-IO0 микросхемы IC11
                                                -- (N1 M1 K2 C3 F5 G1 E2 D2 D1
                                                -- E1 G2 J1 K1 M2 N2 P2)
                                                -- IO15-IO0 микросхемы IC10
                                                -- (R1 P1 L2 J2 H1 F2 P8 D3 B1
                                                -- C1 C2 R5 T5 R6 T8 N7)
CE    : out std_logic_vector(1 downto 0);       -- сигналы выбора микросхем
                                                -- IC11, IC10 (N5 P7)
OE    : out std_logic;                          -- сигнал разрешения выходов
                                                -- микросхем IC11, IC10 (K4)
WE    : out std_logic;                          -- сигнал разрешения записи
                                                -- микросхем IC11, IC10 (G3)
BSEL  : out std_logic_vector(3 downto 0));      -- сигналы разрешения старшего
                                                -- и младшего байтов
                                                -- UB2, LB2 микросхемы IC11
                                                -- (R4 P5)
                                                -- UB1, LB1 микросхемы IC10
                                                -- (T4 P6)
end;

architecture behavior of top is

begin

-- отключение неиспользуемых выводов
ADDR  <= "000000000000000000";
DATA  <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
CE    <= "11";
OE    <= '1';
WE    <= '1';
BSEL  <= "0000";
TXD   <= "11";
PS2   <= "ZZ";
GRB   <= "000";
VH    <= "00";
LED   <= x"00";
SEG   <= x"00";
AN    <= x"F";

end;
```

В примере 6.2 приведен вариант VHDL-модуля верхнего уровня (TOP-модуля) для проверки устройства умножения, проектирование которого рассмотрено в лабораторной работе № 2, на отладочной плате.

Тестовый модуль обеспечивает ввод множимого и множителя с переключателей и отображение полученного произведения на светодиодах. Управление умножителем осуществляется с помощью кнопок:

1) *BTN3* – пользовательский сброс. По нажатии на эту кнопку производится перевод умножителя в исходное состояние – ввод чисел и вычисление их произведения;

2) *BTN2* – выбор для отображения на светодиодах старшего или младшего байтов произведения. Когда эта кнопка не нажата, на светодиодах отображается младший байт произведения. По нажатии на эту кнопку на светодиодах будет отображаться старший байт произведения;

3) *BTN1* – ввод множителя и запуск процесса умножения. По нажатии на эту кнопку осуществляется прием набранного на переключателях значения множителя и запуск на выполнение микропрограммы работы устройства умножения;

4) *BTN0* – ввод множимого. По нажатии на эту кнопку осуществляется прием набранного на переключателях значения множимого и переход к ожиданию ввода множителя.

Для отображения состояния устройства умножения используются три правых 7-сегментных индикатора. Отображение состояния осуществляется с помощью четырех нижних сегментов: *c, d, e, g* (визуально на соответствующем индикаторе высвечивается квадратик).

Квадратик на крайнем правом индикаторе загорается при пользовательском сбросе (была нажата кнопка *BTN3*) и указывает, что умножитель находится в исходном состоянии и ожидает ввода множимого. После нажатия кнопки *BTN0* происходит ввод множимого в соответствующий регистр `rg_mn`. При этом квадратик перемещается на второй справа индикатор и указывает, что умножитель ожидает ввода множителя. По нажатии кнопки *BTN1* осуществляется ввод множителя в соответствующий регистр `rg_mt`, и квадратик перемещается на третий справа индикатор, указывая тем самым, что сомножители введены и умножитель перешел к выполнению операции умножения. При этом формируется строб запуска умножителя длительностью один такт синхронизации, по которому в соответствии с вариантом 4 (см. с. 29, рис. 2.12) осуществляется задержанная последовательная загрузка операндов в модуль умножителя и запуск процесса умножения в соответствии с микропрограммой работы умножителя (см. с. 21, рис. 2.6).

После окончания процесса умножения на светодиодах отображается младший байт произведения. По нажатии на кнопку *BTN2* на светодиодах будет отображаться старший байт произведения. Для ввода новых чисел и вычисления их произведения необходимо выполнить сброс умножителя, нажав на кнопку *BTN3*.

Пример 6.2. VHDL-модуль верхнего уровня для проверки устройства умножения на отладочной плате Spartan-3 Starter Kit

```
----------------------------------------------------------------------------
-- Description:   Модуль верхнего уровня для проверки устройства умножения на
--                отладочной плате Spartan-3 Starter Kit
--                - множимое и множитель вводятся с переключателей
--                - произведение отображается на светодиодах
--                - управление умножителем осуществляется с помощью кнопок:
--                BTN3 - пользовательский сброс:
--                       перевод умножителя в исходное состояние - ввод чисел
--                       и вычисление их произведения
--                BTN2 - выбор для отображения старшего или младшего байтов
--                       произведения: не нажата - младший байт,
--                       нажата – старший байт
--                BTN1 - ввод множителя и запуск процесса умножения
--                BTN0 - ввод множимого
----------------------------------------------------------------------------
-- Structure:
--                top_mult
--                - mult
--                  - mult_oper
--                  - mult_control
----------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity top_mult is
port (
CLK   : in std_logic;                     -- синхросигнал 50 МГц
BTN   : in std_logic_vector(3 downto 0);  -- кнопки
SW    : in std_logic_vector(7 downto 0);  -- переключатели
LED   : out std_logic_vector(7 downto 0); -- светодиоды
-- семисегментный светодиодный дисплей
SEG   : out std_logic_vector(7 downto 0); -- сегменты
AN    : out std_logic_vector(3 downto 0); -- индикаторы
-- выводы интерфейса RS232
RXD   : in std_logic_vector(1 downto 0);  -- входы приемников RXD-A, RXD
TXD   : out std_logic_vector(1 downto 0); -- выходы передатчиков TXD-A, TXD
-- выводы интерфейса VGA
GRB   : out std_logic_vector(2 downto 0); -- видеосигнал
VH    : out std_logic_vector(1 downto 0); -- сигналы синхронизации
-- выводы интерфейса PS2
PS2   : inout std_logic_vector(1 downto 0);  -- сигналы синхронизации и данных
-- выводы подключения статической памяти SRAM (256K × 16 бит)
ADDR  : out std_logic_vector(17 downto 0);   -- шина адреса
DATA  : inout std_logic_vector(31 downto 0); -- шина данных
CE    : out std_logic_vector(1 downto 0);    -- сигналы выбора микросхем
OE    : out std_logic;                        -- сигнал разрешения выходов
WE    : out std_logic;                        -- сигнал разрешения записи
BSEL  : out std_logic_vector(3 downto 0)      -- сигналы разрешения байтов
);
end top_mult;

architecture implementation of top_mult is

----- component mult - устройство умножения -----
component mult
port (
-- входные сигналы
```

Продолжение примера 6.2

```
CLK         : in STD_LOGIC;                          -- сигнал синхронизации
RESET       : in STD_LOGIC;                          -- сброс (активный - высокий)
D_IN        : in std_logic_vector (7 downto 0); -- шина операндов
START_IN    : in std_logic;   -- строб запуска (активный - высокий, 1 такт)
-- выходные сигналы
READY_OUT   : out std_logic;  -- готовность произведения (активный - высокий)
D_OUT       : out std_logic_vector (15 downto 0)     -- шина произведения
);
end component mult;

-- internal signals and registers
signal reset      : std_logic := '0';-- синхронный сброс (активный - высокий)
signal ff_btn0    : std_logic := '0';-- триггер кнопки BTN0
signal ff_btn0_del : std_logic := '0';    -- задержанный сигнал с выхода
                                          -- триггера кнопки BTN0
signal strb_btn0  : std_logic := '0';     -- строб нажатия кнопки BTN0
signal ff_btn1    : std_logic := '0';     -- триггер кнопки BTN1
signal ff_btn1_del : std_logic := '0';    -- задержанный сигнал с выхода
                                          -- триггера кнопки BTN1
signal strb_btn1  : std_logic := '0';     -- строб нажатия кнопки BTN1
signal start_mult : std_logic := '0';     -- строб запуска умножителя
                                          -- (активный - высокий, 1 такт)
signal start_mult_del   : std_logic := '0';     -- задержанный строб запуска
                                                -- умножителя
signal ff_load    : std_logic := '0';     -- флаг загрузки множимого/множителя
signal ready_out  : std_logic := '0';     -- готовность произведения
                                          -- (в примере не используется)
-- регистр множимого (8 разрядов)
signal rg_mn      : std_logic_vector (7 downto 0) := (others => '0');
-- регистр множителя (8 разрядов)
signal rg_mt      : std_logic_vector (7 downto 0) := (others => '0');
-- мультиплексор множимого/множителя
signal mux_mn_mt  : std_logic_vector (7 downto 0) := (others => '0');
-- мультиплексор отображения на светодиодах переключателей/произведения
signal mux_led    : std_logic_vector (7 downto 0) := (others => '0');
-- выход умножителя
signal d_mult_out : std_logic_vector (15 downto 0) := (others => '0');
-- мультиплексор отображения на светодиодах младш./старш. байтов произведения
signal mux_mult_out : std_logic_vector (7 downto 0) := (others => '0');
-- регистр выбора семисегментного индикатора
signal rg_an      : std_logic_vector (2 downto 0) := (others => '1');

begin

-- отключение неиспользуемых выводов
ADDR <= "000000000000000000";
DATA <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
CE <= "11";
OE <= '1';
WE <= '1';
BSEL <= x"0";
TXD <= "11";
PS2 <= "ZZ";
GRB <= "000";
VH <= "00";
AN(3) <= '1';

-- задание квадратика для отображения на семисегментных индикаторах
-- сегменты c, d, e, g
SEG <= "10100011";
```

Продолжение примера 6.2

```
-- пользовательский сброс (активный - высокий)
reset <= BTN(3);

u1: mult          -- устройство умножения
port map    (
      CLK         => CLK,
      RESET       => reset,
      D_IN        => mux_mn_mt,
      START_IN    => start_mult,
      READY_OUT   => ready_out,
      D_OUT       => d_mult_out
);

-- регистр выбора семисегментного индикатора
-- три разряда - правые индикаторы AN2, AN1, AN0
process (CLK) is
begin
      if (CLK'EVENT and CLK = '1') then        -- rising clock edge
            if reset = '1' then
                  rg_an <= "110";
            elsif strb_btn0 = '1' then
                  rg_an <= "101";
            elsif strb_btn1 = '1' then
                  rg_an <= "011";
            end if;
      end if;
end process;

AN(2 downto 0) <= rg_an;

-- триггер нажатия кнопки BTN0
process (CLK) is
begin
      if (CLK'EVENT and CLK = '1') then        -- rising clock edge
            if reset = '1' then
                  ff_btn0 <= '0';
            elsif (ff_btn0 = '0' and BTN(0) = '1') then
                  ff_btn0 <= '1';
            end if;
      end if;
end process;

-- задержка синхронного сигнала нажатия кнопки BTN0
process (CLK) is
begin
      if (CLK'EVENT and CLK = '1') then        -- rising clock edge
            ff_btn0_del <= ff_btn0;
      end if;
end process;
-- строб нажатия кнопки BTN0
-- (активный - высокий, 1 такт)
strb_btn0 <= ff_btn0 and (not (ff_btn0_del));

-- триггер нажатия кнопки BTN1
process (CLK) is
begin
      if (CLK'EVENT and CLK = '1') then        -- rising clock edge
            if reset = '1' then
```

Продолжение примера 6.2

```
                  ff_btn1 <= '0';
            elsif (ff_btn1 = '0' and BTN(1) = '1' and ff_btn0 = '1') then
                  ff_btn1 <= '1';
            end if;
      end if;
end process;

-- задержка синхронного сигнала нажатия кнопки BTN1
process (CLK) is
begin
      if (CLK'EVENT and CLK = '1') then        -- rising clock edge
            ff_btn1_del <= ff_btn1;
      end if;
end process;
-- строб нажатия кнопки BTN1
-- (активный - высокий, 1 такт)
strb_btn1 <= ff_btn1 and (not (ff_btn1_del));

-- регистр множимого (8 разрядов)
process (CLK) is
begin
      if (CLK'EVENT and CLK = '1') then        -- rising clock edge
            if strb_btn0 = '1' then
                  rg_mn <= SW(7 downto 0);
            end if;
      end if;
end process;

-- регистр множителя (8 разрядов)
process (CLK) is
begin
      if (CLK'EVENT and CLK = '1') then        -- rising clock edge
            if strb_btn1 = '1' then
                  rg_mt <= SW(7 downto 0);
            end if;
      end if;
end process;

-- строб запуска умножителя
-- (активный - высокий, 1 такт)
process (CLK) is
begin
      if (CLK'EVENT and CLK = '1') then        -- rising clock edge
            start_mult <= strb_btn1;
      end if;
end process;
-- задержка строба запуска умножителя
process (CLK) is
begin
      if (CLK'EVENT and CLK = '1') then        -- rising clock edge
            start_mult_del <= start_mult;
      end if;
end process;

-- триггер загрузки множимого/множителя
process (CLK) is
begin
      if (CLK'EVENT and CLK = '1') then        -- rising clock edge
```

Продолжение примера 6.2

```
            if reset = '1' then
                  ff_load <= '0';
            elsif start_mult_del = '1' then
                  ff_load <= '1';
            end if;
      end if;
end process;

-- мультиплексор множимого/множителя
mux_mn_mt <= rg_mn when ff_load = '0' else rg_mt;

-- мультиплексор отображения на светодиодах переключателей/произведения
mux_led <= SW(7 downto 0) when ff_load = '0' else mux_mult_out;

-- мультиплексор отображения на светодиодах
-- младшего/старшего байтов произведения
mux_mult_out <= d_mult_out(7 downto 0) when BTN(2) = '0'
                  else d_mult_out(15 downto 8);

LED <= mux_led;

end implementation;
```

6.4. Загрузка конфигурационной последовательности в ПЛИС отладочной платы Spartan-3 Starter Kit

Загрузка файла конфигурации в FPGA выполняется модулем программирования *iMPACT*, который входит в состав среды проектирования Xilinx ISE (можно вызвать из навигатора проекта *ISE Project Navigator*). Прежде чем приступить к процедуре загрузки файла конфигурации в ПЛИС, необходимо присоединить загрузочный кабель к персональному компьютеру и отладочной плате. Для выполнения лабораторной работы используется загрузочный кабель Xilinx Platform Cable USB II. Он представляет собой USB-совместимый кабель для внутрисхемного конфигурирования FPGA, а также программирования PROM- и CPLD-устройств фирмы Xilinx. Кабель полностью адаптирован для использования с программным обеспечением Xilinx, в том числе со средой проектирования Xilinx ISE. Один конец кабеля подключается к разъему USB-порта персонального компьютера, а второй – к разъему порта загрузки/отладки JTAG, совместимого с Xilinx Parallel Cable IV (позиция 24 на рис. 6.1). Далее необходимо убедиться, что перемычка, задающая режим использования внутрисистемной конфигурационной флеш-памяти (позиция 3 на рис. 6.1), и перемычки, задающие режим конфигурирования FPGA (позиция 16 на рис. 6.1), находятся в положении, заданном по умолчанию (см. полное описание отладочной платы Spartan-3 Starter Kit [16]). После этого можно подключать напряжение питания к отладочной плате (позиция 25 на рис. 6.1). Приведенная последовательность подключения к отладочной плате (сначала подключение загрузочного кабеля, а затем – напряжения питания) обеспечивает при запуске модуля iMPACT автоматическое

обнаружение и инициализацию загрузочного кабеля и устройств, которые установлены на отладочной плате и могут быть запрограммированы. В противном случае необходимо выполнять процедуру задания типа и параметров используемого загрузочного кабеля. На отладочной плате Spartan-3 Starter Kit имеются два таких устройства – FPGA Xilinx Spartan-3 XC3S200FT256 (позиция 1 на рис. 6.1) и конфигурационная флеш-память Xilinx XCF02S. Конфигурационная память FPGA построена на статической памяти, поэтому ее содержимое не сохраняется после выключения питания. Конфигурационная флеш-память Xilinx XCF02S является энергонезависимой, поэтому конфигурационную информацию проекта можно один раз занести в эту память, а затем каждый раз при включении питания отладочной платы переносить эту информацию из конфигурационной флеш-память XCF02S в конфигурационную память FPGA. Эти два устройства соединены в последовательную цепочку (отображаются в графическом виде в рабочей области основного окна и в текстовом виде в окне сообщений) модуля iMPACT и программируются одинаковым образом.

После подсоединения загрузочного кабеля процедура загрузки файла конфигурации в FPGA включает следующие шаги:

1) в окне *Processes* развернуть пункт *Generate Programming File*;

2) запустить процесс *Configure Device (iMPACT)*. Появится диалоговое окно *Welcome to iMPACT*, как показано на рис. 6.3. В открывшемся окне отметить пункт *Configure devices using Boundary-Scan (JTAG)* и убедиться, что в раскрывающемся списке выбрано *Automatically connect to a cable and identify Boundary-Scan chain*. Нажать кнопку *OK*;

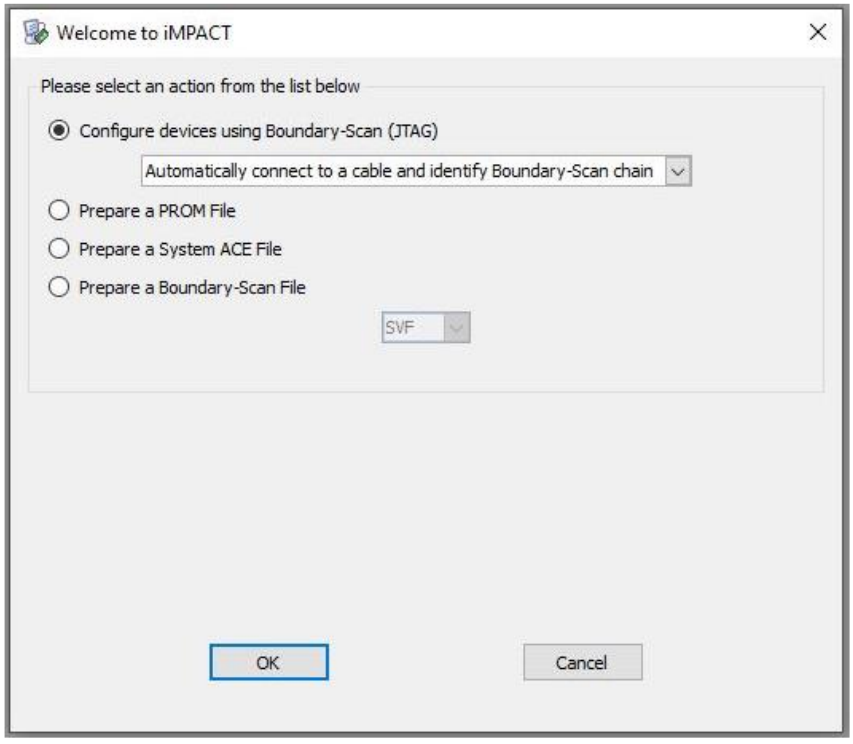

Рис. 6.3. Диалоговое окно *Welcome to iMPACT* модуля *iMPACT*

3) если отображается сообщение о том, что найдено два устройства, нажать кнопку *OK*, чтобы продолжить;

4) появится главное окно модуля *iMPACT* вместе с диалоговым окном выбора конфигурационного файла *Assign New Configuration File*, как показано на рис. 6.4. Устройства, подключенные к цепи JTAG на отладочной плате, должны быть обнаружены и отображены в рабочей области основного окна;

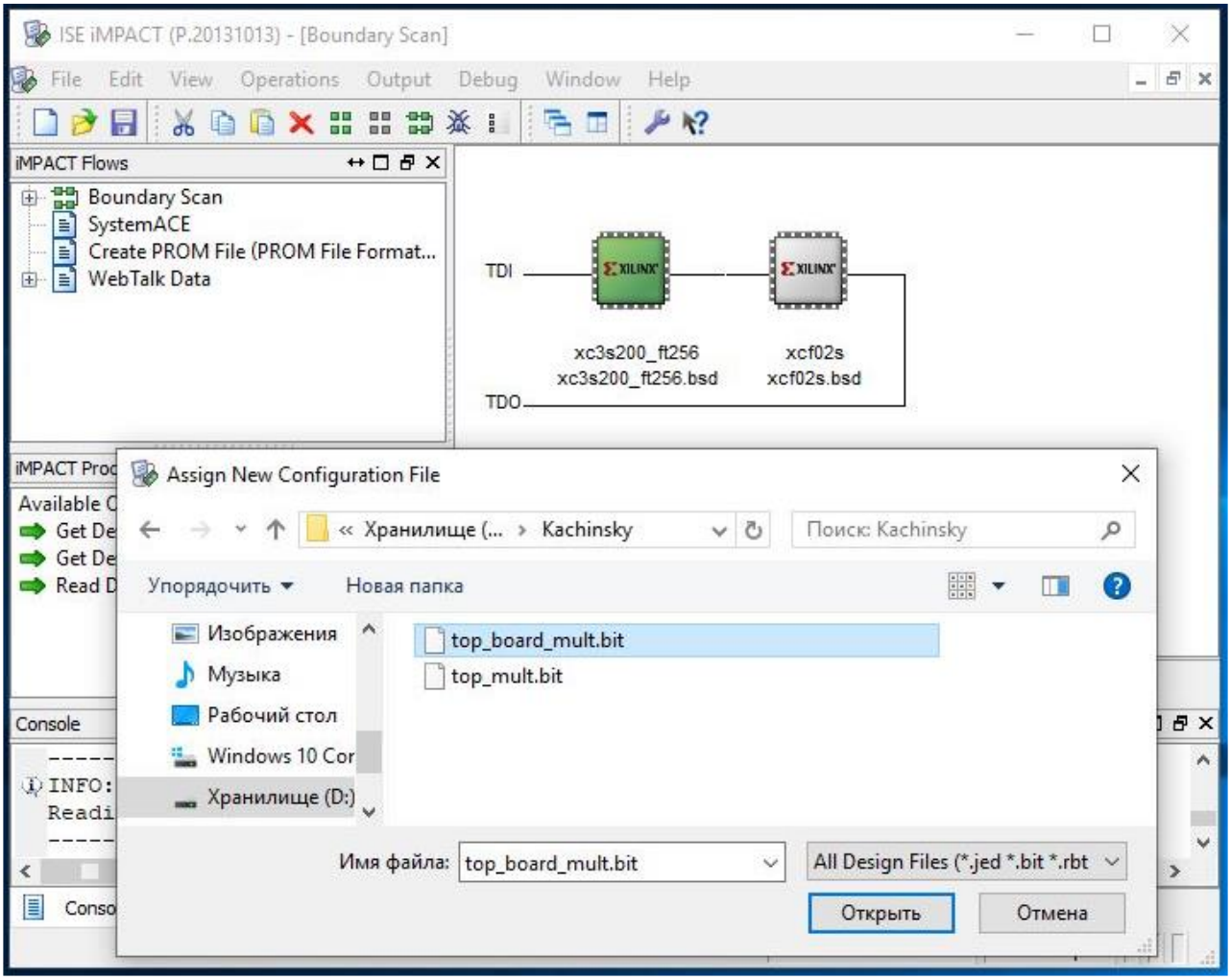

Рис. 6.4. Основное окно модуля *iMPACT*

5) выбрать нужный файл с расширением .bit и нажать кнопку *Open* (*Открыть*), чтобы назначить этот файл конфигурации устройству xc3s200 (микросхема FPGA) в цепочке JTAG;

6) выбрать *Bypass*, чтобы пропустить другое устройство (микросхема флеш-памяти Xilinx XCF02S) в цепочке JTAG;

7) щелкнуть правой кнопкой мыши по изображению устройства xc3s200 и выбрать команду *Program*. Откроется диалоговое окно *Programming Properties*. После установки всех необходимых значений параметров (в большинстве случаев ничего менять не нужно, оставить значения по умолчанию) нажать кнопку *OK*, чтобы запрограммировать микросхему FPGA;

8) после завершения процесса загрузки появится сообщение *Program Succeeded* в рабочей области основного окна и в окне сообщений модуля *iMPACT*.

Альтернативный способ настройки FPGA – загрузить файл конфигурации в конфигурационную флеш-память XCF02S, а затем загрузить конфигурационную информацию из конфигурационной флеш-памяти XCF02S в конфигурационную память FPGA. Процедура загрузки файла конфигурации в конфигурационную флеш-память XCF02S включает те же шаги, что и при загрузке в FPGA, с той лишь разницей, что на шаге 6 пропускается микросхема FPGA.

6.5. Порядок выполнения работы

1. Создать VHDL-модуль верхнего уровня устройства умножения для отладочной платы Spartan-3 Starter Kit, обеспечив ввод множимого и множителя с переключателей и вывод произведения на светодиоды.
2. Создать проект умножителя для отладочной платы Spartan-3 Starter Kit в среде проектирования Xilinx ISE.
3. Провести все этапы построения проекта, включая синтез, размещение и трассировку проекта. Файл временных и топологических ограничений UCF (top.ucf) получить у преподавателя.
4. Определить длительность такта работы умножителя.
5. Определить аппаратные затраты ПЛИС на реализацию умножителя.
6. Выполнить генерацию файла конфигурирования ПЛИС.
7. Запрограммировать полученную конфигурационную информацию в ПЛИС отладочной платы.
8. Проверить правильность работы умножителя.
9. Оформить отчет.

6.6. Содержание отчета

1. Титульный лист.
2. Цель работы.
3. Описание процесса построения модуля верхнего уровня устройства умножения для отладочной платы Spartan-3 Starter Kit.
4. Описание процесса синтеза проекта устройства умножения для отладочной платы Spartan-3 Starter Kit в среде проектирования Xilinx ISE.
5. Описание процесса настройки ПЛИС для реализации проекта устройства умножения с использованием отладочной платы Spartan-3 Starter Kit, включая генерацию файла конфигурирования ПЛИС и его программирование в ПЛИС отладочной платы.
6. Результаты проверки работы умножителя на отладочной плате.
7. Выводы по работе.

ЛАБОРАТОРНАЯ РАБОТА № 7.
ИЗУЧЕНИЕ АССЕМБЛЕРА УЧЕБНОГО ПРОЦЕССОРА С АРХИТЕКТУРОЙ MIPS

Цель работы – изучение структурной организации процессора с архитектурой MIPS; изучение программистской модели процессора с архитектурой MIPS; изучение способов адресации процессора с архитектурой MIPS; изучение основ языка ассемблера MIPS; получение практических навыков кодирования исходной программы на языке ассемблера MIPS.

7.1. Архитектура MIPS

7.1.1. Базовая структурная организация процессора с архитектурой MIPS

Базовый MIPS-процессор (MIPS R2000) состоит из целочисленного процессорного устройства (процессора) CPU и набора сопроцессоров, которые выполняют дополнительные задачи или работают (выполняют операции) с другими типами данных, такими как числа с плавающей запятой (ПЗ) (рис. 7.1).

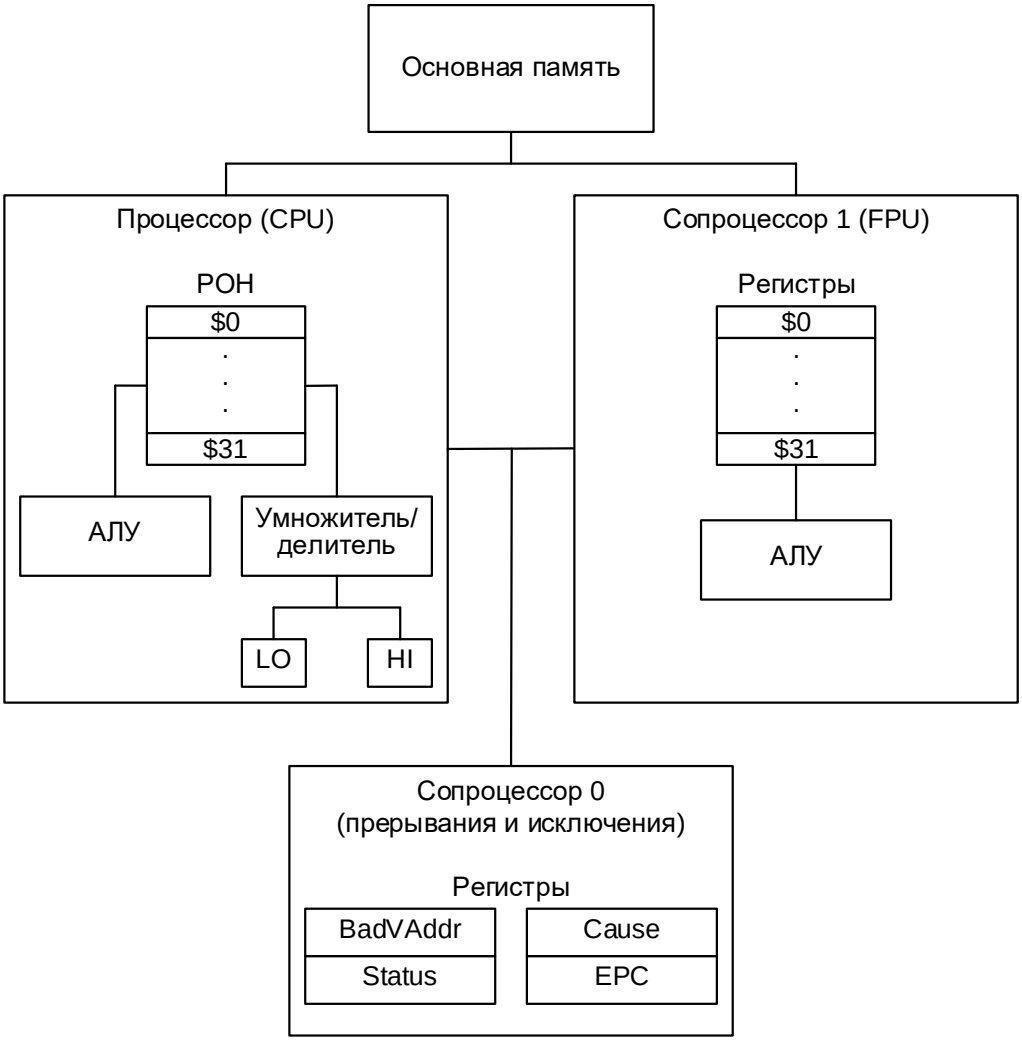

Рис. 7.1. Базовая организация MIPS-процессора

SPIM-симулятор MIPS-процессора поддерживает два сопроцессора:

1) сопроцессор 0 – обрабатывает исключения и прерывания;
2) сопроцессор 1 – устройство обработки чисел с ПЗ (процессор ПЗ) FPU.

7.1.2. Программистская модель (регистры) процессора с архитектурой MIPS

Процессор с архитектурой MIPS содержит 32 регистра общего назначения (РОН), программный счетчик `PC` и два специальных регистра `HI`, `LO` (рис. 7.2). Все регистры являются 32-разрядными. В ассемблере эти регистры обозначаются как `$0`, `$1`, ..., `$31`.

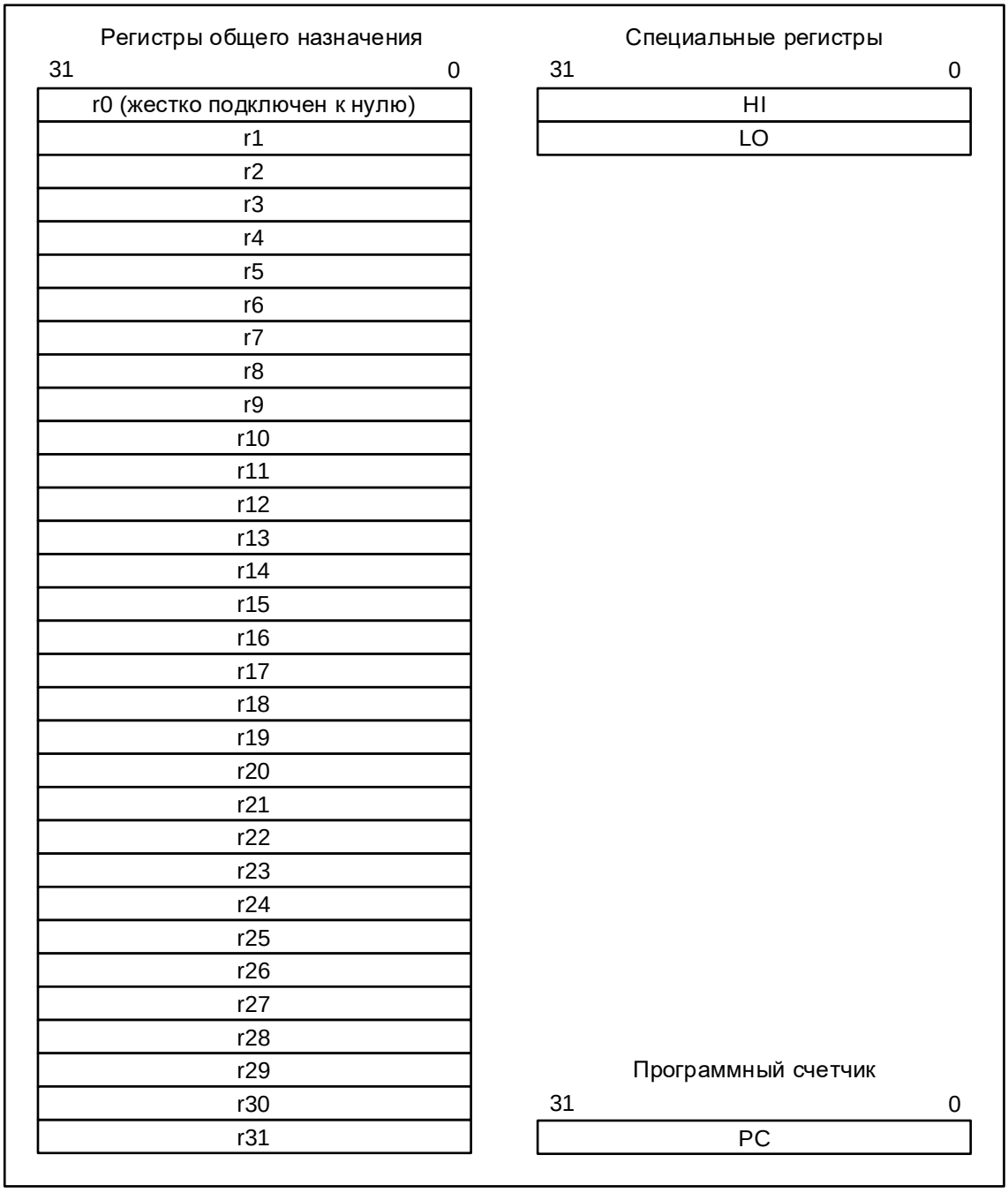

Рис. 7.2. Программистская модель (регистры) MIPS-процессора

Два регистра общего назначения – первый и последний – зарезервированы для определенной функции:

– регистр `$0` аппаратно (жестко) подключен так, чтобы обеспечивать нулевое значение. Он часто используется в качестве регистра источника, когда необходимо нулевое значение. Если этот регистр используется в команде в качестве регистра назначения, то результат отбрасывается;

– последний регистр `$31` используется в качестве регистра связи в команде управления (`jal`), которая используется для вызова процедуры. Регистр `$31` используется для хранения адреса возврата из вызываемой процедуры.

Два специальных регистра, называемых `HI` и `LO`, используются для хранения результатов команд целочисленного умножения и деления:

– в команде целочисленного умножения в регистрах `HI` и `LO` фиксируется 64-разрядный результат – старшие 32 бита в регистре `HI` и младшие 32 бита в регистре `LO`;

– в команде целочисленного деления 32-разрядное частное сохраняется в регистре `LO`, а остаток – в регистре `HI`.

На уровне аппаратуры процессора не существует никаких требований по использованию РОН. На программистском уровне установлены определенные соглашения о том, как эти 32 регистра должны использоваться (табл. 7.1). Так как эти соглашения не поддерживаются аппаратными средствами, мы можем использовать в программах регистры общего назначения произвольным образом. Однако, такие программы могут неправильно работать при взаимодействии с другими программами.

Таблица 7.1

Регистры MIPS-процессора и соглашения по их использованию

Имя регистра	Номер регистра	Назначение регистра
`zero`	0	Константа 0
`$at`	1	Зарезервирован для ассемблера
`$v0, $v1`	2, 3	Результаты процедуры
`$a0-$a3`	4-7	Первые четыре аргумента процедуры (не требуется сохранять при вызове процедуры)
`$t0-$t7`	8–15	Временные регистры (не требуется сохранять при вызове процедуры)
`$s0-$s7`	16–23	Временные регистры (требуется сохранять при вызове процедуры)
`$t8,$t9`	24, 25	Временные регистры (не требуется сохранять при вызове процедуры)
`$k0,$k1`	26, 27	Зарезервирован для ядра операционной системы
`$gp`	28	Указатель на глобальную область памяти
`$sp`	29	Указатель стека
`$fp`	30	Указатель кадра, если необходимо, в противном случае используется в качестве регистра временного хранения `$s8`
`$ra`	31	Адреса возврата (используется для возврата из процедуры)

Регистры `$v0` и `$v1` используются для возврата результатов из процедуры. Регистры `$a0-$a3` используются для передачи первых четырех аргументов в процедуру. Остальные аргументы передаются через стек. Эти регистры не требуется сохранять при вызове процедуры, т. е. вызываемая процедура может свободно изменять содержимое этих регистров.

Регистры `$t0-$t9`, `$s0-$s7` являются временными. Регистры `$t0-$t9` не нужно сохранять при вызове процедуры. Предполагается, что они сохраняются вызывающей процедурой. С другой стороны, регистры `$s0-$s7` требуют сохранения при вызове процедуры, т. е. вызываемая процедура должна сохранять их содержимое.

Регистр `$sp` является указателем стека, который используется для реализации стека. Он указывает на вершину стека. Регистр `$fp` является указателем кадра. MIPS-компилятор не использует указатель кадра, поэтому регистр `$fp` используется в качестве регистра временного хранения `$s8`. Регистр `$ra` используется для хранения адреса возврата при вызове процедуры.

Регистр `$gp` является указателем на глобальную область. Этот регистр указывает на область памяти, содержащую константы и глобальные переменные.

Регистр `$at` предназначен (зарезервирован) для ассемблера. Ассемблер использует этот регистр для перевода псевдокоманд. Псевдокоманды не являются машинными командами процессора, эти инструкции поддерживается только ассемблером. Каждая псевдокоманда переводится ассемблером в одну или несколько машинных команд процессора.

7.1.3. Организация памяти в архитектуре MIPS

Память имеет байтовую организацию. Ячейка памяти имеет разрядность 8 бит, т. е. обеспечивает хранение 1 байта данных. Память адресуется с точностью до байта, т. е. адрес относится к ячейке памяти (или, что тоже самое, к байту). Полуслово (16-разрядное слово) хранится в двух соседних ячейках памяти, имеет четный адрес. Слово (32-разрядное слово) хранится в четырех смежных ячейках памяти, имеет адрес, кратный 4.

Адресное пространство программы состоит из трех частей: кода, данных и стека. Расположение этих трех компонентов в памяти показано на рис. 7.3.

В сегменте кода (Text segment) хранятся команды программы. Эта область программы помещается в нижней части адресного пространства пользователя, начиная с адреса `0x04000000`.

Сегмент данных (Data segment) размещается над сегментом кода и начинается с адреса `0x10000000`. Сегмент данных делится на статическую и динамическую области. Динамическая область выделяется для динамических структур данных и растет в сторону увеличения адресов.

Сегмент стека (Stack segment) помещается в конце адресного пространства пользователя, начиная с адреса `0x7FFFFFFF`. Он растет вниз в сторону уменьшения адресов памяти. Такое размещение сегментов позволяет совместно использовать свободную память сегментом данных и сегментом стека.

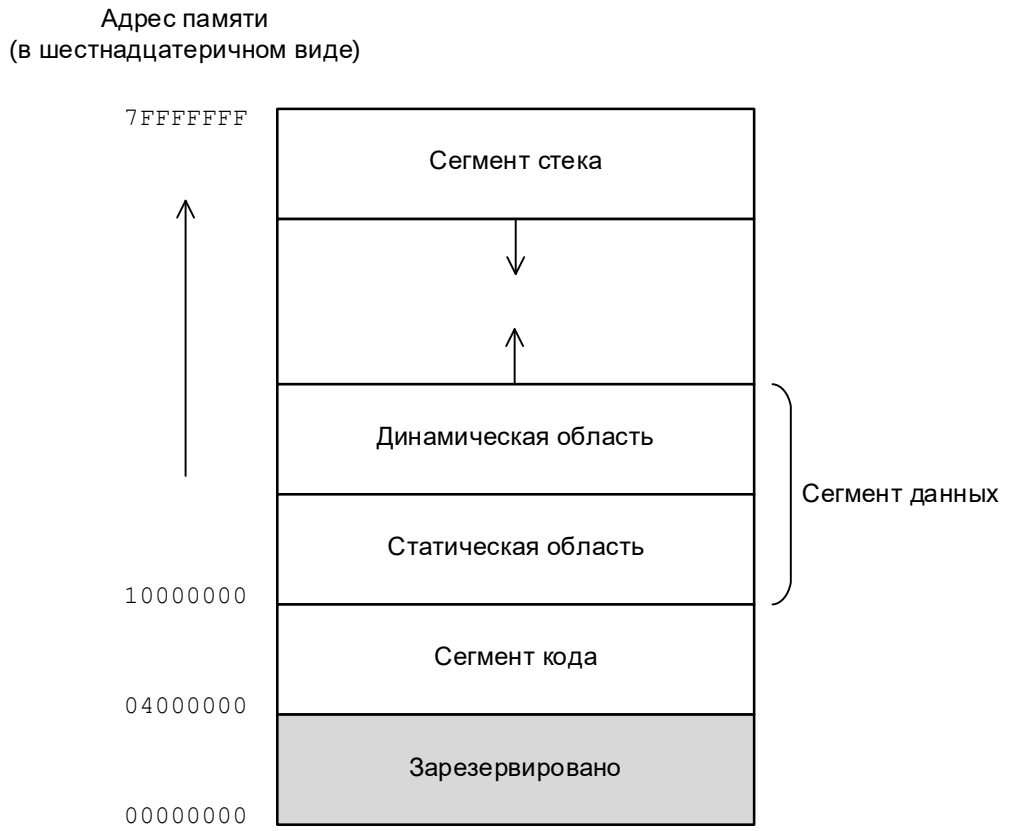

Рис. 7.3. Распределение адресного пространства программы MIPS-процессора

7.1.4. Способы адресации в процессоре с архитектурой MIPS

Так как MIPS-архитектура является архитектурой загрузки/сохранения (что характерно для RISC-архитектуры), то только команды загрузки и сохранения позволяют обращаться к памяти. Все остальные команды предполагают, что их операнды находятся в регистрах процессора. Таким образом, они используют регистровый способ адресации.

На аппаратном уровне MIPS-процессор поддерживает только один способ адресации памяти – регистровая относительная адресация или базовая адресация:

`disp(Rx)`,

где `disp` – смещение, которое является 16-битным значением со знаком.

При этом адрес вычисляется как:

`disp +` содержимое базового регистра `Rx`.

При вычислении адреса можно использовать любой регистр в качестве базового регистра `Rx`.

Ассемблер обеспечивает дополнительные способы адресации, облегчающие программирование на языке ассемблера.

На логическом уровне архитектура MIPS поддерживает следующие способы адресации:

– регистровая адресация;
– непосредственная адресация;
– способы адресации памяти.

Регистровая адресация. В этом режиме адресации операнды находятся в регистрах и результат операции тоже помещается в регистр. Например, в команде

```
add  $t2,$t1,$t0
```

два операнда-источника находятся в регистрах `$t1` и `$t0`. Результат операции заносится в регистр `$t2`. Большинство команд, за исключением загрузки и сохранения, используют этот режим адресации. Эти инструкции кодируются с помощью формата R-типа.

Непосредственная адресация. При непосредственной адресации операнд хранится в самой команде. Этот режим адресации используется для задания констант, как показано в следующем примере:

```
addi $t0,$t0,4.
```

Эта команда кодируется с помощью формата I-типа. Непосредственное значение задается с помощью 16 бит. Как следствие, константа ограничена 16-разрядным значением со знаком. Преимуществом этого способа адресации является то, что непосредственный операнд извлекается вместе с командой и не требуется отдельного доступа к памяти.

Способы адресации памяти. Данные режимы адресации используются для задания операндов, которые находятся в памяти. Они определяют способ вычисления адреса, по которому операнд читается из памяти или записывается в память. Табл. 7.2 показывает режимы адресации памяти, поддерживаемые ассемблером MIPS.

Таблица 7.2

Способы адресации памяти

Способ адресации	Способ вычисления адреса
`(Rx)`	Содержимое регистра `Rx`
`imm`	Значение `imm`
`imm(Rx)`	`imm` + содержимое регистра `Rx`
`symbol`	Значение символического имени `symbol`
`symbol±imm`	Значение `symbol±imm`
`symbol±imm(Rx)`	Значение `symbol±(imm` + содержимое регистра `Rx)`

7.2. Основы ассемблера MIPS

7.2.1. Кодирование исходной программы на языке ассемблера MIPS

Программа, написанная на языке ассемблера MIPS, состоит из последовательности исходных операторов. В свою очередь каждый исходный оператор состоит из последовательности ASCII-символов, заканчивающейся управляющим символом CR (возврат каретки, шестнадцатеричный код 0D). Набор символов, которые распознает ассемблер, является подмножеством стандарта ASCII (табл. 7.3).

Таблица 7.3

Набор символов, которые распознает ассемблер MIPS

Разряды 0–3	Разряды 4–6							
	0	1	2	3	4	5	6	7
0	NULL	DLE	SP	0	@	P	`	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	S1	US	/	?	O	_	o	DEL

Исходный текст программы не должен содержать непечатаемых управляющих символов. Поэтому он может быть подготовлен с использованием любого текстового редактора, который формирует выходной текст в ASCII формате (например, «Блокнот» или специализированный редактор для подготовки исходных текстов программ на языке ассемблера MIPS – Mipster).

Программы на языке ассемблера создаются из трех различных типов операторов. Операторы первого типа указывают процессору, что делать. Эти операторы называют исполняемыми командами или, для краткости, командами. Исполняемые команды соответствуют командам машинного языка (машинным командам). Второй тип операторов образуют псевдокоманды. Они напрямую (аппаратно) не поддерживаются процессором. Псевдокоманды поддерживаются ассемблером. Для их реализации ассемблер генерирует одну или несколько машинных команд. Третий тип операторов образуют директивы ассемблера. Директивы ассемблера являются неисполняемыми инструкциями и не генерируют никаких машинных команд. Они содержат информацию для ассемблера по различным аспектам процесса трансляции (ассемблирования) программы.

В каждой строке исходной программы на языке ассемблера задается только один оператор. Все три типа операторов языка ассемблера используют один и тот же формат:

```
[label:] mnemonic [operands] [#comment]
```

Поля в квадратных скобках являются необязательными в некоторых операторах. Для языка ассемблера обычной практикой является выравнивание полей относительно друг друга для того, чтобы улучшить читаемость программ. Ассемблер не учитывает количество пробелов между полями.

Метки используются, чтобы связать заданные в операторах объекты с адресами ячеек памяти. Метки обычно используют в сегменте кода и сегменте данных. В обоих сегментах метка заканчивается двоеточием.

В сегменте кода метки используются для идентификации команд, как показано ниже:

```
repeat: add $t0,$t0,1 #increment $t0 by 1
```

В этом примере метка `repeat` определяет адрес команды `add`.

В сегменте данных метки используются для идентификации операндов, как показано ниже:

```
number: .word 10 #initializes number to 10
```

В этом примере объявляется слово в памяти с адресом `number`, которое инициализируется значением 10.

В качестве меток не могут использоваться зарезервированные слова, такие как мнемоники команд, например `add`.

7.2.2. Директивы ассемблера

Перечень директив ассемблера:

1) `.TEXT address` – определяет сегмент кода;
2) `.DATA address` – определяет сегмент данных;
3) `.ASCII "string"` – определяет в памяти строку символов, которая не заканчивается символом `NULL`;

4) `.ASCIIZ "string"` – определяет в памяти строку символов, которая заканчивается символом `NULL`;

5) `.SPACE n` – определяет *n* байт в памяти, которые инициализируются кодом пробела;

6) `.BYTE b1,b2,...,bn` – определяет *n* 8-разрядных слов (байтов) в памяти;

7) `.HALF h1,h2,...,hn` – определяет *n* 16-разрядных слов (полуслов) в памяти;

8) `.WORD h1,h2,...,hn` – определяет *n* 32-разрядных слов (слов) в памяти;

9) `.ALIGN n` – выравнивает следующее данное на границу 2^n байт;

10) `.GLOBL` – определяет глобальное символическое имя.

7.2.3. Системные вызовы симулятора SPIM

Для обеспечения поддержки ввода/вывода при отладке программ с помощью симулятора SPIM используются инструкции системного вызова `syscall` (табл. 7.4). Восемь из этих вызовов обеспечивают ввод и вывод четырех основных типов данных: строка, целое число, число с плавающей запятой и двойное число. В этом списке отсутствуют ввод и вывод символов. Для символьного ввода/вывода необходимо использовать строковые системные вызовы.

Таблица 7.4

Системные вызовы симулятора SPIM

Системный вызов	Номер системного вызова (в регистре `$v0`)	Аргументы системного вызова	Результат выполнения системного вызова
`print_int`	1	`$a0 = integer`	
`print_float`	2	`$f12 = float`	
`print_double`	3	`$f12 = double`	
`print_string`	4	`$a0 = string address`	
`read_int`	5		`$v0 = integer`
`read_float`	6		`$f0 = float`
`read_double`	7		`$f0 = double`
`read_string`	8	`$a0 = buffer address` `$a1 = buffer size`	
`sbrk`	9		`$v0 = address`
`exit`	10		

Чтобы вызвать выполнение необходимого действия, код соответствующего системного вызова должен быть помещен в регистр `$v0`. Все необходимые аргу-

менты помещаются в регистры $a0 и $a1 (для действий с плавающей запятой необходимо использовать регистр $f12). Любое значение, возвращаемое системным вызовом, помещается в регистр $v0 ($f0 для плавающей запятой).

Системный вызов print_int принимает из регистра $a0 32-разрядное слово и выводит на экран соответствующее ему целое число.

Системный вызов print_string принимает из регистра $a0 указатель на строку, завершающуюся символом NULL, и выводит на экран эту строку.

Системный вызов read_int вызывает ввод целого числа с клавиатуры.

Системный вызов read_string принимает в регистре $a0 указатель на буфер, в который должна быть помещена входная строка, а в регистре $a1 – размер буфера n. Размер буфера должен быть выражен в байтах. Он считывает не более $n - 1$ символов в буфер и завершает строку символом NULL.

Системный вызов sbrk возвращает указатель на блок памяти, содержащий n байтов. Последний системный вызов exit останавливает выполнение программы пользователя.

7.2.4. Шаблон программы на языке ассемблера MIPS

При составлении программы на языке ассемблера MIPS необходимо следовать приведенному ниже шаблону.

Шаблон программы на языке ассемблера MIPS

```
# Title of the program                                    Filename
#
# Objective:
# Input:
# Otput:
#
# Register usage
#
#################### Data segment ########################
      .data
      . . .
      data go here
      . . .

#################### Code segment ########################
      .text
      .globl main
main:
      . . .
      code goes here
      . . .
      li    $v0,10           # exit
      syscall
```

Шаблон состоит из двух основных частей: сегмента данных и сегмента кода. Сегмент данных указывается директивой .data. Эта часть обычно содержит сообщения, которые будут использоваться в пользовательском интерфейсе, и выделение памяти для переменных (как инициализированных, так и неиници-

ализированных). Сегмент кода указывается директивой `.text`. Выполнение программы начинается с оператора, идентифицируемого в сегменте кода меткой `main`, которая должна быть определена как глобальное символическое имя. Завершается выполнение программы с помощью системного вызова выхода `exit`.

В примере 7.1 приведен фрагмент программы, который запрашивает у пользователя имя и считывает его с использованием системных вызовов симулятора SPIM.

Пример 7.1. Использование системных вызовов симулятора SPIM

```
################### Data segment ########################
      .data
prompt:
      .ASCIIZ "Enter your name: "
in_name:
      .space 31

################### Code segment ########################
      .text
      .globl main
main:
      . . .
      la    $a0,prompt        # prompt user
      li    $v0,4
      syscall

      la    $a0,in_name       # read name
      li    $a1,31
      li    $v0,8
      syscall
      . . .
      li    $v0,10            # exit
      syscall
```

7.2.5. Примеры команд языка ассемблера MIPS

Команды пересылки. Команда

```
move Rdest,Rsrc
```

выполняет пересылку слова между двумя РОН: копируется содержимое регистра `Rsrc` в регистр `Rdest`.

Эта команда является псевдокомандой. Ассемблер заменяет ее машинной командой следующим образом:

```
move Rd, Rs   ⇒   addu Rd, $0, Rs.
```

Команда

```
lui Rdest,const
```

загружает 16-разрядную константу `const` в старшее полуслово регистра `Rdest`. Младшее полуслово устанавливается в нулевое значение (заполняется нулями).

Команда

```
la Rdest,var
```

загружает адрес второго операнда `var` в регистр `Rdest`.

Например, команда

```
la $a0,marks
```

загружает адрес данного отмеченного меткой `marks` в регистр `$a0` (фактически значение символического имени `marks`, которое используется в качестве метки).

Эта команда является псевдокомандой. Ассемблер заменяет ее последовательностью машинных команд следующим образом:

```
la Rd,addr    ⇒    lui $at,%hi(addr)
                   ori Rd,$at,%lo(addr),
```

где `%hi(addr)` и `%lo(addr)` – соответственно старшие и младшие 16 разрядов адреса.

Команда

```
li Rdest,imm
```

загружает непосредственное значение `imm` в регистр `Rdest`.

Эта команда также является псевдокомандой. Ассемблер заменяет ее в зависимости от значения непосредственного операнда последовательностью машинных команд следующим образом:

```
li $4,0x8000      ⇒    ori $4,$0,0x8000,
li $5,0x120000    ⇒    lui $5,0x12,
li $6,0x12345     ⇒    lui $6,0x1
                       ori $6,$6,0x2345.
```

Арифметические команды. Команда

```
add Rdest,Rsrc1,Rsrc2
```

суммирует содержимое регистров `Rsrc1` и `Rsrc2` и сохраняет результат в регистре `Rdest`. Числа рассматриваются как целые со знаком. В случае переполнения генерируется прерывание (исключение).

Команда

```
addu Rdest,Rsrc1,Rsrc2
```

полностью аналогична команде `add`, но не генерирует исключения при переполнении.

Второй операнд может быть задан как 16-разрядное непосредственное значение `imm`:

```
addi(addiu)   Rdest,Rsrc1,imm.
```

В этом случае 16-разрядное значение `imm` расширяется знаком до 32 разрядов и суммируется с содержимым регистра `Rsrc1`.

Псевдокоманды

```
add(addu)Rdest,Rsrc1,Src2
```

позволяют использовать в качестве второго операнда либо 16-разрядное непосредственное значение, либо регистр.

Логические команды. Команда

```
and(or)  Rdest,Rsrc1,Rsrc2
```

выполняет поразрядное И (ИЛИ) содержимого регистров `Rsrc1` и `Rsrc2` и сохраняет результат в регистре `Rdest`.

Второй операнд может быть задан как 16-разрядное непосредственное значение `imm`, которое расширяется нулями до 32 разрядов:

`and(or)  Rdest,Rsrc1,imm`.

7.2.6. Пример программы на языке ассемблера MIPS

В примере 7.2 приведена программа, суммирующая три целых числа, которые вводятся с клавиатуры. Полученная сумма выводится на экран.

Пример 7.2. Программа нахождения суммы трех целых чисел

```
1:   # The sum of three numbers                                  sum3.asm
2:   #
3:   # Objective: Finds the sum of three integers.
4:   # Input: Requests the input of three numbers from the keyboard.
5:   # Output: Displays the sum on the screen.
6:   #
7:   ################### Data segment ###################
8:          .data
9:   prompt:
10:         .asciiz     "Please enter three numbers: \n"
11:  sum_msg:
12:         .asciiz "The sum is: "
13:  newline:
14:         .asciiz "\n"
15
16:  ################### Code segment ###################
17:         .text
18:         .globl main
19:  main:
20:         la    $a0,prompt              # request for input of numbers
21:         li    $v0,4
22:         syscall
23:
24:         li    $v0,5                   # 1st number read is placed into $t0
25:         syscall
26:         move  $t0,$v0
27:
28:         li    $v0,5                   # 2nd number read is placed into $t1
29:         syscall
30:         move  $t1,$v0
31:
32:         li    $v0,5                   # 3rd number read is placed into $t2
33:         syscall
34:         move  $t2,$v0
35:
36:         addu  $t0,$t0,$t1
37:         addu  $t0,$t0,$t2
38:
39:         la    $a0,sum_msg             # displaying a message about the sum
40:         li    $v0,4                   # on the screen
41:         syscall
42:
43:         move  $a0,$t0                 # displaying the sum on the screen
44:         li    $v0,1
45:         syscall
46:
47:         la    $a0,newline             # write newline
48:         li    $v0,4
```

Продолжение примера 7.2

```
49:         syscall
50:
51:         li    $v0,10                  # exit
52:         syscall
```

Сегмент данных состоит из трех сообщений. Все три сообщения объявлены как строки ASCII, завершающиеся символом `NULL` (см. строки 9–14). Фактически исполнимая часть программы начинается со строки 20. Программа предлагает пользователю ввести три целых числа, отображая строку приглашения с помощью системного вызова `print_string` (см. строки 20–22). Затем используется системный вызов `read_int` для чтения трех входных целых чисел. Первое целое число помещается в регистр `$t0` (см. строки 24–26). Следующие два целых числа помещаются в регистры `$t1` и `$t2` соответственно. Три целых числа складываются с помощью двух команд сложения в строках 36 и 37. Эти команды помещают сумму в регистр `$t0`. Затем отображается сообщение о сумме (см. строки 39–41). Сумма выводится с помощью системного вызова `print_int` (см. строки 43–45). После перехода на новую строку (см. строки 47–49) программа завершает работу с помощью системного вызова выхода (см. строки 51 и 52).

7.3. Порядок выполнения работы

1. Изучить основы архитектуры и языка ассемблера MIPS.
2. Записать назначение каждой строки программы, приведенной в п. 7.2.6.
3. Разработать программу на языке ассемблера MIPS для решения следующей задачи: *Найти сумму элементов массива целых чисел. Размер массива и сами элементы вводятся с клавиатуры. Полученная сумма выводится на экран.*
4. Записать назначение каждой строки разработанной программы.
5. Выполнить разработанную программу вручную, записав для каждой команды выполняемое действие и получаемый результат.
6. Оформить отчет.

7.4. Содержание отчета

1. Титульный лист.
2. Цель работы.
3. Краткие сведения по основам архитектуры и языка ассемблера MIPS.
4. Результат выполнения п. 2 подразд. 7.3.
5. Исходный текст программы на языке ассемблера MIPS для решения задачи, заданной в п. 3 подразд. 7.3.
6. Описание разработанной программы в соответствии с п. 4 подразд. 7.3.
7. Результат выполнения разработанной программы в соответствии с п. 5 подразд. 7.3.
8. Выводы по работе.

ЛАБОРАТОРНАЯ РАБОТА № 8.
ИЗУЧЕНИЕ ОТЛАДОЧНОГО СРЕДСТВА (СИМУЛЯТОРА) ДЛЯ УЧЕБНОГО ПРОЦЕССОРА С АРХИТЕКТУРОЙ MIPS

Цель работы – изучение симулятора SPIM процессора с архитектурой MIPS (версия PCSpim для Windows); получение практических навыков создания и отладки программы на языке ассемблера MIPS с помощью симулятора.

8.1. Симулятор SPIM процессора с архитектурой MIPS

8.1.1. Общая характеристика симулятора SPIM

SPIM – это программный симулятор, который запускает программы на языке ассемблера, написанные для процессоров, реализующих архитектуру MIPS32, в частности версию 1 этой архитектуры с фиксированным распределением памяти, без кешей и только для сопроцессоров 0 и 1. Более ранние версии симулятора SPIM (до 7.0) поддерживают вариант архитектуры MIPS-I, которая использовалась в оригинальных процессорах MIPS R2000. Эта архитектура является практически подмножеством архитектуры MIPS32, с отличием, заключающемся в способе обработки исключений. Также MIPS32 содержит около 60 новых инструкций, которые не поддерживались ранними версиями SPIM. Программы, которые работали в более ранних версиях SPIM и не использовали исключения, должны работать в более новых версиях SPIM без изменений. Программы, использующие исключения, потребуют незначительных изменений.

Имя SPIM – это просто MIPS, написанное наоборот. SPIM может читать и немедленно выполнять файлы языка ассемблера. Симулятор SPIM – это автономная система для запуска программ на языке ассемблера MIPS. Он содержит отладчик и предоставляет несколько служб, подобных операционной системе.

По умолчанию симулятор SPIM имитирует виртуальную машину, которую реализует ассемблер, поскольку именно эта машина окажется полезной для большинства программистов. При этом SPIM следует традиции программистов (и компиляторов) языка ассемблера MIPS, которые обычно используют расширенную машину так, как если бы она была реализована физически (аппаратно в виде реальной вычислительной машины на базе процессора с архитектурой MIPS).

8.1.2. Краткое руководство по использованию симулятора SPIM

В данной лабораторной работе описывается симулятор SPIM, который был разработан профессором Джеймсом Ларусом, когда он работал на факультете компьютерных наук университета Висконсина, Мэдисон. Этот симулятор выполняет программы, написанные для процессоров MIPS R2000/R3000. Это продукт «два в одном»: он содержит симулятор запуска программ MIPS, а также отладчик.

SPIM работает на различных платформах, включая UNIX, Linux, Windows и DOS. Версия SPIM для Microsoft Windows назвается PCSpim.

На рис. 8.1 показан интерфейс симулятора PCSpim. Как показано на этом рисунке, PCSpim предоставляет строку меню и панель инструментов вверху, а также строку состояния внизу экрана. Средняя область окна симулятора разделяется на четыре панели.

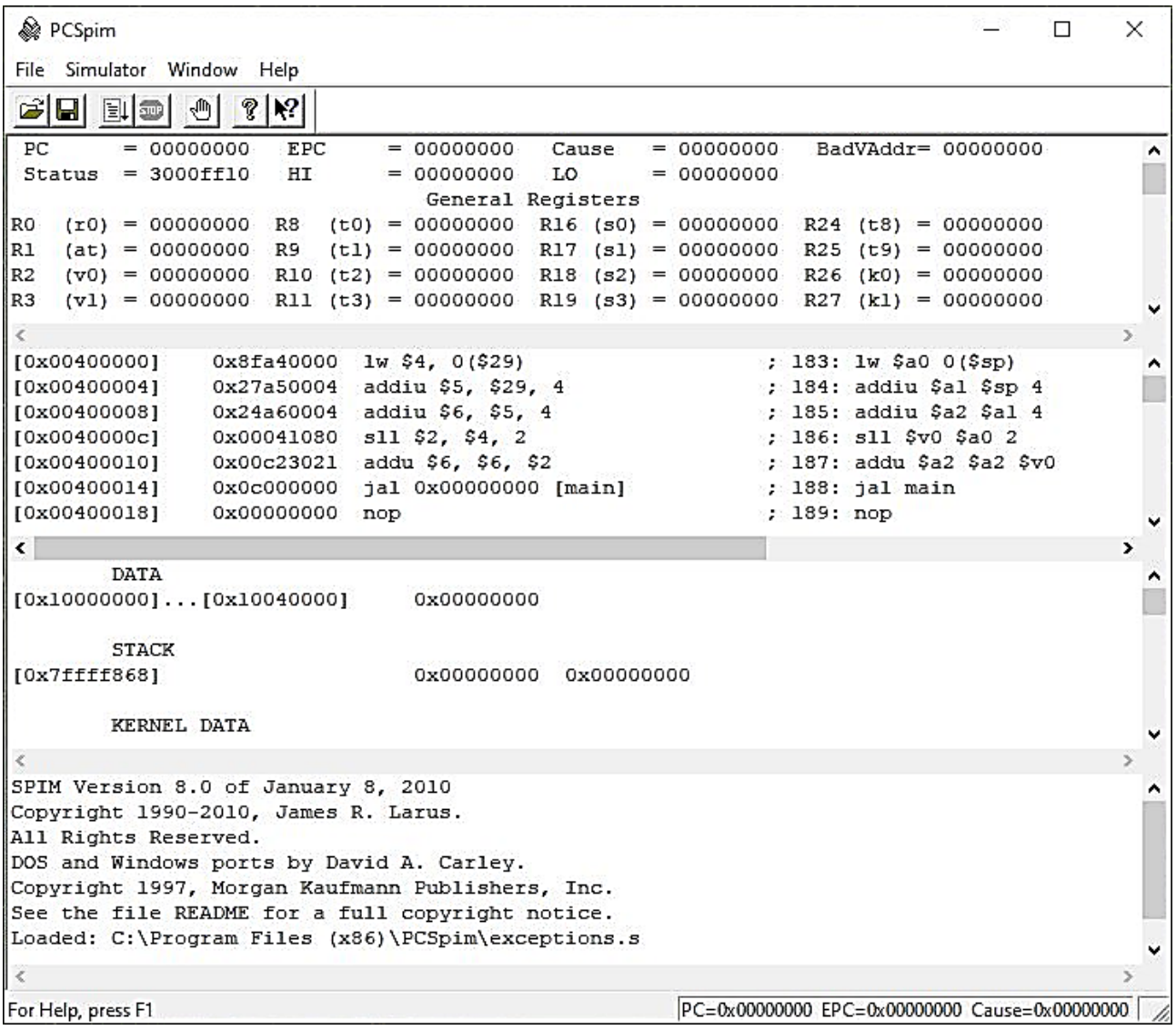

Рис. 8.1. Интерфейс симулятора PCSpim

Строка меню содержит следующие команды для работы симулятора:

– *File*: позволяет выбирать операции с файлами. Можно открыть файл с исходным текстом программы на языке ассемблера с помощью команды `open…` или сохранить файл журнала текущего состояния симулятора. Кроме того, можно выйти из PCSpim, выбрав команду `Exit`. Конечно, также можно выйти из PCSpim, закрыв окно;

– *Simulator*: содержит несколько команд для запуска и отладки программы (будут описаны ниже). Это меню также позволяет выбрать настройки симулятора. При выборе команды `Settings…` открывается окно для установки настроек симулятора, которые будут рассмотрены ниже;

– *Windows*: позволяет управлять навигацией по панелям окна симулятора. Кроме того, при помощи этого меню также можно скрыть или отобразить панель инструментов и строку состояния. Окно консоли `Console` появляется, когда программе пользователя необходимо прочитать данные или записать их на терминал (ввести с клавиатуры или вывести на экран). Оно исчезает после завершения работы программы. Для того чтобы увидеть процесс ввода и вывода пользовательской программы, необходимо активировать это окно, выбрав команду `Console`;

– *Help*: позволяет получить справку по PCSpim.

На панели инструментов имеются кнопки для открытия и закрытия файла, содержащего исходный текст программы на языке ассемблера MIPS, запуска и вставки точек останова, а также для получения справки.

Строка состояния в нижней части окна PCSpim состоит из трех частей:

– левая часть используется для предоставления информации о пунктах меню и кнопках панели инструментов. Например, когда стрелка мыши находится на значке открытия файла (первая кнопка) на панели инструментов, в этой области отображается сообщение «*Open an assembly file*»;

– в средней части в ранних версиях симулятора SPIM (до 7.0) показываются текущие настройки симулятора. В поздних версиях в этой части отображается содержимое программного счетчика `PC` и двух регистров сопроцессора 0, связанных с обработкой исключений и прерываний (`EPC` – содержит адрес команды, при выполнении которой возникло исключение, и `Cause` – фиксирует запросы прерывания и задает код исключения, определяющий причину исключения);

– правая часть в ранних версиях симулятора SPIM (до 7.0) используется для отображения того, активны ли клавиши Caps Lock (CAP), Num Lock (NUM) и Scroll Lock (SCRL). В поздних версиях эта часть не используется.

Средняя область окна симулятора включает следующие четыре панели:

– верхняя панель показывает значения всех регистров ЦП (CPU), сопроцессора 1 (FPU) и сопроцессора 0 (прерывания и исключения) MIPS-процессора. Эта панель обновляется всякий раз, когда программа перестает выполняться (останавливается);

– на следующей панели отображаются команды как программы пользователя, так и системного кода, который загружается автоматически при запуске PCSpim. Каждая команда отображается в строке, которая выглядит как

```
[0x00400000] 0x8fa40000 lw $4, 0($29)   ; 89: lw $a0, 0($sp)
```

Первый элемент в строке в квадратных скобках – это шестнадцатеричный адрес команды. Второй элемент – это машинный код команды, отображаемый также в шестнадцатеричном виде. Третий элемент – мнемоническое описание машинной команды. Все, что следует за точкой с запятой, является фактической строкой из исходного текста программы. Число 89 – это номер строки в этом

тексте. Иногда после точки с запятой в строке ничего нет. Это означает, что команда была создана SPIM как часть трансляции псевдоинструкции в более чем одну реальную машинную команду MIPS;

– третья панель отображает данные, загруженные в память программы, и данные в стеке программы;

– в нижней панели PCSpim записывает сообщения. Здесь появляются сообщения об ошибках.

Чтобы запустить программу, нужно либо перейти в меню *Simulator* и нажать `Go`, либо щелкнуть по значку *Go*, либо просто нажать на клавишу F5. В любом случае PCSpim выводит диалоговое окно с двумя записями и двумя кнопками. В первой записи задается начальный адрес программы, а во второй можно ввести параметры командной строки для запуска программы. В большинстве случаев эти записи содержат правильные значения для запуска программы, поэтому их можно не изменять. Обратите внимание: когда программа работает, PCSpim не обновляет панель отображения регистров, поскольку регистры постоянно изменяются. Вы всегда можете определить, запущен ли PCSpim, посмотрев на эту панель или на строку заголовка в верхней части окна PCSpim.

Чтобы остановить программу, нужно перейти в меню *Simulator* и нажать `Break`. Это приведет к тому, что PCSpim откроет диалоговое окно с двумя кнопками. Прежде чем нажать любую кнопку, можно просмотреть регистры и память, чтобы узнать, что делала программа, когда ее остановили. Когда вы поймете, что произошло, можно либо продолжить работу программы, нажав `Yes`, либо остановить ее, нажав `No`.

SPIM имеет две функции, помогающие отлаживать программу. Первая и, возможно, самая полезная – пошаговый режим, который позволяет запускать программу по одной команде за раз. Для этого нужно нажать `Single Step` в меню *Simulator*. При этом PCSpim будет выполнять одну команду и обновлять свое окно, чтобы можно было видеть, что команда изменила в регистрах или памяти.

Второй функцией является использование точки останова, которая сообщает PCSpim, что нужно остановить выполнение программы непосредственно перед выполнением определенной команды. В меню *Simulator* есть пункт `Breakpoints`, по которому программа PCSpim откроет диалоговое окно с полем ввода и списком. В это поле необходимо ввести адрес команды, по которой PCSpim должен остановиться. Если команда имеет глобальную метку, можно просто ввести имя метки. Маркированные точки останова – особенно удобный способ остановиться на первой команде процедуры. Чтобы установить точку останова, нужно нажать кнопку `Add`. Тогда она появится в списке активных точек останова. Можно установить любое количество точек останова. Чтобы удалить точку останова, нужно выбрать ее в списке и нажать `Remove`. После установки точек останова необходимо закрыть диалоговое окно и запустить программу.

Когда симулятор собирается выполнить команду с точкой останова, PCSpim выводит диалоговое окно с адресом инструкции и двумя кнопками. Кнопка `Yes` продолжает работу программы, а кнопка `No` останавливает программу. Прежде чем нажать любую кнопку, можно проверить значения в регистрах или памяти, а также можно добавить или удалить точки останова.

Пошаговое выполнение и установка точек останова помогут быстро найти ошибку в программе. Для того чтобы исправить ошибку, нужно вернуться в редактор, который использовался для создания исходного текста программы, и изменить его. Чтобы снова запустить программу, нужна новая копия PCSpim, которую можно получить одним из двух способов. Для этого нужно либо выйти из PCSpim, выбрав пункт `Exit` в меню *File* (или нажать комбинацию клавиш Alt + F4), либо просто перезагрузить программу, выбрав пункт `Reload` в меню *Simulator*. При перезагрузке программы PCSpim спросит, нужно ли повторно инициализировать симулятор. Необходимо ответить «да», если вы изменили свою программу и хотите попробовать выполнить ее еще раз.

8.1.3. Настройки симулятора SPIM

Настройки PCSpim можно просмотреть, выбрав команду `Settings…` в меню *Simulator*. Откроется окно настройки, как показано на рис. 8.2.

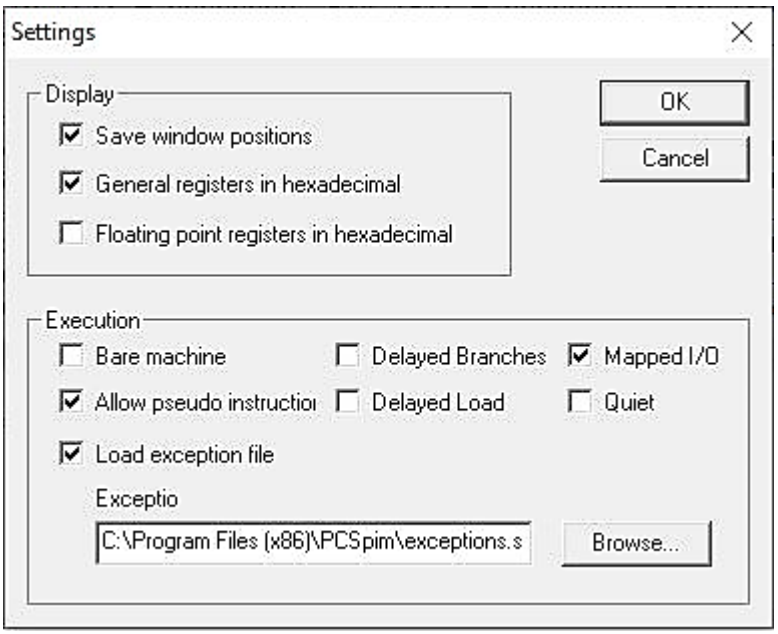

Рис. 8.2. Окно настройки симулятора PCSpim

PCSpim использует эти настройки, чтобы определить, как загружать и запускать программу MIPS. Неправильная настройка может привести к ошибкам. Настройки разделены на две группы: `Display` (отображение) и `Execution` (выполнение). Настройки `Display` определяют, сохраняются ли позиции окон и как отображается содержимое регистров. Если выбран параметр `Save windows positions`, PCSpim запомнит положение своих окон при выходе и восстановит их при последующем запуске PCSpim. Если выбрать

опцию отображения регистров, содержимое регистров общего назначения и регистров с плавающей запятой будет отображаться в шестнадцатеричном формате. В противном случае содержимое регистра отображается в виде десятичных чисел.

Часть настроек `Execution` определяет, как будет выполняться программа:

– `Bare machine`. Если выбран этот параметр, SPIM имитирует «голую» машину MIPS. Это значит, что не допускаются как псевдоинструкции, так и дополнительные режимы адресации, предоставляемые ассемблером. Поскольку программы MIPS удобнее разрабатывать, используя приведенные дополнительные возможности ассемблера, эту опцию не следует выбирать для запуска разрабатываемых вами программ;

– `Allow pseudoinstructions`. Этот параметр определяет, разрешены ли псевдоинструкции в исходном тексте программы. Следует выбирать эту опцию, поскольку наши программы используют псевдоинструкции;

– `Mapped I/O`. Если выбран этот параметр, нельзя использовать системные вызовы SPIM для чтения с терминала. Таким образом, этот параметр не следует выбирать для запуска разрабатываемых вами программ;

– `Quiet`. Если выбран этот параметр, PCSpim будет печатать сообщение при возникновении исключения;

– `Load exeption file`. Выбор этого параметра заставляет PCSpim загружать стандартный обработчик исключений и код запуска. Обработчик ловушек можно выбрать с помощью кнопки `Browse…` . При загрузке код запуска в файле ловушки вызывает основную процедуру. В этом случае можно пометить первый исполняемый оператор программы как `main`. Если файл ловушки не выбран, PCSpim начинает выполнение с оператора, помеченного как `start`. Ваши программы написаны с учетом того, что файл ловушки загружен (используется метка `main`). Файл ловушки находится в папке, в которую установлена программа симулятора PCSpim.

ВНИМАНИЕ! Проверяйте, какой путь задан в окне настроек.

8.2. Порядок выполнения работы

1. Изучить основные сведения по симулятору SPIM процессора с архитектурой MIPS, версия PCSpim для Windows (или по симулятору MARS самостоятельно, если вы выбрали его).

2. Разработать программу на языке ассемблера MIPS для решения следующей задачи (задача из лабораторной работы № 7): *Найти сумму элементов массива целых чисел. Размер массива и сами элементы вводятся с клавиатуры. Полученная сумма выводится на экран.* **Предусмотреть размещение исходного массива в памяти.**

3. Загрузить исходный текст разработанной программы в симулятор PCSpim. Найти в соответствующей панели окна симулятора PCSpim команды программы и ее данные.

4. Найти в разработанной программе все псевдоинструкции. Записать для каждой псевдоинструкции набор машинных команд, в которые она была преобразована ассемблером (симулятором).

5. Выполнить разработанную программу в пошаговом режиме, записав для каждой команды выполняемое действие и получаемый результат (содержимое памяти и регистров).

6. Выполнить разработанную программу с использованием точек останова (не менее трех), записав для каждой точки останова получаемый результат (содержимое памяти и регистров).

7. Для пп. 3–6 данного подраздела, привести скриншоты окна симулятора (или его фрагментов) PCSpim, поясняющие полученные результаты.

8. Оформить отчет.

8.3. Содержание отчета

1. Титульный лист.
2. Цель работы.
3. Краткие сведения по симулятору SPIM процессора с архитектурой MIPS, версия PCSpim для Windows (или по симулятору MARS самостоятельно, если вы выбрали его).
4. Исходный текст программы на языке ассемблера MIPS для решения задачи, приведенной в п. 2 подразд. 8.2.
5. Описание разработанной программы с учетом п. 3 подразд. 8.2.
6. Результат выполнения разработанной программы в соответствии с пп. 4–7 подразд. 8.2.
7. Выводы по работе.

ЛАБОРАТОРНАЯ РАБОТА № 9. ПРОГРАММИРОВАНИЕ НА АССЕМБЛЕРЕ УЧЕБНОГО ПРОЦЕССОРА С АРХИТЕКТУРОЙ MIPS

Цель работы – изучение методики программирования на языке ассемблера MIPS с использованием нотации псевдокода высокого уровня; получение практических навыков разработки программ на языке ассемблера MIPS.

9.1. Разработка программ на языке ассемблера MIPS

9.1.1. Общая характеристика методики программирования на языке ассемблера MIPS с использованием нотации псевдокода высокого уровня

Языки высокого уровня при разработке программных приложений обеспечивают удобную абстракцию базовой системы, подходящую для решения конкретных задач. Разработка программы происходит быстрее. Многие языки высокого уровня предоставляют структуры (последовательные, выборочные, итеративные), которые облегчают разработку программ. Программы, написанные на языке высокого уровня, относительно малы по сравнению с эквивалентными программами, написанными на языке ассемблера. Их также легче кодировать и отлаживать. Такие программы легче понять и, при соблюдении хороших правил программирования, легче поддерживать. Программы на языке ассемблера, как правило, сложно разрабатывать, они большие и требуют больше времени для написания кода и отладки. В результате их также сложно поддерживать.

Однако есть три основные причины, по которым программирование попрежнему выполняется на языке ассемблера. Первые две – это эффективность и доступность системных аппаратных средств вычислительной машины.

Эффективность означает способность программы хорошо решать задачу с точки зрения двух критериев, основанных на пространстве (эффективность использования пространства) и времени (эффективность использования времени). Эффективность использования пространства относится к требованиям программы к памяти, т. е. к размеру исполняемого кода. Говорят, что программа А более экономична, если для выполнения той же задачи ей требуется меньше памяти, чем программе В. Очень часто программы, написанные на языке ассемблера, оказываются более компактными, чем программы, написанные на языке высокого уровня. Эффективность использования времени относится к времени, затраченному на выполнение программы. Очевидно, что программа, которая работает быстрее, считается лучшей с точки зрения эффективности использования времени. Если мы тщательно создаем программы на языке ассемблера, они, как правило, работают быстрее, чем их аналоги на языке высокого уровня.

Основной причиной написания программ на языке ассемблера является создание кода, который выполняется быстрее. Превосходство программ на языке ассемблера в создании эффективного кода является прямым проявлением того,

что программы на ассемблере содержат только тот код, который необходим для выполнения поставленной задачи.

Третья основная причина написания программ на языке ассемблера – прямой контроль над аппаратными средствами вычислительной машины. Языки высокого уровня намеренно предоставляют ограниченное (абстрактное) представление о базовом оборудовании. Из-за этого практически невозможно выполнить некоторые задачи, требующие непосредственного доступа к аппаратным средствам. Например, написание драйвера устройства. Поскольку язык ассемблера не накладывает никаких ограничений, можно напрямую управлять аппаратными средствами системы. Если разрабатывается системное программное обеспечение, то нельзя избежать написания программ на языке ассемблера.

Разумным подходом к упрощению программирования на языке ассемблера MIPS является первоначальная разработка алгоритма с использованием нотации псевдокода высокого уровня. Опытные программисты разрабатывают свои алгоритмы, используя конструкции программирования высокого уровня, такие как

– `If (condition) do {this block of code} else do {that block of code};`

– `While (condition) do {this block of code};`

– `For (t0=1, t0 < s0, t0++) do {this block of code};`

Затем на заключительном этапе эти выражения псевдокода высокого уровня переводятся на язык ассемблера MIPS. Другими словами, на заключительном этапе программист выполняет ту же функцию, что и компилятор, а именно переводит код высокого уровня в эквивалентный код на языке ассемблера.

9.1.2. Краткое руководство по использованию нотации псевдокода высокого уровня при программировании на языке ассемблера MIPS

При реализации алгоритма на таком языке, как Паскаль, C или JAVA, программисты используют описательные имена переменных, такие как: сумма, размер, количество и т. д. После компиляции программы эти имена переменных соответствуют ячейкам памяти. Значения, хранящиеся в данных ячейках памяти, соответствуют значениям этих переменных. Компилятор пытается разработать код таким образом, чтобы сохранить переменные, на которые чаще всего ссылаются, в регистрах процессора, поскольку доступ к переменной в регистре процессора происходит быстрее, чем доступ к памяти. Процессор MIPS имеет 32 регистра. При этом архитектура MIPS предполагает, чтобы операнды команд обработки данных и команд управления находились в регистрах.

Программист на языке ассемблера MIPS должен указать в каждой инструкции, какие регистры процессора будут использоваться. Например, в регистре `$t0` может быть значение, соответствующее скорости, в регистре `$t2` значение,

соответствующее размеру, и в регистре `$t3` значение, соответствующее счетчику. При использовании псевдокода для разработки программы на ассемблере необходимо использовать имена регистров, которые в дальнейшем будут использованы в программе на ассемблере. Целесообразно создать таблицу перекрестных ссылок, которая определяет, для чего используется каждый регистр процессора в алгоритме. Например, `$t0: Sum`, `$t3: Count`. Имена регистров используются в псевдокоде для того, чтобы перевод на язык ассемблера был простым процессом. Кроме того, это позволяет описать, как использовались средства архитектуры MIPS, чтобы реализовать алгоритм. Если не идентифицировать используемые регистры, псевдокод будет ограничен с точки зрения создания программы на языке ассемблера и ничто не упростит этот процесс (для чего собственно говоря и предлагается использовать псевдокод).

Псевдокод для программ на языке ассемблера будет иметь вид Паскаля или С с точки зрения управляющих структур и арифметических выражений. Описательные имена переменных обычно появляются только в инструкции загрузки адреса (`la`), где есть ссылка на символический адрес памяти. На языке ассемблера пространство для переменных в сегменте данных памяти определяется и выделяется использованием директив ассемблера, таких как `.word` и `.space`. Строкам выделяется место в памяти с помощью директив ассемблера `.ascii` и `.asciiz`.

Проиллюстрируем использование псевдокода, чтобы написать программу на языке ассемблера для нахождения суммы целых чисел от единицы до *N*, где *N* – это значение, введенное с клавиатуры. Другими словами, необходимо решить следующую задачу: *Ввести число N и вычислить сумму 1 + 2 + 3 + 4 + 5 + + 6 + 7 + ... + N*. В примере 9.1 приведены псевдокод алгоритма для решения этой задачи (*А*) и соответствующая этому псевдокоду программа на языке ассемблера (*Б*).

Пример 9.1:
А. Псевдокод алгоритма вычисления суммы целых чисел

```
# Cross References:
# v0: N,
# t0: Sum

main:       cout << "Please input a value for N"
            cin >> v0
            If (v0 <= 0) then stop
            else
               t0 = 0;
               while (v0 > 0) do
                  {
                  t0 = t0 + v0;
                  v0 = v0 - 1;
                  }
            cout << t0;
            go to main
```

Продолжение примера 9.1:
Б. Программа на языке ассемблера, соответствующая псевдокоду алгоритма вычисления суммы целых чисел

```
            .data
prompt:     .asciiz "\n Please Input a value for N = "
result:     .asciiz " The sum of the integers from 1 to N is "
            .globl main
            .text
main:
            li $v0, 4               # system call code for Print String
            la $a0, prompt          # load address of prompt into $a0
            syscall                 # print the prompt message
            li $v0, 5               # system call code for Read Integer
            syscall                 # reads the value of N into $v0
            blez $v0, end           # branch to end if $v0 < = 0
            li $t0, 0               # clear register $t0 to zero
loop:
            add $t0, $t0, $v0       # sum of integers in register $t0
            addi $v0, $v0, -1       # summing integers in reverse order
            bnez $v0, loop          # branch to loop if $v0 is != zero
            li $v0, 4               # system call code for Print String
            la $a0, result          # load address of message into $a0
            syscall                 # print the string
            li $v0, 1               # system call code for Print Integer
            move $a0, $t0           # move value to be printed to $a0
            syscall                 # print sum of integers
            b main                  # branch to main
end:
            li $v0, 10              # terminate program run and
            syscall                 # return control to system
```

В псевдокоде и программе регистр процессора `$t0` используется для накопления суммы, а регистр процессора `$v0` – в качестве счетчика цикла, который начинается со значения *N* и ведет обратный отсчет до 0.

Ниже приведены примеры перевода структур псевдокода в соответствующую последовательность команд на языке ассемблера.

Пример 9.2. Арифметическое выражение

Псевдокод:

```
$s0 = srt($a0 * $a0 + $a1 * $a1)
```

Результат перевода псевдокода на язык ассемблера MIPS (в этом примере предполагается, что существует библиотечная функция, которую можно вызвать и которая вернет квадратный корень аргумента):

```
mult $a0, $a0     # square $a0
mflo $t0          # $t0 = Lower 32-bits of product
mult $a1, $a1     # square $a1
mflo $t1          # $t1 = Lower 32-bits of product
add $a0, $t0, $t1 # $a0 = $t0 + $t1
jal srt           # call of sqr function
move $s0, $v0     # result of sqr is returnet in $v0
```

Пример 9.3. Управляющая структура «IF THEN ELSE»

```
if (condition) then do {this block of code} else do {that block of code}
```

Псевдокод:

```
      if ($s1 < 0) then
            {$s0 = 0 - $s1;
             $t1 = $t1 + 1}
      else
            {$s0 = $s1;
             $t2 = $t2 + 1}
```

Результат перевода псевдокода на язык ассемблера MIPS:

```
      bgez $s1, else          # if ($s1 is > or = zero) branch to else
      sub $s0, $zero, $s1     # s0 gets the negative of $t1
      addi $t1, $t1, 1        # increment $t1 by 1
      b next                  # branch around the else code
else:
      ori $s0, $s1, 0         # $s0 gets a copy of $s1
      addi $t2, $t2, 1        # increment $t2 by 1
```

Пример 9.4. Управляющая структура «FOR LOOP»

```
for (condition) do {this block of code}
```

Псевдокод:

```
      $s0 = 0;
      for ($t0 = 10; $t0 > 0; $t0 = $t0 - 1) do
            {$s0 = $s0 + $t0}
```

Результат перевода псевдокода на язык ассемблера MIPS:

```
      li $s0, 0               # $s0 = 0
      li $t0, 10              # initialize loop counter to 10
loop:
      add $s0, $s0, $t0       # $s0 = $s0 + $t0
      addi $t0, $t0, -1       # decrement loop counter
      bgtz $t0, loop          # if ($t0 > 0) branch to loop
```

Пример 9.5. Управляющая структура «WHILE»

```
while (condition) do {this block of code}
```

Псевдокод:

```
      $s0 = 0;
      while ($a1 < $a2) do
            {$t1 = mem[$a1];
             $t2 = mem[$a2];
             $s0 = $s0 + $t1 + $t2;
             $a1 = $a1 + 1;
             $a2 = $a - 1}
```

Продолжение примера 9.5

Результат перевода псевдокода на язык ассемблера MIPS:

```
      li $s0, 0               # $s0 = 0
loop:
      bgeu $a1, $a2, done
      lb $t1, 0($a1)
      lb $t2, 0($a2)
      add $s0, $s0, $t1       # $s0 = $s0 + $t0
      add $s0, $s0, $t2       # $s0 = $s0 + $t0
      addi $a1, $a1, 1        # decrement loop counter
      addi $a2, $a2, -1       # decrement loop counter
      b loop                  # if ($t0 > 0) branch to loop
done:
```

Пример 9.6. Управляющая структура « SWITCH»

```
switch (expression)
{
      case value1:
            {this block of code if expression is equal to value1}
      case value2:
            {this block of code if expression is equal to value2}
      ...
      case valueN:
            {this block of code if expression is equal to valueN}
      default:
            {this block of code if expression is not equal to any value}
}
```

Псевдокод:
двоичное значение в регистре $s0 сдвигается влево на один, два или три разряда в зависимости от того, какое значение считывается в регистре $v0

```
top:  cin >> $v0;
      switch ($v0)
            {case(1): {$s0 = $s0 << 1; break;}
             case(2): {$s0 = $s0 << 2; break;}
             case(3): {$s0 = $s0 << 3; break;}
             default: goto top;}
```

Результат перевода псевдокода на язык ассемблера MIPS:

```
       .data
       .align 2
table: .word top, case1, case2, case3
       .text
top:
       li $v0, 5
       syscall                 # read an integer
       blez $v0, else          # default for less than one
       li $t3, 3
       bgt $v0, $t3, top       # default for greater than 3
       la $a1, table           # load address of table
```

Продолжение примера 9.6

```
        sll $t0, $v0, 2         # compute word offset
        add $t1, $a1, $t0       # form a pointer into table
        lw $t2, 0($t1)          # load an address from table
        jr $t2                  # jump to specific case "switch"
case1:  sll $s0, $s0, 1         # shift left logical one bit
        b next
case2:  sll $s0, $s0, 2         # shift left logical two bits
        b next
case3:  sll $s0, $s0, 3         # shift left logical three bits
next:
```

9.2. Порядок выполнения работы

1. Изучить краткое руководство по использованию нотации псевдокода высокого уровня при программировании на языке ассемблера MIPS.
2. Получить у преподавателя номер варианта задания.
3. Разработать псевдокод алгоритма для решения заданной задачи **(при разработке алгоритма НЕ использовать процедуры и стек)**.
4. В соответствии с разработанным псевдокодом алгоритма разработать программу на языке ассемблера MIPS **(при разработке программы НЕ использовать процедуры и стек)**.
5. Загрузить исходный текст разработанной программы в симулятор PCSpim. Найти в соответствующей панели окна симулятора PCSpim команды и данные программы.
6. Выполнить отладку разработанной программы, используя пошаговый режим и точки останова. Записать для каждой команды выполняемое действие и получаемый результат (содержимое памяти и регистров). Сделать скриншоты окна симулятора (или его фрагментов) PCSpim, поясняющие полученные результаты.
7. Оформить отчет.

9.3. Содержание отчета

1. Титульный лист.
2. Цель работы.
3. Краткие сведения по методике программирования на языке ассемблера MIPS с использованием нотации псевдокода высокого уровня.
4. Псевдокод алгоритма для решения задачи в соответствии с вариантом задания. Описание псевдокода.
5. Исходный текст программы на языке ассемблера MIPS для решения задачи в соответствии с разработанным псевдокодом алгоритма. Описание исходного текста программы.
6. Результат выполнения разработанной программы в соответствии с пп. 5, 6 подразд. 9.2.
7. Выводы по работе.

9.4. Варианты задания

1. Найти максимальный элемент и сумму элементов массива целых чисел. Элементы массива вводятся с клавиатуры. Ввод заканчивается либо после ввода нулевого значения элемента, либо после ввода 40 элементов.

2. Выполнить циклическую перестановку элементов массива целых чисел на три элемента вправо. Элементы массива вводятся с клавиатуры. Ввод заканчивается либо после ввода нулевого значения элемента, либо после ввода 30 элементов.

3. Найти сумму цифр целого числа. Целое число вводится с клавиатуры как число.

4. Отсортировать массив целых чисел по возрастанию. Элементы массива вводятся с клавиатуры. Ввод заканчивается либо после ввода нулевого значения элемента, либо после ввода 20 элементов.

5. Выполнить циклическую перестановку элементов массива целых чисел на два элемента влево. Элементы массива вводятся с клавиатуры. Ввод заканчивается либо после ввода нулевого значения элемента, либо после ввода 35 элементов.

6. Найти сумму цифр целого числа. Целое число вводится с клавиатуры как строка символов.

7. Отсортировать массив целых чисел по убыванию. Элементы массива вводятся с клавиатуры. Ввод заканчивается либо после ввода нулевого значения элемента, либо после ввода 15 элементов.

8. Найти минимальный элемент и сумму элементов массива целых чисел. Элементы массива вводятся с клавиатуры. Ввод заканчивается либо после ввода нулевого значения элемента, либо после ввода 25 элементов.

9. Вычислить факториал целого числа n ($n \leq 10$). Число вводится с клавиатуры.

10. Вычислить функцию Фибоначчи для целого числа n ($n \leq 21$). Число вводится с клавиатуры. Функция Фибоначчи для целых чисел n ($n \geq 0$) определяется следующим образом: $F(0) = F(1) = 1$, $F(n) = F(n-1) + F(n-2)$.

11. Вычислить произведение двух целых чисел A и B. Найти все числа, кратные A, от A до $A \cdot B$. Числа A и B вводятся с клавиатуры.

12. Определить, является ли строка текста палиндромом (палиндром – это слово или предложение, которое пишется одинаково как в прямом, так и в обратном направлении). Строка текста вводится с клавиатуры. Максимальная длина строки – 32 символа.

13. Преобразовать целое десятичное число в шестнадцатеричный вид. Целое число вводится с клавиатуры.

14. Найти сумму элементов заданного столбца двумерного (4 × 4) массива целых чисел. Номер столбца и элементы массива вводятся с клавиатуры. Элементы массива вводятся построчно.

15. Найти заданное число в массиве ненулевых целых чисел. Искомое число и элементы массива вводятся с клавиатуры. Ввод элементов массива заканчивается либо после ввода нулевого значения элемента, либо после ввода 28 элементов.

16. Посчитать количество заглавных букв, знаков препинания и пробелов в заданной строке текста. Строка текста вводится с клавиатуры. Максимальная длина строки – 64 символа.

ЛАБОРАТОРНАЯ РАБОТА № 10. ИСПОЛЬЗОВАНИЕ МЕХАНИЗМА ПРОЦЕДУР И СТЕКА В ПРОГРАММАХ НА АССЕМБЛЕРЕ УЧЕБНОГО ПРОЦЕССОРА С АРХИТЕКТУРОЙ MIPS

Цель работы – изучение механизма реализации процедур на языке ассемблера MIPS; изучение порядка использования стека в программах на языке ассемблера MIPS; получение практических навыков разработки программ с использованием процедур и стека на языке ассемблера MIPS.

10.1. Механизм реализации процедур и стека в программах на языке ассемблера MIPS

10.1.1. Механизм реализации процедур на языке ассемблера MIPS

Процедура представляет собой логически самостоятельную единицу программы, предназначенную для выполнения конкретной функции. Процедуры еще называют подпрограммами. Это важная программная конструкция, облегчающая модульное программирование. Их можно разделить на *leaf* и *nonleaf*. *Leaf*-процедура не вызывает другие процедуры, а *nonleaf*-процедура – вызывает. В архитектуре MIPS можно реализовать простые *leaf*-процедуры без использования стека. *Nonleaf*-процедурам всегда необходимо использовать стек, по крайней мере для хранения обратного адреса вызывающей процедуры.

Архитектура MIPS предоставляет две команды для поддержки процедур: `jal` и `jr`. Команда `jal` используется для вызова процедуры, а `jr` – для возврата из процедуры.

Команда `jal` (переход и ссылка)

```
jal proc_name
```

передает управление метке (команде по адресу) `proc_name` так же, как это делает инструкция перехода. Поскольку в данном случае нужен адрес для возврата из вызванной процедуры, она сохраняет адрес команды, следующей за командой вызова, в регистре `$ra`. Действия, выполняемые при вызове процедуры с помощью команды `jal`, можно описать следующим образом:

```
$ra = PC + 4

PC = PC[31:28] || offset<<2
```

Обратите внимание, что регистр `$ra` (т. е. `$31`) загружается значением `PC + 4`, поскольку процессор MIPS R2000 не поддерживает задержанные переходы.

Команда `jal` использует 26 бит для указания смещения. Целевой адрес перехода вычисляется следующим образом: 26-битное смещение, указанное в команде, сдвигается влево на два разряда для выравнивания на границу 32-разрядного слова. Это дает нам 28 бит, а остальные 4 бита берутся из программного счетчика `PC`.

Для пояснения механизма вызова процедуры, рассмотрим пример 10.1, содержащий фрагмент кода программы, полученный из симулятора SPIM.

Пример 10.1. Пример кода для объяснения вызовов процедур

```
Адрес          Машинный код    Мнемоническое описание     Исходный текст
               команды         машинной команды           программы

                               ;  main:
                                     . . .
[0x0040004c]   0x0c100022      jal   0x00400088 [sum]  ;  37: jal  sum
                                     . . .
                               ;  end of main procedure
;*************************************************************************
                               ;  sum:
[0x00400088]   0x34020000      ori   $2, $0, 0         ;  60: li   $v0, 0
[0x0040008c]   0x0c10002c      jal   0x004000b0 [avg]  ;  61: jal  avg
                                     . . .
                               ;  end of sum procedure
;*************************************************************************
                               ;  avg:
[0x004000b0]   0x00044021      addu  $8, $0, $4        ;  76: move $t0, $a0
                                     . . .
[0x004000c0]   0x0c100022      jal   0x00400088 [sum]  ;  80: jal  sum
                                     . . .
                               ;  end of avg procedure
```

Этот фрагмент кода состоит из трех процедур: `main`, `avg` и `sum`. В первом столбце указан адрес каждой команды (префикс `0x` означает, что адрес указывается в шестнадцатеричном виде). Сгенерированный машинный код указан во втором столбце. В третьем столбце приведен ассемблерный перевод исходной инструкции, приведенной в четвертом столбце. Например, `li $v0,0` переводится как `ori $2,$0,0`.

Команда вызова процедуры в основной процедуре `jal sum` кодируется как `0x0c100022`. Старшие 6 бит представляют код операции, а оставшиеся 26 разрядов дают смещение, как показано на рис. 10.1.

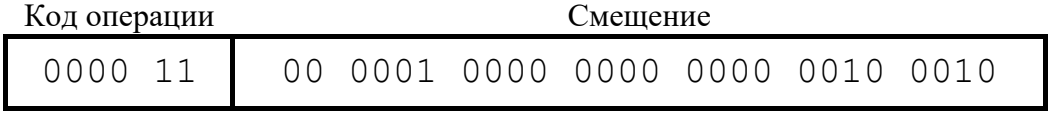

Рис. 10.1. Машинный код команды вызовова процедуры `jal sum`

Сдвинув это смещение влево на два разряда, мы получим 28-битное значение `0x0400088`. Добавляя 4 старших бита из `PC`, мы получаем целевой адрес `0x00400088`, который является адресом процедура суммирования. Такая же кодировка используется для вызова суммы в процедуре `avg`.

Команда `jal` в процедуре суммирования `jal avg` кодируется как `0x0c10002c`, которая состоит из `000011` в качестве кода операции и `0x010002c` в качестве значения смещения, как показано на рис. 10.2.

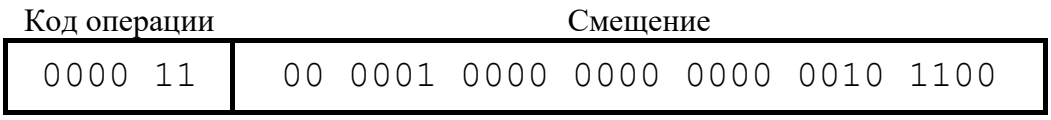

Рис. 10.2. Машинный код команды вызовова процедуры `jal avg`

Сдвинув смещение `10002c` влево на два разряда, мы получим 28-битное значение `04000b0`. Добавляя четыре старших разряда из `PC`, мы получаем целевой адрес процедуры `avg`, равный `0x004000b0`.

Для возврата из процедуры используется команда `jr $ra`, которая читает обратный адрес из регистра `$ra` и передает управление на этот адрес. Эта команда является частным случаем общей команды перехода `jr rs`, в которой регистр `rs` содержит целевой адрес перехода. Формат этой команды показан на рис. 10.3.

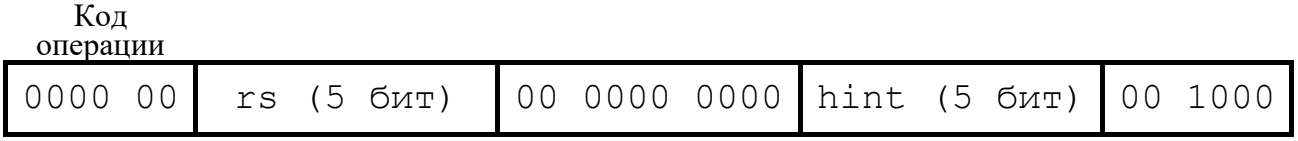

Рис. 10.3. Формат команды перехода `jr rs`

В рассматриваемой версии архитектуры MIPS только 0 определяется как допустимое значение для поля `hint`. Это значение симулятор SPIM использует при кодировании инструкции `jr`. В поле `rs` указывается регистр, который содержит целевой адрес перехода. Для возврата из процедуры используется `$ra`, т. е. регистр `r31`. Таким образом, подставив 31 в поле `rs`, мы получим кодировку `0x03e00008` для команды `jr $ra`.

Передача параметров на языке ассемблера отличается от передачи параметров в языках высокого уровня. Она более сложна. На языке ассемблера вызывающая процедура сначала помещает все параметры, необходимые вызываемой процедуре, в область хранения, доступную обеим процедурам (обычно в регистры или память). Только после этого процедура может быть вызвана.

Существует два основных метода в зависимости от типа области хранения, используемой для передачи параметров: метод на основе регистров или метод на основе стека. Как следует из названий, метод на основе регистров использует регистры общего назначения для передачи параметров, а другой метод использует стек. Сначала рассмотрим метод на основе регистров. Передачу параметров на основе стека рассмотрим отдельно вместе со стеком.

В соответствии с соглашениями по использованию регистров MIPS процессора (см. лабораторную работу № 7), регистры `$v0` и `$v1` используются для возврата результатов из процедуры. Регистры от `$a0` до `$a3` используются для передачи процедуре первых четырех параметров. Остальные параметры передаются через стек.

Как отмечалось выше, процедуры можно разделить на два основных типа: *leaf*-процедуры и *nonleaf*-процедуры. *Leaf*-процедуры не вызывают другие процедуры, в отличие от *nonleaf*-процедур. Простым *leaf*-процедурам не обязательно использовать стек, если их локальные переменные могут быть размещены в регистрах от `$t0` до `$t9`. Если это не так, *leaf*-процедура должна будет использовать стек. *Nonleaf*-процедуры должны использовать стек, по крайней мере, для хранения адреса возврата. Подробнее эти процедуры будут рассмотрены ниже.

Для иллюстрации вопроса использования процедур на языке ассемблера MIPS, рассмотрим пример 10.2.

Пример 10.2. Программа для вычисления суммы трех целых чисел

```
1:   # Find the sum of three numbers                              sum.asm
2:   #
3:   # Objective: Finds the sum of three integers.
4:   # To demonstrate register-based
5:   # parameter passing.
6:   # Input:  Requests three numbers from the user.
7:   # Output: Outputs the sum.
8:   #
9:   # $a0, $a1, $a2 - three numbers are passed via
10:  # these registers
11:  # $v0 - returns sum
12:  #
13:  ################### Data segment ###################
14:         .data
15:  prompt:
16:         .asciiz "Please enter three numbers: \n"
17:  sum_msg:
18:         .asciiz "The sum is: "
19:  newline:
20:         .asciiz "\n"
21:
22:  ################### Code segment ###################
23:         .text
24:         .globl main
25:  main:
26:         la $a0,prompt                # prompt user for input
27:         li $v0,4
28:         syscall
29:
30:         li $v0,5                     # read 1st number into $a0
31:         syscall
32:         move $a0,$v0
33:
34:         li $v0,5                     # read 2nd number into $a1
35:         syscall
36:         move $a1,$v0
37:
38:         li $v0,5                     # read 3rd number into $a2
39:         syscall
40:         move $a2,$v0
41:
42:         jal find_sum
43:         move $t0,$v0
44:
```

Продолжение примера 10.2

```
45:         la $a0,sum_msg                # write sum message
46:         li $v0,4
47:         syscall
48:
49:         move $a0,$t0                  # output sum
50:         li $v0,1
51:         syscall
52:
53:         la $a0,newline                # write newline
54:         li $v0,4Chapte
55:         syscall
56:
57:         li $v0,10 # exit
58:         syscall
59:
60:   #-----------------------------------------------
61:   # FIND_SUM receives three integers in $a0, $a1,
62:   # and $a2 and returns their sum in $v0
63:   #-----------------------------------------------
64:   find_sum:
65:         move $v0,$a0
66:         addu $v0,$v0,$a1
67:         addu $v0,$v0,$a2
68:         jr $ra
```

Основная программа запрашивает три целых числа и передает их процедуре `find_sum`. Процедура возвращает сумму. Регистры используются для передачи параметров, а также для возврата результата. Регистры `$a0`, `$a1` и `$a2` используются для передачи трех целых чисел. Процедура возвращает сумму в регистре `$v0`. Чтобы вызвать процедуру, используется команда `jal find_sum`, как показано в строке 42. После завершения суммирования в процедуре команда `jr $ra` возвращает управление основной программе (см. строку 68).

Метод на основе регистров имеет свои преимущества и недостатки.

Преимущества:

1. Метод на основе регистров удобнее и проще для передачи небольшого количества параметров. Например, MIPS обычно использует регистры `$a0`–`$a3` для передачи первых четырех параметров.

2. Этот метод быстрее, поскольку все параметры доступны в регистрах, а не в памяти.

Недостатки:

1. Метод на основе регистров не может выступать в качестве общего механизма передачи параметров. Например, с помощью регистров можно передать лишь несколько параметров, поскольку их количество ограничено. Если передается переменное количество параметров, необходимо использовать стек. Рекурсивные процедуры также требуют использования стека.

2. Регистры часто используются вызывающей процедурой для каких-то других целей. Таким образом, необходимо временно сохранять содержимое этих

регистров в стеке, чтобы освободить их для использования при передаче параметров перед вызовом процедуры, и восстанавливать после возврата из вызванной процедуры.

10.1.2. Реализация стека в архитектуре MIPS

Концептуально стек представляет собой структуру данных «последним пришел – первым обслужен» (Last-In-First-Out – LIFO). Со стеком связаны две операции: вставка (запись в стек) и удаление (чтение из стека). Если мы рассматриваем стек как линейный массив элементов, операции вставки и удаления стека ограничиваются одним концом массива. Таким образом, единственным элементом, к которому возможен прямой доступ, является элемент на вершине стека (Top-Of-Stack – TOS). В терминологии стека операции вставки и удаления называются операциями `push` и `pop` соответственно.

Существует еще одна похожая структура данных – очередь. Очередь можно рассматривать как линейный массив, в котором вставки выполняются на одном конце массива, а удаления – на другом. Таким образом, очередь представляет собой структуру данных «первым пришел – первым обслужен» (First-In-First-Out – FIFO).

В качестве примера предположим, что мы вставляем в стек числа от 1 до 4 в порядке возрастания. Состояние стека при выполнении этого действия показано на рис. 10.4. Стрелки указывают на вершину стека при выполнении каждой операции вставки.

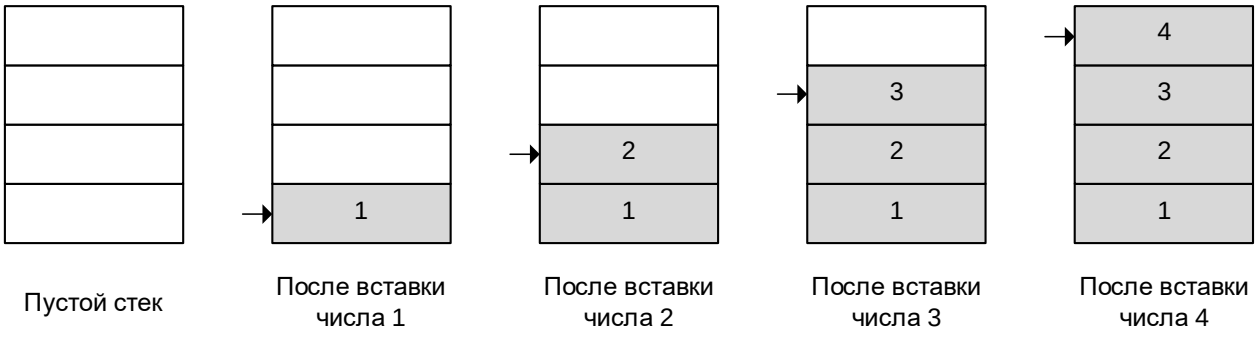

Рис. 10.4. Пример работы стека при выполнении операции вставки

Когда числа удаляются из стека, они выходят в порядке, обратном вставке, как показано на рис. 10.5, т. е. сначала удаляется 4, потом 3 и т. д. Говорят, что после удаления последнего числа стек находится в пустом состоянии.

В отличии от стека очередь поддерживает порядок данных. Предположим, что числа от 1 до 4 вставлены в очередь, как в примере со стеком. При удалении чисел из очереди первым числом, которое выйдет из очереди, будет то, которое вошло первым, т. е. 1. Таким образом, числа, удаленные из очереди, сохранят свой порядок, т. е. порядок вставки.

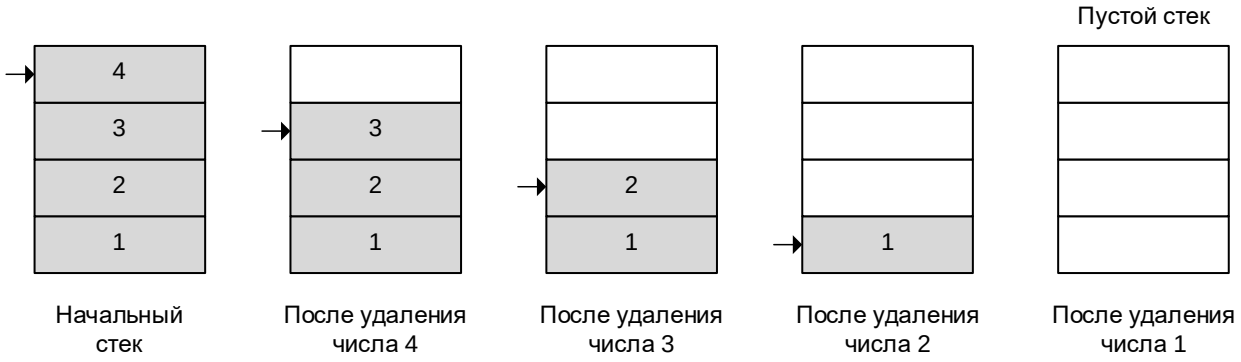

Рис. 10.5. Пример работы стека при выполнении операции удаления

Архитектура MIPS не предоставляет никакой специальной аппаратной поддержки для реализации стека. Стек должен поддерживаться программно, т. е. задача реализации стека возлагается на программиста. Для реализации стека используется пространство памяти, зарезервированное в сегменте стека. По соглашению сегмент стека начинается в конце (вверху) адресного пространства памяти (т. е. по адресу `0x7FFFFFFF`) и растет вниз таким образом, что при записи данных в стек адрес памяти уменьшается. С другой стороны, область динамических данных растет вверх. Это эффективный способ распределения неиспользуемой памяти между сегментами стека и данных.

Так как стек представляет собой структуру данных LIFO, то для его реализации нужно просто поддерживать указатель на вершину стека. В некоторых архитектурах предусмотрен специальный регистр, указывающий на вершину стека. Например, в архитектуре Intel для этой цели служит регистр `ESP`. Однако в архитектуре MIPS специального регистра нет. Для этой цели мы можем использовать любой регистр общего назначения. По соглашению регистр `r29` используется для указания вершины стека. Этот регистр называется регистром указателя стека (`$sp`). Обратите внимание, что при этом TOS соответствует последнему элементу, помещенному в стек.

Архитектура MIPS явно не поддерживает операции стека. Напротив, архитектура Intel предоставляет такие инструкции, как `push` и `pop`, для облегчения операций со стеком. В процессоре MIPS программисту необходимо самому манипулировать регистром указателя стека, чтобы реализовать стек. В соответствии с этим можно разместить сегмент стека в любом месте адресного пространства памяти. Важно только, чтобы эта область адресного пространства не использовалась для других целей. Для этого прежде, чем использовать стек, в регистр указателя стека необходимо занести начальный адрес выбранной для стека области адресного пространства памяти.

Операция занесения в стек. Как указывалось выше, вершина стека соответствует последней заполненной позиции в стеке. Поэтому при выполнении операции вставки в стек нужно сначала уменьшить значение в регистре `$sp`, чтобы освободить место для элемента данных, помещаемого в стек, т. е. переместить указатель стека так, чтобы он указывал на первую свободную позицию в

стеке (памяти). Например, если мы хотим отправить содержимое регистра $a0 в стек, необходимо зарезервировать 4 байта пространства стека (четыре ячейки в памяти) и использовать команду sw для записи слова, как показано ниже:

```
sub  $sp,$sp,4      # reserve 4 bytes of stack
sw   $a0,0($sp)     # save the register
```

Операция чтения из стека. Для выполнения операции чтения необходимо извлечь содержимое из последней заполненной позиции в стеке, т. е. по адресу, который находится в указателе стека, и увеличить значение в регистре $sp, чтобы переместить указатель стека на последнюю заполненную позицию в стеке (памяти). Это можно реализовать с помощью команд загрузки и сложения. Например, чтобы восстановить значение в регистре $a0 из стека, мы должны использовать команду lw для копирования значения из памяти и команду addu для увеличения содержимого регистра $sp на 4, как показано ниже:

```
lw   $a0,0($sp)     # restore the two registers
addu $sp,$sp,4      # clear 4 bytes of stack
```

Проиллюстрируем организацию стека на примере 32-разрядных данных, хотя стек можно использовать и для других типов данных. В примере 10.3 приведены фрагменты программы на языке ассемблера для выполнения операций занесения в стек (*А*) и чтения из стека (*Б*), а на рис. 10.6 показаны состояния стека при выполнении этих операций.

Пример 10.3:
А. Занесение в стек

```
sub   $sp,$sp,8          # reserve 8 bytes of stack
sw    $a1,0($sp)         # save registers
sw    $a0,4($sp)
```

Б. Чтение из стека

```
lw    $t0,0($sp)
addu  $sp,$sp,4
```

Начальное состояние стека показано на рис. 10.6, *а*. Когда в этот стек помещается содержимое регистров $a0 (5678) и $a1 (9012), указатель стека $sp уменьшается на 8, чтобы зарезервировать 8 байтов стекового пространства. Затем значения 5678 и 9012 сохраняются в стеке с помощью последовательности команд записи в память с использованием регистра указателя стека. Итоговое состояние стека после занесения данных показано на рис. 10.6, *б*. Предположим, необходимо прочитать слово данных из стека и поместить в регистр $t0. Это делается с помощью команды чтения из памяти по адресу, находящемуся в регистре указателя стека, после чего содержимое регистра $sp должно быть увеличено на 4. Состояние стека после выполнения операции чтения слова данных из стека показано на рис. 10.6, *в*.

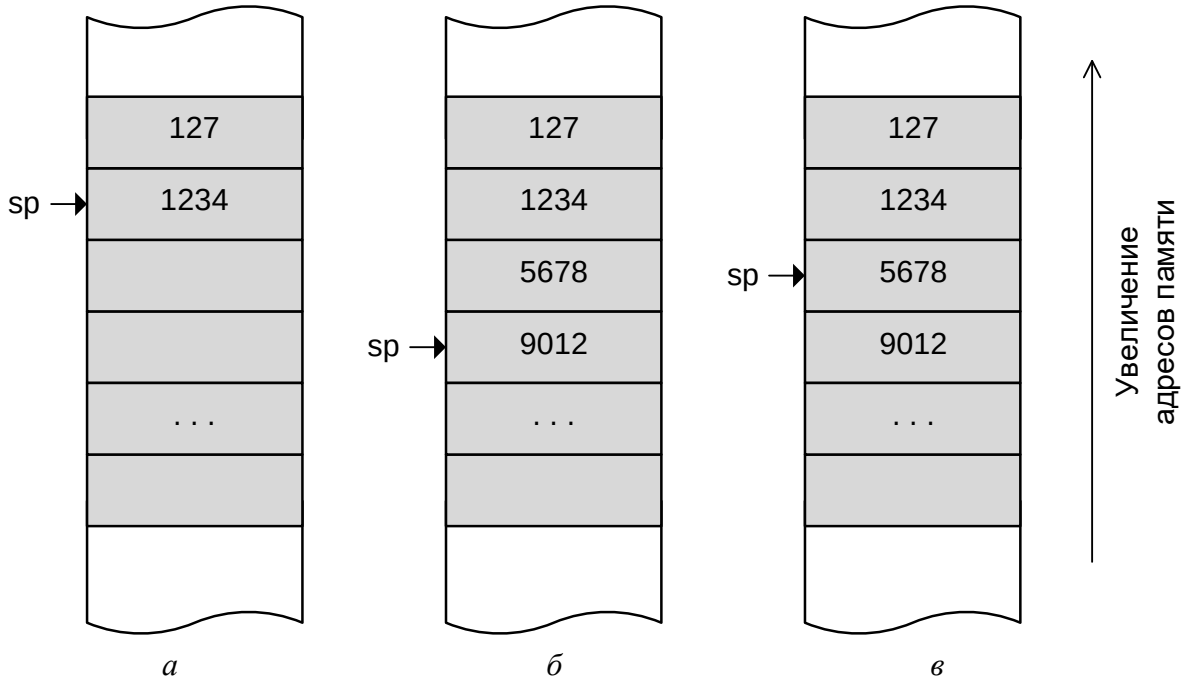

Рис. 10.6. Пример работы стека при выполнении операций со стеком в архитектуре MIPS: *а* – начальное состояние стека (`sp` указывает на вершину стека); *б* – состояние стека после записи слов данных 5678 и 9012; *в* – состояние стека после чтения слова данных 9012

Стек используется для трех основных целей: в качестве буфера для временного хранения данных, для передачи управления программой и для передачи параметров во время вызова процедуры.

1. *Временное хранение данных*. Стек можно использовать как буфер для временного хранения данных. Например, стек часто используется как буфер для сохранения и восстановления регистров. Такая необходимость возникает, когда нужно освободить набор регистров для того, чтобы их можно было использовать в текущей процедуре. Предположим, что основная программа `main` вызывает процедуру `test`, которая использует временные регистры `$s0`, `$s1` и `$s2`. Прежде чем использовать эти регистры, процедура `test` должна сохранить содержимое этих регистров, чтобы иметь возможность восстановить их перед возвратом управления в основную программу `main`. Для этой цели можно использовать стек, как показано в примере 10.4.

Пример 10.4. Использование стека для временного хранения данных:
А. Сохранение содержимого регистров `$s0`, `$s1` и `$s2` в стеке

```
test:
      sub   $sp,$sp,12        # reserve 12 bytes of stack
      sw    $s0,0($sp)        # save registers
      sw    $s1,4($sp)
      sw    $s2,8($sp)
```

Б. Восстановление содержимого регистров `$s0`, `$s1` и `$s2` из стека

```
      lw    $s0,0($sp)
      lw    $s1,4($sp)
      lw    $s2,8($sp)
      add   $sp,$sp,12
```

2. *Передача управления*. Стек также используется для передачи управления. В частности, при вызове *nonleaf*-процедуры адрес возврата сохраняется в стеке, чтобы можно было передать управление обратно вызывающей программе.

3. *Передача параметров*. Еще одним важным применением стека является использование его в качестве средства для передачи параметров вызываемой процедуре.

Чтобы продемонстрировать использование стека для передачи параметров процедуре, воспользуемся задачей вычисления суммы трех целых чисел, рассмотренной ранее в примере 10.2. Но на этот раз три целых числа будут передаваться через стек. Основная программа запрашивает для ввода три целых числа и передает их процедуре `find_sum` через стек. Как и в примере 10.2, в примере 10.5 процедура суммирования возвращает сумму в регистре `$v0`.

Пример 10.5. Программа для вычисления суммы трех целых чисел

```
1:   # Find the sum of three numbers              sum_stack.asm
2:   #
3:   # Objective: Finds the sum of three integers.
4:   # To demonstrate stack-based
5:   # parameter passing.
6:   # Input:  Requests three numbers from the user.
7:   # Output: Outputs the sum.
8:   #
9:   ################### Data segment ###################
10:         .data
11:  prompt:
12:         .asciiz "Please enter three numbers: \n"
13:  sum_msg:
14:         .asciiz "The sum is: "
15:  newline:
16:         .asciiz "\n"
17:
18:  ################### Code segment ###################
19:         .text
20:         .globl main
21:  main:
22:         la    $a0,prompt        # prompt user for input
23:         li    $v0,4
24:         syscall
25:
26:         li    $t0,3             # loop count = 3
27:  read_more:
28:         li    $v0,5             # read 1st number into $a1
29:         syscall                 # number is in $v0
30:         sub   $sp,$sp,4         # reserve 4 bytes on stack
31:         sw    $v0,($sp)         # store the number on stack
32:         sub   $t0,$t0,1         # decrement loop count
33:         bnez  $t0,read_more     # if not zero, read more
34:
35:         jal   find_sum
36:         move  $t0,$v0
37:
38:         la    $a0,sum_msg       # write sum message
39:         li    $v0,4
40:         syscall
41:
```

Продолжение примера 10.5

```
42:         move  $a0,$t0           # output sum
43:         li    $v0,1
44:         syscall
45:
46:         la    $a0,newline       # write newline
47:         li    $v0,4
48:         syscall
49:
50:         li    $v0,10            # exit
51:         syscall
52:
53:   #-----------------------------------------------
54:   # FIND_SUM procedure receives three integers
55:   # via the stack and returns their sum in $v0.
56:   #-----------------------------------------------
57:   find_sum:
58:         li    $v0,0             # $v0 = 0 (sum in $v0)
59:         li    $t0,3             # loop iteration count
60:   sum_loop:
61:         lw    $t1,($sp)         # pop into $t1
62:         add   $sp,$sp,4
63:         add   $v0,$v0,$t1
64:         sub   $t0,$t0,1
65:         bnez  $t0,sum_loop
66:         jr    $ra
```

В данном примере для сложения чисел используется цикл. Этот цикл суммирования состоит из строк 60–65. Поскольку сумму необходимо вернуть в регистре `$v0`, этот регистр используется для хранения суммы. В строке 58 он инициализируется нулем. Как и в основной программе, регистр `$t0` используется для счетчика циклов. Две команды в строках 61 и 62 реализуют операцию извлечения из стека. Значение в регистре `$t1` добавляется к сумме в регистре `$v0`. В конце цикла окончательная сумма находится в регистре `$v0`, как и в примере 10.2. Обратите внимание, что в конце цикла суммирования стек очищается от трех значений, переданных в процедуру.

Одно из основных отличий этой программы от программы в примере 10.2 заключается в том, что для ввода чисел и их сложения используется цикл. Когда для указанной цели используются регистры, это сделать невозможно. Чтобы понять гибкость передачи параметров на основе стека, рассмотрим добавление 20 чисел вместо 3. В случае использования стека все, что нам нужно сделать, это изменить инициализацию счетчика циклов с 3 на 20 (в строках 26 и 59).

При использовании механизма процедур важно сохранять содержимое регистров при вызове процедуры. Необходимость этого иллюстрируется примером 10.6.

Пример 10.6. Фрагмент программы с многократным вызовом процедуры

```
            li $s0,50
repeat:     call compute
            . . .
            subu $s0,$s0,1
            bnez $s0,repeat
```

В этом примере процедура `compute` вызывается 50 раз. Регистр `$s0` хранит количество оставшихся итераций. Теперь предположим, что процедура `compute` использует регистр `$s0` во время своего выполнения. Тогда, когда процедура `compute` вернет управление вызывающей программе, содержимое регистра `$s0` будет измененным, и логика программы будет нарушена. Чтобы содержимое регистра `$s0` имело правильное значение, его следует сохранить в вызываемой процедуре `compute`. Для этой цели удобно использовать стек.

Важным моментом при использовании механизма процедур является вопрос: какие регистры следует сохранять? Сохранять нужно те регистры, которые используются вызывающей процедурой, но изменяются вызываемой процедурой. Это приводит к следующему вопросу: какая процедура, вызывающая или вызываемая, должна сохранять регистры?

Если вызывающая процедура должна сохранить необходимые регистры, ей необходимо знать регистры, используемые вызываемой процедурой. Это вызывает две серьезные трудности:

1) затрудняется сопровождение программы. Если вызываемая процедура позже будет изменена и будет использован другой набор регистров, то каждую процедуру, вызывающую эту процедуру, придется также изменить;

2) программы становятся, как правило, длиннее. Поскольку если процедура вызывается несколько раз, приходится включать инструкции по сохранению и восстановлению регистров при каждом вызове процедуры.

Поэтому предпочтительно возлагать на вызываемую процедуру ответственность за сохранение используемых ею регистров и их восстановление перед возвратом к вызывающей процедуре. Это также соответствует принципам модульного проектирования программ.

Эта стратегия правильная с логической точки зрения, но не самая эффективная. Чтобы понять, почему так, предположим, что вызываемая процедура использует 10 регистров. Однако ни один из этих регистров не используется вызывающей процедурой. В этом случае мы зря тратим ресурсы и время на сохранение и восстановление этих 10 регистров. Чтобы решить эту проблему, в архитектуре MIPS регистры делятся на две группы:

1) *caller-save*-регистры. Эти регистры сохраняются вызывающей процедурой. В архитектуре MIPS регистры от `$t0` до `$t9` используются как *caller-save*-регистры. Эти регистры могут быть перезаписаны вызываемой процедурой без каких-либо ограничений. Если вызывающая сторона сохраняет в них что-то полезное, она должна позаботиться о сохранении их содержимого при вызовах процедур;

2) *callee-save*-регистры. Эти регистры сохраняются вызываемой процедурой. В архитектуре MIPS регистры от `$s0` до `$s8` используются как *callee-save*-регистры. Эти регистры, если они используются вызываемой проце-

дурой, должны быть сохранены вызываемой процедурой. Часто это делается путем помещения названных регистров в стек и их восстановления перед возвратом из процедуры.

10.2. Порядок выполнения работы

1. Изучить особенности использования механизма процедур и стека при разработке программ на языке ассемблера MIPS.
2. Получить у преподавателя номер варианта задания.
3. Разработать псевдокод алгоритма для решения заданной задачи **с использованием процедур и стека**.
4. В соответствии с разработанным псевдокодом алгоритма разработать программу на языке ассемблера MIPS **с использованием процедур и стека**.
5. Загрузить исходный текст разработанной программы в симулятор PCSpim. Найти в соответствующих панелях окна симулятора PCSpim команды основной программы и используемых в ней процедур, а также данные, используемые ими, включая передаваемые параметры и области для сохранения/восстановления содержимого регистров.
6. Выполнить отладку разработанной программы, используя пошаговый режим и точки останова. Записать для каждой команды выполняемое действие и получаемый результат (содержимое памяти и регистров). Сделать скриншоты окна симулятора (или его фрагментов) PCSpim, поясняющие полученные результаты.
7. Оформить отчет.

10.3. Содержание отчета

1. Титульный лист.
2. Цель работы.
3. Краткие сведения по использованию механизма процедур и стека при разработке программ на языке ассемблера MIPS.
4. Псевдокод алгоритма для решения задачи в соответствии с вариантом задания. Описание псевдокода.
5. Исходный текст программы на языке ассемблера MIPS для решения задачи в соответствии с разработанным псевдокодом алгоритма. Описание исходного текста программы.
6. Результат выполнения разработанной программы в соответствии с пп. 5, 6 подразд. 10.2.
7. Выводы по работе.

10.4. Варианты задания

1. Найти максимальный элемент и сумму элементов массива целых чисел. Размер массива и элементы массива вводятся с клавиатуры.

2. Выполнить циклическую перестановку элементов массива целых чисел на n элементов вправо. Размер массива, элементы массива и значение n вводятся с клавиатуры.

3. Найти сумму цифр целого числа. Целое число вводится с клавиатуры как число. Диапазон вводимых чисел (минимальное и максимальное значения) вводится с клавиатуры.

4. Отсортировать массив целых чисел по возрастанию. Размер массива и элементы массива вводятся с клавиатуры.

5. Выполнить циклическую перестановку элементов массива целых чисел на n элементов влево. Размер массива, элементы массива и значение n вводятся с клавиатуры.

6. Найти сумму цифр целого числа. Целое число вводится с клавиатуры как строка символов. Количество цифр целого числа вводится с клавиатуры.

7. Отсортировать массив целых чисел по убыванию. Размер массива и элементы массива вводятся с клавиатуры.

8. Найти минимальный элемент и сумму элементов массива целых чисел. Размер массива и элементы массива вводятся с клавиатуры.

9. Вычислить факториал целого числа n $(n \leq 10)$. Число вводится с клавиатуры.

10. Вычислить функцию Фибоначчи для целого числа n $(n \leq 21)$. Число вводится с клавиатуры. Функция Фибоначчи для целых чисел n $(n \geq 0)$ определяется следующим образом: $F(0) = F(1) = 1$, $F(n) = F(n-1) + F(n-2)$.

11. Сформировать последовательность чисел Фибоначчи для целого числа n $(n \leq 21)$. Число вводится с клавиатуры. Функция Фибоначчи для целых чисел n $(n \geq 0)$ определяется следующим образом: $F(0) = F(1) = 1$, $F(n) = F(n-1) + F(n-2)$.

12. Определить, является ли строка текста палиндромом (палиндром – это слово или предложение, которое пишется одинаково как в прямом, так и в обратном направлении). Размер строки и сама строка текста вводятся с клавиатуры.

13. Преобразовать целое десятичное число в шестнадцатеричный вид. Целое число вводится с клавиатуры как строка символов. Количество цифр целого числа вводится с клавиатуры.

14. Найти сумму элементов заданного столбца двумерного $n \times n$ массива целых чисел. Номер столбца, размерность массива и элементы массива вводятся с клавиатуры. Элементы массива вводятся построчно.

15. Найти заданное число в массиве ненулевых целых чисел. Искомое число, размер массива и элементы массива вводятся с клавиатуры.

16. Подсчитать количество заглавных букв, знаков препинания и пробелов в заданной строке текста. Размер строки и сама строка текста вводятся с клавиатуры.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Угрюмов, Е. П. Цифровая схемотехника : учеб. пособие для студентов вузов / Е. П. Угрюмов. – 3-е изд. – СПб. : БХВ-Петербург, 2010. – 816 с.
2. Качинский, М. В. Проектирование цифровых устройств на интегральных микросхемах. Курсовое проектирование : пособие / М. В. Качинский, В. Ю. Герасимович, А. В. Станкевич. – Минск : БГУИР, 2018. – 67 с.
3. Бибило, П. Н. Основы языка VHDL : учеб. пособие / П. Н. Бибило. – 3-е изд., доп. – М. : ЛКИ, 2007. – 328 с.
4. Сергиенко, А. М. VHDL для проектирования вычислительных устройств / А. М. Сергиенко. – Киев : ЧП «Корнейчук», 2003. – 208 с.
5. Суворова, Е. А. Проектирование цифровых систем на VHDL / Е. А. Суворова, Ю. Е. Шейнин. – СПб. : БХВ-Петербург, 2003. – 576 с.
6. Зотов, В. Ю. Проектирование цифровых устройств на основе ПЛИС фирмы XILINX в САПР WebPACK ISE / В. Ю. Зотов. – М. : Горячая линия – Телеком, 2003. – 624 с.
7. ISE In-Depth Tutorial. UG695 (v14.1) April 24, 2012 [Электронный ресурс]. – Режим доступа: https://docs.amd.com/v/u/en-US/ise_tutorial_ug695.
8. Соловьев, В. В. Проектирование функциональных блоков встраиваемых систем на FPGA / В. В. Соловьев. – М. : Горячая линия – Телеком, 2021. – 348 с.
9. Орлов, С. А. Организация ЭВМ и систем : учеб. / С. А. Орлов, Б. Я. Цилькер. – 3-е изд. – СПб. : Питер, 2014. – 688 с.
10. Качинский, М. В. Арифметические основы электронных вычислительных средств : учеб.-метод. пособие / М. В. Качинский, В. Б. Клюс, А. Б. Давыдов. – Минск : БГУИР, 2014. – 64 с.
11. Токхейм, Р. Л. Основы цифровой электроники / Р. Л. Токхейм ; пер. с англ. В. А. Курочкина, В. М. Матвеева ; под ред. Е. К. Масловского. – М. : Мир, 1988. – 391 с.
12. Баранов, С. И. Автоматы и программируемые матрицы / С. И. Баранов, В. Н. Синев. – Минск : Выш. шк., 1980. – 135 с.
13. Баранов, С. И. Синтез микропрограммных автоматов / С. И. Баранов. – 2-е изд., перераб. и доп. – Л. : Энергия, 1979. – 232 с.
14. Chu, Pong P. FPGA Prototyping by VHDL Examples: Xilinx Spartan-3 Version / Pong P. Chu. – Wiley-Interscience, 2008. – 468 с.
15. Spartan-3 FPGA Family Data Sheet. DS099 (v3.1) June 27, 2013 [Электронный ресурс]. – Режим доступа: https://docs.amd.com/v/u/en-US/ds099.
16. Spartan-3 Starter Kit Board User Guide. UG130 (v1.2) June 20, 2008 [Электронный ресурс]. – Режим доступа: https://docs.amd.com/v/u/en-US/ug130.
17. Dandamudi, S. P. Guide to RISC Processors : for Programmers and Engineers / S. P. Dandamudi. – Springer, 2010. – 404 с.

18. Dandamudi, S. P. Introduction to Assembly Language Programming: For Pentium and RISC Processors / S. P. Dandamudi. – 2nd edition. – Springer, 2004. – 716 с.

19. Sweetman, D. See MIPS Run / D. Sweetman. – 2nd edition. – Morgan Kaufmann, 2006. – 512 с.

20. Patterson, D. A. Computer Organization and Design / D. A. Patterson, J. L. Hennessy. – 5th edition. – Morgan Kaufmann, 2014. – 793 с.

21. Britton, R. MIPS Assembly Language Programming. Computer Science Department California State University, Chico Chico, California [Электронный ресурс]. – 2002. – Режим доступа: https://www.cs.csub.edu/~eddie/cmps2240/doc/britton-mips-text.pdf.

22. MIPS Architecture For Programmers. Volume I-A: Introduction to the MIPS32 Architecture. Document Number: MD00082. Revision 6.01. August 20, 2014 [Электронный ресурс]. – 2014. – Режим доступа: https://hades.mech.northwestern.edu/images/a/af/MIPS32_Architecture_Volume_I-A_Introduction.pdf.

23. MIPS Architecture For Programmers. Volume II-A: The MIPS32 Instruction Set Manual. Document Number: MD00086. Revision 6.04. November 13, 2015 [Электронный ресурс]. – 2015. – Режим доступа: https://hades.mech.northwestern.edu/images/1/16/MIPS32_Architecture_Volume_II-A_Instruction_Set.pdf.

Св. план 2025, поз. 50

Учебное издание

Качинский Михаил Вячеславович

СТРУКТУРНАЯ И ФУНКЦИОНАЛЬНАЯ ОРГАНИЗАЦИЯ ВЫЧИСЛИТЕЛЬНЫХ МАШИН. ЛАБОРАТОРНЫЙ ПРАКТИКУМ

ПОСОБИЕ

Редактор *А. Ю. Шурко*
Компьютерная правка, оригинал-макет *А. А. Лущикова*

Подписано в печать 30.07.2026. Формат 60×84 1/16. Бумага офсетная. Гарнитура «Таймс».
Отпечатано на ризографе. Усл. печ. л. 6,63. Уч.-изд. л. 7,0. Тираж 60 экз. Заказ 22.

Издатель и полиграфическое исполнение: учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники».
Свидетельство о государственной регистрации издателя, изготовителя,
распространителя печатных изданий №1/238 от 24.03.2014,
№2/113 от 07.04.2014, №3/615 от 07.04.2014.
Ул. П. Бровки, 6, 220013, г. Минск