

В.Г. Козырев

ПРОГРАММИРОВАНИЕ МИКРОКОНТРОЛЛЕРОВ

**Учебное пособие
для вузов**

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное
учреждение высшего образования
«СЕВАСТОПОЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

В.Г. Козырев

ПРОГРАММИРОВАНИЕ МИКРОКОНТРОЛЛЕРОВ

Учебное пособие

Севастополь
СевГУ
2023

УДК 004.382.7:004.42(075.8)

ББК 32.973.26-018я73

К593

Р е ц е н з е н т:

В. С. Чернега - канд. техн. наук, доцент кафедры информационных систем
ФГАОУ ВО «Севастопольский государственный университет»

Козырев В. Г.

К593 Программирование микроконтроллеров: учебное пособие для вузов /
В. Г. Козырев; Севастопольский государственный университет. – Севасто-
поль: СевГУ, 2023. – 244 с. – Текст : электронный.

Книга представляет собой учебное пособие по архитектуре и программированию микроконтроллеров. Предназначено для студентов университетов, обучающихся по направлению подготовки 27.03.04 – Управление в технических системах.

В пособии изложены теоретические основы архитектурного построения и машинно-ориентированного программирования микроконтроллеров: принципы хранения и обработки информации, ее структурирование и размещение по системе программно-доступных устройств, способы адресации устройств, система прерываний, основы функционирования интерфейса, система команд и директивы ассемблера. Общие положения демонстрируются на примере одного из широко распространенных в настоящее время семейства микроконтроллеров AVR.

Рассмотренная проблематика актуальна для специалистов по системной инженерии и прикладному программированию, занимающихся разработкой и эксплуатацией компьютеризированных систем различного технического назначения.

УДК 004.382.7:004.42(075.8)

ББК 32.973.26-018я73

Рекомендовано к изданию на заседании Ученого совета Института ИТУТС, протокол № 10 от 31 августа 2022 г.

© Козырев В. Г., 2023
© ФГАОУ ВО «Севастопольский
государственный университет», 2023

Оглавление

Предисловие	5
Введение	7
Глава 1. Концепции построения компьютеров	9
1.1. Компьютер. Аппаратура и программы.....	9
1.2. Информация в компьютере. Программы и данные.....	13
1.3. Гарвардская и принстонская модели вычислительных систем.....	18
1.4. CISC и RISC процессоры.....	19
Глава 2. Функциональные органы компьютера	23
2.1. Процессор.....	23
2.2. Память.....	40
2.3. Контроллеры ввода-вывода.....	47
2.4. Системная магистраль.....	53
2.5. Периферийные устройства. Интерфейс.....	54
Глава 3. Микроконтроллеры	60
3.1. Знакомство с микроконтроллером.....	60
3.2. Классификация МК.....	64
3.3. Применение микропроцессоров и микроконтроллеров.....	66
3.4. Семейства МК.....	68
Глава 4. Микроконтроллеры AVR	71
4.1. Устройство МК AVR.....	71
4.2. Выводы МК.....	75
4.3. Сброс микроконтроллера	77
4.4. Виды МК AVR и сферы применения.....	79
Глава 5. Память AVR	81
5.1. Состав памяти микроконтроллера.....	81
5.2. Программная модель.....	83
5.3. Регистровая память.....	89
5.4. Память программ и память данных.....	94
5.5. Стек. Подпрограммы.....	98
Глава 6. Система команд AVR	106
6.1. Команды процессора.....	106
6.2. Команды обработки данных.....	109
6.3. Команды передачи управления.....	119
6.4. Команды изменения состояния системы.....	127
6.5. Расшифровка мнемоник команд.....	128
Глава 7. Ассемблер	130
7.1. Лексемы.....	130
7.2. Выражения.....	132
7.3. Структура программы. Предложения.....	135
7.4. Команды.....	140
7.5. Директивы	141
Глава 8. Прерывания	151

8.1. Понятие прерываний.....	151
8.2. Обработка прерываний.....	156
8.3. Прерывания микроконтроллеров AVR.....	161
8.4. Обработка прерываний микроконтроллерами AVR.....	164
8.5. Регистры системы внешних прерываний.....	168
Глава 9. Периферийные компоненты и пространство ввода-вывода.....	172
9.1. Состав периферийных компонентов AVR.....	172
9.2. Ввод-вывод.....	173
9.3. Пространство ввода-вывода AVR.....	174
Глава 10. Порты ввода-вывода.....	176
10.1. Характеристика портов ввода-вывода.....	176
10.2. Регистры портов.....	177
10.3. Примеры.....	179
Глава 11. Таймер T0.....	184
11.1. Таймеры.....	184
11.2. Таймер-счетчик T0 ATmega16 (8 бит).....	185
11.3. Регистры таймера-счетчика T0.....	187
11.4. Регистры, разделяемые таймерами-счетчиками.....	191
Глава 12. Таймер T1.....	194
12.1. Таймер-счетчик T1 ATmega16 (16 бит).....	194
12.2. Регистры таймера-счетчика T1.....	196
12.3. Использование разделяемых регистров.....	203
Глава 13. Аналого-цифровой преобразователь.....	205
13.1. Общее описание АЦП.....	205
13.2. АЦП микроконтроллера AVR.....	206
13.3. Характеристика преобразования.....	207
13.4. Регистры.....	209
Список литературы.....	214
Приложение А. Арифметические операции над двоичными числами.....	216
Приложение Б. Обмен информацией в компьютере.....	225
Б.1. Структура аппаратуры и потоки информации компьютера..	225
Б.2. Базовые машинные операции.....	225
Б.3. Операции внешних пересылок.....	228
Б.4. Машинные циклы.....	234
Приложение В. Порядок работы компьютера.....	238
В.1. Программный процесс.....	238
В.2. Размещение программы в основной памяти.....	238
В.3. Продвижение процессора по программе.....	239
В.4. Пуск компьютера и его функционирование после пуска. Командные циклы.....	240
В.5. Разложение команд по машинным циклам.....	242

Предисловие

Настоящее учебное пособие предназначено для студентов направления подготовки «Управление в технических системах». Оно написано по материалам лекций, которые читались автором на протяжении многих лет в Севастопольском государственном университете по дисциплинам «Архитектура компьютерных систем», «Микропроцессорные системы», «Проектирование и программирование МП систем», «Программирование микроконтроллеров» и другим.

Пособие непосредственно рекомендуется для изучения дисциплины «Программирование микроконтроллеров», входящей в программу подготовки бакалавров по указанной специальности. Имеющаяся учебная литература по этой дисциплине и по микроконтроллерам (МК) в целом не вполне отвечает потребностям учебного процесса. Одной из причин является особенность учебного плана, согласно которому данная дисциплина читается раньше предметов общего компьютерного цикла, в то время как литература по МК опирается на этот цикл. Возникающий диссонанс затрудняет ее изучение.

Имеются и другие осложняющие обстоятельства. Они связаны с преимущественно прикладным характером существующей литературы, в которой основной акцент делается на лабораторный практикум и разработку устройств на микроконтроллерах [1-4, 6-10, 14, 17, 20, 21, 24]. В то же время архитектура самих МК и компьютеров [5, 11, 12, 15, 16, 18, 19, 22, 23] практически не обсуждается, в расчете, по-видимому, на то, что студенты справятся с этой задачей самостоятельно. В итоге ряд общетеоретических положений об устройстве и работе МК и компьютера остаются за рамками указанных книг.

Описание системы команд МК также делается чересчур кратким, главным образом путем перечисления мнемокодов команд и их псевдокодов. Однако, вводя понятие машинной команды (инструкции), следует, как представляется, тщательно прорабатывать ряд вопросов, актуальных именно для студенческой, а не профессиональной аудитории: назначение, классификацию и структуру команд, режимы адресации операндов и переходов, действие команд с пояснением работы общих и специальных регистров процессора, в частности, программного счетчика, рассмотрение примеров команд, их форматов и особенностей выполнения. Отдельного обсуждения требуют события арифметического переполнения в знаковой и натуральной арифметике, их критерии, особенности работы флажков, использование дополнительных кодов чисел со знаком для реализации знаковой арифметики. Особое место должно занимать изучение команд вызова и возврата из подпрограмм и прерываний, взаимодействующих со стеком. Эта тема, к сожалению, а также понятие и организация стека обычно вообще не затрагивается.

Повышенное внимание следует уделять изложению системы прерываний МК, имеющей ключевое значение для работы МК. В этой части курса необходимо давать полноценное определение режима прерываний, классификацию

прерываний и векторов прерываний, понятие которых подчас смешивается с адресами векторов, и исчерпывающе характеризовать систему прерываний существующих моделей МК.

Упомянутые обстоятельства склонили автора к написанию собственной книги. По исходному замыслу она должна была отталкиваться от ключевых идей, воплощенных в структуре любых вычислительных систем. Одновременно книга обязана была содержать образцы их реализации в имеющихся типах МК (в нашем случае – в МК семейства AVR) для того, чтобы общие положения приобрели конкретные формы осуществления и лучше усваивались студентами. Автор постарался добиться достижения поставленных задач и снабдить пособие необходимыми материалами.

Чтобы обеспечить методологическую совместимость пособия с существующими в настоящий момент в профессиональном сообществе представлениями об архитектуре и программировании МК пособие составлено с опорой на материалы многих известных литературных источников. Их список приводится в конце учебника. Цитируемые места из них берутся в кавычки и/или даётся ссылка на источник.

Автор полагает, что данное учебное внесет продуктивный вклад в получение образования студентами университетов, обучающимися в сфере информационных технологий и систем управления техническими объектами.

Введение

Расширение интеллектуальных возможностей современной техники базируется на использовании микропроцессоров (МП), программируемых логических интегральных схем (ПЛИС), приборов типа «система на кристалле» (System-On-Chip – SOC), например микроконтроллеров, и других современных цифровых приборов. Встраивание этих приборов в технические объекты позволяет автоматизировать процессы измерения, управления, контроля, регулирования и обработки информации, а также обеспечить такие свойства приборных комплексов, как многофункциональность, модифицируемость, адаптивность, обучаемость и ряд других.

В отличие от персональных компьютеров, предназначенных для личного использования при решении задач хранения, обработки и анализа информации, проведения исследований, управления проектами и других, для интеграции в различные устройства широко применяют микроконтроллеры. Микроконтроллер (МК; англ. Micro Controller Unit, MCU; controller – регулятор, управляющее устройство) – микросхема, предназначенная для управления различными техническими объектами – от бытовых приборов до промышленного оборудования. Типичный микроконтроллер содержит на одном кристалле процессор, ОЗУ, ПЗУ и периферийные компоненты: устройства ввода-вывода, таймеры и др. По сути, микроконтроллер – это однокристалльный компьютер, способный выполнять относительно простые задачи.

Первый патент на однокристалльный компьютер (или, как в то время говорили, однокристалльную микро-ЭВМ) был выдан в 1971 г. инженерам М. Кочрену и Г. Буну, сотрудникам американской компании Texas Instruments. Именно они предложили на одном кристалле разместить не только процессор, но и память с устройствами ввода-вывода. Разработанный ими 4-х разрядный микроконтроллер получил марку TMS1000 от Texas Instruments и был выпущен в 1972 году. Он содержал ОЗУ (32 байта), ПЗУ (1 Кбайт), часы и поддержку ввода-вывода, а также имел новую по тем временам возможность – добавление новых инструкций.

В 1976 году (через 5 лет после создания первого микропроцессора) на свет появился первый микроконтроллер фирмы Intel, получивший имя 8048. Помимо центрального процессора, на кристалле находились 1 Кбайт памяти программ, 64 байта памяти данных, два восьмибитных таймера, генератор часов и 27 линий портов ввода-вывода. МК семейства 8048 использовались в игровых консольных приставках Magnavox Odyssey, в клавиатурах первых IBM PC и в ряде других устройств.

В 1978 г. фирма Motorola выпустила свой первый микроконтроллер MC6801, совместимый по системе команд с выпущенным ранее микропроцессором MC6800.

Через 4 года, в 1980 г., Intel выпускает следующий микроконтроллер i8051. Удачный набор периферийных устройств, возможность гибкого выбора

внешней или внутренней программной памяти и приемлемая цена обеспечили этому микроконтроллеру успех на рынке. С точки зрения технологии микроконтроллер i8051 являлся для своего времени очень сложным изделием – в кристалле было использовано 128 тыс. транзисторов, что в 4 раза превышало количество транзисторов в 16-разрядном микропроцессоре i8086. Аналоги i8051 выпускали советские предприятия в Минске, Киеве, Воронеже, Новосибирске, на них выросло целое поколение отечественных разработчиков.

На сегодняшний день существуют сотни модификаций микроконтроллеров различных типов. Популярностью у разработчиков пользуются 8-битные микроконтроллеры семейства MCS-51 компании Intel (с родоначальником – i8051), 8-битные микроконтроллеры PIC фирмы Microchip Technology и AVR фирмы Atmel, 16-битные MSP430 фирмы TI, а также 32-битные микроконтроллеры архитектуры ARM, которую разрабатывает фирма ARM Limited и продает лицензии другим фирмам для их производства. Разработкой микроконтроллеров занимаются практически все крупнейшие мировые производители компьютеров, бытовой техники, промышленных систем и электронных компонентов – всем известные: Intel, AMD, Atmel, Microchip, Mitsubishi, Motorola, Analog Device, National Semiconductor, Texas Instruments, Philips, Siemens, Dallas, OKI и др.

Российские производители конкурентно способных микроконтроллеров представлены компаниями АО «Ангстрем» (Зеленоград, Москва), ЗАО «ПМК Миландр» (Зеленоград) и ОАО «НИИЭТ» (Воронеж). Есть еще группа компаний «Микрон» (Зеленоград, Москва), но они нацелены больше на создание маломощных специализированных чипов, вроде чипов для smart-карт. Вот перечень некоторых типов современных отечественных микроконтроллеров:

- 8-разрядный МК КР1878ВЕ1 (ядро RISC, «Тесей») (АО «Ангстрем»);
- 8-разрядная серия 1886ВЕхх (1886ВЕ1,2У – аналоги PIC17; 1886ВЕ3,4У – аналоги ST7хх, AT89C5131A-M); 32-разрядная серия 1986ВЕ9х (ядро RISC, ARM Cortex-M3) и МК 1901ВЦ1Т (ЗАО «ПМК Миландр»);
- 8-разрядная серия Н1830ВЕхх, МК 1882ВМ1Т (ядро MCS-51), МК 1887ВЕ4У (AVR RISC), МК 1887ВЕ7Т (архитектура и система команд AMCS-96); 16-разрядные Л1874ВЕхх, 1874ВЕхх, 1887ВЕ3Т (на базе лицензированного ядра C166SV1); 32-разрядные 1921ВКхх (32-битное ядро на базе ARM Cortex-M4F) (ОАО «НИИЭТ»).

В учебном пособии излагаются принципы построения и программирования микроконтроллеров, подробно рассматривается семейство микроконтроллеров AVR на примере ATmega, описывается его архитектура, организация и язык ассемблера.

Глава 1. Концепции построения компьютеров

1.1. Компьютер. Аппаратура и программы

Поскольку микроконтроллеры – это однокристальные компьютеры, то изучение их начнем с краткого напоминания элементарных основ построения компьютеров.

Понятие ЭВМ. *Компьютером* (от англ. computer — «вычислитель») или *электронной вычислительной машиной (ЭВМ)* называют электронную, программируемую (т.е. управляемую хранимой в памяти ЭВМ программой) машину для автоматической обработки цифровых данных [15].

Под *данными* понимаются не только и не столько числа, а всякая информация в дискретном (цифровом) представлении: всевозможные тексты, сигналы, изображения и т.п.

Компьютер, в отличие от программируемого калькулятора, обладает, кроме того, *высокой производительностью, большим объемом памяти и развитой многоуровневой организацией* [23].

Термины «компьютер» и «ЭВМ» имеют равноправное хождение. При возможном появлении в будущем компьютеров, работающих на оптических, квантовых и других физических принципах, понятие «электронная вычислительная машина», отражающее электронную природу процессов в машине, станет более узким, чем понятие «компьютер». Однако общее выражение «вычислительная машина» вполне равнозначно термину «компьютер».

Компьютерной или электронной вычислительной системой, вычислительной системой обычно называют компьютер или несколько компьютеров в комплексе с периферийным оборудованием и специальным программным обеспечением для определенного применения или класса применений. Например: 1. Компьютерная система разработки программ для встраиваемых микроконтроллеров; 2. Информационно-управляющая вычислительная система (ИУВС), например бортовой компьютер на самолете в составе автопилота. К простым ИУВС относятся микроконтроллеры; 3. Компьютерная система автоматизированного обучения; 4. Система автоматизированного проектирования (САПР) (называемая также АРМ – автоматизированное рабочее место или Workbench – буквально «рабочая скамья»).

При оснащении компьютерной системы соответствующими устройствами ввода-вывода, например аналого-цифровыми и цифро-аналоговыми преобразователями, она может производить обработку не только цифровой, но и аналоговой информации, что характерно для ИУВС, большинства МК и других управляющих компьютеров. Иными словами, ЭВС – это проблемно ориентированная ЭВМ или комплекс ЭВМ.

Коротко: *Компьютер – это универсальная машина для обработки информации. Компьютерная система – специализированное применение компьютера.*

Компьютерной, вычислительной системой можно называть, разумеется, и компьютер общего назначения или любой другой компьютерный прибор – мобильное устройство, микроконтроллер и т.п., подчеркивая тем самым, что он представляет собой сложную совокупность большого числа взаимодействующих друг с другом компонентов, т.е. систему.

Аппаратура и программное обеспечение [5]. В ЭВМ или системе прежде всего различают *аппаратуру (аппаратные, аппаратные средства)* и *программное оснащение, или обеспечение (программные средства)* (ПО). Под аппаратными средствами понимают разнообразные механические, электромеханические, магнитные и электронные элементы, устройства, приборы, блоки, из которых реализована машина. К программным средствам (ПО) относят все программы, связанные с машиной. Неотъемлемой частью ПО являются также языки программирования.

Основные компоненты ЭВМ перечислены в табл.1.1.

Таблица 1.1 – Состав компьютера

ЭВМ			
Аппаратура		Программы	
Центральная часть (ядро):	Периферийные устройства (периферия):	Системное ПО:	Прикладное ПО:
• ЦП – центральный процессор • ОП – основная память • КВВ – контроллеры ввода/вывода • Внутренняя магистраль	• УСП – устройства для связи ЭВМ с пользователем • Внешняя память • УСО – устройства для связи ЭВМ с техническими объектами • СТ – средства телекоммуникации	• ОС – операционная система • ИС – инструментальные системы (системы программирования): C++, Java, Assembler и др.) • Сервисные программы	• Расчетные (Mathcad, Matlab, Ansys) и конструкторские (Acad, Inventor) • системы управления базами данных (dBase, Visual FoxPro) и табличные процессоры (Excel) • Управляющие программы для систем автоматического управления; др. программы

Структура аппаратуры. В аппаратуре компьютера или системы различают внутреннюю и внешнюю (центральную и периферийную) части (рис.1.1).

Центральная часть компьютера. К *внутренней (центральной) части, или ядру* компьютера – *собственно компьютеру* относятся:

- 1) Центральный процессор (ЦП) или комплекс процессоров и непосредственно взаимодействующие с ЦП органы;
- 2) Основная (главная, первичная, оперативная) память (ОП);
- 3) Контроллеры ввода-вывода (КВВ) и контроллер прерываний; и сопрягающая все эти органы между собой
- 4) Системная (внутренняя, внутримашинная) магистраль («шина») или магистрали («шины»), состоящие каждая из шин данных, адресов и управления.

Периферийные устройства, или «периферию» (ПУ), принято разделять на:

- 1) Устройства ввода и вывода данных для связи компьютера с пользователем (УСП): клавиатура, мышь, мониторы, принтеры и др.;

- 2) Устройства внешней (периферийной, вторичной, массовой) памяти для хранения больших объемов информации: жесткие диски, флэш-карты, съемные диски и пр.;
- 3) Устройства ввода-вывода для связи компьютера с различного рода техническими объектами – источниками сигналов (датчиками) и приемниками сигналов (исполнительными механизмами, линиями связи и т.д.) (УСО). К УСО относятся аналого-цифровые (АЦП) и цифро-аналоговые (ЦАП) преобразователи и пр. устройства. К этой группе причисляют и устройства, обслуживающие компьютер (ОУ), например таймеры;
- 4) Средства телекоммуникации (телекоммуникационные средства), предназначенные для передачи данных на большие расстояния, например телекоммуникационные сети.

Более полная характеристика периферийных устройств представлена в параграфе (п.) 2.5.

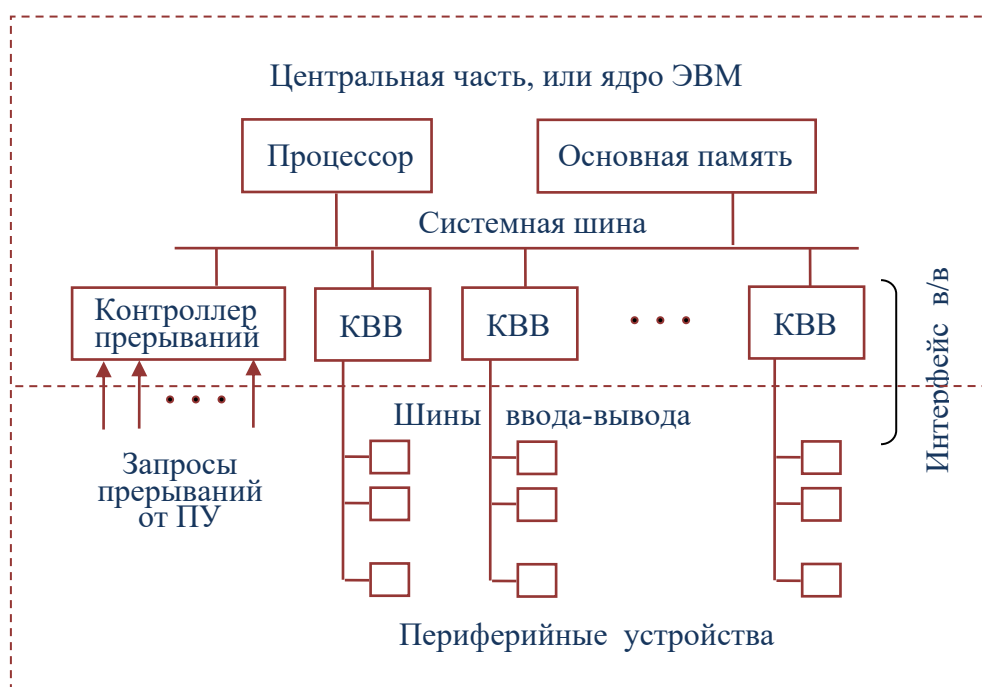


Рисунок 1.1 – Структура аппаратных средств ЭВМ.

КВВ – контроллер ввода-вывода

Структура программного обеспечения. В отличие от аппаратуры программное оснащение, или обеспечение (ПО) не разделяют на центральное и периферийное, или внутреннее и внешнее. ПО делится на *системное (общее)* и *прикладное (специальное, функциональное)* [5]. Программы, относящиеся к системному ПО, называются *системными*, а к прикладному ПО – *прикладными*.

Системное ПО состоит из двух главных частей:

- Операционной системы. Например: Windows, UNIX, Linux;
- Инструментальной системы (системы программирования, системы автоматизации программирования): C++, Pascal, Java, Ассемблер и т.д.

Кроме этих частей в системное ПО входят

- Сервисные программы, к которым относят системы технического обслуживания и программы-утилиты, например, антивирусы.

Операционная система является важнейшей и центральной частью системного ПО. Она представляет собой комплекс взаимосвязанных программ, постоянно находящихся в основной памяти компьютера и предназначенных для управления работой всех других программ, а также устройств компьютера и организации взаимодействия с пользователем. *Основная функция всех ОС – посредническая*. Она заключается в обеспечении интерфейсов пользователя (между пользователем и программно-аппаратными средствами), между программным и аппаратным обеспечением и между разными видами программного обеспечения. Операционная система взаимодействует с аппаратным обеспечением, прикладными программами и пользователем и может переносить информацию между ними (рис.1.2).

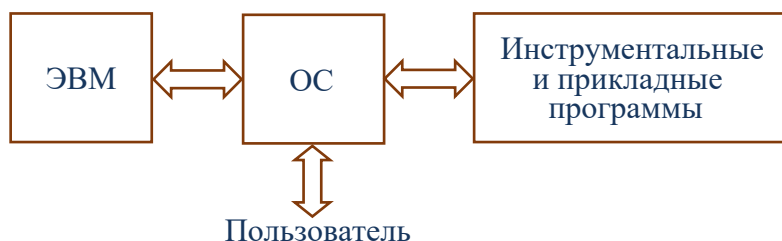


Рисунок 1.2 – Назначение операционной системы

Система автоматизации программирования (инструментальные средства, инструментальная система ЭВМ) предназначена для создания новых программ.

ИС включает в себя:

язык или языки программирования, на которых создаются исходные тексты программ, и набор таких программ, как:

- а) Транслятор: компилятор или/и интерпретатор (см. *примечание* ниже);
- б) Редактор;
- в) Компоновщик (редактор связей);
- г) Библиотеки прикладных программ;
- д) Загрузчик;
- е) Отладчик.

Эти программы могут быть выполнены в виде отдельных, автономных приложений (быть, как говорят, портативными – portable); или же они встраиваются в единую *интегрированную среду разработки* (IDE, Integrated Development Environment), обеспечивающую быстрый и удобный доступ к любой из них из любой точки среды.

Примечание. Трансляторы бывают двух типов: компиляторы и интерпретаторы [23]. *Компилятор* (англ. Compiler – составитель, собиратель) читает всю программу *целиком*, делает ее перевод и создает законченный вариант программы на машинном языке, сохраняя его в виде так называемого объектного (object), или абсолютного, файла или файлов (кодов, модулей) (с расширением ".obj"). *Интерпретатор* (англ. Interpreter – истолкователь, устный переводчик) осуществляет пооператорную трансляцию и выполнение исходной программы: переводит и выполняет программу *строка за строкой*: берет один оператор языка высокого уровня, транслирует и выполняет его, а затем берет следующий оператор. Он не генерирует машинный код всей программы. Например, язык Java имеет как компилятор, так и интерпретатор.

Сервисные программы расширяют возможности ОС по обслуживанию системы и обеспечивают удобство работы пользователя. К этой категории относят системы технического обслуживания и программы-утилиты.

Системы технического обслуживания – это совокупность программно-аппаратных средств, которые выполняют контроль, тестирование и диагностику устройств компьютера и обнаруживают неисправности в их работе. Они являются инструментом специалистов по эксплуатации и ремонту технических средств компьютера.

Программы-утилиты («утилиты», служебные программы) – это полезные специализированные программы, не включаемые обычно в серийные комплекты ОС и предназначенные для решения узкого круга вспомогательных задач. К ним относятся: диагностика, тестирование аппаратных и программных средств, оптимизация использования дискового пространства, восстановление разрушенной на магнитном диске информации, сжатие (архивирование) информации, запись информации на съемные диски (например, программа Nero Start Smart), защита от компьютерных вирусов (Dr.Web, Avast и т.п.) и др.

Прикладное ПО представляет собой пакет прикладных программ, созданных для пользователя и предназначенных для выполнения научных, проектных, производственных и учебных заданий, а также для бытового применения и развлечений:

- текстовые и графические редакторы, с помощью которых можно готовить различные тексты, создавать рисунки, строить чертежи; проще говоря, писать, чертить, рисовать (Word, WordPad, Paint и др.);
- табличные процессоры, позволяющие организовывать очень распространенные на практике расчеты для электронных таблиц (например, программа Microsoft Excel);
- коммуникационные (сетевые) программы, предназначенные для обмена информацией с другими компьютерами, объединенными с данным в компьютерную сеть: веб-браузеры, или веб-обозреватели (средства просмотра Web-документов), вспомогательные браузеры (обозреватели) и др.;
- очень популярным видом прикладного программного обеспечения являются компьютерные игры. Большинство пользователей именно с них начинает свое общение с ЭВМ;
- бухгалтерские программы, производящие начисления заработной платы и другие расчеты, которые делаются в бухгалтериях (1 С: Бухгалтерия и др.);
- системы автоматизированного проектирования (САПР, CAE – Computer Aided Engineering) и конструирования, дизайна (CAD – Computer Aided Design, ACAD – Automatic CAD, Inventor), которые помогают инженерам и конструкторам разрабатывать проекты различных технических устройств;
- пакеты *, позволяющие выполнять сложные математические расчеты при решении научных и инженерных задач без составления программ или с составлением программ-сценариев – скриптов (MCAD – Mathematical Computer Aided Design, MatLab – Matrix Laboratory, ANSYS – Analysis of Systems – программа для расчета механических и электромагнитных полей);
- программы управления технологическими процессами и техническими объектами в реальном масштабе времени (в темпе с процессом), используемые в компьютеризированных системах автоматического управления (САУ), в ИУВС;
- и многое другое.

* *Пакет прикладных программ* (англ. application package) или *пакет программ* – набор взаимосвязанных модулей, предназначенных для решения задач определенного класса некоторой предметной области.

1.2. Информация в компьютере. Программы и данные

Представление информации в компьютере. Любая информация (числа, команды, тексты и т.д.) представляется в компьютере в виде двоичных кодов (двоичных чисел, двоичных слов) фиксированной или переменной длины. Отдельные, минимальные элементы двоичного кода, имеющие значение 0 или 1 (двоичных цифр), называют *разрядами* (двоичными) или *битами* (от англ. сокращения словосочетания Binary digiT, bit – двоичная цифра). В компьютере

слова часто разбивают на части, называемые *слогами* или *байтами* («байт», англ. «byte» – сокращение от BinarY TErm – «двоичный терм», «двоичный член»). *Байтом* называют *наименьшую* порцию информации, назначаемую (адресуемую) процессором в основной памяти или периферийных устройствах, передаваемую по шине данных в параллельном формате, в один прием, и обрабатываемую процессором одномоментно, как одну, единую величину. В современных компьютерах используется байт, состоящий из 8 битов. Однако в истории компьютерной техники существовали решения с иными размерами байта (например, 6, 32 или 36 битов), поэтому иногда в компьютерных стандартах и официальных документах для однозначного обозначения группы из 8 битов используется термин «октет» (лат. *octet*).

Понятие слова является менее определенным, чем байт. В содержательном смысле *словом* называют любую единицу информации в компьютере, имеющую законченное логическое значение и воспринимаемую как целостную, не расчленяемую конструкцию. Например: многобайтное число, машинную команду *, слово русского языка в тексте, записанное с помощью ASCII-кодов ** и т.д. Обычно подобное слово состоит из целого числа байтов, в частном случае из одного байта, хотя никому не возбраняется разместить, например, какое-нибудь двоичное число по нецелому числу байтов, но такое число-слово неудобно обрабатывать программно.

* Понятие машинной команды (инструкции) дано в п.2.1, 6.1.

** ASCII (англ. American Standard Code for Information Interchange) – двоичные коды символов внешнего алфавита (латинских и национальных букв, десятичных цифр, знаков препинания) и управляющих символов (возврат каретки, перевод строки, звонок и др.).

Иную трактовку понятия слова связывают не с содержательным аспектом, а с разрядностью процессора. *Словом* (машинным, физическим словом) называют в этом случае *наибольшую* порцию информации, передаваемую по шине данных в параллельном формате и обрабатываемую процессором в один прием. Поэтому размер машинного слова равен ширине шины данных и в байтах составляет обычно целую степень 2. В процессорах архитектуры IA-32 (Intel Architecture 32 bit) и AMD x86-64 (Advanced Micro Devices x86 64 bit, коротко «AMD64») машинное слово состоит из восьми байт (64 бита), в МК AVR – из 2 байт для программ и из 1 байта для данных. Итак, если *байт* – это *наименьшая адресуемая* порция информации в памяти или ПУ, то *машинное слово* – *наибольшая*. *Словом* называют также вообще любую порцию информации, передаваемую по ШД за один раз и обрабатываемую в один прием (в том числе один байт).

Иногда размер слова определяют некоторым условным стандартом, единым для целого класса или семейства процессоров. Так, для процессоров IA-32 и AMD64 *словом* (но не машинным словом) называют не 64-битное, а 16-битное значение. Соответственно 32-битное значение называют *двойным словом*, а 64-битное – *четверным словом*, чтобы обеспечить совместимость терминологии для всего семейства. 80-битное (10-байтное) данное (поддерживаемое встроенными в процессоры IA-32 и AMD64 арифметически-

ми сопроцессорами FPU) – *tbyte* (ten byte – десять байт). Такая терминология удобна для программистов, не заботящихся об особенностях физического представления данных в компьютере.

Таким образом, *базовыми единицами информации (типами данных)* процессора служат: биты, байты и слова. Из них байты и слова могут передаваться по шине данных в параллельном формате и обрабатываться процессором в один прием (и посредством одной машинной команды) (подобные слова не должны превышать размера машинных слов). Биты таким свойством не обладают. Для обработки отдельных битов приходится применять последовательность из нескольких машинных команд, хотя ряд процессоров имеют и бит-ориентированные команды – команды, обрабатывающие отдельные биты внутри байта или слова.

Физические носители информации. Вся информация в компьютере представляется двоичными числами – комбинациями нулей и единиц. Позиция в двоичном числе, отводимая для записи нуля или единицы, называется двоичным разрядом или битом. Физическим носителем двоичного разряда в компьютере служит некоторое техническое устройство, двум различным состояниям которого приписываются значения 0 и 1. К такого рода устройствам относятся триггеры, конденсаторы, магниты, оптические элементы. В триггере – электронном элементе с двумя устойчивыми состояниями логический «0» представляется чаще всего сигналом от 0 до 1 В, а логическая «1» – сигналом от 2 до 5 В. Напряжение за пределами указанных величин недопустимо. В конденсаторе наличие заряда предполагает «1», а отсутствие – «0». На самом деле, отдельного конденсатора нет, а его роль выполняет обычно электрическая емкость между затвором и истоком полевого транзистора. Но для простоты изложения емкость выделена нами как бы в отдельный элемент. В магните «1» выражается его намагниченностью, а «0» – размагниченностью или противоположной намагниченностью. Роль магнита выполняет крошечный участок тонкого ферромагнитного слоя, нанесенного на поверхность ленты или диска. «Нули» и «единицы» записываются на этот слой при помощи магнитного поля, создаваемого сложной микроскопической конструкцией (магнитной «головкой»), находящейся на самом кончике механического держателя, и располагаются по прямой дорожке ленты и по спиральной дорожке диска. В оптических дисках «0» и «1» записываются на поверхность диска (тоже вдоль спиральной дорожки) при помощи лазерного луча, формируемого оптической головкой: «0» – в виде мельчайшей выемки, впадины – «питы» (от англ. *pit* – «ямка», «углубление»), а «1» – в виде ровного участка на поверхности диска, или «ленда» (*land* – ровная поверхность) – промежутка между питами. Чтение производится с помощью считывающего лазерного луча, направляемого на вращающийся диск. При сканировании лазерный луч по-разному отражается от углубления и от ровной основной поверхности. Если от основной поверхности (*land*) свет отражается полностью, то в углублениях (*pits*) свет поглощается (из-за интерференции).

Поскольку оба состояния 0 и 1 указанных устройств устойчивы, т.е. мало подвержены воздействию различного рода помех, то эти устройства могут надежно сохранять один двоичный разряд числа, и по этой причине они используются в качестве *элементов памяти* компьютера. Для представления многоразрядного двоичного числа (байта или слова) используется упорядоченный набор (блок, линейка) таких устройств (элементов памяти), называемый *ячейкой памяти* (п.2.2). В частности, линейка триггеров образует так называемый *регистр* – электронное устройство для запоминания двоичного числа. Регистры используются в качестве собственных регистров процессора, портов ввода-вывода и, наряду с емкостными устройствами, ячеек основной памяти ЭВМ. Магнитные и оптические элементы применяются для создания устройств внешней, периферийной памяти ЭВМ – дисков и лент. Съёмные накопители флэш-карты представляют собой разновидность полупроводниковой технологии электрически стираемой, перепрограммируемой, энергонезависимой памяти (EEPROM, англ.

Electrically Erasable Programmable Read-Only Memory), информация в которой хранится в массиве транзисторов с плавающим затвором.

Программы и данные. Любую информацию, хранящуюся в памяти компьютера и обрабатываемую процессором, можно подразделить по смыслу на команды (программы) и операнды (данные). Команды (иначе – «инструкции») указывают процессору, какие операции он должен выполнять, а операнды – это те числа, над которыми эти операции выполняются, команды отвечают на вопрос «что делать», а операнды – «над чем». Говоря коротко, МП обрабатывает *данные* (собственно данные – операнды) в соответствии с *программой*. Понятие программы и данных является центральным в ЭВМ. Вся цифровая информация, находящаяся в памяти ЭВМ, интерпретируется процессором либо как программы, либо как данные. Программы и данные – это тот материал, с которым работает процессор. Для процессора очень важно отличать программы от данных, так как первые (т.е. программы) диктуют процессору порядок его действий, а последние (т.е. данные) служат теми числами, над которыми эти действия выполняются. Понятия программы и данных проходят через всю идеологию построения ЭВМ и, в конечном счете, определяют принципы ее архитектуры и организации.

Программа, непосредственно воспринимаемая машиной, располагается в основной памяти. Она представляет собой последовательность машинных команд, записанных в двоичных кодах и расположенных в памяти в том порядке, в каком они должны выполняться процессором – подряд, одна за другой, по смежным ячейкам памяти, а в случае переходов по программе – по адресам перехода (см. также п.2.1).^{*} Вся программа занимает в памяти один или несколько непрерывных массивов информации. Такую общую форму имеет любая машинная программа. Содержание программы определяется совокупным смыслом составляющих ее команд, которые будучи расположенные в заданном порядке, реализуют требуемый алгоритм обработки данных по решению поставленной задачи. Программа выступает в качестве задания процессору на работу.

^{*} Согласно стандарту ISO 2382/1-84 *машинная программа* – это упорядоченный набор команд, подлежащий обработке. *Ячейка памяти* – это устройство, хранящее двоичный код, двоичное число. Смежные ячейки памяти – это ячейки, имеющие соседние адреса. Адреса служат для нахождения ячеек в основной памяти процессором. Структура основной памяти и ее ячеек и их адресация подробно разбираются в п.2.2.

Перед запуском на счет программа должна быть загружена в основную память компьютера. Часть ОП, где размещаются программы, называется **программной памятью**. Программная память располагается в постоянном (ПЗУ) и оперативном (ОЗУ) запоминающих устройствах. Управляющие ЭВМ, встраиваемые в технологическое оборудование, (например, однокристальные микро-ЭВМ – микроконтроллеры) содержат в ПЗУ, как правило, все необходимые управляющие и системные программы. Универсальные ЭВМ хранят в ПЗУ лишь стартовую, иницилирующую часть программного оснащения, служащую для начальной загрузки операционной системы с жесткого магнитного диска в

ОЗУ. В ПЗУ этих ЭВМ находятся также некоторые вспомогательные программы: тестовые программы для проверки работоспособности устройств и объема памяти, программы-драйверы периферийных устройств, подпрограммы обработки прерываний и др. Совокупность упомянутых программ ПЗУ составляет так называемую базовую систему ввода-вывода ROM BIOS (Read Only Memory Basic Input/Output System). В ПЗУ могут храниться, кроме того, некоторые константовые данные, например марка процессора, год выпуска и пр.

Данные (цифровая информация, интерпретируемая процессором как **операнды**, т.е. как числа, над которыми производятся операции) записаны в другой части ОП, называемой **памятью данных**, обычно в ОЗУ, а также могут находиться в программной памяти (когда они входят в состав команд). Данные (операнды), размещенные в памяти данных, называются **переменными**, так как они могут изменяться в ходе работы программы. Они тоже объединяются обычно в непрерывные массивы информации (как и команды в программной памяти). Данные (операнды), находящиеся в программной памяти, т.е. входящие в состав команд (встроенные в коды команд в качестве непосредственных операндов или адресов) называются **константами**. Их нельзя изменить во время работы программы (программа не изменяет саму себя).

Заметим, что *данными* называют зачастую любую информацию, хранящуюся в компьютере, т.е. не только операнды, но и команды, поскольку и те и другие служат отправным материалом для работы компьютера. К слову сказать, и шина компьютера, предназначенная для считывания процессором команд из программной памяти и обмена операндами между устройствами ЭВМ, именуется шиной данных.

Источники программ и данных. Программы и данные, накопленные в основной памяти, поступают в нее от пользователя через устройства ввода (с клавиатуры, манипуляторов типов «мышь» и «джойстик», сканера и др. устройств), от технических объектов (датчиков физических величин через аналого-цифровые преобразователи или др. сопрягающие устройства), из внешней памяти (с магнитного диска, магнитной ленты, флэш-карты и т.д.) или из телекоммуникационных, например компьютерных, сетей. Запись программ в ПЗУ производится с помощью специальных приборов – программаторов ПЗУ, или в процессе изготовления. Данные формируются в основной памяти также в процессе счета программы.

Результаты счета выводятся через устройства вывода пользователю (на экран монитора, принтер и др. устройства) или на технические объекты (на исполнительные механизмы через цифро-аналоговые преобразователи и усилители) или во внешнюю память (на диски, ленты, флэш-карты) или в телекоммуникационные сети.

1.3. Гарвардская и фон-неймановская модели вычислительных систем

Гарвардская и фон-неймановская концепции построения вычислительных систем представляют собой наиболее общую совокупность идей, воплощенных в структуре вычислительных систем. Эти идеи были сформулированы в 40-х годах 20 века на этапе зарождения компьютерной техники и до сих пор остаются актуальными. Мы знаем их как гарвардскую и фон-неймановскую (более известную как фон-неймановскую) модели или архитектуры.

Гарвардская архитектура была разработана Говардом Эйкенем (в конце 1930-х гг. в Гарвардском университете), фон-неймановская (в 1944-46 гг. в Принстонском университете) – Джоном (Яношем) фон Нейманом (математиком венгерского происхождения). Их основное отличие заключалось в том, что архитектура фон Неймана использовала единую память (и общую шину данных), а гарвардская предполагала наличие нескольких шин (в оригинале две: шина данных и шина команд), к одной из которых подключалась память данных, к другой – память команд (рис.1.3).

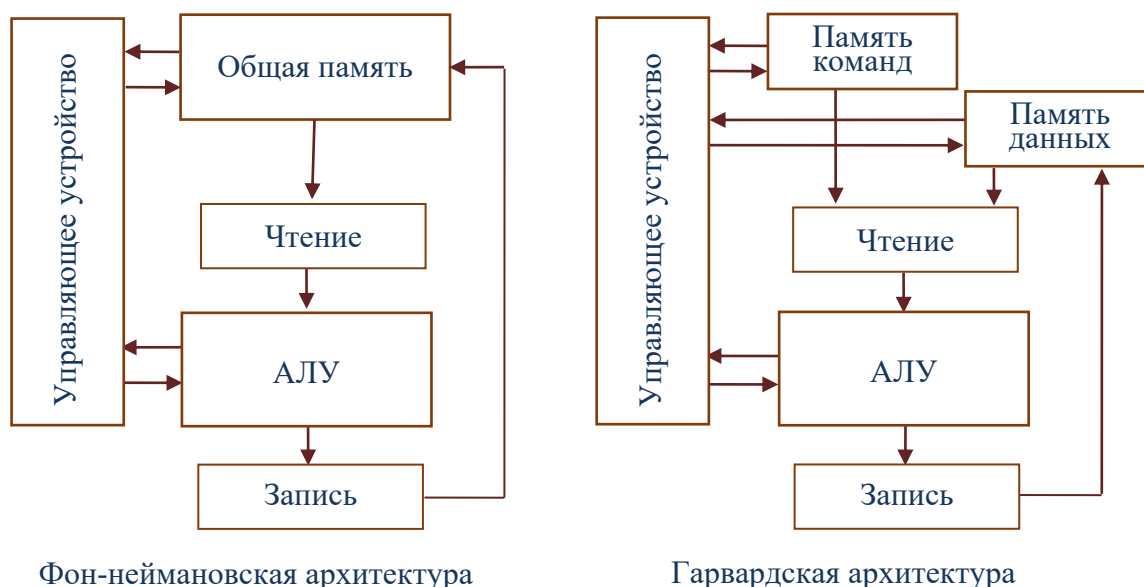


Рисунок 1.3 – Две модели построения вычислительных систем

Согласно этой схеме в чистой архитектуре фон Неймана процессор в каждый момент времени может либо читать инструкцию, либо читать/записывать единицу данных из/в памяти. Оба действия одновременно происходить не могут, поскольку инструкции и данные используют один и тот же поток (шину).

В компьютере с гарвардской архитектурой применяется принцип распараллеливания операций: процессор может считывать очередную команду и оперировать памятью данных одновременно. Поэтому компьютер с гарвардской архитектурой при определенной сложности схемы быстрее, чем компьютер с архитектурой фон Неймана, поскольку потоки команд и данных расположены на отдельных физически не связанных между собой аппаратных каналах. Однако из-за более сложной схемы гарвардскую архитектуру используют обычно для построения простых специализированных машин, например, мик-

роконтроллеров, а универсальные, например, настольные персональные компьютеры строятся по фон-неймановскому принципу.

Обобщая содержание рассмотренных подходов, можно назвать следующие общие принципы построения ВС, присущие принстонской концепции, называемые также *принципами фон Неймана*:

1) *Принцип двоичного кодирования.*

Для представления данных и команд используется двоичная система счисления.

2) *Принцип однородности памяти.*

Как программы (команды), так и данные хранятся в одной и той же памяти.

3) *Принцип адресуемости памяти.*

Структурно основная память состоит из пронумерованных ячеек; процессору в произвольный момент времени доступна любая ячейка.

4) *Принцип последовательного программного управления.*

Все команды располагаются в памяти и выполняются последовательно, одна после завершения другой, в порядке, определяемом программой.

5) *Принцип жесткости архитектуры.*

Неизменяемость в процессе работы топологии, архитектуры, списка команд.

Компьютеры, построенные на этих принципах, относят к типу фон-неймановских.

Гарвардская концепция ВС отличается от принстонской во втором принципе, который для нее звучит так:

2) *Раздельное хранение команд и данных и их одновременная обработка.*

Хранилище инструкций и хранилище данных представляют собой разные физические устройства; канал инструкций и канал данных также физически разделены.

Таким образом, отличительными признаками гарвардской архитектуры от принстонской являются:

- Хранилище инструкций и хранилище данных представляют собой разные физические устройства.
- Канал инструкций и канал данных также физически разделены.

Первым компьютером, в котором была использована идея гарвардской архитектуры, был «Марк I». Гарвардская архитектура используется в программируемых логических контроллерах и микроконтроллерах, таких как Microchip PIC, Atmel AVR, Intel 4004, Intel 8051.

1.4. CISC и RISC процессоры

В зависимости от принципа построения процессорного ядра (технологии построения управляющего устройства) все процессоры можно условно поделить на две большие разновидности: CISC-процессоры (Complex Instruction Set Computer – компьютер с набором комплексных инструкций) и RISC-

процессоры (Reduced Instruction Set Computer – компьютер с набором сокращенных инструкций). [6, 14, 17]

CISC-процессоры.

Особенности архитектуры CISC-процессоров:

- ☐ нефиксированное значение длины команд – от коротких (в 1 байт) до длинных (в один-два десятка байт);
- ☐ арифметические действия кодируются в одной команде;
- ☐ небольшое число регистров, каждый из которых выполняет строго определённую функцию.

CISC-процессоры являются традиционными, а их система команд (инструкций) включает большое количество функционально разнообразных команд, имеющих и малую и большую длину и допускающих все возможные методы адресации. При считывании команды из памяти процессор осуществляет ее интерпретацию – обращается в ПЗУ микрокоманд и получает набор микроинструкций, реализующий алгоритм выполнения данной команды.

Фактически в CISC-процессорах реализован перенос акцентов с программного уровня системы на аппаратный. Разработчики CISC-процессоров стремились ввести в их состав как можно большее число различных АЛУ, сопроцессоров и т.д., поскольку очевидно, что аппаратная реализация какой-либо функции обеспечивала более быстрое ее выполнение, чем программная. Да и длина кода для реализуемых аппаратным путем функций существенно меньше. Поэтому в 80-х годах процессоры «росли вширь», оснащались новыми узлами, совершенствуя уже имеющиеся. В результате появились системы команд, подобные системе семейства x86, включающие в себя тысячи различных инструкций. Эти системы должны были выполнять как можно больше работы с целью облегчения ручного написания программ на языке ассемблера или прямо в машинных кодах, а также упрощения реализации компиляторов. Нередко в наборы команд включались инструкции для прямой поддержки конструкций языков высокого уровня. Другая особенность этих наборов – большинство инструкций, как правило, допускали все возможные методы адресации – к примеру и операнды, и результат в арифметических операциях доступны не только в регистрах, но и через непосредственную адресацию, и прямо в памяти. Позднее такие архитектуры были названы CISC (Complex instruction set computer).

Однако, начиная с некоторого момента, CISC-технология стала не ускорителем, а тормозом роста производительности систем. Дело в том, что большое разнообразие команд и различная их длина существенно усложняют аппаратуру управления внутри процессора. Большое число действий, которое должен совершить микропроцессор при выполнении команды, заставляло отводить под устройство управления большую часть кристалла микросхемы (до 70% в процессорах x86). В результате для АЛУ, сопроцессоров и других устройств, повышающих производительность процессора, оставалось всего 30...40% площади кристалла.

Обнаружился и другой нежелательный эффект. Многие компиляторы не задействовали все возможности больших наборов инструкций CISC-процессоров, а на сложные методы адресации уходило много времени из-за дополнительных обращений к медленной памяти. Было показано, что такие функции лучше исполнять не одной, сложной, инструкцией, а последовательностью более простых инструкций, если при этом процессор упрощается и в нем остается место для большего числа регистров, за счет которых можно сократить количество обращений к памяти. Эти идеи привели к созданию RISC-процессоров. В первых архитектурах, причисляемых к RISC, большинство инструкций для упрощения декодирования имеют одинаковую длину и похожую структуру, арифметические операции работают только с регистрами, а работа с памятью идет через отдельные инструкции загрузки (load) и сохранения (store). Эти свойства и позволили лучше сбалансировать этапы конвейеризации, сделав конвейеры в RISC значительно более эффективными и позволив поднять тактовую частоту.

RISC-процессоры.

Особенности архитектуры RISC-процессоров:

- ☐ фиксированная и небольшая длина команды (1-2 слова). За счет упрощения инструкций их декодирование проще, и они не нуждаются в интерпретации, а время выполнения – короче, и быстродействие процессора увеличивается;
- ☐ минимально возможное число команд, имеющих простую и регулярную структуру. В этих процессорах «центр тяжести» перенесен на программные средства;
- ☐ относительно большое число регистров общего назначения, выполняющих все функции.

Нередко слова «сокращенный набор команд» понимаются как минимизация количества инструкций в системе команд. На самом деле, термин «сокращенный» в названии описывает тот факт, что сокращен объем (и время) работы, выполняемый каждой отдельной инструкцией (как максимум один цикл доступа к памяти), тогда как сложные инструкции CISC-процессоров могут требовать сотен циклов доступа к памяти для своего выполнения. Поэтому «Reduced Instruction Set» уместней было бы переводить как «набор сокращенных команд».

Другими словами, в RISC-процессорах принята стратегия «длинная программа – короткие команды» в противовес господствующей «короткая программа – длинные команды».

Инструкции в архитектуре RISC имеют фиксированную и небольшую длину. За счет упрощения инструкций их декодирование проще, и они не нуждаются в интерпретации, а время выполнения – короче, и быстродействие процессора увеличивается. Первые RISC-процессоры даже не имели инструкций умножения и деления. Это облегчает повышение тактовой частоты и делает более эффективной суперскалярность (распараллеливание инструкций между несколькими исполнительными блоками). Конвейеризация, как вариант распараллеливания действий, – это особенность, которая позволила первым RISC-процессорам на голову обогнать своих конкурентов в тестах производительности. В этих процессорах «центр тяжести» перенесен на программные средства, оставлено минимально возможное число команд, имеющих простую и регулярную структуру. Другими словами, в RISC-процессорах принята стратегия «длинная программа – короткие команды» в противовес господствующей «короткая программа – длинные команды». Аббревиатура RISC расшифровывается как Reduced Instruction Set Computer и переводится обычно как компьютер с сокращенным набором команд, т.е. речь идет о процессорах, система команд которых существенно сокращена в сравнении с системой команд x86.

Четыре основных принципа, которые положены в основу процессоров группы RISC:

- ☐ любая операция, к какому бы типу она ни принадлежала, должна выполняться за один такт;
- ☐ операции обработки данных реализуются только в формате «регистр – регистр». Обмен между памятью и регистрами (т. е. модификация памяти и чтение из нее необходимых данных) осуществляется только лишь с помощью специальных команд чтения/записи;
- ☐ система команд должна содержать минимальное число наиболее часто используемых простейших команд одинаковой длины;
- ☐ состав системы команд должен быть оптимизирован с учетом требований компиляторов языков высокого уровня.

Первая система, которая может быть названа RISC-системой, – суперкомпьютер CDC 6600, который был создан в 1964 г., за десять лет до появления соответствующего термина. CDC 6600 имел RISC-построение всего с двумя режимами адресации («регистр + регистр» и

«регистр + непосредственное значение») и 74 кодами команд (тогда как процессор 8086 имел 400 кодов команд).

Наиболее известные RISC-системы были разработаны позднее. В 1982 г. в университете Беркли появился на свет процессор RISC-I. В нем было 44 420 транзисторов (для сравнения: в CISC-процессорах того времени их было около 100 тыс.). RISC-I имел всего 32 инструкции, но превосходил по скорости работы любой одночиповый процессор того времени. Через год, в 1983 г., был выпущен RISC-II, который состоял из 40 760 транзисторов, использовал 39 инструкций и работал в три раза быстрее RISC-I.

Практически в то же время, в 1981 г., в Стэнфордском университете начат аналогичный проект, названный «MIPS-архитектура». (MIPS – сокращение от англ. Microprocessor without Interlocked Pipeline Stages – микропроцессор без взаимосвязанных конвейерных этапов – системы команд и микропроцессорных архитектур, разработанных компанией MIPS Computer Systems (в настоящее время Wave Computing).) Создатели MIPS практически полностью сфокусировались на конвейерной обработке, что позволило разработанному чипу работать значительно быстрее аналогов. Наиболее важным было требование выполнения любой из инструкций процессора за один такт. Однако его исполнение имело негативный побочный эффект в виде удаления из набора инструкций таких полезных операций, как умножение или деление.

В первые годы развития RISC-архитектуры многие в компьютерной индустрии считали, что преимущества RISC-процессоров не проявятся при использовании в реальных продуктах из-за низкой эффективности использования памяти в составных инструкциях. Однако с 1986 г. исследовательские проекты RISC начали выпускать первые работающие продукты.

Как оказалось, в начале 1990-х гг., RISC-архитектуры позволяют получить бóльшую производительность, чем CISC, за счет распараллеливания, а также за счет возможности серийного повышения тактовой частоты и упрощения кристалла с высвобождением площади под кэш-память, достигающий огромных емкостей. Также, RISC-архитектуры позволили сильно снизить энергопотребление процессора за счет уменьшения числа транзисторов.

В настоящее время многие архитектуры процессоров являются RISC- подобными, к примеру, ARM, DEC Alpha, SPARC, AVR, MIPS, POWER и PowerPC. Наиболее широко используемые в настольных компьютерах процессоры архитектуры x86 ранее являлись CISC-процессорами, однако новые процессоры, начиная с Intel 486DX, являются CISC-процессорами с RISC-ядром. Они непосредственно перед исполнением преобразуют CISC-инструкции x86-процессоров в более простой набор внутренних инструкций RISC.

После того, как процессоры архитектуры x86 были переведены на суперскалярную RISC-архитектуру, можно сказать, что большинство существующих ныне процессоров основаны на архитектуре RISC.

Коротко:

- CISC-процессоры – процессоры с развитой системой команд. Система команд процессорного ядра имеет инструкции разного формата – от однобайтного до многобайтного. Различные инструкции при этом имеют и существенно разное время исполнения.

- RISC-процессоры – процессоры с набором сокращенных команд. Одна инструкция, как правило, занимает только одну ячейку памяти, и все инструкции имеют равное и малое время исполнения.

Глава 2. Функциональные органы компьютера

2.1. Процессор

Центральный процессор (ЦП), или просто «процессор» (Пр) представляет собой программно-управляемый (т.е. управляемый хранимой в памяти ЭВМ программой) функциональный орган ЭВМ, непосредственно производящий вычисления – арифметические и логические операции над данными и иные действия по их обработке. Помимо этого, он руководит работой других устройств компьютера.

Процессор называют центральным потому, что наряду с ним в компьютере могут находиться другие – специализированные процессоры, например процессоры периферийных устройств (процессор клавиатуры и пр.).

Для обозначения процессора используют также аббревиатуры ЦПУ (центральное процессорное устройство), CPU (Central Processing Unit, дословно – центральное обрабатывающее устройство), МП (микропроцессор), МП БИС (микропроцессорная большая интегральная схема) и др.

Современные ЦП, выполняемые в виде отдельных микросхем (чипов), реализующих все особенности, присущие данного рода устройствам, называют микропроцессорами. С середины 1980-х последние практически вытеснили прочие виды ЦП, вследствие чего термин «процессор» стал чаще всего восприниматься как обыкновенный синоним слова «микропроцессор». Тем не менее, это не всегда так: центральные процессорные устройства некоторых суперкомпьютеров даже сегодня представляют собой сложные комплексы больших (БИС) и сверхбольших (СБИС) интегральных микросхем.

Многоядерные процессоры. Поскольку тактовые частоты современных процессоров приблизились к физическому пределу, для повышения их производительности применяется объединение нескольких копий процессора (так называемых «процессорных ядер», cores) в одном корпусе. Подобные приборы называются *многоядерными* процессорами. Существующие многоядерные процессоры имеют в своем составе от 2 до 32 ядер, параллельно работающих на общей памяти. В многоядерном процессоре поток команд, поступающий из памяти, разделяется (распараллеливается) внутри процессора между его ядрами, обрабатывающими каждое свою часть потока. Кроме многоядерных, в компьютерах применяются и *гибридные* процессоры, состоящие из центрального и графического процессоров.

Принцип обработки данных процессором – *принцип последовательного программного управления*. Обработка данных в процессоре производится путем последовательного выполнения им некоторых простых операций, проводимых по указке машинных команд. Команды поступают на процессор (считываются им) из *машинной программы*, хранимой в основной памяти ЭВМ. Программа представляет собой упорядоченный набор машинных команд («инструкций») (п.1.2). Она служит заданием процессору на решение определенной задачи. Поэтому программа по форме – это набор машинных команд, а по содержанию – задание процессору на работу. Команды и данные записываются в машине с помощью двоичных чисел – комбинаций нулей и единиц. Сказанное означает,

что все, что умеет делать компьютер, запрограммировано программистом и исполняется процессором.

Машинная команда (машинная инструкция) служит приказом процессору на выполнение какой-нибудь простой операции. Электронные схемы каждого процессора могут распознавать и выполнять ограниченный набор простых команд, которые обычно не сложнее чем: сложить 2 числа; проверить, не является ли число нулем; сдвинуть код числа на один бит влево или вправо; скопировать кусок данных из одной части памяти в другую; и т.п. Более сложные действия производятся процессором посредством выполнения группы команд, т.е. программы. Все программы, на каком бы языке они не были бы написаны, перед выполнением должны быть превращены в последовательность команд. [22]

Эти примитивные команды в совокупности составляют язык, на котором люди могут общаться с компьютером. Каждая команда на нем представлена в виде так называемого *опкода* (операционного кода) – двоичного кода отдельной операции. Такой язык называется *машинным языком*. Разработчик при создании нового компьютера должен решать, какие команды включать в машинный язык этого компьютера. Это зависит от назначения компьютера, от того, какие задачи он должен выполнять. Обычно стараются сделать машинные команды как можно проще, чтобы избежать сложностей при конструировании компьютера и снизить затраты на необходимую электронику. [23]

Для удобства программирования вместо числовых опкодов, которые только и понимает процессор, обычно используют их условные буквенные обозначения – *мнемоники* команд. Набор таких мнемоник, вместе с некоторыми дополнительными возможностями, называется *языком ассемблера*.

Порядок функционирования процессора. Процессор ведет себя, в известном смысле, довольно просто. Он начинает функционировать сразу после пуска при условии, что к нему подведено электропитание и поступают сигналы от тактового генератора. Тогда для пуска надо подать на процессор сигнал пуска (системного сброса, перезагрузки Сбр – Сброс, Пуск, RESET). По этому сигналу процессор стартует – начинает отработку команд с некоторой пусковой ячейки памяти, скажем, с нулевой (0) *. В этой ячейке должна храниться самая первая, стартовая команда компьютера. Номер пусковой ячейки заранее определен, он «известен» процессору, встроен в его электронную схему. Поэтому процессор всегда «знает», с какой ячейки памяти ему следует начать работу после пуска. Ему не надо сообщать об этом заранее.

* Мы уже упоминали в сноске ранее, что *ячейка памяти* – это устройство, хранящее двоичное число, а местоположение ячейки в основной памяти определяется ее номером (адресом). См. также п.2.2.

После пуска процессор приступает к работе, он начинает выбирать и выполнять команды программы из памяти. Выборка и выполнение команд происходит последовательно, одна за другой, из очередных ячеек памяти, в которых они расположены, в том порядке, как они записаны в памяти. Обработка команд начинается, как мы сказали, с predetermined – пусковой ячейки (например,

с 0-й). Процессор выбирает (считывает) находящуюся в ней команду и расшифровывает ее. Благодаря этому он «узнаёт», какие операции над данными он должен выполнить по этой команде. «Узнав» состав операций, процессор принимается за их выполнение: читает число из ячейки памяти, или преобразует содержимое собственного регистра *, или совершает другие действия, предписанные командой. Выполнив эти действия, процессор переходит к следующей, второй команде программы. Для этого он обращается к следующей ячейке памяти, в которой хранится вторая команда, в нашем случае, к 1-й ячейке, и выбирает из нее и расшифровывает вторую команду. Расшифровав, он выполняет все операции над данными, заданные и этой командой. Затем, в таком же порядке процессор выбирает и выполняет третью команду из 2-й ячейки памяти, четвертую – из 3-й ячейки и так далее, поочередно, одну за другой все другие команды программы.

* *Регистр* – это, как и ячейка памяти, устройство, предназначенное для хранения двоичного числа (см. далее в этом параграфе и рис.2.4).

Подобным образом процессор функционирует при выполнении обычных команд – *команд обработки данных*. Эти команды предназначены для преобразования и пересылки *операндов* – данных, над которыми производятся операции. По указанию этих команд осуществляется сложение и вычитание чисел, передача кодов из регистра на регистр, запись результатов в память, чтение из памяти и т.д. После выполнения каждой такой команды процессор переходит к следующей, соседней ячейке памяти и выполняет команду из этой ячейки. Команды в этом случае выполняются в *линейном, естественном порядке*, так, как они размещены в памяти – от ячейки к ячейке (рис.2.1). Такой же порядок поддерживается и при отработке команд изменения состояния процессора, упоминаемых ниже.

Если же по ходу программы процессору попадает *команда перехода (передачи управления)*, то линейный порядок выполнения команд нарушается. Процессор ведет себя в этом случае иначе. Он прекращает линейное продвижение по программе и под действием команды перехода делает переход, «скачок» (*go to, jump*) вперед или назад по программе на новую ячейку памяти, содержащую нужную в данный момент команду. Адрес этой, новой, ячейки задается самой командой перехода. Далее процессор продолжает счет уже с указанной, принудительно назначенной ячейки. Он начинает выбирать и выполнять команды из нее и следующих за ней ячеек. Например, если в 100-й ячейке находится команда «перейти на ячейку под номером 40», то после ячейки 100 процессор обратится не к 101-й ячейке, а вернется назад на 40-ю ячейку и начнет функционировать заново с этой, 40-й, ячейки памяти. В результате в программе образуется замкнутый цикл: процессор вновь будет выполнять команды из ячеек 40, 41, 42 и т.д. (см. рис.2.1). Этот цикл будет бесконечным (с бесконечным числом повторений), если в ячейке 100 находится команда безусловного перехода на ячейку 40. Выход из него возможен только аппаратными средствами – перезапуском процессора сигналом RESET (Сбр, Пуск) или подачей сигнала

прерывания INTR (Interrupt, ЗПр – Запрос прерывания). Если же используется команда перехода по условию (например, по условию повторения цикла, скажем, 7 раз), то после 7 повторений процессор уже не вернется назад, на начало цикла, а двинется дальше по программе (выйдет из цикла) и обратится за следующей командой к ячейке 101.

Программа в памяти

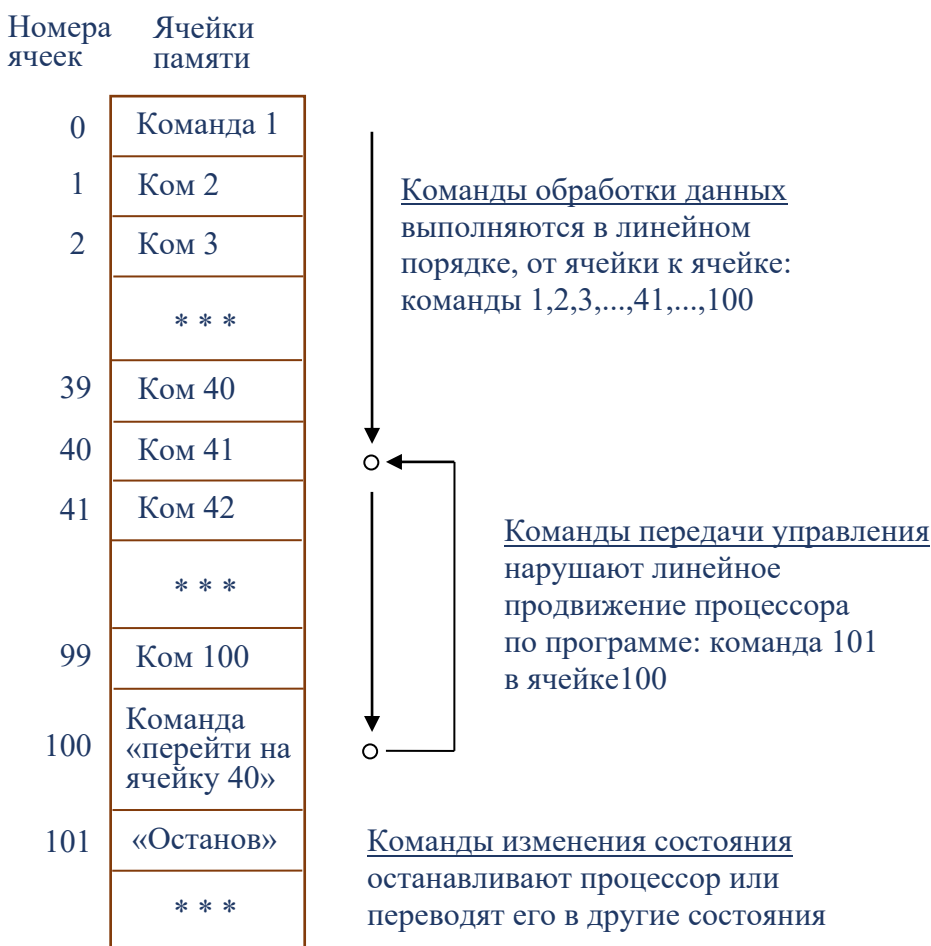


Рисунок 2.1 – Три категории команд процессора

Образование циклов в программе является главным средством автоматизации вычислений и достижения производительного эффекта компьютера. С помощью команд передачи управления осуществляются циклы в программе, ветвления (переходы по условию), безусловные переходы, вызовы и возвраты из подпрограмм. Продвижение процессора по программе с помощью команд передачи управления называется *принудительным*. Поскольку номера ячеек называют также *адресами*, то говорят, что под действием команд передачи управления процессор переходит *по заданному адресу памяти*, а сам этот адрес называется *адресом перехода* (40 в примере на рис.2.1).

Кроме команд обработки данных и передачи управления у любого процессора имеется третья категория команд, служащих для *изменения его состояния*. Одной из них является команда останова `halt` (от англ. `halt` – остановка,

прекращение; она принадлежит процессорам IA-32/AMD64) или *sleep* (сон; для МК AVR) (см. рис.2.1). По ее приказу процессор прекращает выполнение программы и переходит в состояние останова. В состоянии останова никакие команды (операции над данными) не выполняются. Процессор перестает продвигаться по программе, «замирает» в этом состоянии (в точке останова) и сохраняет содержимое всех своих регистров неизменным, каким оно было перед останом. Это состояние длится до тех пор, пока процессор не будет выведен из останова. Вывести процессор из состояния «Останов» можно повторным запуском (путем подачи сигнала сброса/пуска RESET (Сбр, Пуск), а также (для большинства процессоров) сигналом прерывания INTR (ЗПр). Под действием этих сигналов процессор снова возвращается в состояние «Счет программы» и либо заново стартует с пусковой ячейки памяти (по сигналу Сбр), либо переходит на подпрограмму обработки прерывания (по сигналу ЗПр) (гл.4, 5). («гл.» – это «глава»; здесь: главы 4 и 5.) Перезапустить (сигналом Сбр) или прервать (сигналом ЗПр) процессор можно и в других точках программы, а не только в момент останова.

Говоря коротко, процессор является автоматическим устройством, которое сразу после пуска начинает выбирать из памяти и выполнять команды программы одну за другой, по очереди, в порядке возрастания их адресов; или по адресам переходов. Процессор беспрерывно повторяет однотипные действия: он *выбирает* и *выполняет* команды программы. Он не «умеет» делать, по существу, ничего иного, кроме *выборки* и *выполнения* команд. Это – автоматическое, циклически функционирующее устройство, предназначенное для *выборки* и *выполнения* команд.

Одновременно мы установили, что любой процессор располагает **тремя категориями команд**:

1. Командами *обработки данных*: пересылки (*mov* – от англ. «move») и ввода-вывода данных (*in*, *out* – от «input», «output»), арифметических (*add*, *sub* – «subtract» и т.д.), логических (*and*, *or* и т.д.), сдвиговых (*sal* – «Shift Arithmetic operand Left», *sar* – «...Right», *rol* – «Rotate operand Left», *ror* – «...Right» и т.д.) и др. операций;
2. Командами *передачи управления*: простого перехода (*jmp* – от «jump»), перехода по условию, или ветвления (*jc* – «jump on condition c» или *brxx* – «branch on condition xx»), вызова (*call*) и возврата (*ret* – «return») из подпрограмм и др.;
3. Командами *изменения состояния* процессора, например командой *hlt*, переводящей процессор в состояние останова HALT, или *sleep* – спать. Сигналы прерывания INTR (Interruption) и сброса RESET выводят процессор из состояния HALT/SLEEP.

Здесь и далее во всех примерах демонстрируются машинные команды процессоров архитектуры IA-32/AMD64 и микроконтроллеров семейства AVR. Последние приведены все в главе 6.

Автоматическое программируемое процессирование. Мы выяснили, что процессор – это та часть компьютера, которая «знает», как складывать и вычитать числа, как проводить простые логические операции, но которая делает это только по заданию программы, хранимой в оперативной памяти ЭВМ. Отдельные команды и реализующие их операции выполняются процессором самостоятельно, с помощью собственной *аппаратуры*, без поясняющих указаний со стороны других устройств, а порядок следования этих операций устанавливается хранимой в памяти ЭВМ *программой*, процессор «умеет» выполнять отдельные команды (операции), а программа диктует ему порядок их выполнения. Благодаря подобному разделению функций между аппаратурой и программой процессор генерирует под управлением *программы* последовательность простых, *аппаратно* реализуемых операций – *процесс*, т.е. осуществляет *автоматическое программируемое процессирование*. Поэтому *процессор* – это *устройство, генерирующее процесс под управлением программы*, это – *автоматическое программно процессирующее устройство*. «Процессор можно назвать «мозгом» компьютера, его двигателем или, если хотите, сердцем этой удивительной машины» [5].

Процессирование и программирование. Процессор выполняет действия, обратные алгоритмизации и программированию. При программировании решаемая задача разбивается на простые задания, понятные машине, которые она может выполнить с помощью известных ей простых операций. При процессировании, наоборот, эти простые операции складываются процессором в целостную задачу. Если алгоритмизация и программирование – это *микрорасчленение* задачи на машинные операции, то процессирование – это, наоборот, *сочленение* элементарных машинных операций в законченное решение.

Таким образом, все многообразие функций, предоставляемых компьютером пользователю, достигается благодаря двум основополагающим факторам: способности процессора к *автоматическому программируемому процессированию* и искусству программиста описать этот процесс в терминах машинных операций путем *алгоритмизации и программирования* исходной задачи. В настоящей книге мы познакомимся с элементами *организации* автоматического программируемого процессирования и подробно изучим основу для написания программ – *архитектуру* процессоров и ЭВМ на примере однокристальных ЭВМ – микроконтроллеров AVR.

Архитектура и организация компьютера.

В буквальном, прямом смысле под *архитектурой компьютера, процессора* (*архитектурой набора команд*) понимается абстрактное представление процессора и машины в терминах основных – программно-доступных функциональных модулей и узлов, языка ЭВМ, структуры данных. Архитектура процессора это воплощенная в аппаратуре и встроенных программах основа программируемого процессирования, включающая те аспекты структуры ЭВМ, которые являются видимыми для пользователя. [16] К ним относятся:

- набор компонентов компьютера, доступных пользователю. Это регистры самого процессора, ячейки основной памяти и порты (регистры) ввода-вывода периферийных устройств;
- типы, форматы и длина размещаемых в этих регистрах и ячейках данных (форматом данного называют его двоичное представление с описанием назначения каждого бита);
- способы (режимы) адресации регистров, ячеек и портов;
- наконец, сама система команд;
- а также система прерываний.

Коротко, *архитектура* – это все то, что машина предоставляет программисту, работающему на уровне машинных команд, все, что нужно знать о машине системному программисту, что видит в машине программист, когда он работает на языке машинных команд. [16]

«Поскольку сейчас ни один нормальный человек не пишет программ на машинном языке, можно переделать это определение. Программа уровня архитектуры процессора — это то, что выдает компилятор (в данный момент мы игнорируем вызовы операционной системы и символический язык ассемблера). Чтобы произвести программу уровня команд, составитель компилятора должен знать, какая модель памяти используется в машине, какие регистры, типы данных и команды имеются в наличии и т. д. Вся эта информация в совокупности и определяет архитектуру процессора.» [23]

Таким образом, архитектура – это, прежде всего, язык команд, вместе с тем, что за ним стоит, что в нем отражено. Это «видимая» часть внутренней конструкции процессора, обеспечивающая его совместимость с определенным набором команд. В отличие от организации, архитектура не определяет особенностей реализации аппаратной части ЭВМ, времени выполнения команд, степени параллелизма при выполнении программ. Программно не доступные, не отраженные в языке команд объекты: магистрали, буферные регистры, скрытые механизмы убыстрения доступа к памяти, схемы исправления ошибок, средства распараллеливания операций и т.п. к архитектуре не относятся – они относятся к организации.

Организация процессора – это то, как процессор устроен, как реализованы выполняемые им функции. Это система логико-арифметических, схемотехнических и системотехнических мероприятий, направленных на рациональное построение процессора и придание ему нужных пользовательских свойств, включая заданную архитектуру. [16]

Говоря коротко, термин «архитектура» используется для описания возможностей, предоставляемых ЭВМ, а термин «организация» определяет, как эти возможности реализованы, архитектура отвечает на вопрос «что ЭВМ может?», а организация – «как она это делает?».

В широком смысле архитектура компьютера отражает не только программно-доступные объекты и язык ЭВМ, но и главные аспекты его аппаратной организации. [16] Она представляет построение компьютера также с точки зрения системного инженера – разработчика компьютерного «железа», а не только

системного программиста (иногда, чтобы окончательно не запутаться, используется термин «микроархитектура»). В этом случае к архитектуре относят ключевые компоненты аппаратуры компьютера, отвечающие за принципиальные вопросы его функционирования. Именно: *архитектурой компьютера, микроархитектурой* называют концептуальную структуру компьютера, отражающую общий принцип его аппаратного устройства и организации работы, характеристики основных узлов и блоков, структуру управляющих и информационных связей между ними, способы реализации ими заданных целей и параметров. Архитектура в широком смысле включает такие высокоуровневые, общесистемные компоненты разработки компьютера и процессора, как

- структурная схема компьютера с описанием ее состава по функциональным органам и связей между ними;
- структура, «внутренняя конструкция» процессора: состав, назначение и связи между блоками, идеология обработки процессором потока команд, механизмы распараллеливания операций, кэширование памяти и др.;
- система памяти, пространство адресуемых ячеек памяти, их размер, организацию чтения/записи в ячейки;
- система портов (регистров) ввода/вывода, пространство адресов и размер портов, организация ввода/вывода;
- организация и разрядность интерфейсов ЭВМ: собственных, «первичных», внешних шин и сигналов процессора; системной шины или шин; внешних (периферийных) шин (шин ввода-вывода); контроллеров ввода-вывода;
- способы представления и форматы данных ЭВМ;
- наряду с указанными компонентами к архитектуре, понимаемой в широком смысле, относят и систему команд, способы адресации данных, систему прерываний и т.п., т.е. те аспекты построения компьютера, которые составляют архитектуру набора команд – построение компьютера с точки зрения системного программиста.

Все эти сведения или многие из них бывают нужны системному инженеру для обслуживания и разработки компьютерных систем на основе использования серийной элементной базы покупных, промышленно выпускаемых микрочипов процессора и его окружения.

Коротко: термин «архитектура компьютера» не имеет в настоящее время однозначного толкования, и его можно понимать как,

во-1-х, «абстрактное представление процессора в виде набора его команд и отраженной в них «видимой» части внутренней конструкции компьютера» (архитектура в узком смысле, архитектура с точки зрения системного программиста),

во-2-х, «комплекс высокоуровневых аппаратных компонентов компьютера, отвечающих за принципиальные вопросы его функционирования» (архитектура в широком смысле, архитектура с точки зрения системного инженера).

Во всех случаях *архитектура* даёт общее описание структуры и функций компьютера на уровне, достаточном для понимания его принципов работы и

системы команд, не включающее деталей технического и физического устройства компьютера.

Слово «архитектура» в понятии «архитектура компьютера» сходно по значению с его привычным употреблением в понятии «архитектура зданий». Архитектурой зданий, сооружений называют их облик, внешний вид. Подобно этому и архитектура компьютера означает функциональный облик компьютера с точки зрения системного программиста или системного инженера. Понятие архитектуры компьютера закономерно появилось ввиду сложности современных вычислительных машин как понятие, охватывающее комплекс общих вопросов построения компьютера, существенных в первую очередь для пользователя, интересующегося главным образом возможностями машины, а не частностями ее технического исполнения.

Состав процессора. Создано и выпускается множество модификаций микропроцессоров и микроконтроллеров, выполненных по полупроводниковой технологии БИС и отличающихся совокупностью аппаратных средств и программного обеспечения. При рассмотрении общих принципов построения и работы микропроцессоров / процессорных ядер микроконтроллеров целесообразно не касаться особенностей конкретного изделия, а использовать понятие гипотетического (иллюстративного) прибора, имеющего минимальный состав оборудования и простой набор команд.

Ниже мы опишем состав процессора (ЦП) неймановской модели. Процессорное ядро микроконтроллеров во многом схоже с описанным. Однако в отличие от неймановского процессора оно построено согласно гарвардской концепции и имеет не одну системную магистраль, а две (или более) магистралей – данных и команд (п.1.2). Эти магистрали являются внешними по отношению к процессорному ядру, но размещены внутри микрочипа МК. Процессорное ядро МК будет рассмотрено в главах 3, 4, однако принципиально, за исключением числа системных магистралей, оно ничем не отличается от неймановской модели.

Любой процессор содержит две главные функциональные части, или устройства (рис.2.2):

- 1) Управляющее устройство (УУ) и
- 2) Арифметико-логическое, или операционное, устройство (АЛУ, ОУ).

К ним добавляется, кроме того,

- 3) Устройство сопряжения процессора с системной магистралью компьютера – интерфейсное устройство (ИУ).

Все эти устройства соединены между собой посредством

- 4) Внутренней магистрали (или магистралей) процессора.

Для обмена информацией с внешними устройствами – основной памятью и периферией служит

- 5) Внешняя магистраль процессора, представленная набором выводов его микросхемы, подлежащих соединению с системной магистралью ЭВМ.

Структуру отдельных частей типичного микропроцессора иллюстрирует рис.2.3. **Внутренние устройства** процессора включают в себя:

- 1) УУ – *управляющее устройство*:

БУ и С – *блок управления и синхронизации*, вырабатывающий сигналы для синхронной работы отдельных устройств,

ДшК – *дешифратор команд*,

набор управляющих регистров:

СчК (PC, IP) – счетчик команд, указывающий на следующую команду для выборки из памяти (Program Counter – программный счетчик, Instruction Pointer – указатель инструкций; например, EIP/RIP процессоров архитектуры IA-32/AMD64 или PC микроконтроллера AVR),

РгК – регистр (или регистры) команд, размещающий в себе код текущей команды,

некоторые другие регистры (например, указатель кодового сегмента CS – Code Segment процессоров IA-32/AMD64),

другие блоки;

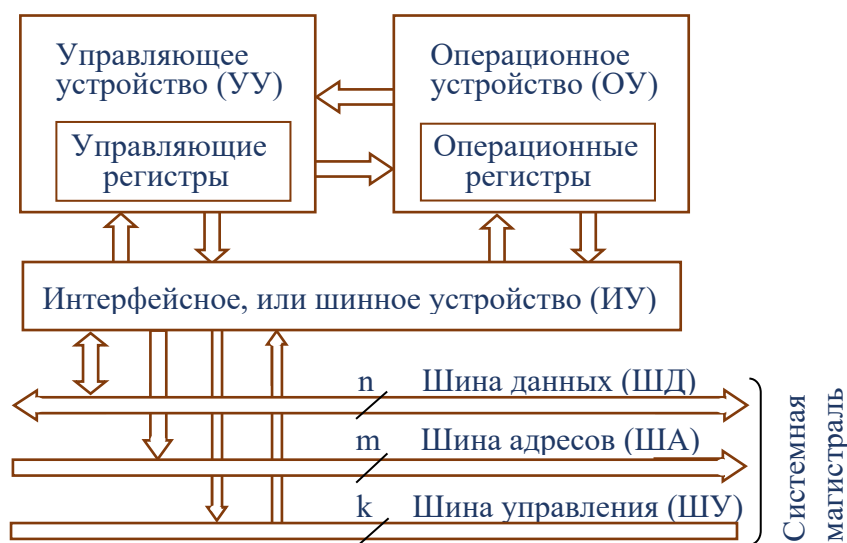


Рисунок 2.2 – Обобщенная структура процессора.
n, m, k – разрядности шин данных, адресов, управления

2) ОУ – операционное устройство:

АЛУ, или АЛБ – арифметико-логическое устройство, или блок, выполняющий арифметические и логические операции над данными, и

набор операционных регистров:

общих, таких как:

РгАк – регистр-аккумулятор (EAX/RAX – Accumulator для IA-32/AMD64),

РОН – регистры общего назначения (EBX/RBX, ECX/RCX, ... для IA-32/AMD64 или R0...R31 для МК AVR);

специальных:

УС (SP) – указатель стека (Stack Pointer; ESP/RSP для IA-32/AMD64, SP для AVR),

РгСП (Flags) – регистр состояния процессора, или регистр флагов,

другие, например сегментные регистры CS, SS, DS, ... процессора IA-32/AMD64;

а также *вспомогательных, буферных регистров данных (БРГД)*, предназначенных для временного хранения операндов и скрытых от программиста;

3) ИУ – *интерфейсное устройство* (устройство шинного интерфейса), образованное *буферными регистрами (БРГ)*, усилителями-формирователями шин данных, адресов и управления и др. компонентами;

4) *внутреннюю магистраль или магистрали.*

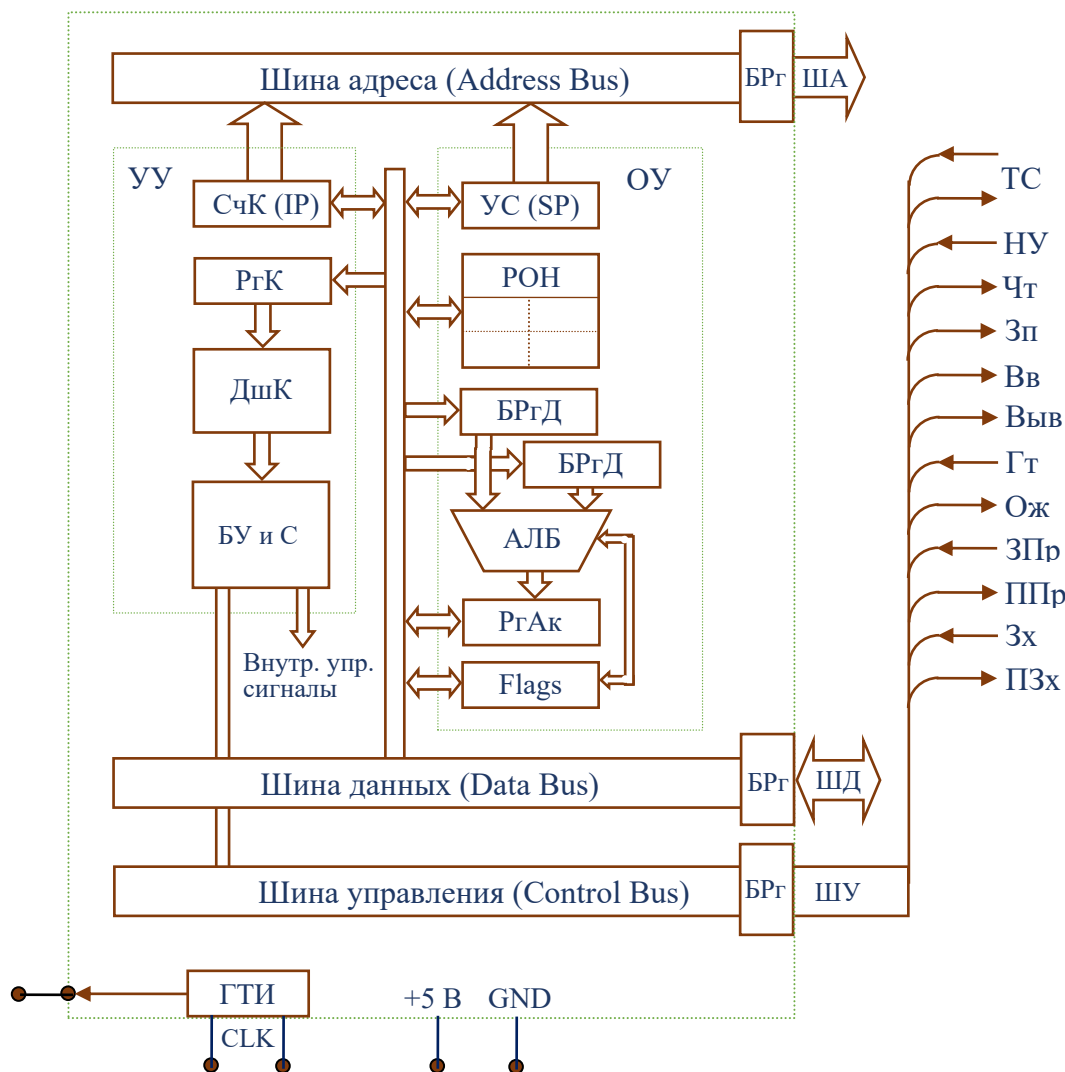


Рисунок 2.3 – Типовая структура процессора

Внешние сигналы процессора. По внешней шине данных (ШД) процессор считывает коды команд из памяти, а также выдает и принимает коды данных из памяти и периферийных устройств, а по внешней шине адресов (ША) назначает ячейки памяти или регистры ПУ, в которых эти команды и данные хранятся (см. рис.2.2, 2.3). Кроме того, процессор формирует и реагирует на различные управляющие сигналы, передаваемые по шине управления (ШУ):

Чт (чтение) и Зп (запись) – выходные сигналы процессора, обозначающие вид его обращения к памяти – прием информации из памяти (чтение) и выдача на память (запись);

Вв (ввод) и Выв (вывод) – выходные сигналы, обозначающие вид обращения процессора к устройствам ввода-вывода (периферийным устройствам) – прием информации из УВВ (ввод) и выдача на УВВ (вывод);

ЗПр (запрос прерывания) и ППр (подтверждение прерывания) – входной и выходной сигналы для организации прерываний;

Зх (захват) и ПЗх (подтверждение захвата) – входной и выходной сигналы для организации работы многопроцессорных систем,

Гт (готовность) и Ож (ожидание) – входной и выходной сигналы для синхронизации обмена данными процессора с медленно действующими внешними устройствами;

ТС – тактовые сигналы: входные от кварцевого генератора и выходные на внешние устройства, сообщающие устройствам, например, о начале цикла обмена данными по шине – шинного цикла;

НУ – сигнал начальной установки (сброса RESET).

Внешние сигналы подробно разбираются в приложениях Б и В, а также в литературе [11, 12, 18, 19]).

Функции отдельных устройств процессора. *Управляющее устройство* выбирает команды из памяти, расшифровывает их, узнавая, что требует от него очередная команда, и посылает всем другим устройствам процессора и компьютера сигналы, включающие их на выполнение затребованных операций. Далее эти устройства выполняют заданные операции уже самостоятельно, без участия УУ.

УУ следит также за правильной очередностью выполнения команд и извлекает их из памяти в порядке этой очередности.

Для временного хранения команд и связанных с ними данных устройство управления может использовать набор регистров. Один из этих регистров называется *регистром команд* и служит для размещения кода выполняемой команды, выбираемой из основной памяти. Команда, находящаяся в данный момент в регистре команд, сообщает устройству управления следующую информацию:

- а) какую *операцию* он должен выполнить;
- б) какие *данные (операнды)* потребуются (и потребуются ли) для этого, где они находятся и как их вызвать;
- в) где искать *следующую команду*.

Команды обработки данных и изменения состояния процессора поддерживают естественный порядок выполнения команд – вслед за ними по умолчанию, согласно их коду операции, выполняется всегда *команда из очередной, соседней* ячейки программной памяти. Команды передачи управления вызывают принудительное продвижение по программе – они заставляют процессор перейти на *произвольную* команду программы, где бы она ни находилась, для чего сообщают устройству управления ее адрес – *адрес перехода*.

После выполнения команды другой регистр – *счетчик команд*, имеющийся в составе УУ, выдает адрес. Этот адрес обеспечивает выборку из основной памяти (и исполнение) следующей команды. Другие названия счетчика команд: *программный счетчик* (Program Counter, PC), *указатель инструкций* (Instruction Pointer, IP). Порядок функционирования счетчика команд описывается в п.5.3.

Таким образом, основными функциями УУ являются:

- а) слежение за правильной очередностью выполнения команд и извлечение (выборка) их из памяти в порядке этой очередности;
- б) дешифрация выбранных команд и посылка всем другим устройствам процессора и ЭВМ сигналов, включающих их на выполнение заданных командами операций.

Для осуществления указанных функций УУ содержит такие блоки:

- программный счетчик;
- регистр команд;
- дешифратор команд;
- блок управления и синхронизации.

Операционное (арифметико-логическое) устройство можно рассматривать как устройство, помогающее устройству управления выполнять команды [19]. Оно может выполнять *арифметические* операции (например, сложение и вычитание). Для этой цели в его составе содержится двоичный сумматор. Кроме того, ОУ способно выполнять *логические* операции, т.е. обрабатывать коды чисел с помощью логических операций И, ИЛИ, сложение по модулю два и др. В распоряжении ОУ имеется также вспомогательная память в виде накапливающего регистра, принимающего результат операции, который принято называть *аккумулятором*, и других рабочих регистров.

Когда команда, находящаяся в данный момент в регистре команд устройства управления, требует выполнения некоторой арифметической или логической операции над данными, эти данные извлекаются из основной памяти и посылаются в ОУ. Затем ОУ по указанию устройства управления производит требуемую операцию. После завершения операции результат (новые данные) запоминаются в аккумуляторе. Таким образом, ОУ выполняет все арифметические и логические операции над данными, поступающими из памяти компьютера или из периферийных устройств.

Данные (числа, буквы и другие символы внешнего алфавита ЭВМ, коды пикселей на экране монитора, прочие данные в двоичном представлении), над которыми производится операция, называются *операндами*. Для работы с ними ОУ имеет в своем составе несколько рабочих регистров и функциональных схем. *Регистры* ОУ предназначены для хранения операндов или их адресов в процессе выполнения операций, и в них заносятся результаты операций. Поэтому их принято называть *операционными* или *операндными* (или, как мы уже говорили, *рабочими*) регистрами. Те операционные регистры, которые предназначены для хранения собственно операндов, называют *регистрами данных*, а для хранения адресов операндов в памяти или в периферийных устройствах – *регистрами адреса*. У большинства процессоров рабочие регистры универ-

сальны: способны хранить и операнды, и их адреса. Тогда их называют *регистрами общего назначения (РОН), общими регистрами*.

Например, микропроцессор IA-32 имеет восемь рабочих 32-битных регистров общего назначения, а AMD64 – шестнадцать 64-битных РОН, способных хранить как операнды и результаты операций, так и их адреса. Наоборот, рабочие регистры микропроцессора MC68000 компании Motorola делятся на две группы: восемь 32-битных регистров данных и семь 32-битных регистров адреса. Рабочие (общие) регистры процессора МК AVR представлены 32-мя единицами 8-битных РОН – R0, R1, ..., R31.

Регистром (как, впрочем, и ячейкой памяти) называется электронное устройство, предназначенное для запоминания двоичного числа. Регистр представляет собой упорядоченный набор («линейку») триггеров (рис.2.4), обычно D.



Рисунок 2.4 – Регистр – это линейка триггеров

Триггер (англ. trigger – курок; защелка) – электронное устройство, обладающее двумя устойчивыми состояниями и способное скачком переходить из одного состояния в другое под воздействием внешнего импульса. Каждому состоянию триггера соответствует определённый (высокий или низкий) уровень выходного электрического напряжения:

- 1) триггер установлен в единичное состояние – уровень «1» (обычно – напряжение от 2 до 5 В).
- 2) триггер сброшен в нуль – уровень «0» на выходе (напряжение от 0 до 1 В).

Установившееся состояние сохраняется сколь угодно долго и может быть изменено внешним импульсом или отключением напряжения питания. Таким образом, триггер является элементарным компонентом памяти, способным хранить наименьшую единицу информации – один двоичный разряд: «0» или «1».

Триггер строится обычно на двух транзисторных логических элементах И-НЕ или ИЛИ-НЕ, охватывающих друг друга положительной обратной связью и за счет этого взаимно поддерживающих свое состояние.

Функциональные схемы («вентили», от нем. Ventil – клапан) вычисляют различные функции от двоичных операндов, представленных двужначными сигналами – логическими нулями («0») и единицами («1») [23]. Логический «0» представляется обычно сигналом от 0 до 1 В, а логическая «1» – сигналом от 2 до 5 В. Напряжение за пределами указанных величин недопустимо. С помощью вентиля производятся все необходимые преобразования операндов при передаче их из одного операционного регистра на другой. Одним из наиболее важных вентилях операционного устройства процессора является двоичный сумматор, осуществляющий сложение двоичных чисел. Простейшие вентили (логические элементы) – это схемы И, ИЛИ, НЕ, ПОВТОРЕНИЕ.

К числу основных, аппаратно реализуемых операций по обработке данных, выполняемых ОУ по указке машинных команд, относятся:

- а) пересылка (прием и выдача) операнда (слова или байта) на регистр. Это могут быть внутренние, межрегистровые пересылки, происходящие между регистрами самого ОУ или процессора; и внешние пересылки, осуществ-

- ляемые между регистрами ОУ, с одной стороны, и ячейками памяти или регистрами периферийных устройств, с другой;
- б) сдвиг операнда на заданное число разрядов (битов) влево или вправо;
- в) прибавление к слову или вычитание из него единицы (в более общем случае, некоторой *const*). Эта операция называется инкрементом и декрементом слова соответственно;
- г) сравнение операндов по принципу «больше-меньше-равно»;
- д) поразрядные логические операции «или» – *or* (дизъюнкция), «и» – *and* (конъюнкция) «исключающее или» – *xor* (*exclusive or*, сложение по модулю 2), «не» – *not* (побитное инвертирование);
- е) суммирование и вычитание двух операндов, представляющих числа в одной и той же системе счисления; другие бинарные арифметические действия: умножение, деление и т.д.
- ж) преобразование кодов операндов, включая дополнение, дешифрацию и др.

Часть регистров УУ (управляющих регистров) и ОУ (операционных регистров) доступны программисту, т.е. их содержимое можно изменять при помощи машинных команд. К программно-доступным регистрам относятся *программный счетчик* (управляющий регистр), *регистры общего назначения* и *аккумулятор* (операционные регистры) и др. Содержимое программного счетчика изменяется при помощи команд передачи управления (например, *jmp*), а РОН – при помощи команд обработки данных *mov*, *add* и др. (Однако, нельзя написать: *mov PC, 1000*).

Количество программно-доступных (адресуемых, «видимых») регистров процессора сравнительно невелико. Оно варьируется от нескольких единиц (у простейших микропроцессоров) до нескольких десятков и более (у высокоорганизованных микропроцессоров и процессоров супер-ЭВМ).

Кроме ОУ и УУ в структуре процессора можно выделить третью, вспомогательную функциональную часть – устройство сопряжения (интерфейса) процессора с внешними по отношению к нему устройствами – памятью и периферийными устройствами – *интерфейсное устройство* (ИУ). ИУ осуществляет электрическое (по уровню и по мощности) и логическое (по составу и моментам выработки) согласование собственных сигналов процессора (микропроцессорной БИС) и сигналов, которые надо подавать на память и периферийные устройства, – сигналов системной магистрали (шины). Поэтому ИУ называют по-другому *шинным устройством*, *устройством шинного интерфейса* (ШИ) или *формирователем системной магистрали* (ФСМ) или *системным контроллером*, поскольку оно управляет системной шиной, формируя стандартные сигналы на системной шине ЭВМ, отвечающие нормативным требованиям. Сам же процессор (МП БИС) зачастую не вырабатывает всей номенклатуры нормативных системных сигналов ввиду ограничения на число выводов микросхемы. В связи с этим интерфейсное устройство выполняется обычно в виде отдельного, внешнего по отношению к МП БИС модуля (например, в виде так называемого «северного моста» – системного контроллера), помогающего процессору формировать нужные сигналы.

В заключение отметим, что при всей сложности внутреннего построения процессор, с точки зрения пользователя, представляет собой функционально законченный прибор с отдельными шинами адреса и данных, набором внешних управляющих сигналов и группой собственных программно-доступных регистров (рис.2.5).

Быстродействие и производительность процессора и компьютера. Термины «быстродействие» и «производительность» процессора или компьютера часто используют как синонимы, но им придают также и нормативно различающиеся значения.

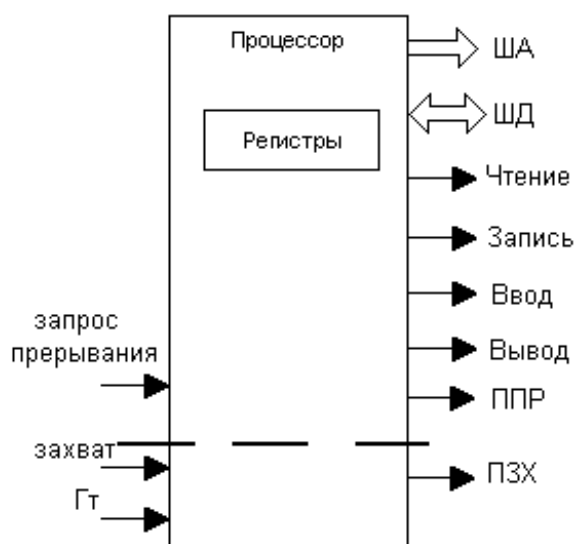


Рисунок 2.5 – Микропроцессор с точки зрения пользователя

Быстродействие процессора (оно же – *быстродействие ЭВМ*) – это число команд (или, как часто говорят, «операций»), выполняемых процессором за единицу времени (число операций в секунду, или *IPS* – от англ. instructions per second). Поскольку разные команды («операции») выполняются с разной скоростью, то говорят о *среднем быстродействии процессора (ЭВМ)* на разных классах задач, которое определяется с учетом вероятностей использования каждой операции при решении данного класса задач. В частности, для любой данной программы среднее быстродействие процессора есть просто отношение количества команд в программе к времени ее выполнения. Сейчас стало общепринятым характеризовать процессор по числу выполняемых им *MIPS* (Million of Instructions Per Second – «миллион команд в секунду»).

В последние годы мощность вычислительных машин попадает в диапазон *BIPS*, или *GIPS* (Billion, Giga – миллиард команд в секунду).

Быстродействие процессора (не задерживаемого, в идеале, медленно функционирующими внешними устройствами и системной шиной) пропорционально его тактовой частоте. *Тактовая частота* (частота ядра, Internal Clock) – это количество электрических импульсов в секунду, подаваемых на процессор внешним генератором тактовых импульсов – одним из блоков, расположенных на материнской плате. Тактовая частота выдерживается с очень высокой точностью и лежит в мега или гигагерцовом диапазоне. Процессор, работающий на тактовой частоте 800 МГц, выполняет 800 миллионов рабочих тактов в секунду. В зависимости от сложности обрабатываемой команды процессору для ее выполнения иногда необходимы десятки и более тактов. Самые простые команды (например, команды межрегистровых пересылок данных внутри процессора) выполняются обычно за один такт. Быстродействие процессора в этом случае – *предельное, пиковое быстродействие* – совпадает с тактовой частотой. Чем выше тактовая частота ядра, тем выше скорость обработки данных. Современные микропроцессоры работают на частотах от 800 МГц до 4,7 ГГц. Соответственно их предельное быстродействие составляет от сотен до тысяч Mips. Микроконтроллеры (у них все или почти все команды обрабатываются, как правило, за 1 такт) обладают быстродействием порядка десятков Mips.

Развитые процессоры построены так, что даже сложные, многотактные, команды практически не снижают скорости их работы, и при обработке потока команд эта скорость зачастую доходит до пиковых значений – сотен и тысяч миллионов операций в секунду ($\sim 10^3$ Mips). Это обеспечивается за счет реализации *общего принципа достижения высокого*

быстродействия, который звучит так: *компьютер (процессор) должен запускать как можно больше команд в секунду* [23]. С этой целью в структуру процессора вводятся средства распараллеливания (одновременного выполнения нескольких) операций, такие как буфер опережающей выборки команд, один или несколько конвейеров дешифрации и выполнения команд (п.2.2). Также увеличивается ширина (разрядность) шины данных, что позволяет считывать из памяти по несколько команд сразу за один такт, помещая их в буфер опережающей выборки команд, и обмениваться данными с внешним окружением тоже большими порциями. Кроме того, используется кэш-память – быстродействующая память, расположенная между процессором и основной памятью (п.2.2). Благодаря всему этому процессор способен запускать достаточно много команд в секунду. После считывания из памяти поток команд поступает на конвейер внутри процессора, и если конвейер справляется с этим потоком раньше, чем его надо обновлять, то собственно время обработки команд (в смысле времени на выполнение заданных ими операций) как бы выпадает из общего времени обработки потока. В результате быстродействие начинает определяться лишь скоростью считывания команд из памяти, например, из кэша, и ограничивается только частотой, а также шириной системной шины или шины кэша.

Частота системной шины (System Clock). Даже если процессор располагает развитыми механизмами распараллеливания операций и кэш-памятью, а также достаточно широкой шиной данных, его быстродействие может существенно ограничиваться частотой системной шины, которая всегда меньше частоты работы ядра процессора. Системная шина служит для связи процессора с остальными устройствами, и медленные пересылки информации по ней задерживают работу ядра. Частота системной шины современных микропроцессоров от 66 МГц до 1-1.5 ГГц. Если умножить ее на среднее число команд, приходящихся на одну выборку из памяти за один такт шины, то это, как правило, и будет некое суммарное *быстродействие* системы с развитым процессором. Таким образом, процессор имеет две частоты: тактовую частоту ядра и частоту системной шины, и системная шина является наиболее «узким» местом компьютера.

Умножение частоты. Соотношение частот ядра и шины не является произвольным. В современных процессорах используется умножение тактовой частоты: собственная частота процессора (частота ядра) определяется путем умножения частоты системной шины на определенный коэффициент – коэффициент умножения, который может быть равен 1.5, 2.0, 2.5, 3.0, 4.0 и доходит до 20 и более.

Учитывая зависимость работы процессора (и вообще любого устройства ЭВМ) от взаимодействия с другими компонентами, величину быстродействия уместно относить к отдельным устройствам: процессору, памяти, устройствам ввода-вывода и т.д., и говорить о собственном, внутреннем быстродействии этих устройств. В этом случае понятие быстродействия не распространяется на всю ЭВМ, а вместо быстродействия используется понятие производительности ЭВМ.

Производительность ЭВМ, в отличие от быстродействия, определяется в общем случае как объем работ, осуществляемых ЭВМ в единицу времени. Производительность можно измерять, например, средним числом некоторых стандартных задач, решаемых за единицу времени, или средним числом отрабатываемых команд в секунду на этих классах задач (т.е. вышеупомянутым средним быстродействием ЭВМ). Однако чаще производительность измеряется временем выполнения ЭВМ некоторых образцовых программ – типичных программ, которые могли бы выполняться на данной установке. Производительность зависит от быстродействия процессора и устройств ввода-вывода, частоты и ширины системной шины, порядка прохождения задач в ЭВМ и т.п. Она увеличивается при распараллеливании обработки потока команд: совмещении операций чтения-записи с обработкой, использовании мультипрограммного и мультипроцессорного режимов работы. Таким образом, единственной подходящей и надежной единицей измерения производительности является время выполнения реальных программ, и в первую очередь, стандартных тестовых программ.

Генератор тактовых импульсов (ГТИ) (также *синтезатор частот, тактовый генератор*, сленг. «клокер» – от англ. «clocker», clock generator, clock synthesizer) – устройство, формирующее основные тактовые частоты, используемые на материнской плате и в процессоре. Источником опорной, чрезвычайно стабильной частоты для него служит кварцевый резонатор («кварц»). Типичные частоты подобных устройств составляют от единиц до десятков МГц.

Тактовый генератор в современных персональных компьютерах и микроконтроллерах состоит из опорного кварцевого генератора и микросхемы генерации. Опорная частота кварцевого генератора подается на вход микросхемы, которая на своих выходах формирует из опорного сигнала тактовые сигналы для различных компонентов системы: для процессора, для системной шины, для чипсета, для оперативной памяти и т. д. (рис.2.6). Микросхема генерации умножает и делит частоту опорных импульсов, то есть выступает в качестве делителя и умножителя частоты. Современные микросхемы генерации являются программируемыми. Внутри отдельных компонентов (включая процессор) частоты от ГТИ могут еще дополнительно умножаться или делиться местными схемами синхронизации.

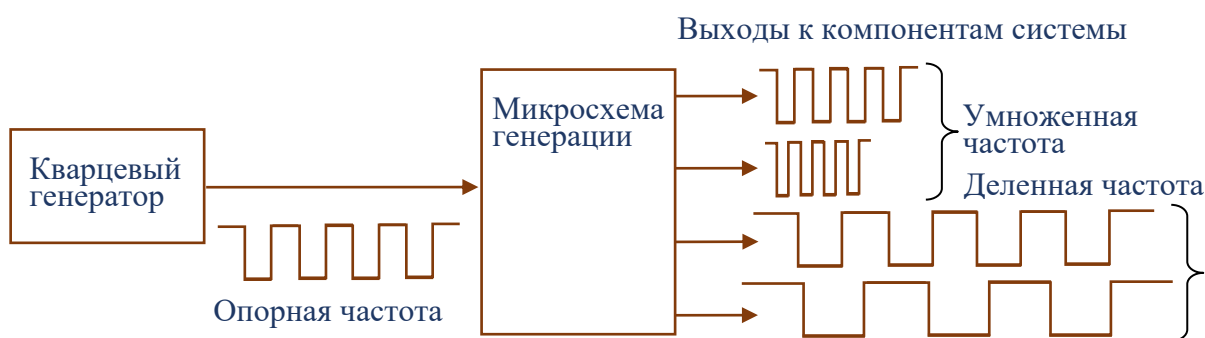


Рисунок 2.6 – Выработка тактовых сигналов в компьютере

2.2. Память

Назначение памяти и ее виды. Чтобы иметь возможность выполнять обработку данных по заданной программе, ЭВМ должна иметь память, в которой хранятся команды и данные [19]. Память вычислительной системы состоит из следующих частей: регистров процессора и периферийных устройств, основной памяти (часть которой может быть продублирована кэш-памятью) и внешней памяти (рис.2.7).

Многоуровневая организация памяти. Указанные части памяти образуют многоуровневую иерархическую структуру. По мере продвижения по уровням сверху вниз (на рис.2.7 – слева направо) – от регистров процессора до внешней памяти меняются три параметра: время доступа, объем и стоимость единицы хранимой информации. Время доступа и стоимость уменьшаются, а объем увеличивается.

На самом верху иерархии находятся **регистры процессора**. Доступ к ним осуществляется быстрее всего, поскольку они работают на тактовых частотах процессора, время доступа составляет десятые доли наносекунд. Поэтому **регистровую память процессора** называют сверхоперативной памятью ЭВМ (СОП). Если в понятие СОП включать, как это обычно делается, только про-

граммно-доступные регистры, то объем таких регистров – десятки машинных слов. Регистры процессора используются для промежуточного, кратковременного хранения данных в процессе их обработки. Это энергозависимый вид памяти, т.е. при отключении питания от процессора информация в регистрах стирается.

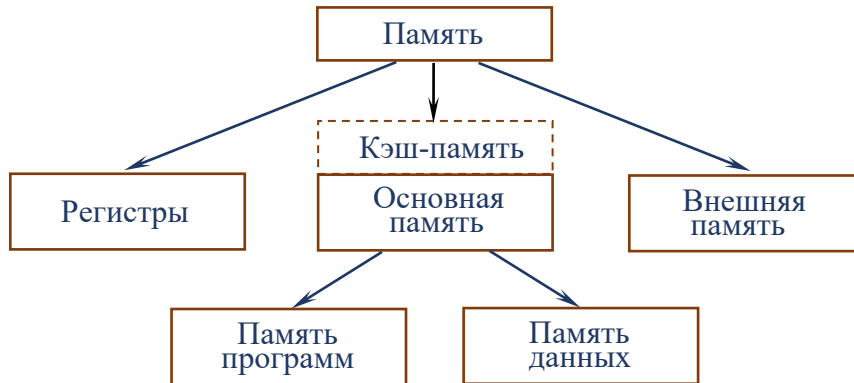


Рисунок 2.7 – Виды памяти ЭВМ

Регистры имеются не только в процессоре, но и в периферийных устройствах. Это – регистры, или порты, ввода-вывода (п.2.3). Свойства **периферийных регистров** сопоставимы со свойствами основной памяти. **Регистры**, процессорные и периферийные, представляют собой вспомогательную память, распределенную по всей системе.

Дальше в иерархической структуре идет **кэш-память** *. Так называют память небольшого объема с высокой скоростью работы, «прозрачную» для программиста (невидимую программой). Кэш-память играет роль буфера между основной памятью и процессором. Основная идея кэш-памяти проста: в ней находятся слова, которые чаще всего используются. Если процессору нужно какое-нибудь слово: команда или операнд, сначала он обращается к кэш-памяти. Только в том случае, если слова там нет, он обращается к основной памяти. Если значительная часть слов находится в кэше, среднее время доступа значительно сокращается. Непосредственное время доступа составляет единицы наносекунд. Объем кэша составляет от 32 Кбайт до десятков мегабайт (см. «Кратные числовые единицы» ниже). Отличается относительно высокой стоимостью и высоким энергопотреблением. Энергозависимый вид памяти.

* Микроконтроллеры не оснащаются кэш-памятью

Кратные числовые единицы. Для измерения больших величин в информатике приняты следующие кратные единицы:

1 К (K) = 2^{10} = 1024 – двоичная тысяча (кило);

1 М (M) = 2^{20} – двоичный миллион (мега);

1 Г (G) = 2^{30} – двоичный миллиард (гига);

1 Т (T) = 2^{40} – двоичный триллион (тера).

1 П (P) = 2^{50} – двоичный квадриллион (пета).

1 Э (E) = 2^{60} – двоичный квинтиллион (экса).

1 З (Z) = 2^{70} – двоичный секстиллион (зетта).

1 И (Y) = 2^{80} – двоичный септиллион (йотта).

Основная память (ОП) предназначена для хранения текущих выполняющихся программ и обрабатываемых в них данных – операндов. Та часть ОП, в которой хранятся команды, называются *памятью программ*, а та часть, в которой хранятся данные, – *памятью данных*. ОП занимает среднее положение в иерархии памяти. Время доступа к ней составляет десятки наносекунд, а объем в настоящее время лежит в диапазоне от 1 Гбайт до сотен гигабайт. ОП более дешевая, чем кэш-память, так как реализуется с помощью динамических микросхем. Обладает высокой надежностью хранения данных и программ, расчетная вероятность ошибки – одна в десять лет.

Основная память реализуется на микросхемах ОЗУ и ПЗУ. ОЗУ – это оперативное запоминающее устройство, или RAM – Random Access Memory – память с произвольным доступом, как для чтения, так и для записи. ОЗУ – энергозависимый вид памяти: после выключения питания хранимая в нем информация искажается либо полностью разрушается. ПЗУ – постоянное запоминающие устройства, или ROM – Read Only Memory – память только для чтения. ПЗУ – энергонезависимый вид памяти, содержащаяся в нем информация сохраняется постоянно как при включенном, так и выключенном напряжении питания. Служит, например, для хранения BIOS (Basic Input/Output System – базовая система ввода-вывода). В последнее время широко используются другие виды энергонезависимой перезаписываемой памяти, например, флэш-память, или флэш-карты. (Имя «флэш» было придумано разработчиком этого вида памяти потому, что процесс стирания содержимого памяти напомнило ему фотовспышку, от англ. flash – вспышка, мгновение).

Внутренняя память. Это понятие используют иногда для обозначения памяти внутренней части ЭВМ. К ней относят основную память, память процессора – набор его адресуемых рабочих регистров (сверхоперативная память ЭВМ – СОП), а также память контроллеров периферийных устройств – набор регистров (портов) ввода-вывода. Совокупность внутренней и внешней памяти ЭВМ образует всю память ЭВМ.

Внешняя память – магнитные диски (жесткие, или так называемые «винчестеры», и гибкие) и оптические (компакт-) диски, флэш-карты, накопители на магнитных лентах и проволоках и др. Внешняя память служит информационным складом, местом для долгосрочного хранения данных. Она самая дешевая и самая емкая – объем современных жестких дисков доходит до десятков терабайт. Также велик объем накопителей на лентах и проволоках. У флэш-карт, оптических и гибких магнитных дисков объем меньше. Однако внешняя память – это самый медленный уровень памяти. Доступ к дискам происходит, по крайней мере, в 10 раз медленнее для жестких, твердотельных дисков и в сотни раз медленнее для съемных накопителей (оптических дисков и флэш-карт), чем доступ к основной памяти. К тому же съемные накопители нужно еще взять и поместить в соответствующее устройство. Быстродействие накопителей на гибких дисках, магнитных лентах и проволоках еще меньше.

Состав основной памяти. Основная память состоит из большого числа идентичных технических устройств – блоков одинакового размера, служащих для запоминания двоичных чисел в некоторой физической форме (рис.2.8). Эти запоминающие блоки называются *ячейками памяти*.

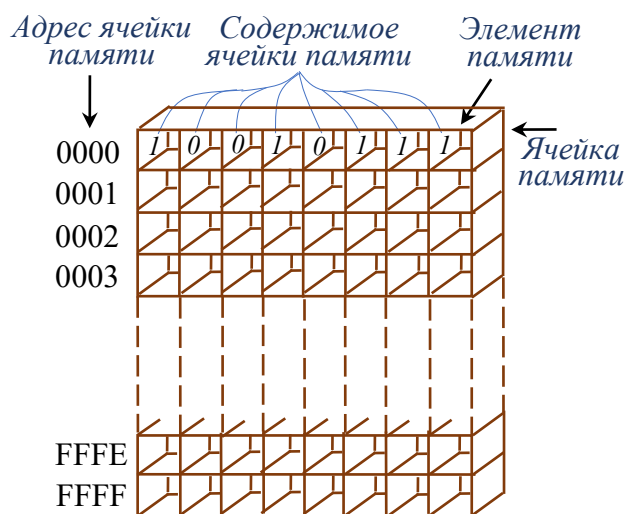


Рисунок 2.8 – Ячейки памяти.
Адреса ячеек записаны 16-ричными числами

Ячейка памяти состоит из *элементов памяти*.

В большинстве ЭВМ ячейка состоит из 8 элементов. В каждом элементе может храниться наименьшая единица информации – одно двоичное одnorазрядное число 0 или 1. Таким образом, в каждой ячейке содержится 8 двоичных разрядов. Поскольку эти 8 бит составляют 1 байт, говорят, что длина ячейки ЭВМ равна 1 байту.

Упорядоченная совокупность нулей и единиц, находящихся в восьми элементах памяти, представляет собой *содержимое* ячейки памяти.

Физически в качестве элемента основной памяти (простейшего запоминающего устройства) применяется триггер или микроскопический электроемкостной элемент («конденсатор»). Высокий уровень напряжения на выходе триггера (обычно от 2 до 5 В) трактуется как логическая «1», низкий (от 0 до 1 В) – как логический «0». Аналогично заряженность «конденсатора» означает «1», разряженность – «0». Упорядоченная совокупность («линейка») подобных триггеров или «конденсаторов» и образует базовый запоминающий блок – ячейку памяти.

Ячейки памяти нумеруются от нуля до некоторого положительного числа. Номер ячейки называют её *адресом*. Каждая ячейка памяти имеет свой, уникальный *адрес (номер)*, являющийся целым числом без знака. Если память содержит N ячеек, их адреса пробегают последовательный ряд значений от 0 для первой ячейки до максимального значения N-1 для последней ячейки: 0, 1, 2, ..., N-1.

Адреса ячеек известны программисту. Используя их, программист с помощью машинных команд может ссылаться на нужные ему ячейки (назначать,

адресовать их) и заказывать процессору определенные действия над их содержанием. По этим командам процессор заносит в ячейки исходные операнды или извлекает ранее занесенные, выполняет над ними заданные операции и запоминает результаты в надлежащих ячейках; или извлекает новые команды для исполнения, когда был задан их адрес с помощью команды перехода. В отсутствие переходов процессор самостоятельно, без дополнительных указаний со стороны команд, считывает из ячеек памяти очередные команды в порядке возрастания их адресов.

Таким образом, адреса ячеек определяют местоположение их в памяти. Они задаются программистом в программе чтобы обеспечить нахождение процессором конкретной ячейки в процессе обмена информацией с памятью.

Максимальное количество адресов составляет 2^m , где m – разрядность адреса (его размер в битах), и адреса ячеек пробегают последовательный ряд значений $0, 1, 2, \dots, 2^m - 1$. Этим числом ограничивается размер основной памяти. Поскольку бóльшие значения адресов недопустимы, большее число ячеек памяти использовать нельзя. Например, в Intel-совместимых компьютерах $m = 32$ или $m = 64$, и $2^m = 4\,294\,967\,296$ (4 Гбайт информации, 4 двоичных миллиарда байт) или $2^m = 16$ экса байт (16 двоичных квинтиллионов байт). Характеристики памяти микроконтроллеров AVR вы найдете в конце этого параграфа, а ее полное описание в главе 5.

Для передачи данных на и из памяти процессор использует системную шину. При отработке каждой машинной команды адрес целевой ячейки (в которой хранится команда или операнд для нее) выставляется процессором на шину адреса и далее поступает на схемы, управляющие выборкой ячеек – *селекторы*, или *дешифраторы адреса*, которые разрешают подключение этой ячейки к шине данных. Другие сигналы, тоже вырабатываемые процессором, выставляются на шину управления. Они заводятся на *схемы управления* памятью и запускают затребованные операции: чтение команды или операнда, запись, ввод или вывод операнда. Таким образом, процессор имеет доступ ко всем ячейкам памяти.

Структура функционально законченной системы основной памяти представлена на рис.2.9 и ОП включает в себя:

- а) набор запоминающих блоков – *ячеек памяти*, каждая из которых имеет свой уникальный *номер*, или *адрес*, известный программисту;
- б) схемы, управляющие выборкой (селекцией) ячеек – *селекторы*, или *дешифраторы адреса*;
- в) управляющие схемы, активизирующие выбранную ячейку на выполнение нужной операции: чтение или запись (согласно сигналам Чт или Зп от процессора) – *схемы управления чтением/записью*.

Процессор (а в режиме прямого доступа к памяти – периферийное устройство *) выполняет с основной памятью две операции:

- 1) чтение и
- 2) запись.

* *Прямой доступ периферийного устройства к памяти ЭВМ* представляет собой способ передачи данных между ПУ (например, жестким диском; или видеокартой) и основной памятью ЭВМ, минуя центральный процессор. В настоящем пособии этот режим работы ЭВМ не рассматривается.

При чтении процессор извлекает одну порцию данных (байт или слово) из выбранной им, адресованной ячейки. При записи – заносит данное в ячейку. В промежутках между этими операциями память хранит записанную в нее информацию. Операции чтения и записи в память называются *обращениями к памяти*.

Процессоры МК AVR обращаются к памяти, например, по командам `lds` и `sts`:

`lds R0, 0x60 ; R0 ← m(0x60)` – чтение 0x60-й ячейки памяти данных (ОЗУ) в регистр процессора R0 (п.5.2);

`sts 0x61, R0 ; m(0x61) ← R0` – запись в 0x61-ю ячейку ОЗУ из регистра R0.

Память – пассивное устройство, она никогда не выступает инициатором проведения каких-либо операций в компьютере.

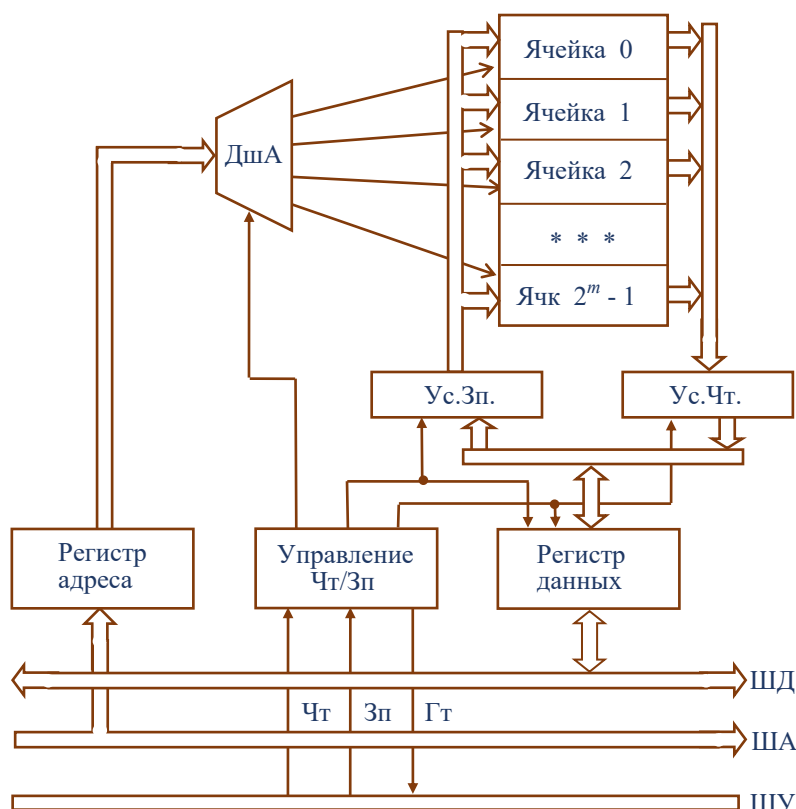


Рисунок 2.9 – Структура основной памяти.

Чт (MEMR – Memory Read) и Зп (MEMW – Memory Write) – сигналы управления чтением и записью; Гт (READY) – сигнал готовности запоминающего устройства; ДшА – дешифратор адреса; Ус.Чт и Ус.Зп – усилители чтения и записи

Байтная организация памяти. Мы уже указали, что ячейки основной памяти имеют порядковые *номера*, по которым их можно различать. Эти номе-

ра называются *адресами ячеек*. Задавая их, можно обращаться к ячейкам посредством машинных команд. Собственные, уникальные, номера присваиваются однобайтным ячейкам – ячейкам размером в 1 байт (8 бит). Номера однобайтных ячеек принимают последовательные целочисленные значения от 0 до $2^m - 1$, где m – разрядность номера (адреса). В этом смысле говорят, что *память компьютера организована по байтам*. У процессоров IA-32/AMD64 $m = 32 / 64$. Соответственно количество однобайтных ячеек памяти высокоорганизованных процессоров архитектуры AMD64 может достигать $2^m = 16$ двоячных квинтиллионов (16 Э), а их номера (адреса) изменяются от 0000000000000000h до FFFFFFFFh (здесь h – суффикс обозначения шестнадцатеричного числа; от англ. hexadecimal).

Соседние однобайтные ячейки могут объединяться в более крупные, многобайтные ячейки, которые, как и однобайтные ячейки, могут адресоваться и обрабатываться процессором по одной машинной команде. Размер таких ячеек в байтах составляет обычно целую степень двойки: 2 байта, 4 байта, 8 байт. Многобайтные ячейки отдельно не нумеруются, а идентифицируются по уже назначенным адресам однобайтных ячеек. *Адресом многобайтной ячейки* считается младший из адресов составляющих ее однобайтных ячеек. Например, если число 1234h (длиной в 16 бит) хранится в двух однобайтных ячейках памяти с адресами 75 и 76, то адресом всего числа, всей двухбайтной ячейки считается 75. При этом многобайтное число хранится в памяти по правилу: *младшие байты по младшим адресам*: «младшее по младшему адресу». В нашем примере байт 34h хранится по адресу 75, а байт 12h – по адресу 76: |34h| |12h| (рис.2.10).

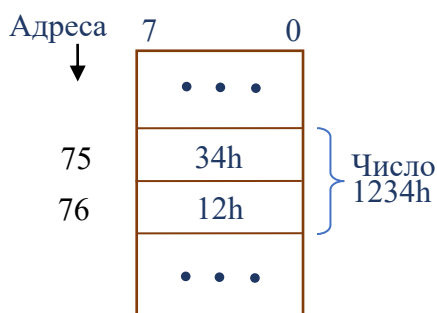


Рисунок 2.10 – Размещение длинного числа по байтам памяти: «младшее по младшему адресу»

Современные запоминающие устройства обычно имеют размер физической ячейки (максимальной по размеру ячейки, содержимое которой передается по шине данных ЗУ в один прием) 8 бит, 16 бит, 32 бита, 64 бита. В ранних ЭВМ использовались и более экзотические размеры, например 39 (БЭСМ-1) или 48 (БЭСМ-6). Длина ячеек микросхем ЗУ может оказаться короче машинного слова процессора. Тогда для приема/выдачи слова они соединяются параллельно по две, четыре или более микросхем, каждая из которых подключается к своей группе линий шины данных.

Аналогично ячейкам памяти организованы и регистры ввода-вывода периферийных устройств. Они тоже имеют стандартную длину 1 байт (8 бит) и собственные оригинальные номера (адреса), пробегающие последовательные

значения в некотором диапазоне целых чисел, отдельном или совместном с ячейками. Соседние 1-байтные регистры могут объединяться в более крупные, многобайтные, регистры по тем же правилам, что и ячейки памяти.

Что касается собственных регистров процессора, то их длина составляет чаще всего 1, 2, 4, 8 байт, и они различаются своими мнемоническими именами (на языках программирования) или укороченными номерами (встроенными в машинные коды команд). Например, общие 32-битные регистры процессоров IA-32 именуются как EAX, EBX, ..., ESP, а общие 8-битные регистры микроконтроллеров AVR – как R0, R1, ..., R31.

Упомянутое выше единообразное построение основной памяти из одинаковых, однобайтных, ячеек с уникальными адресами характерно для компьютеров принстонского (неймановского) типа. В них и программа, и данные располагаются в совместной, общей, памяти без жесткой привязки к каким-либо областям адресов. Поэтому ячейки такой памяти должны иметь одинаковые размеры и сквозную нумерацию, чтобы по ним можно было бы свободно перемещать информацию различного назначения.

В машинах же гарвардской архитектуры программа и данные располагаются отдельно, в двух разных видах памяти с независимой нумерацией ячеек для каждого вида. Это побуждает зачастую по тем или иным соображениям реализовывать указанные виды памяти на ячейках разных размеров. В микроконтроллерах AVR, например, ячейки памяти программ имеют размеры 16 бит (условно 2 байта) и нумеруются подряд от 0 до $2^{m1} - 1$, где $m1 = 22$ – разрядность адреса памяти программ (используется 17 разрядов), а ячейки памяти данных (статического ОЗУ и регистров) имеют размеры 8 бит (1 байт) и нумеруются независимо от ячеек программной памяти (и от ячеек EEPROM) от 0 до $2^{m2} - 1$, где $m2 = 16$ – разрядность адреса памяти данных. При этом ячейки статического ОЗУ в зависимости от модели микроконтроллера начинаются с адресов \$60/\$100/\$200. Соответственно объем памяти программ достигает до $2^{17} = 128$ кило слов (16-битных), а памяти данных (при подключении внешнего статического ОЗУ) – до $2^{m2} = 64$ кило байт.

2.3. Контроллеры ввода-вывода

Назначение контроллеров ввода-вывода. *Контроллеры ввода-вывода (КВВ), или контроллеры периферийных устройств,* – это электронные устройства, предназначенные для подключения к системной магистрали компьютера разных по принципу действия, интерфейсу и конструктивному исполнению периферийных устройств. КВВ образуют каналы для передачи данных между центральной частью компьютера и ПУ и далее – внешней средой. Они также управляют периферийными устройствами во время такой передачи (отсюда их название «контроллеры», т.е. устройства управления).

Ввод и вывод. При указанной передаче данные могут вводиться или выводиться из компьютера. Подчеркнем, что операции ввода и вывода понимаются всегда по отношению к внутренней (центральной) части компьютера. Иначе,

вводом называется передача данных из внешней части (периферии) компьютера (включая передачу из внешней среды) во внутреннюю часть – в процессор или память, а *выводом* называется обратная передача – из внутренней части (из процессора или основной памяти) во внешнюю часть и далее, возможно, во внешнюю среду (рис.2.11). Коротко: *ввод* – это передача данных из периферии в ядро компьютера, а *вывод* – обратная передача: из ядра в периферию.

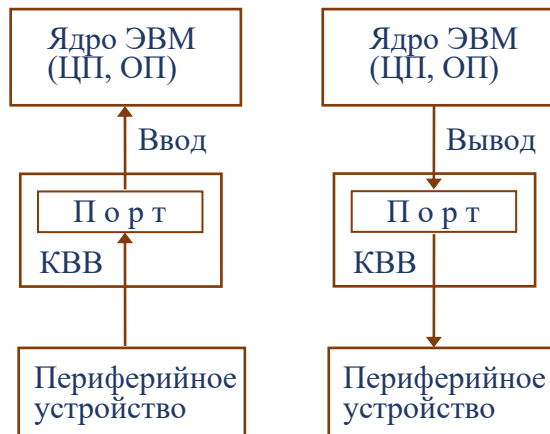


Рисунок 2.11 – Потоки данных при вводе и выводе.

ЦП – центральный процессор;
ОП – основная память;
КВВ – контроллер в/в;
Порт – регистр ввода-вывода, принадлежащий КВВ

Функции контроллеров ввода-вывода. Образуя каналы ввода-вывода – каналы передачи данных между ядром и периферией, контроллеры ввода-вывода в процессе передачи выполняют следующие функции (рис.2.12):

- 1) осуществляют временное хранение (буферирование) передаваемых данных на своих внутренних регистрах («портах») в/в с целью синхронизации передачи. Порты в/в совместно с адресными и управляющими схемами составляют первый функциональный блок контроллера, называемый *модулем портов в/в* (МВВ на рис.2.12 и, далее, на рис.2.13, 2.14);
- 2) преобразуют данные в нужные форматы – из форматов внешних сигналов периферийного устройства в форматы сигналов системной магистрали, понятные процессору, – при вводе и обратно – при выводе. Системно отформатированные данные и помещаются в порты (регистры) в/в для временного хранения. Совокупность компонентов, осуществляющих подобную функцию, образует второй функциональный блок КВВ – *преобразователь форматов* (ПФ на рис.2.12);
- 3) управляют функционированием периферийных устройств по запросам от процессора, посылая на ПУ нужные сигналы, активизирующие устройства на выполнение целевых операций. Этим занимается третий функциональный блок КВВ, носящий имя *блока непосредственного управления периферийным устройством* или *локального контроллера* (УПУ на рис.2.12).

Порты (регистры) ввода-вывода. Процессор общается с периферийными устройствами через адресуемые регистры контроллеров ввода-вывода, названные выше *регистрами* или *портами ввода-вывода* (см. рис.2.11). Это означает, что для процессора любое периферийное устройство представляется в виде одного или нескольких портов в/в.

Порты образуют отдельные каналы для передачи данных между ядром и периферией компьютера. Они служат для временного хранения вводимых и выводимых данных, являются буфером для передаваемой информации, представленной двоичными кодами стандартной длины, совместимой с длиной регистров процессора. Порты имеют свои оригинальные *адреса* на системной шине, по которым процессор находит их в процессе работы. Каждый КВВ снабжен от нескольких единиц до нескольких десятков портов в/в.

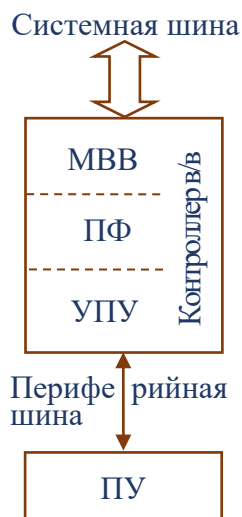


Рисунок 2.12 – Состав контроллера в/в.

МВВ – модуль портов (регистров) в/в;
ПФ – преобразователь форматов;
УПУ – блок управления ПУ
(локальный контроллер)
ПУ – периферийное устройство

Наличие буферных портов (регистров) в/в обусловлено необходимостью синхронизации работы процессора и периферийного устройства. Одновременно эти органы компьютера не всегда оказываются готовыми к обмену данными друг с другом. Так, процессор не способен в произвольный момент времени принять данные от ПУ (при вводе), поскольку в этот момент он может быть занят выполнением других операций, например, записью числа в память по системной магистрали. При этом и сама системная магистраль, по которой ПУ собирается ввести данные в процессор, тоже будет занята в этот момент. Аналогично и ПУ не в состоянии в любой момент времени принять данные от процессора (при выводе) из-за того, например, что оно еще не успело обработать предыдущее данные. Все это говорит о том, что при осуществлении ввода-вывода требуется *синхронизация* работы процессора и периферийного устройства – выполнение операций обмена данными между ними должно быть обеспечено в строго определенные для каждого из них моменты времени. Эта задача и решается с помощью портов в/в.

Порты функционируют подобно почтовым ящикам для передачи корреспонденции. Они кратковременно хранят вводимые и выводимые данные. При *вводе* периферийное устройство помещает код данного (слово) в порт ввода независимо от того, готов процессор принять это слово в данный момент времени или нет и свободна ли системная шина от других передач данных. Одновременно ПУ сообщает процессору (тем или иным способом), что слово помещено в порт. Для этого ПУ может оставить сообщение в другом порте ввода, установив, например, определенный его бит в 1; или просигнализировать процессору

напрямую специальным сигналом – запросом прерывания. В свою очередь, процессор принимает отправленное ему данное не сразу в тот самый момент, когда оно поступило в порт, а спустя какое-то время, после того как он и системная шина освободятся от выполнения текущих операций.

Аналогично при *выводе* процессор просто заносит порцию данных в порт вывода, не заботясь о том, чем в данный момент занято периферийное устройство и готово ли оно принять данное. Далее процессор продолжает текущую работу. Со своей стороны ПУ принимает выведенное данное не одновременно с его поступлением в порт, а лишь при очередном просмотре порта. Просмотры могут повторяться с некоторой заранее обусловленной частотой или по мере поступления очередных данных.

Предварительно в этих передачах до занесения информации в порт передающее устройство обычно удостоверяется, что порт свободен, т.е. предыдущее данное из него выбрано. Сообщение о выборке данного может быть оставлено принимающим устройством в другом порте путем перевода его специального бита в 0.

Таким образом, порты ввода и вывода позволяют согласовать во времени работу процессора и периферийных устройств несмотря на то, что эти органы компьютера обособлены друг от друга, функционируют независимо друг от друга, каждый сам по себе, с разными скоростями и возбуждаются разными событиями.

Виды портов (регистров) ввода-вывода. В процессе ввода-вывода передается информация двух видов: *управляющие данные* (управляющие слова) и *собственно данные* (данные-сообщения, данные-операнды). Соответственно порты, служащие для передачи управляющих слов, называются *портами управления* (или *управления/состояния*), а для передачи данных – *портами данных*.

Управляющие слова от процессора называются также *командными словами*. Они инициируют действия, не содержащие непосредственно передачу данных, например, запуск устройства, запрещение прерываний, перемещение каретки принтера и т.п.

Управляющие слова от периферийного устройства называются *статусными словами, словами состояния*. Они содержат информацию об определенных признаках, например, о готовности устройства к обмену данными, о наличии ошибок при обмене и т.п. Состояние обычно представляется в декодированной форме – один бит для каждого признака.

Наиболее общая программная модель модуля в/в содержит *четыре порта*: два для ввода данных (входной порт данных) и параметров состояния (порт состояния), и два для вывода данных (выходной порт данных) и параметров управления (порт управления) (рис.2.13). В зависимости от особенностей устройства общая модель конкретизируется, например, отдельные порты состояния и управления объединяются в один порт, в устройстве ввода (вывода) имеется только порт вводимых (выводимых) данных, для ввода и вывода операндов используется только один двунаправленный порт.

Модули портов (регистров) ввода-вывода. Мы уже упоминали, что все порты (регистры) ввода-вывода доступны программисту, адресуемы, «видимы». Каждому порту (как и ячейке памяти) присвоен свой индивидуальный номер (адрес), известный программисту. Поэтому программист может вывести через порт любую информацию в периферийное устройство и извлечь из любого порта имеющееся в нем сообщение от периферийного устройства.

Для ввода и вывода в системе команд ряда процессоров служат специальные машинные команды. МК AVR имеет для этой цели команды ввода `in` (input) и вывода `out` (output). Например:

`in R2, port ; R2 ← p(port)` – ввести в регистр процессора R2 8-битное число из порта ввода с номером port;
`out port, R2 ; p(port) ← R2` – вывести в порт вывода по адресу port содержимое R2.

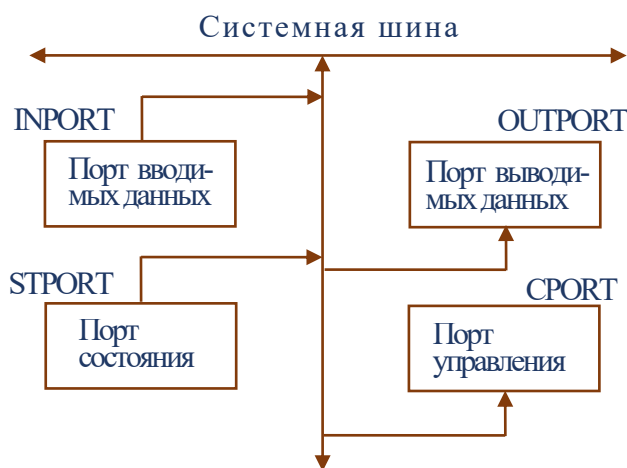


Рисунок 2.13 – Общая программная модель периферийного устройства

Номер порта AVR Atmega16 6-битный. Поэтому максимальное количество портов ввода и вывода составляет $2^6 = 64$ (по 64 тех и других).

Выборка (селекция) портов в/в осуществляется при помощи электронных схем – *селекторов (дешифраторов) адреса*. Для управления записью в порты (выводом) и чтением из портов (вводом) используются электронные *управляющие схемы*.

Все перечисленные устройства составляют отдельный функциональный блок контроллера ввода-вывода, отвечающий за буферирование потоков вводимых и выводимых данных. Выше мы назвали его *модулем портов в/в* или просто *модулем в/в* (MBV). Таким образом, полный состав MBV такой (рис.2.14):

- а) *порты ввода-вывода* – адресуемые регистры периферийного устройства;
 - б) *дешифратор (селектор) их адресов*;
 - в) *управляющие схемы*, открывающие порты по вводу и выводу,
- и процессор выполняет с модулями (портами) ввода-вывода две операции:
- 1) ввод данных (из порта в процессор);
 - 2) вывод данных (из процессора в порт).

Эти операции называются *обращениями к портам* периферийных устройств.

По своему составу модули портов в/в напоминают память ЭВМ, но, в отличие от памяти, хранящей текущие программы и обрабатываемые в них данные, МВВ осуществляют буферирование (временное хранение) потоков передаваемой информации в операциях ввода и вывода.

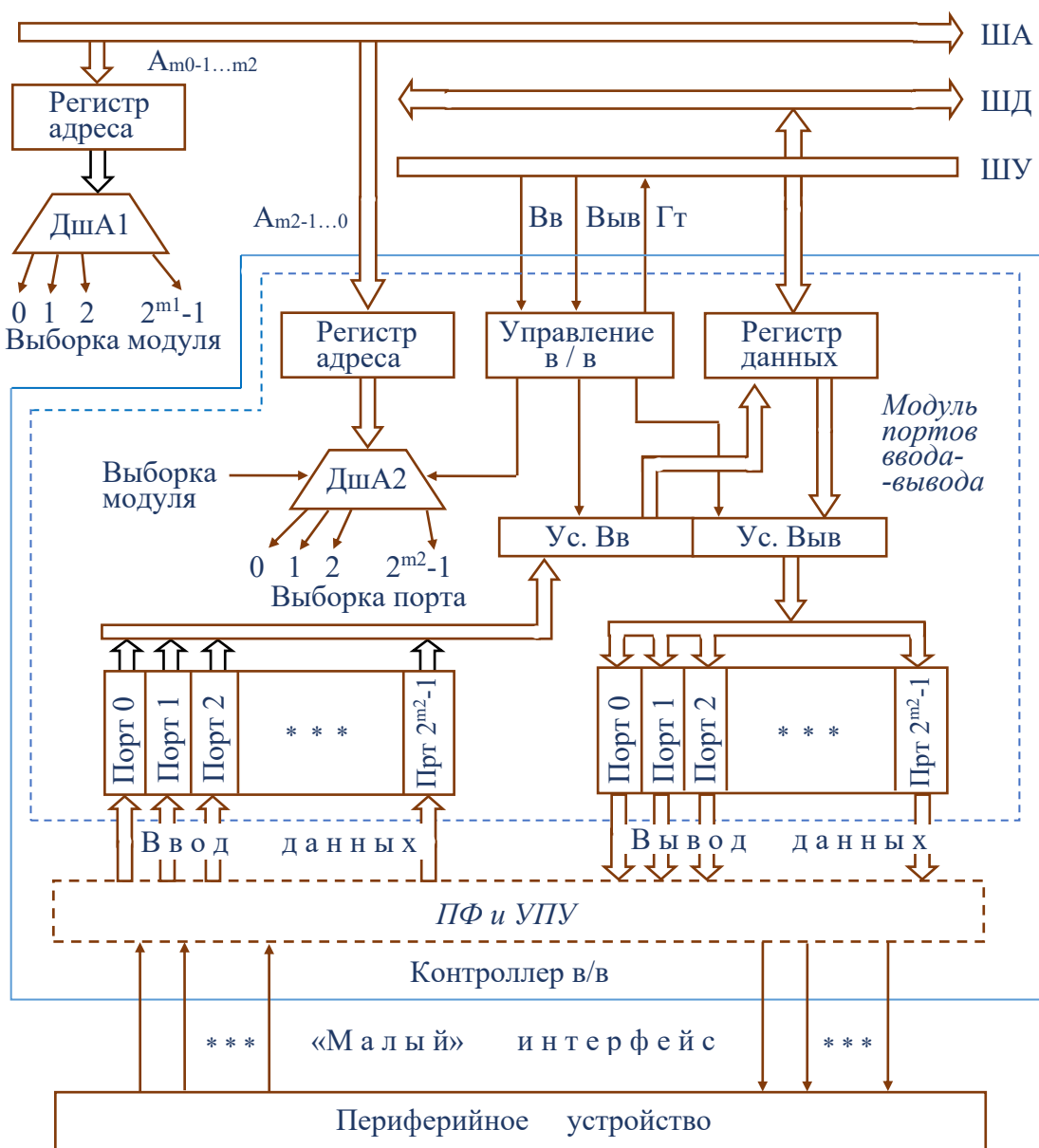


Рисунок 2.14 – Структура модуля портов ввода-вывода.

Вв (IOR) и Выв (IOW) – сигналы управления вводом-выводом; Гт – сигнал готовности периферийного устройства; Ус.Вв – усилитель-формирователь ввода; Ус.Выв – усилитель-формирователь вывода; ПФ – преобразователь форматов данных; УПУ – блок управления периферийным устройством; $A_{m0-1...0}$ – адрес порта, состоящий из номера модуля в/в $A_{m0-1...m2}$ (старшие разряды адреса) и номера порта внутри модуля $A_{m2-1...0}$ (младшие разряды адреса); $m0=m1+m2$ – разрядность адреса порта

В более широкой трактовке **порт** (буквально с англ. port – «отверстие», «проход», «ворота») – это «вход» в компьютер – физическое соединение, служащее для подключения к системной шине периферийных устройств. Через порт осуществляется ввод данных в компьютер из ПУ и вывод данных из компьютера в ПУ.

Технически такой «порт» представляет собой контроллер ввода-вывода (КВВ) на системной шине компьютера, снабженный разъемом для ручного подключения к нему извне тех или иных периферийных устройств. Разъем вынесен на какой-либо элемент аппаратного обеспечения компьютера и в него включается кабель, шнур или вилка периферийного устройства. Иногда разъем отдельно тоже именуют портом – входом в компьютер.

В микроконтроллерах (МК) порт выступает наружу своими выводами (ножками) на корпусе микросхемы, к которым можно подсоединять внешние устройства (ВУ).

Каждому порту, как и любому КВВ, выделяется группа адресов для входящих в его состав регистров в/в. Благодаря этому компьютер/МК может напрямую, с помощью машинных команд принимать данные от ПУ/ВУ и отправлять данные на ПУ/ВУ через порт по системной шине – собственно данные, служебную информацию для программирования параметров порта и текущее состояние порта.

Таким образом, порт – это функционально законченное устройство (канал ввода-вывода), состоящее из контроллера в/в и соединительного разъема на аппаратуре компьютера или группы ножек на корпусе микросхемы МК. Это своего рода *вакантный контроллер ввода-вывода*, контроллер, к которому снаружи компьютера/микроконтроллера изначально не подключено никакое внешнее устройство, а последнее можно подключать по мере необходимости. Такого рода порт называется также *интерфейсным, соединительным, коммуникационным портом*.

Название «порт», применительно к компьютеру, заимствовано из электронной техники, где портом ввода-вывода называют техническое обеспечение для обмена данными между контроллером (или процессором) и подключаемым устройством. Так же и в компьютере, порты принимают и передают информацию с устройства, расположенного вне системного блока.

Различают несколько типов портов: внутренний (таймерный), клавиатурный, коммуникационный, игровой (джойстик). Через таймерный порт можно программировать частоту следования меток времени, используемых в электронных часах компьютера. Сигналы порта подсчитываются операционной системой, для хранения накопленного значения используется область памяти с определенным адресом. Клавиатурный порт обеспечивает ввод кодов нажатых клавиш. Данные от порта накапливаются и обрабатываются в специальном кольцевом буфере клавиатуры в определенной области памяти.

Коммуникационные порты обеспечивают подключение таких внешних устройств как мышь, принтер, сканер, внешний модем и некоторых других. Эти порты подразделяются на последовательные (например, СОМ, классическое наименование которого RS232, – для подключения монитора, клавиатуры, мыши) и параллельные (LPT – для подключения принтеров, сканеров и плоттеров). Широкое распространение получил последовательный порт USB. Он обеспечивает высокоскоростное подключение к компьютеру сразу нескольких типов периферийных устройств: съемных флэш-карт, принтера, сканера, цифровых камер и т.п.

Коммуникационные порты используются также для межкомпьютерной связи в режиме Link.

2.4. Системная магистраль

Системная (внутренняя, внутримашинная) магистраль (или магистрали) предназначена для электрического соединения функциональных органов ядра компьютера (процессора, памяти и контроллеров ввода-вывода) между собой и передачи информации между ними.

По системной магистрали процессор считывает команды из памяти, вводит данные из периферийных устройств, считывает операнды из памяти и выдает результаты счета на память и ПУ. По ней же он назначает устройство, с

которым он будет производить обмен информацией в текущем машинном цикле, выставляя в магистраль адрес устройства – ячейки памяти или порта ввода-вывода. По системной магистрали пересылаются также управляющие сигналы и, кроме того, данные в режиме прямого доступа к памяти.

Физически системная магистраль представляет собой набор проводников, или линий, которые проложены на плате или в гибком шлейфном кабеле параллельно друг другу. По каждому проводнику передается один двоичный разряд – бит информации. Высокий уровень электрического потенциала в проводнике соответствует логической единице, а низкий – нулю.

Проводники магистрали разделяются на **три группы (подмагистрали, шины)**:

- шину данных (по которой передаются данные – команды и операнды);
- шину адресов (в нее процессор выставляет адреса ячеек и портов в/в, где данные хранятся);
- шину управления (для передачи сигналов, определяющих тип операции – чтение, запись, ввод, вывод, подтверждение прерывания и пр.).

Отметим, что магистрали (системные и периферийные; последние описываются в п.2.5) нередко называют шинами, а шины (данных, адресов, управления) – магистралями.

Простые компьютеры имеют одну, общую для всех устройств системную магистраль, к которой подключаются все устройства центральной части компьютера. Развитые ЭВМ располагают в своей структуре несколькими (двумя-тремя) системными магистралями. Например, местной (локальной) – высокоскоростной, но «короткой» (для подключения небольшого числа быстродействующих устройств памяти и ввода-вывода) и общесистемной – низкоскоростной, но «длинной» (для большого числа устройств). Эти магистрали отделяются друг от друга буферными схемами – так называемыми мостами (bridge). Мосты регулируют прохождение данных из одной магистрали в другую. В супер ЭВМ используется большое число различных внутримашинных магистралей, передача информации по которым происходит одновременно и независимо друг от друга.

2.5. Периферийные устройства. Интерфейс

Периферийные устройства. Отдаленные от центра территории нередко называют периферией. Периферия бывает не только в обыденной жизни, но и в компьютерах. По отношению к компьютерам под термином «периферия» подразумевают периферийные устройства. *Периферийные устройства, компьютерная периферия* – это различные дополнительные устройства, которые пусть недалеко, но все-таки отдалены от «сердца» компьютера – центрального процессора и оперативной памяти. Они подключаются к компьютеру в дополнение к этим основным устройствам. Можно сказать и иначе: периферия персонального компьютера – это все то, что можно отключить от компьютера, и он сможет продолжить свою работу, пусть с некоторыми ограничениями (кроме от-

дельных обслуживающих периферийных устройств, таких как кварцевый генератор и, возможно, таймер).

Периферийные устройства («периферию») (внешнюю по отношению к процессору и основной памяти часть компьютера) удобно подразделять на:

1. Устройства ввода-вывода данных для связи компьютера с пользователем (УСП): а) устройства ввода – клавиатура, переключатели, мышь, сканер, джойстик, микрофон, видео- и веб-камера, световое перо, графический планшет и т.п.; б) устройства вывода – индикаторы, дисплеи, мониторы, принтеры, плоттеры (графопостроители), звуковые громкоговорители («динамики»), наушники и т.п.; в) интерактивные устройства – терминалы, ЖК-планшеты с сенсорным вводом и др.
2. Устройства периферийной (внешней, вторичной, массовой) памяти для хранения больших объемов информации: жесткие диски, флэш-карты, съемные диски (магнитные дискеты, оптические и магнитооптические диски) и их дисководы, стримеры (накопители на магнитной ленте) и др.
3. Устройства ввода-вывода для связи компьютера с различного рода внешними техническими объектами (УСО) – источниками сигналов (датчиками) и приемниками сигналов (исполнительными механизмами, линиями связи и т.д.). УСО включают в свой состав аналого-цифровые (АЦП) и цифро-аналоговые (ЦАП) преобразователи, сигнализаторы, реле, мультиплексоры (коммутаторы) и др. Кроме УСО к данной группе можно отнести и те устройства, которые выполняют внутренние операции по обслуживанию компьютера, например, отсчитывают промежутки времени, задают режимы его работы и т.д. (ОУ – обслуживающие устройства). Они тоже обмениваются информацией с процессором в форме операций ввода-вывода. Обслуживающие периферийные устройства представлены таймерами, счетчиками событий, тактовыми генераторами, цепями сброса и пр.
4. Средства телекоммуникации, реализующие передачу данных на большие расстояния и образующие, как правило, телекоммуникационную сеть (сеть связи, сеть передачи данных). Телекоммуникационными сетями являются телефонные, радио, телевизионные и компьютерные сети. Для построения компьютерных сетей применяются сетевые адаптеры, модемы (модуляторы-демодуляторы), кабели, телефонные линии, мультиплексоры, демультиплексоры. В качестве узлов связи в компьютерных сетях используются специализированные сетевые устройства: концентраторы, мосты, коммутаторы, маршрутизаторы (роутеры), шлюзы.

Напомним (п.2.3), что периферийные устройства подключаются к ядру компьютера – ЦП и ОП не напрямую, а через контроллеры ввода-вывода (КВВ). КВВ содержат адресуемые регистры (порты) ввода-вывода, служащие для программно-управляемого обмена информацией между ядром и периферией. Поэтому процессор воспринимает любое периферийное устройство как один или нескольких портов в/в. Поскольку КВВ, как и основная память, непосредственно доступны процессору, мы их относим к центральной части компьютера (п.1.1).

Интерфейс ввода-вывода. С контроллерами в/в связано одно из важнейших понятий ЭВМ – понятие интерфейса ввода-вывода. Контроллеры в/в как раз и составляют физическую часть этого интерфейса (рис.2.15).

Интерфейсом ввода-вывода называется совокупность аппаратных и программных средств, служащих для сопряжения периферийных устройств (клавиатуры, мыши, монитора и т.п.) с центральной частью компьютера (процессором и памятью) и обеспечивающих обмен информацией между ними в операциях ввода-вывода. *Аппаратная часть* интерфейса включает в себя контроллеры ввода-вывода (п.1.4) и шины ввода-вывода (периферийные шины), составленные из наборов электрических проводов и передаваемых по ним сигналов. *Программная часть* интерфейса образована совокупностью программ, называемых *драйверами периферийных устройств* и предназначенных для управления обменом данными между периферийными устройствами и центральной частью компьютера. Драйверы ПУ осуществляют ввод-вывод собственно данных, управляющих слов и слов состояния ПУ, их анализ, дешифрацию и сохранение итоговой информации в памяти. Они идентифицируют тип информации (данные, управляющие символы и т.д.), преобразуют форматы данных, переводят команды от операционной системы в команды, которые понимает данное ПУ, и обратно, и т.п.

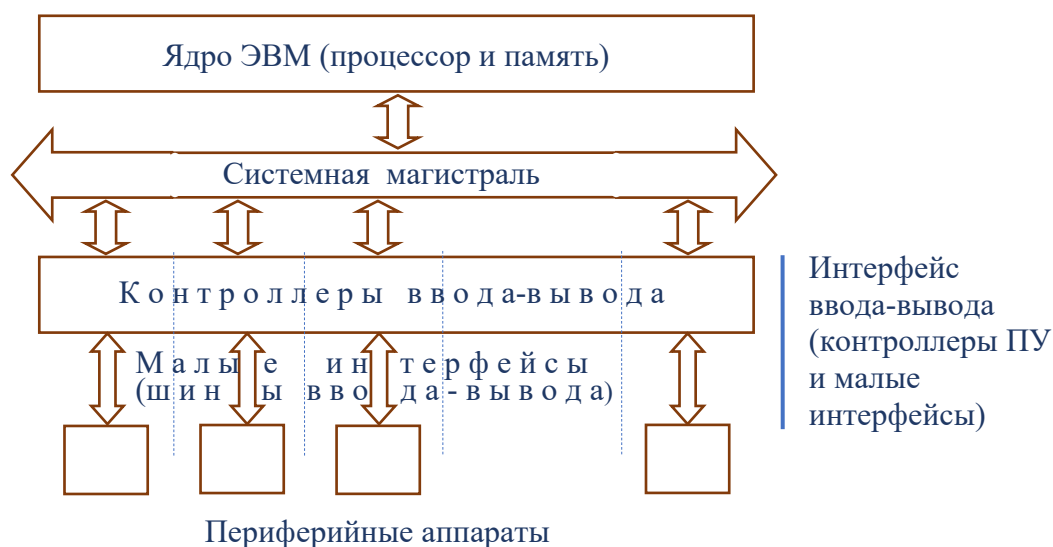


Рисунок 2.15 – Состав интерфейса ввода-вывода

Общая координация вводом-выводом осуществляется операционной системой, компонентом, который чаще всего называют *супервизором ввода-вывода*. Данная программа выполняет арбитраж и проверку на корректность запросов на ввод-вывод от прикладных задач и от программных модулей самой ОС, инициирует операции ввода-вывода (передает управление соответствующим драйверам), идентифицирует ПУ, от которых поступили запросы прерываний, и передает управление соответствующей подпрограмме обработки прерываний, посылает сообщения об ошибках, если таковые происходят в процес-

се ввода-вывода, а также сообщения о завершении операции ввода-вывода запрошившему эту операцию процессу, снимая его с состояния ожидания в/в, если процесс ожидал завершения операции.

Малые интерфейсы. Каждый тип периферийного устройства функционирует по своим особым физическим принципам, и нуждается в собственном интерфейсе ввода-вывода – контроллере (или адаптере) и шине в/в, соединяющей контроллер и ПУ. Подобный интерфейс (т.е. контроллер и шина в/в) предназначен для сопряжения ЭВМ с определенными типами периферийных аппаратов, и он называется *малым* или *внешним* или *периферийным* или *интерфейсом периферийных устройств*. Малые интерфейсы отличаются друг от друга принципами передачи данных. Они могут быть устройство зависимыми и устройство независимыми, и информация по ним может передаваться целыми словами (*параллельные* интерфейсы) или отдельными битами (*последовательные* интерфейсы). К унифицированным устройством независимым малым интерфейсам относятся, например, COM-порт (RS-232C), USB-порт (последовательные интерфейсы), LPT-порт, Centronics, IDE – встроенный интерфейс накопителя или SCSI – интерфейс малых вычислительных машин (параллельные интерфейсы) и т.п. Полный интерфейс ввода-вывода представлен совокупностью малых интерфейсов.

Необходимость в нескольких малых интерфейсах объясняется широкой номенклатурой периферийных устройств, среди которых есть как сравнительно медленно действующие (например, принтеры), так и высокоскоростные (по сравнению с процессором) устройства (накопители на магнитных дисках и лентах, видеомониторы), и которые к тому же отличаются разнообразием типов, имеют различную физическую природу и архитектуру, могут располагаться на значительных расстояниях от ЭВМ и характеризоваться многими другими особенностями. Все это делает задачу построения всего интерфейса в/в или отдельных внешних интерфейсов достаточно сложной в схемотехническом отношении.

Обязательным компонентом малого интерфейса является, как мы указали, *шина ввода-вывода (периферийная шина)* – набор линий связи, соединяющих контроллер непосредственно с периферийным аппаратом – принтером, дисководом, мышью и т.д. Подобный набор линий, а также система передаваемых по ним сигналов, отвечающих заданным, обычно унифицированным протоколам обмена, тоже называют *внешним* (или *малым*) *интерфейсом* (или *интерфейсом ПУ, периферийным интерфейсом*) (см. рис.2.15). Шины внешних интерфейсов выводятся на внешние разъемные гнезда компьютера (размещаемые преимущественно на задней панели системного блока). Через эти разъемы («порты») к компьютеру и подключаются периферийные аппараты: клавиатура, монитор, мышь, флэш-карты и т.п. при помощи соединительных шлангов или собственных разъемных штекеров.

Другие виды интерфейсов ЭВМ. Наряду с внешними интерфейсами ЭВМ говорят и о внутреннем, или внутримашинном, или машинном (системном) интерфейсе. Под ним понимают совокупность средств соединения всех

устройств центральной части ЭВМ в единую систему. Это, прежде всего, сама системная магистраль компьютера, а также дополнительные согласующие и усилительные электронные схемы. Ввиду высокой степени совместимости серийных микросхем процессора, памяти и контроллеров в/в объединение их в целостную систему не представляет сложной задачи и производится в некоторых случаях при помощи одной только системной магистрали с минимальным привлечением микросхем окружения. Поэтому системную магистраль часто тоже именуют системным интерфейсом.

Нами упомянуты два вида интерфейса ЭВМ: *внутренний* и *внешние* (рис.2.16). Кроме них можно говорить и об *интерфейсах между отдельными устройствами* ЭВМ, например, между центральным процессором и арифметическим сопроцессором, между процессором (микропроцессорной интегральной схемой) и системной магистралью, между разными периферийными устройствами и т.д. И, вообще, *интерфейсом ЭВМ* называют любой набор аппаратных, программных и конструктивных средств, предназначенных для сопряжения устройств ЭВМ между собой. В интерфейс входят линии, шины и магистрали, сигналы и электронные схемы, алгоритмы, программы и протоколы обмена, конструктивные элементы (например, разъемные соединители) и т.п. средства сопряжения.

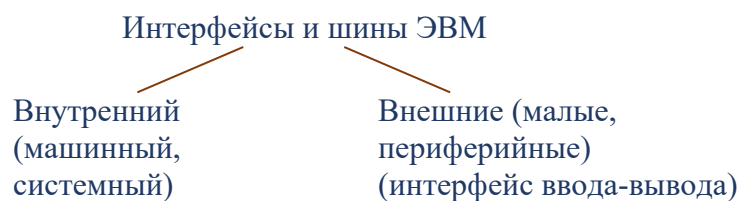


Рисунок 2.16 – Интерфейсы ЭВМ

Из сказанного можно также заключить, что смысловое содержание терминов «интерфейс», «интерфейсный модуль», «канал», «контроллер», «адаптер» довольно многообразно. Эти термины не являются ни однозначными, ни общепринятыми, и значение каждого из них зависит от текущего контекста.

Унифицированный интерфейс. Большую роль для практического создания интерфейсов ЭВМ играет их стандартизация и унификация. *Унифицированный интерфейс* – это совокупность правил, устанавливающих единые принципы взаимодействия устройств ЭВМ, а также сами физические компоненты и машинные программы, обеспечивающие взаимодействие устройств по этим правилам. Стандартизации и унификации в интерфейсе обычно подлежат: форматы передаваемой информации; сигналы управления и состояния; номенклатура и характер линий связи; порядок (алгоритмы) функционирования; передающие и приемные электронные схемы, в т.ч. аппаратурные средства соединения устройств (разъемы, связи); электрофизические параметры сигналов и их временные диаграммы; конструктивные решения; наконец, программные средства, описывающие характер сигналов, их логические соотношения и режимы

передачи. Унифицированный интерфейс может отвечать требованиям на все или только часть этих характеристик.

Выбор унифицированного интерфейса или, в общем случае, совокупности интерфейсов является одним из центральных, определяющих моментов в проектировании ЭВМ и других вычислительных систем. Говоря коротко, построить ЭВМ на серийной элементной базе – это и значит, прежде всего, спроектировать ее интерфейс. Как правило, ЭВМ строятся с использованием нескольких интерфейсов, например, интерфейса модулей центральной части системы, т.е. внутримашинного интерфейса (соответственно МП, ОЗУ, ПЗУ), интерфейсов подключения периферийных устройств (внешних интерфейсов). Выбор внутримашинного интерфейса определяется типом микропроцессора. Внешние интерфейсы (адаптеры и контроллеры ПУ и шины в/в) выбираются исходя из особенностей периферийных устройств и требований к характеристикам ввода/вывода. В случае МП ИУВС разрабатывается интерфейс подключения объекта управления (датчиков информации, исполнительных механизмов, аналого-цифровых и цифро-аналоговых преобразователей, устройств регистрации и т.п. устройств). В многопроцессорных системах необходим интерфейс между отдельными микропроцессорами.

Глава 3. Микроконтроллеры

3.1. Знакомство с микроконтроллером

Микроконтроллер – микросхема, предназначенная для управления различными устройствами – бытовыми приборами, периферийными аппаратами компьютеров, техническим оборудованием в промышленности, на транспорте и др. отраслях экономики *. Типичный микроконтроллер содержит на одном кристалле процессор, ОЗУ, ПЗУ и периферийные компоненты: устройства ввода-вывода, таймеры и др. Иными словами, *микроконтроллер – это однокристалльный компьютер, способный выполнять относительно простые задачи*. Впрочем, ранее для обозначения МК как раз и употреблялся термин «однокристалльная микро ЭВМ».

* *Контроллером* в технике регулирования считается управляющее устройство, осуществляющее регулирующие или контролирующие функции в системе. Контроллер, реализованный на одном кристалле, называется *микроконтроллером*. Современный микроконтроллер является большой цифровой интегральной схемой, объединяющей миллионы выполненных по микронным технологиям транзисторов.

Особенности построения микроконтроллеров. При проектировании микроконтроллеров приходится соблюдать баланс между размерами и стоимостью, с одной стороны, и гибкостью и производительностью – с другой. Для разных приложений оптимальное соотношение этих и других параметров может различаться очень сильно. Поэтому существует огромное количество типов микроконтроллеров, отличающихся архитектурой процессорного модуля, размером и типом встроенной памяти, набором периферийных устройств, типом корпуса и т.д. В отличие от обычных компьютерных микропроцессоров, в микроконтроллерах часто используется гарвардская архитектура памяти, т.е. раздельное хранение данных и команд в ОЗУ и ПЗУ соответственно.

Кроме ОЗУ, микроконтроллер может иметь встроенную энергонезависимую память для хранения программы и данных. Во многих контроллерах вообще нет шин для подключения внешней памяти. Наиболее дешевые типы памяти допускают лишь однократную запись. Такие устройства подходят для массового производства в тех случаях, когда программа контроллера не будет обновляться. Другие модификации контроллеров обладают возможностью многократной перезаписи энергонезависимой памяти.

Ограничения по цене и энергопотреблению сдерживают также рост тактовой частоты контроллеров. Хотя производители стремятся обеспечить работу своих изделий на высоких частотах, они, в то же время, предоставляют заказчикам выбор, выпуская модификации, рассчитанные на разные частоты и напряжения питания. Во многих моделях микроконтроллеров используется статическая память для ОЗУ и внутренних регистров. Это дает контроллеру возможность работать на меньших частотах и даже не терять данные при полной

остановке тактового генератора. Часто предусмотрены различные режимы энергосбережения, в которых отключается часть периферийных устройств и вычислительный модуль.

Структура микроконтроллера. Обобщенная структура микроконтроллера изображена на рис.3.1, а на рис.3.2 показана структурная схема типичного МК с характерным набором периферийных устройств и внешнего оборудования. Микроконтроллер состоит из трех, связанных системными шинами, элементов: процессорного ядра («процессора»), памяти и набора периферийных устройств различного назначения («устройств ввода-вывода»). [7, 10]

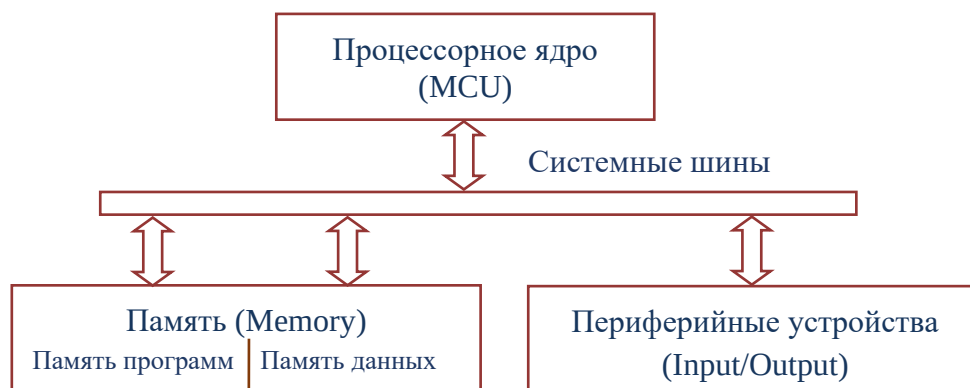


Рисунок 3.1 – Структура микроконтроллера

Процессорное ядро (CPU Core – Central Processor Unit Core, или MCU – Microprocessor Core Unit), или *центральное процессорное устройство* (ЦПУ, CPU – Central Processor Unit), или просто «процессор», является основой микроконтроллера. По системной шине процессорное ядро обменивается данными с памятью и различными периферийными устройствами и получает из памяти команды. Для временного хранения команд и связанных с ними данных оно содержит набор регистров (вспомогательную, регистровую память). К ним относятся: регистр команд, служащий для размещения кода выполняемой команды, указатель команд (счетчик команд, программный счетчик), содержащий адрес следующей команды для исполнения, регистры общего назначения для хранения операндов и др. Разрядность процессорного ядра определяет разрядность микроконтроллера. Наиболее распространены в настоящее время 8-битные микроконтроллеры. Вместе с тем, широкое применение в простых задачах находят и 4-битные изделия, а в сложных высокопроизводительных системах 16- и 32-битные.

В памяти (Memory) хранится программа работы микроконтроллера, исходные данные и все промежуточные результаты вычислений. Память состоит из множества идентичных технических устройств – блоков одинакового размера, служащих для запоминания отдельных слов команд и данных. Эти запоминающие блоки называются *ячейками памяти*. Каждая из ячеек имеет свой номер, называемый *адресом*. По этому адресу процессорное ядро находит кон-

кретную ячейку памяти в процессе обмена. Память микроконтроллера обычно разделена на две части: память данных (Data Memory) и память программ (Program Memory). В памяти данных хранятся переменные операнды, представляющие результаты промежуточных вычислений, в памяти программ – программа, по которой работает микроконтроллер. У большинства микроконтроллеров в памяти данных расположен также стек. Память программ микроконтроллера энергонезависима, при отключении питания в ней сохраняется записанная программа. *Программа* работы микроконтроллера хранится в памяти программ в

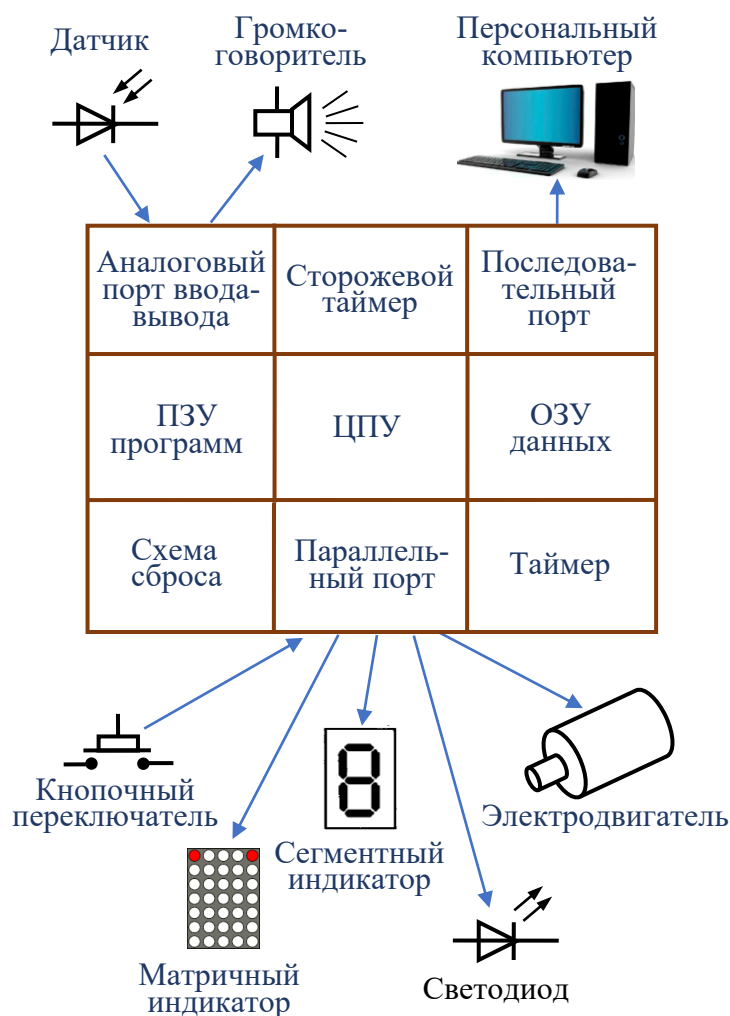


Рисунок 3.2 – Состав и применение типичного микроконтроллера

виде последовательности команд (инструкций), каждая из которых занимает (для большинства МК) одну или две соседние ячейки памяти. В ходе работы процессорное ядро последовательно извлекает из памяти инструкции, расшифровывает и выполняет их. В зависимости от инструкции в ядре выполняются различные арифметические и логические операции и пересылки данных. При необходимости в процессе выполнения инструкции процессорное ядро обращается за данными к ячейкам памяти данных и функциональным блокам, либо пе-

ресылает в них результаты вычислений. Множество инструкций, которые понимает процессорное ядро, образует *систему команд микроконтроллера*. Все команды и данные записываются в МК, как и в любом компьютере, с помощью двоичных чисел – комбинаций нулей и единиц.

Периферийные устройства (устройства ввода-вывода, Input/Output). В зависимости от модели МК в нем могут присутствовать то или иное число дополнительных, или, как еще говорят, «периферийных» устройств. Периферийные они по отношению к центральному процессорному устройству микроконтроллера. Но находятся они тоже внутри микросхемы. Периферийные устройства, традиционно, как и для всякого компьютера, называют еще *устройствами ввода-вывода*, поскольку процессорное ядро вводит из них и выводит в них данные.

Периферийные устройства обеспечивают связь микроконтроллера с внешним миром, а также выполняют внутренние операции по обслуживанию МК, например, отсчитывают промежутки времени, задают режимы его работы и т.д. Для процессорного ядра любое периферийное устройство представляется в виде одного или нескольких регистров. Каждый регистр имеет свой оригинальный *адрес*, по которому процессорное ядро находит его в процессе работы. Адресуемые регистры периферийных устройств принято называть *регистрами ввода-вывода*.

Все периферийные устройства работают сами по себе, т.е. отдельно от процессора МК и не мешают выполнению программы. Когда периферийное устройство выполнит свою работу, оно может сообщить процессору (с помощью сигнала прерывания или через адресуемый бит – «флажок»), а может и не сообщать – зависит от нашего желания, сами потом посмотрим на результаты с помощью программы.

Типичными периферийными устройствами являются:

- Тактовый генератор. Этот генератор определяет скорость работы микроконтроллера.
- Цепь сброса. Эта цепь служит для правильного запуска микроконтроллера.
- Последовательный порт – очень полезный элемент микроконтроллера. Он позволяет обмениваться данными с внешними устройствами при малом количестве проводов (благодаря последовательной передаче битов данных по одной проводной линии).
- Цифровые линии (параллельный порт) ввода-вывода. По сравнению с последовательным портом с помощью параллельного порта можно управлять одновременно несколькими линиями (или проверять несколько линий).
- Таймер. Используется для отсчета временных интервалов.
- Сторожевой таймер. Это специальный таймер, предназначенный для предотвращения сбоев программы. Он работает следующим образом: после запуска он начинает отсчет заданного временного интервала. Если программа не перезапустит его до истечения этого интервала времени, сторожевой таймер перезапустит микроконтроллер. Таким образом, программа

должна давать сторожевому таймеру сигнал – все в порядке. Если она этого не сделала, значит, по какой-либо причине произошел сбой.

Многие МК содержат также:

- Аналого-цифровые и цифро-аналоговые преобразователи.
- Компараторы.
- Широтно-импульсные модуляторы (ШИМ, PWM – Pulse-Width Modulation – процесс управления мощностью методом пульсирующего включения и выключения прибора).
- Контроллеры бесколлекторных двигателей.
- Контроллеры дисплеев и клавиатур.
- Радиочастотные приемники и передатчики.

3.2. Классификация МК

Основным классификационным признаком микроконтроллеров является **разрядность** данных, обрабатываемых арифметико-логическим устройством (АЛУ). По этому признаку МК делятся на:

- 4-битные (Atmel MARC4, Winbond W742, NEC uPD75 и др.) – на сегодняшний день морально устаревшие;
- 8-битные (Intel MCS-48, Intel MCS-51, Atmel ATtiny/ATmega/ATXmega, Microchip PIC12/16/18, Zilog Z86, Ангстрем KP1878BE1, Миландр 1886BExx, НИИЭТ Н1830BExx и др.);
- 16-битные (Intel MCS-96, Texas Instruments MSP430, Motorola 68HC16, Fujitsu MB90, Infineon C16, Mitsubishi M16C, Microchip PIC24, НИИЭТ Л1874BExx и др.);
- 32-битные (Atmel ARM32, Fujitsu MB91, NEC V850, NXP LPC2xx, НИИЭТ К1921BK01Т и Л1874BExx, Миландр 1986BE9x и др.).

8-разрядные МК имеют относительно низкую производительность, которая вполне достаточна для решения широкого круга задач управления различными объектами. Это простые и дешевые МК, ориентированные на использование в относительно несложных устройствах массового выпуска. Основными областями их применения являются бытовая и измерительная техника, промышленная автоматика, автомобильная электроника, теле-, видео- и аудиоаппаратура, средства связи. Внутренняя память программ обычно имеет объем от нескольких единиц до десятков килобайт. Для хранения данных используется регистровый блок, организованный в виде нескольких регистровых банков, или внутреннее ОЗУ. Объем внутренней памяти данных составляет от нескольких десятков байт до нескольких килобайт. Ряд МК этой группы позволяет, в случае необходимости, дополнительно подключать внешнюю память команд и данных, объемом до 64...256 килобайт. МК этой группы обычно выполняют относительно небольшой набор команд (30-100), использующих наиболее простые способы адресации, обеспечивают выполнение большинства команд за один такт машинного времени.

16-разрядные МК во многих случаях являются усовершенствованной модификацией своих 8-разрядных прототипов. Они характеризуются не только увеличенной разрядностью обрабатываемых данных, но и расширенной системой команд и способов адресации, увеличенным набором регистров и объемом адресуемой памяти, а также рядом других дополнительных возможностей. Обычно эти МК позволяют расширить объем памяти программ и

данных до нескольких мегабайт путем подключения внешних микросхем памяти. Во многих случаях реализуется их программная совместимость с более младшими 8-разрядными моделями. Основная сфера применения таких МК – сложная промышленная автоматика, телекоммуникационная аппаратура, медицинская и измерительная техника.

32-разрядные МК содержат высокопроизводительный процессор, соответствующий по своим возможностям младшим моделям микропроцессоров общего назначения. В ряде случаев процессор, используемый в этих МК, аналогичен CISC- или RISC- процессорам, которые выпускаются или выпускались ранее в качестве микропроцессоров общего назначения. Например, в 32-разрядных микроконтроллерах компании Intel используется процессор i386, в микроконтроллерах компании Motorola широко применяется процессор 68020, в ряде других микроконтроллеров в качестве процессорного ядра служат RISC-процессоры типа PowerPC. На базе данных процессоров были реализованы различные модели персональных компьютеров.

Введение этих процессоров в состав МК позволяет использовать в соответствующих системах управления огромный объем прикладного и системного программного обеспечения, созданный ранее для соответствующих персональных компьютеров. Кроме 32-разрядного процессора на кристалле МК размещается внутренняя память команд емкостью до десятков килобайт, память данных емкостью до нескольких килобайт, а также сложно-функциональные периферийные устройства – таймерный процессор, коммуникационный процессор, модуль последовательного обмена и ряд других. Такие МК могут работать с внешней памятью объемом до 16 Мбайт и выше. Они находят широкое применение в системах управления сложными объектами промышленной автоматике (двигатели, робототехнические устройства, средства комплексной автоматизации производства), в контрольно-измерительной аппаратуре и телекоммуникационном оборудовании. Во внутренней структуре этих МК реализуется неймановская (классическая) и гарвардская архитектура.

К подобным приборам относится 32-разрядный микроконтроллер K1921BK01T отечественного производства компании ОАО «НИИЭТ». Он создан на базе ядра ARM Cortex-MF4 с функцией Motor Control и является высокопроизводительной микросхемой для управления техникой: современными российскими автомобилями, кораблями, электрическим транспортом, робототехническими комплексами и пр. Производительность контроллера составляет 125 MIPS, что значительно опережает большинство отечественных аналогов, а также находится на уровне самых современных зарубежных моделей.

64-разрядные МК. На текущий момент очень большое количество разнообразных 64-х битных микроконтроллеров различных производителей работают на ядре ARM.

Вторым классификационным признаком можно выделить **принцип построения процессорного ядра** (технология построения управляющего устройства): CISC- процессоры и RISC- процессоры (п.1.4).

Первые МК имели стандартную CISC-архитектуру, которая применялась в настольных компьютерах того времени. Особенности CISC: команды выполняются поочередно друг за другом и имеют разную длину и структуру. Выборка команды из памяти осуществляется побайтно и выполняется за несколько тактов. CISC-архитектуру имеют: МК из семейства Motorola HC05/HC08, МК с ядром MCS-51, МК из семейства Infineon C500 и ряд других. Наиболее известные микроконтроллеры с CISC- архитектурой выпускаются фирмами Zilog, Intel, Motorola, Siemens.

В начале 1980-х годов была разработана новая архитектура с многообещающим названием RISC (аббревиатуру предложил Д. Паттерсон из Калифорнийского университета в г. Беркли, США). Основная идея заключается в замене сложных команд однотипными простыми и выполнении их единым потоком на параллельном конвейере. Все команды имеют фиксированную длину и в идеале должны выполняться за один, а не за несколько, тактов, чем достигается повышенное быстродействие.

Одним из первых МК с архитектурой RISC стал PIC-контроллер 16C54 фирмы Microchip. Благодаря высокой производительности и трём десяткам легко запоминающихся ко-

манд, PIC-контроллеры быстро завоевали популярность во всём мире. Вскоре их примеру последовали разработчики из фирм Atmel, Scenix и др.

Подавляющее большинство современных микроконтроллеров построено на RISC-процессорах (п.1.4). RISC-архитектура в МК сейчас вне конкуренции. Даже новейшие клоны контроллеров, имеющие совместимость с MCS-51, отличаются от прародителей в первую очередь сменой архитектуры. Это чётко прослеживается на примере продукции фирмы Atmel – «старая» микросхема AT89C2051 (CISC) против улучшенных «новых» микросхем AT89S2051, AT89LP2052 (RISC).

Третьим классификационным признаком является **архитектура микропроцессора** – неймановская (классическая) и гарвардская архитектура (п.1.2). Существуют МК как с *классической, принстонской*, архитектурой, так и с *гарвардской* архитектурой.

Также МК можно условно разбить на две группы в зависимости от **назначения: универсальные и специализированные**.

Первыми появились МК общего назначения с универсальным набором функций. В «джентльменский набор» их аппаратных узлов входили: порты ввода/вывода, таймеры/счётчики, канал последовательного доступа UART, аналоговый компаратор. По мере встраивания МК в различные бытовые приборы, выяснилось, что большинство функций у них «простаивает», а на выполнение специфических задач тратится слишком много машинного времени. Для заполнения образовавшейся ниши были разработаны специализированные МК, «заточенные» под конкретные сферы применения, например, для драйвера шагового двигателя, для МРЗ-декодера, для подключения символьного ЖКИ, для адаптера Ethernet и т.д.

Специализированные МК имеют в своей структуре дополнительные периферийные схемы или функции, предназначенные специально для использования в конкретных задачах. К группе специализированных МК относят и контроллеры для управления приводами. Данная группа МК определяется наличием дополнительных генераторов ШИМ сигналов, генераторов «мертвого времени», каналов АЦП, схем прерывания, таймеров. Среди производителей наиболее популярных МК данной серии можно отметить компании, производящие МК как с классической архитектурой – Intel, так и с гарвардской архитектурой – Texas Instruments, Analog Devices. Отдельно выпускаются DSP-МК (Digital Signal Processor – цифровой сигнальный процессор), ориентированные на использование в системах обработки сигналов.

Диапазон радиолюбительских разработок настолько велик, что требуемый набор аппаратных узлов заранее не предугадаешь. Поэтому практичнее использовать универсальные МК. В дальнейшем переход на специализированные контроллеры будет происходить гораздо легче, поскольку методология у них общая.

3.3. Применение микропроцессоров и микроконтроллеров

В современном мире трудно найти область техники, где не применялись бы микропроцессоры. Они применяются при вычислениях, они выполняют функции управления, они используются при обработке звука и изображения. В зависимости от области применения микропроцессора меняются требования к нему, что накладывает отпечаток на его внутреннюю структуру.

В настоящее время определилось три **направления развития микропроцессоров:**

- универсальные микропроцессоры;

- сигнальные микропроцессоры;
- микроконтроллеры.

Универсальные микропроцессоры используются для построения вычислительных машин. В них используются самые передовые решения по повышению быстродействия, не обращая особого внимания на габариты, стоимость и потребляемую энергию. В технике связи компьютеры используются для управления системами связи или устройствами связи, обладающими большими габаритами и стоимостью. Такие компьютеры называются контроллерами.

Сигнальные процессоры используются для решения задач, которые традиционно решала аналоговая схемотехника. Цифровой сигнальный процессор (англ. Digital Signal Processor, DSP, цифровой процессор обработки сигналов (ЦПОС)) — специализированный микропроцессор, предназначенный для обработки оцифрованных сигналов (обычно, в режиме реального времени).

Архитектура сигнальных процессоров, по сравнению с микропроцессорами общего применения, имеет некоторые особенности, связанные со стремлением максимально ускорить выполнение типовых задач цифровой обработки сигналов, таких, как цифровая фильтрация, преобразование Фурье, поиск сигналов и т.п. Математически эти задачи сводятся к поэлементному перемножению элементов многокомпонентных векторов действительных чисел, последующему суммированию произведений (например, в цифровой фильтрации выходной сигнал фильтра с конечной импульсной характеристикой равен сумме произведений коэффициентов фильтра на вектор выборок сигнала, аналогичные вычисления производятся при поиске максимумов корреляционных и автокорреляционных функций выборок сигналов). Поэтому сигнальные процессоры оптимизированы по быстродействию для выполнения именно таких операций.

Микроконтроллеры. В то время как 8-разрядные процессоры общего назначения полностью вытеснены более производительными моделями, 8-разрядные микроконтроллеры продолжают широко использоваться. Это объясняется тем, что существует большое количество применений, в которых не требуется высокая производительность, но важна низкая стоимость. В МК, в отличие от универсальных микропроцессоров, максимальное внимание уделяется именно габаритам, стоимости и потребляемой энергии. Именно поэтому МК применяются преимущественно во встроенных системах, в игрушках, в станках, в массовой домашней технике, в домашней автоматике — там, где нужна не мощность процессора, а, скорее, баланс между ценой и достаточной функциональностью.

Вместе с тем, есть МК, обладающие большими вычислительными возможностями, например, цифровые сигнальные процессоры. Использование в современном МК достаточного мощного вычислительного устройства с широкими возможностями, построенного на одной микросхеме вместо целого набора, значительно снижает размеры, энергопотребление и стоимость построенных на его базе устройств. Используются в управлении различными устройствами и их отдельными блоками:

- в вычислительной технике: материнские платы, контроллеры дисководов жестких и гибких дисков, CD и DVD;
- в электронике и разнообразных устройствах бытовой техники, в которой используется электронные системы управления – стиральных машинах, микроволновых печах, посудомоечных машинах, телефонах и современных приборах;
- в устройствах промышленной автоматики – от программируемого реле и встраиваемых систем, до систем управления станками и роботами.

В качестве одного из **примеров применения МК** можно назвать построение регуляторов для управления некоторым объектом регулирования, например, электроприводом (рис.3.3). Так как объект регулирования обычно аналоговый, то необходимо преобразование из цифрового кода в аналоговый и обратно. Для этого в МК существует аналого-цифровой (АЦП) и цифроаналоговый (ЦАП) преобразователь. [14]

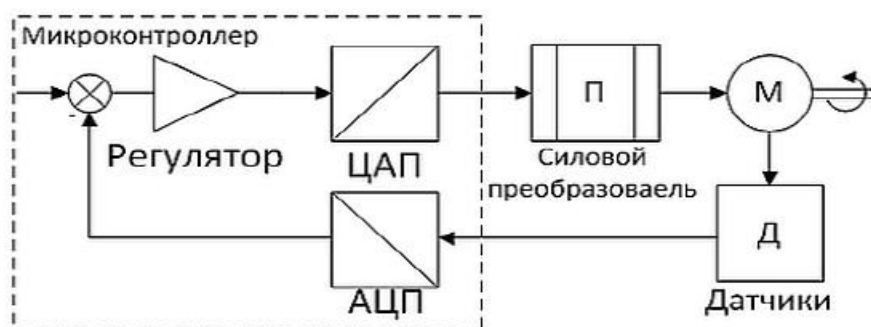


Рисунок 3.3 – Структура САУ электроприводом на базе МК

3.4. Семейства МК

Практически все ведущие производители разрабатывают целые **семейства** микроконтроллеров с так называемой модульной структурой. При этом процессорное ядро для всего семейства неизменно, а память и состав функциональных блоков у каждого микроконтроллера различны. Процессорное ядро всегда имеет свою оригинальную схему и, обязательно, оригинальное имя. Например, микроконтроллеры фирмы Motorola построены на базе ядра HC05 и HC08, фирма Intel создала ядро MCS-51 и MCS-96, контроллеры фирмы Microchip строятся на базе ядра PIC12, PIC16, PIC17, PIC18, фирма Atmel усиленно развивает семейство микроконтроллеров с ядром AVR. Наиболее важная особенность семейства – программная совместимость на уровне двоичного кода всех входящих в него МК.

Ниже охарактеризованы наиболее распространенные семейства МК:

- MCS 51 (Intel),
- MSP430 (TI),

- ARM (ARM Limited),
- AVR (Atmel) (ATmega, ATtiny, XMega),
- PIC (Microchip).

MCS-51 (Intel) – это однокристальный МК (не путать с процессором) гарвардской архитектуры, который был впервые произведен Intel в 1980 году, для использования во встраиваемых системах. В течение 1980-ых и начале 1990-ых годов был чрезвычайно популярен. Однако, в настоящее время устарел и вытеснен более современными устройствами, с 8051-совместимыми ядрами, производимыми более чем 20 независимыми производителями, такими как Atmel, Maxim IC (дочерняя компания Dallas Semiconductor), NXP (ранее Philips Semiconductor), Winbond, Silicon Laboratories, Texas Instruments и Cypress Semiconductor. Существует также советский клон данной микросхемы, KP1816BE51. Официальное название 8051-семейства микроконтроллеров Intel – MCS-51.

PIC (Microchip) – МК Гарвардской архитектуры, производимые американской компанией Microchip Technology Inc. Название PIC является сокращением от Peripheral Interface Controller, что означает «периферийный интерфейсный контроллер».

В основу концепции PIC, единую для всех выпускаемых семейств, была положена RISC-архитектура с системой простых однословных команд, применение встроенной памяти программ и данных и малое энергопотребление.

В номенклатуре Microchip Technology Inc. представлен широкий спектр 8-и, 16-и и 32-битных МК и цифровых сигнальных контроллеров под маркой PIC. Отличительной особенностью PIC-контроллеров является хорошая преемственность различных семейств. Это и программная совместимость (единая бесплатная среда разработки MPLAB IDE), и совместимость по выводам, по периферии, по напряжениям питания, по средствам разработки, по библиотекам и стекам наиболее популярных коммуникационных протоколов. Номенклатура насчитывает более 500 различных МК со всевозможными вариациями периферии, памяти, количеством выводов, производительностью, диапазонами питания и температуры и т. д.

AVR (Atmel) – концепция новых скоростных МК была разработана группой разработчиков исследовательского центра Atmel в Норвегии, инициалы которых затем сформировали марку AVR (Alf Bogen / Vergard Wollan / RISC architecture). Первые микроконтроллеры AVR AT90S1200 появились в середине 1997 г. и быстро нашли расположение потребителей.

AVR-архитектура, на основе которой построены МК семейства AT90S, объединяет мощный гарвардский RISC-процессор с отдельным доступом к памяти программ и данных, 32 регистра общего назначения, каждый из которых может работать как регистр-аккумулятор, и развитую систему команд фиксированной 16-бит длины. Большинство команд выполняются за один машинный такт с одновременным исполнением текущей и выборкой следующей команды, что обеспечивает производительность до 1 MIPS на каждый МГц тактовой частоты.

МК AVR имеют более развитую систему команд, насчитывающую до 133 инструкций, производительность, приближающуюся к 1 Mips (1 МГц), Flash ПЗУ программ с возможностью внутрисхемного перепрограммирования. Многие чипы имеют функцию самопрограммирования. AVR-архитектура оптимизирована под язык высокого уровня Си.

Огромную роль сыграла доступность программного обеспечения и средств поддержки разработки. Хорошо известно, что развитые средства поддержки разработок при освоении и знакомстве с любым микроконтроллерным семейством играют не менее значимую роль, чем сами кристаллы. Фирма Atmel уделяет этому вопросу большое внимание, представляя бесплатную среду разработки AVR Studio. В дополнение к этому ведущие сторонние производители выпускают полный спектр компиляторов, программаторов, ассемблеров, отладчиков, разъемов и адаптеров. Самым популярным способом программирования этих микроконтроллеров являются пять проводков, подсоединенных к параллельному порту персонального компьютера

Можно считать, что AVR постепенно становится еще одним промышленным стандартом среди 8-разрядных МК общего назначения. Они легкодоступны в России и отличаются в среднем невысокой стоимостью, успешно конкурируя с изделиями компании Microchip.

ARM (ARM Limited) – архитектура ARM (Advanced RISC Machine, Acorn RISC Machine, усовершенствованная RISC-машина) – семейство лицензируемых 32-битных и 64-битных микропроцессорных ядер разработки компании ARM Limited. Компания занимается исключительно разработкой ядер и инструментов для них (компиляторы, средства отладки и т. п.), зарабатывая на лицензировании архитектуры сторонним производителям.

В 2007 году около 98% из более чем миллиарда мобильных телефонов, продаваемых ежегодно, были оснащены, по крайней мере, одним процессором ARM. По состоянию на 2009 на процессоры ARM приходится до 90% всех встроенных 32-разрядных процессоров. Процессоры ARM широко используются в потребительской электронике – в том числе КПК (карманные персональные компьютеры), мобильных телефонах, цифровых носителях и плеерах, портативных игровых консолях, калькуляторах и компьютерных периферийных устройствах, таких как жесткие диски или маршрутизаторы.

Данные процессоры имеют низкое энергопотребление, поэтому находят широкое применение во встраиваемых системах и доминируют на рынке мобильных устройств, для которых важно низкое энергопотребление.

Среди лицензиатов: Analog Devices, Atmel, Xilinx, Altera, Cirrus Logic (англ.), Intel (до 27 июня 2006 года), Marvell (англ.), NXP, STMicroelectronics, Samsung, MediaTek, MStar, Qualcomm, Sony Ericsson, Texas Instruments, nVidia, Freescale, российские компании Миландр, Модуль.

Глава 4. Микроконтроллеры AVR

4.1. Устройство МК AVR

AVR – семейство восьмиразрядных микроконтроллеров, ранее выпускаемых фирмой Atmel, затем Microchip. Они представляют собой мощный инструмент, универсальную основу для создания современных экономичных встраиваемых систем многоцелевого назначения – от электронных игрушек до робототехники и сложных систем управления. [2, 6, 7, 10, 17, 20]

Идея разработки нового RISC-ядра принадлежит двум студентам Норвежского университета наук и технологий (г. Тронхейм) – Альфу Богену (Alf-Egil Bogen) и Вегарду Воллену (Vegard Wollen). В 1995 г. Боген и Воллен решили предложить американской корпорации Atmel выпускать новый 8-битный RISC-микроконтроллер и снабдить его Flash-памятью для программ на одном кристалле с вычислительным ядром.

Идея была одобрена Atmel Corporation, и в конце 1996 г. была завершена разработка и выпущен опытный микроконтроллер AT90S1200, а во второй половине 1997 г. корпорация Atmel приступила к серийному производству нового семейства микроконтроллеров.

Новое ядро было запатентовано и получило название AVR. Существует несколько трактовок данной аббревиатуры. Кто-то утверждает, что это Advanced Virtual RISC, другие полагают, что не обошлось здесь без инициалов разработчиков Alf Egil Bogen Vegard Wollen RISC.

Микроконтроллеры AVR изготавливаются по малопотребляющей КМОП - технологии, имеют гарвардскую архитектуру (программа и данные находятся в разных адресных пространствах) и систему команд, близкую к идеологии RISC.

Наряду с семейством AVR выпускаются приборы **AVR32**. Это 32-битные микроконтроллеры архитектуры RISC, анонсированные фирмой Atmel в 2006 году. Призваны конкурировать с архитектурами ARM. Могут использоваться в карманных персональных компьютерах (КПК, PDA – *Personal Digital Assistant*, «личный цифровой секретарь») и других мобильных высокопроизводительных устройствах. Обладают примечательным соотношением производительность/тактовая частота. Имеются 2 семейства: AVR32 AP и AVR32 UC3.

AVR32 имеют сходство с 8-битными контроллерами AVR только в месте их разработки и средствах отладки, так как обе архитектуры были спроектированы норвежским университетом науки и технологии с участием Oyvind Strom, аспирантами и процессорным архитектором Эриком Ренно, являющимся M.Sc (A Master of Science – магистр наук) в норвежском центре дизайна Atmel.

Структура AVR-контроллера. [7, 10, 14, 17, 24]

AVR-микроконтроллеры содержат на кристалле 8-разрядное процессорное ядро, память программ, оперативную память данных, энергонезависимую память данных, регистры ввода-вывода, схему прерываний, схему программирования (обеспечивающую прошивку памяти программ и ПЗУ данных), а также периферийные устройства (рис.4.1).

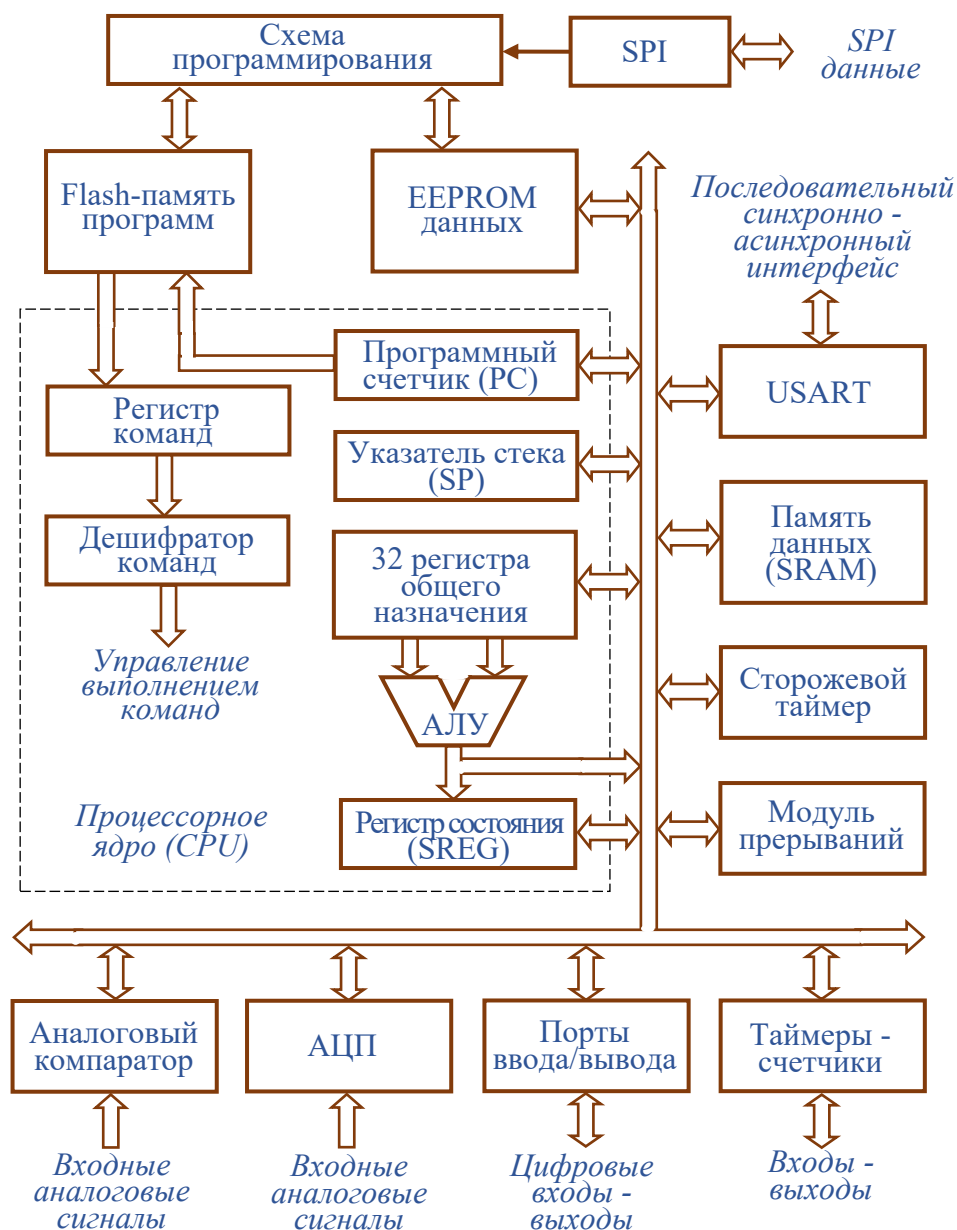


Рисунок 4.1 – Структура микроконтроллеров семейства AVR ATmega.

SPI (Serial Peripheral Interface) – последовательный периферийный интерфейс; USART (Universal Synchronous and Asynchronous Receiver and Transmitter) – универсальный синхронно-асинхронный последовательный приемопередатчик

Процессорное ядро. *Процессорное ядро* (CPU Core – Central Processor Unit Core, или MCU – Microprocessor Core Unit) именуется также *центральной процессорным устройством* (ЦПУ, CPU – Central Processor Unit), или просто «процессором». Процессор AVR-микроконтроллеров содержит арифметико-логическое устройство (АЛУ), регистры общего назначения (РОН), программный счетчик (PC – Program Counter), указатель стека (SP – Stack Pointer), регистр состояния (SREG – Status Register), регистр команд, дешифратор команд, схему управления выполнением команд.

В АЛУ выполняются все вычислительные операции. Операции производятся только над содержимым РОН, которые подключены непосредственно к АЛУ. В связи с этим доступ к РОН происходит быстро: на выборку содержимого регистров, выполнение операции и запись результата обратно в РОН затрачивается один машинный такт (один период тактовой частоты). РОН являются по существу сверхбыстрой оперативной памятью данных.

РОН (регистры общего назначения) представляют собой 32 8-битных регистра, имеющие имена R0, R1,..., R31. РОН составляют так называемый файл регистров общего назначения.

В отличие от «идеального» RISC, регистры не абсолютно равноправны:

- три «сдвоенных» 16-битных регистра X (R27:R26), Y (R29:R28) и Z (R31:R30) используются как указатели при косвенной адресации памяти;
- некоторые команды работают только с регистрами R16...R31;
- результат умножения (в тех моделях, в которых есть модуль умножения) всегда помещается в R1:R0.

Характеристики РОН описаны в п.5.2, 5.3, а машинные команды в главе 6.

Специальные регистры процессора PC, SP, SREG.

Программный счетчик (PC) содержит всегда адрес следующей по порядку, еще не исполненной команды программы, которую процессор будет выбирать из памяти и выполнять вслед за текущей, обрабатываемой в данный момент командой (п.5.3).

Указатель стека (SP) служит для хранения адреса вершины стека (п.5.5).

Регистр состояния (SREG) содержит слово состояния (иначе «флаги») процессора (п.5.3).

Регистр команд, дешифратор команд и схема управления выполнением команд обеспечивают выборку из памяти программ команды, адрес которой содержится в программном счетчике, и занесение ее в регистр команд, декодирование ее, определение способа доступа к указанным в команде аргументам и собственно выполнение команды. Для ускорения выполнения команд используется механизм конвейеризации, который заключается в том, что во время исполнения текущей команды программный код следующей выбирается из памяти и декодируется.

Основная память AVR-микроконтроллеров состоит из памяти программ и памяти данных. Она организована по схеме гарвардского типа – адресные пространства памяти программ и памяти данных разделены.

Память программ реализована на массивах встроенной электрически перепрограммируемой постоянной (энергонезависимой) памяти типа Flash – флэш-памяти (Флэш-ПЗУ – постоянное запоминающее устройство, Flash-ROM – Read Only Memory, память только для чтения).

Память данных выполнена в виде статического оперативного запоминающего устройства (СОЗУ, SRAM – Static Random Access Memory*), представляющего собой энергонезависимую память – при отключении питания вся информация в ней стирается. Кроме СОЗУ МК AVR содержит еще одно устройство памяти данных – электрически стираемое перепрограммируемое ПЗУ

(ЭСППЗУ, EEPROM – Electrically Erasable Programmable Read-Only Memory). Оно энергонезависимо (как PROM и EPROM) и служит для запоминания важных операндов.

* Статическая память с произвольным доступом или по-русски чаще – статическое оперативное запоминающее устройство (СОЗУ).

Детальное описание памяти МК AVR содержится в главе 5.

Периферийные устройства. Любой микроконтроллер AVR обязательно содержит набор *периферийных устройств* (называемых также *устройствами ввода-вывода*). Они осуществляют обмен информацией микроконтроллера с внешней средой, подсчет внешних событий и интервалов времени, аналого-цифровые и цифроаналоговые преобразования сигналов, сравнение различных входных величин, контроль напряжения питания, прием внешних и внутренних запросов прерываний на процессорное ядро и управление их прохождением в процессор и др.

Процессор общается с периферийными устройствами через их адресуемые регистры, называемые *регистрами ввода-вывода*.

Некоторые периферийные устройства и их регистры описаны в главах 10-13.

Некоторые характеристики встроенной памяти и периферии (для AVR ATmega).

Память.

Внутренняя флэш-память команд до 256 Кб (не менее 10 000 циклов перезаписи). Отладка программ осуществляется с помощью интерфейсов JTAG или debug WIRE.

Внутреннее EEPROM данных до 4 Кб (100 000 циклов).

Внутренняя SRAM до 16 Кб время доступа 1 такт.

Внешняя память объемом до 64 Кб (Mega8515 и Mega162).

Периферия.

Многофункциональные, двунаправленные порты ввода-вывода со встроенными подтягивающими резисторами. Конфигурация портов ввода-вывода задается программно.

Таймеры с разрядностью 8, 16 бит.

ШИМ-модулятор (PWM, Pulse-Width Modulation) 8-, 9-, 10-, 16-битный.

Аналоговые компараторы.

АЦП (ADC, Analog to Digital Converter) с дифференциальными входами, разрядность 10 бит (12 для XMega AVR):

- программируемый коэффициент усиления перед АЦП 1, 10 и 200;
- опорное напряжение 2,56 В.

Различные последовательные интерфейсы, включая:

- двухпроводной интерфейс TWI (Two Wire Interface), совместимый с I²C (для XMega);
- универсальный синхронно/асинхронный приемопередатчик UART/USART;

– синхронный последовательный порт Serial Peripheral Interface (SPI) (для Xmega).

Схема прерываний (которую можно одинаково отнести как к периферийным устройствам, так и к процессорному ядру) обеспечивает возможность асинхронного прерывания процесса выполнения программы при определенных условиях.

Тактовая частота. AVR ATmega – до 20 МГц, ATmega16 – до 16 МГц. Производительность 1 MIPS/МГц (1 млн. команд в секунду при частоте 1 МГц).

В качестве *источника тактовых импульсов* может быть выбран:

- кварцевый резонатор;
- внешний тактовый сигнал;
- внутренний RC-генератор (частота 1, 2, 4, 8, 16 МГц для ATmega16).

Система команд микроконтроллеров AVR весьма развита и насчитывает в различных моделях от 90 до 133 различных инструкций.

Большинство команд занимает 1 ячейку памяти (16 бит), остальные – 2 ячейки (32 бита) – это команды, у которых одним из операндов является 16-битный адрес.

Большинство команд выполняется за 1 такт.

4.2. Выводы МК

С внешним миром микроконтроллер общается через **выводы** (ножки, контакты, «пины» *) – металлические штырьки, торчащие из корпуса микрочипа контроллера и предназначенные для подсоединения к микрочипу внешних устройств. По этим выводам микроконтроллер осуществляет прием (ввод) и выдачу (вывод) на внешние устройства цифровых или аналоговых сигналов (в зависимости от программной или аппаратной настройки вывода).

* От англ. pin – штифт, штырь, шпилька, штырек, шплинт; иголка, булавка; вообще любой продолговатый предмет для соединения чего-либо. В нашем случае – вывод на микрочипе.

Каждый микроконтроллер снабжен определенным числом выводов. МК AVR различных серий (п.4.4) имеют различное количество выводов: 6-32 – МК серии Tiny, 28-100 – Mega и 44-100 – Xmega.

На рис.4.2 показана **разводка выводов** (цоколёвка, «распиновка») МК ATmega16, а в табл.4.1 – **описание их аббревиатур**. Большинство ножек МК принадлежит *цифровым портам* (РА, РВ, РС, РD). То есть мы можем либо управлять чем-либо (например, зажигать светодиод), либо принимать данные с чего-либо (например, обрабатывать нажатие кнопки). Также почти каждая ножка имеет *дополнительные функции*. На распиновке они подписаны в скобках. Это могут быть ножки приемопередатчиков, разные последовательные интерфейсы, аналоговые входы, выходы ШИМ генераторов и т.п. По умолчанию все эти функции отключены, а ножка управляется как цифровая. Но если включить какую-либо дополнительную функцию, то тут уже управление может полно-

стью или частично перейти под контроль периферийного устройства. До тех пор пока не выключить периферию, занимающую эти ножки. Например, приемник USART. Стоит только выставить бит разрешения приема RXEN по ножке RXD, то, как бы она ни была настроена до этого, эта ножка переходит в режим приема (входа) в USART.

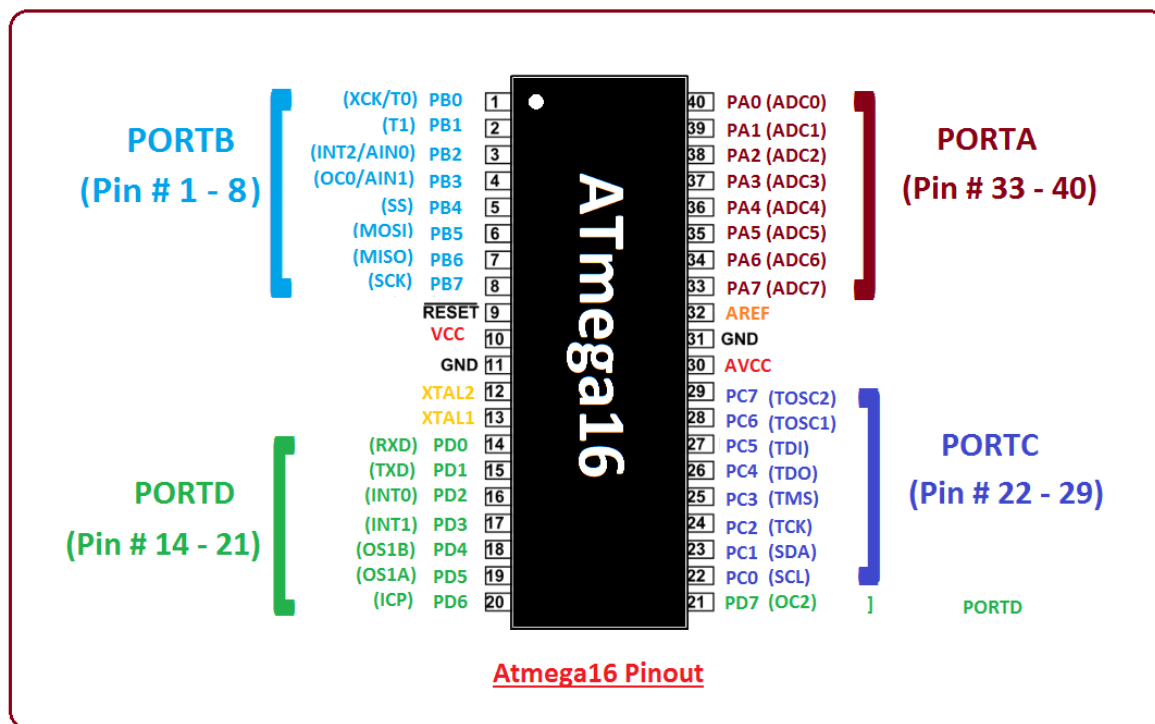


Рисунок 4.2 – Выводы МК ATmega16

Таблица 4.1 – Описание аббревиатур ножек AVR

Pxn	n-й разряд порта x: x=A,B,C,D; n=0, 1, ..., 7 (Port)
ADCn	n-й вход АЦП (Analog to Digital Converter)
AREF	Вход опорного напряжения для АЦП (ADC Reference Voltage)
AVCC ¹⁾	Вывод питающего напряжения для АЦП (ADC Voltage Collector)
AIN0	Положительный вход компаратора (Analog Input 0)
AIN1	Отрицательный вход компаратора (Analog Input 1)
INT0,1,2	Входы внешнего прерывания 0, 1, 2 (Interruption)
XTAL1 ²⁾	Вход тактового генератора
XTAL2 ²⁾	Выход тактового генератора
XCK	Вход / выход внешней синхронизации USART ³⁾
MOSI ⁴⁾	Вывод для ведущей / ввод для подчиненной шины SPI ⁶⁾ модуля USI ⁷⁾
MISO ⁵⁾	Ввод для ведущей / вывод для подчиненной шины SPI модуля USI
SCK	Вход тактового сигнала при программировании
TDI	Вход данных модуля USI в режиме SPI (Test Data Input)
TDO	Выход данных модуля USI в режиме SPI (Test Data Output)
SCK ⁸⁾	Синхронизация последовательной связи шины SPI модуля USI
SDA	Вход/выход данных модуля USI в режиме TWI ⁹⁾
SCL	Вход/выход тактового сигнала модуля USI в режиме TWI
RXD	Вход USART ⁹⁾ (Receive Data)

TXD	Выход USART (Transmit Data)
XCK	Вход/выход внешнего тактового сигнала USART (Transfer Clock)
RESET	Сброс
Tn	Вход внешнего тактового сигнала таймера/счетчика Tn, n=0, 1
OC0	Выход сравнения таймера/счетчика T0 (Output Compare)
OC1X	Выход сравнения X таймера/счетчика T1, X=A, B (Output Compare)
ICP1	Вход захвата таймера/счетчика T1 (Input Capture)
OC2	Выход сравнения таймера/счетчика T2 (Output Compare)
TOSC1	Вход таймера/счетчика T2 для внешнего кварцевого генератора (TOS Cycle)
TOSC2	Выход таймера/счетчика T2 для внешнего кварцевого генератора (TOS Cycle)
GND	Общий провод (Ground)
VCC	Питание микросхемы (Voltage Collector)

¹⁾ Vcc – плюс напряжения питания; AVcc – плюс напряжения питания для АЦП. Почему буквы «с» (collector, коллектор) две? Чтобы отличать напряжение питания (Vcc) от напряжения на выводе транзистора (Vc). GND – общий провод, «земля» (минус напряжения питания)

²⁾ XTAL – неофициальная аббревиатура от crystal. XTAL, XTAL OSC, crystal oscillator – генератор с кварцевой стабилизацией частоты, кварцевый генератор

³⁾ USART (Universal Synchronous and Asynchronous serial Receiver and Transmitter) – универсальный синхронно-асинхронный последовательный приемо-передатчик.

⁴⁾ MOSI – Master Output / Slave Input

⁵⁾ MISO – Master Input / Slave Output

⁶⁾ SPI (Serial Peripheral Interface) – последовательный периферийный трехпроводной интерфейс

⁷⁾ USI (Universal Serial Interface) – универсальный периферийный интерфейс

⁸⁾ Он же USCK

⁹⁾ TWI – аппаратный двухпроводной интерфейс I²C, называемый по терминологии Atmel Two Wire Interface

4.3. Сброс микроконтроллера AVR

Сбросом (он же реинициализация, перезагрузка – Reset) считается перевод микроконтроллера в исходное состояние. При этом все регистры микропроцессорного ядра устанавливаются во вполне определенные начальные состояния, и микроконтроллер переходит к выполнению программы с нулевого начального адреса [6].

Источники сброса. Причинами (источниками) сброса могут являться различные воздействия: включение питания и кратковременные его изменения, сигналы, формируемые аппаратно вне и внутри микроконтроллера, а также инструкции программы. В частности, инструкция безусловного перехода на адрес 0x0 всегда приводит к сбросу устройства.

Микроконтроллер ATmega16 имеет пять источников сброса:

- Сброс при подаче питания (Power-on Reset). Сброс происходит, если напряжение питания ядра ниже определенного порога – порога сброса (VPOT). Типовое значение порогового напряжения VPOT= 1,4 В.

- Внешний сброс (External Reset). Сброс происходит при поступлении сигнала низкого уровня длительностью более 500 ns на внешний контакт RESET микросхемы.
- Сброс при кратковременном провале напряжения питания (Brownout Reset). Сброс происходит, если напряжение питания V_{CC} контроллера в процессе работы опускается ниже порогового значения (VBOT) и разрешена работа схемы контроля питания BOD.
- Сброс сторожевым таймером (Watchdog Reset). Сброс происходит по сигналу сторожевого таймера (если разрешена его работа и истек период его сбрасывания).
- Сброс через интерфейс периферийного сканирования JTAG (Joint Test Action Group). (См. раздел «Граничное сканирование IEEE 1149.1 (JTAG)»).

По любой из этих причин микроконтроллер переходит к выполнению программы с адреса 0x0. В этой ячейке обычно размещают инструкцию jmp с адресом программы инициализации системы.

Регистр управления и состояния процессорного ядра MCUCSR (MCU Control and Status Register) позволяет найти источник сброса. Он доступен программно в пространстве регистров ввода/вывода по адресу \$34 (\$54) (рис.4.3) [24].

Bit	7	6	5	4	3	2	1	0	
	JTD	ISC2	–	JTRF	WDRF	BORF	EXTRF	PORF	MCUCSR
Read/Write	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	See Bit Description					

Рисунок 4.3 – Управляющий и статусный регистр MCU

В системе сброса используются четыре младших бита регистра:

- Бит 0 – PORF (Power-on Reset Flag) – флаг сброса при включении питания. Бит устанавливается при включении питания, сбрасывается только путем записи логического нуля.
- Бит 1 – EXTRF (External Reset Flag) – флаг внешнего сброса. Флаг устанавливается при сбросе от внешнего источника, сбрасывается при включении питания или путем записи логического нуля. Флажки сброса используются для определения причины сброса. Если регистр очищен программно до момента сброса, то источник сброса может быть найден чтением регистра целиком или его отдельных флажков.
- Бит 2 – BORF (Brownout Reset Flag) – флаг сброса при кратковременном провале питания. Бит устанавливается при сбросе от кратковременного провала напряжения питания, сбрасывается при включении питания или путем записи логического нуля.
- Бит 3 – WDRF (Watchdog Reset Flag) – флаг сторожевого таймера. Бит устанавливается при сбросе процессорного ядра сторожевым таймером, сбрасывается при включении питания или путем записи логического нуля.

4.4. Виды МК AVR и сферы применения

Существует три вида (серии) МК AVR:

1. Tiny AVR (ATtinyxxx) – недорогие миниатюрные микроконтроллеры, преимущественно в 8-выводном исполнении (tiny – очень маленький, крошечный, юный, молодой, младший). Объем их памяти, набор периферийных устройств и функции обычно скромнее, чем в следующем виде – mega: флэш-память до 16 Кб; SRAM до 512 б; EEPROM до 512 б; число линий ввода-вывода 4-18 (общее количество выводов 6-32); ограниченный набор периферийных устройств.

Наряду с tiny производились микроконтроллеры classic AVR. Первоначально они позиционировались как основная линия микроконтроллеров. В настоящее время не выпускаются. Производительность отдельных модификаций classic до 16 MIPS, Flash-память программ 2...8 Кб, память данных EEPROM 64...512 байт, оперативная память данных SRAM 128...512 байт.

2. Mega AVR (ATmegaxxx) с производительностью 4...16 MIPS для сложных приложений, требующих большого объема памяти. Flash-памятью программ до 256 Кб, памятью данных EEPROM до 4 Кб, оперативной памятью данных SRAM до 16 Кб, числом линий ввода-вывода 23-86 (общее количество выводов 28-100), встроенным 10-разрядным 8-канальным АЦП, аппаратным умножителем 8×8 . Расширенная система команд и периферийных устройств. Первые микроконтроллеры mega AVR были выпущены в 2002 г.

3. XMeta AVR (ATxmegaxxx) – 8/16 битные микроконтроллеры. Выпускаются в корпусах с огромным количеством выводов, от 44 до 100. Столько необходимо для проектов с большим количеством датчиков и исполнительных механизмов. Имеют увеличенный объем памяти и скорость работы, что позволяет достичь высокого быстродействия: флэш-память до 384 Кб, SRAM до 32 Кб, EEPROM до 4 Кб, четырехканальный DMA-контроллер, инновационная система обработки событий.

Некоторые характеристики AVR-микроконтроллеров приведены дополнительно в табл.4.2.

Таблица 4.2 – Характеристики микроконтроллеров с AVR-архитектурой

Семейство	Разрядность	Флэш-память (кБ)	Корпуса (выводы)	Максимальная частота (МГц)	Производительность (MIPS/МГц)
tinyAVR	8	0,5–8	6–32	20	1
megaAVR	8	4–256	28–100	20	1
AVR XMEGA	8	16–384	44–100	32	1
AVR UC3	32	16–512	48–144	66	1,5

Система команд всех видов AVR совместима при переносе программы со слабого на более мощный микроконтроллер.

Сферы применения МК AVR.

Миниатюрные микроконтроллеры AVR семейства "tiny" используются для построения следующих устройств:

- электронные игрушки;

- различные датчики в автомобильной промышленности;
- детекторы дыма и пламени, датчики температур, измерители разных величин;
- недорогие зарядные устройства, индикаторы напряжения и тока;
- пульты управления для разнообразной бытовой и промышленной техники;
- другие не дорогие и миниатюрные электронные устройства.

Чипы семейства **"mega"**, **"xmega"** и **32-bit AVR** применяются в более сложных устройствах:

- робототехника;
- спутниковые навигационные системы;
- функциональные разрядно-зарядные устройства с программированием;
- сложные дистанционные системы управления;
- сетевые устройства;
- быстродействующие системы для передачи и обработки данных;
- сложная бытовая техника;
- устройства ввода и отображения информации с тач-скринами (Touch-screen);
- другие многофункциональные устройства.

Примеры применения микроконтроллеров AVR можно посмотреть в литературе [2, 3, 4, 10].

Популярность микроконтроллеров AVR очень высока. С каждым годом они захватывают все новые и новые ниши на рынке. Не последнюю роль в этом играет соотношение показателей цена / быстродействие / энергопотребление, являющееся весьма привлекательным на рынке 8-битных микроконтроллеров. Кроме того, постоянно растет число выпускаемых сторонними производителями разнообразных программных и аппаратных средств поддержки разработок устройств на их основе.

Глава 5. Память AVR

5.1. Состав памяти микроконтроллера

Назначение памяти и ее части. Чтобы иметь возможность выполнять обработку данных по заданной программе, микроконтроллер, как и любой другой компьютер, должен иметь память, в которой хранятся команды и данные. Память МК состоит из следующих частей (рис.5.1): регистров, памяти программ и памяти данных. При необходимости к микроконтроллеру может подключаться дополнительная, внешняя, память данных.

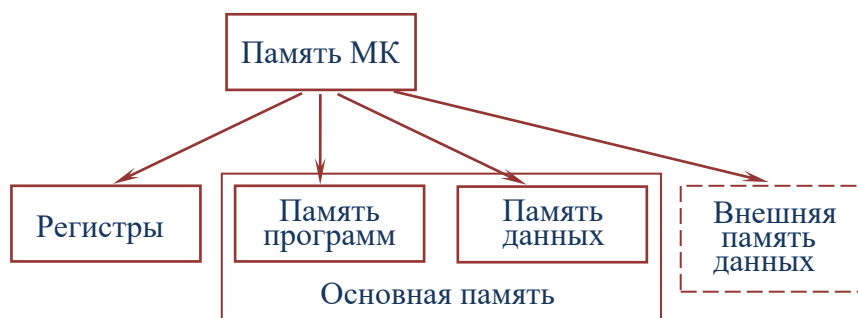


Рисунок 5.1 – Виды памяти микроконтроллера

Регистры распределены по всей системе. Они имеются во всех модулях микроконтроллера – начиная от процессора (процессорного ядра), памяти программ и данных и заканчивая периферийными устройствами. Регистры представляют собой вспомогательную память для временного хранения информации.

Память программ и память данных составляют так называемую *основную память*, служащую для хранения программ и основных объемов данных. При разработке микроконтроллеров AVR используется гарвардская архитектура. Смысл ее заключается в том, что память программ и память данных располагаются в разных адресных пространствах.

Памяти программ и данных состоят из большого числа запоминающих блоков – ячеек памяти. Ячейки отдельных видов памяти идентичны по своей физической реализации, имеют одинаковый размер и нумеруются начиная с некоторого неотрицательного числа, например, с нуля. Конечный номер ячейки может достигать максимально возможного значения, например $2^m - 1$, где m – двоичная разрядность номера. Номер ячейки называют также её *адресом* (п.2.2).

Внутренняя память. Это понятие используют иногда для обозначения всей памяти, встроенной в чип микроконтроллера. К ней относят основную память, а также память процессора и периферийных устройств, представленную наборами их регистров. Совокупность внутренней и внешней памяти образует всю память МК.

Программно-доступные компоненты микроконтроллера. Компоненты микроконтроллера называются *программно-доступными*, если программист может использовать или изменять их содержимое или состояние по своему усмотрению посредством машинных команд. К ним относятся ячейки основной памяти, часть регистров процессора и периферийных устройств, а также некоторые узлы управления/состояния периферийных устройств, не обладающие свойством запоминать информацию.

Посредством команд обработки данных программист может задавать преобразование и пересылку операндов, содержащихся в ячейках памяти данных и программных регистрах, на другие ячейки/регистры. С помощью команд передачи управления – выборку и исполнение нужных в данный момент команд, находящихся в ячейках памяти программ. Ряд микроконтроллеров допускает также выполнение операций над кодами команд из памяти программ как над операндами с помощью команд обработки данных.

Далеко не все регистры находятся в распоряжении программиста. Большая часть их используется самим микроконтроллером и программно недоступна. Например, при выборке из памяти очередной команды она помещается в регистр команд, и программист обратиться к этому регистру не может. Все регистры, располагаемые в модулях основной памяти, и значительное количество регистров в процессоре и периферийных устройствах являются программно недоступными, «невидимыми» для программы. Они выполняют подсобные функции поддержки работы модулей, и программа «не подозревает» об их существовании.

Кроме запоминающих устройств, программно-доступными могут быть, как мы отметили, и некоторые узлы микроконтроллера, не способные запоминать информацию. Например, входные каналы PIN, принимающие текущее состояние ножек цифровых портов ввода-вывода AVR-контроллеров, входы запуска аналого-цифровых преобразователей, получающие сигнал от процессора на начало преобразования, др. узлы. По указке команд обработки данных процессор может принимать от них сигналы состояния или отправлять на них сигналы управления.

Мы говорили, что ячейки памяти имеют порядковые *номера (адреса)*, по которым их можно различать. Такого же рода *номера (адреса)* существуют и у программно-доступных регистров/узлов периферийных устройств, и они тоже пробегают последовательный ряд значений, причем независимо от номеров ячеек. Программные регистры процессора обозначаются на языке ассемблера *собственными именами* (например, R0, R1, ..., R31, PC, SP, SREG у МК AVR), и эти имена в машинных кодах команд представлены определенными укороченными номерами.

Назначая адреса программно-доступных компонентов (или их имена на языке ассемблера), можно обращаться к ним посредством машинных команд (глава 6). Таковыми могут быть адреса/имена ячеек памяти данных или номера/имена регистров процессора и периферии, содержащих операнды для обработки, или адреса/имена функциональных, не запоминающих узлов периферийных устройств (в командах обработки данных). Либо это могут быть адреса/имена ячеек памяти программ, содержащих команды для исполнения (в командах передачи управления). У микроконтроллеров бывают также команды обработки данных, адресующие ячейки памяти программ для выполнения действий над кодами команд как над операндами. Например, команда `lpm` МК AVR адресует байт в памяти программ и пересылает его в регистр общего назначения (п.5.4 и 6.2).

Резюмируя, заключим, что *программно-доступные компоненты микроконтроллера – это его адресуемые, именуемые компоненты: ячейки памяти, регистры процессора и периферийных устройств, узлы управления/состояния ПУ. Ячейки и периферийные компоненты имеют адреса на системной шине микроконтроллера, процессорные регистры – номера (соответствующие их именам) на внутренней шине процессора.*

Программно-доступные регистры процессорного ядра микроконтроллеров AVR подразделяются на общие, или универсальные, и специальные, или особые. К общим относятся регистры общего назначения (РОН), к специальным – программный счетчик, указатель стека и регистр состояния. РОН служат для временного хранения операндов, специальные – для выполнения индивидуальных функций (п.5.1-5.3).

Программно-доступные регистры периферийных устройств (называемые также *регистрами ввода-вывода*) используются программистом для управления и контроля состояния периферийных устройств и передачи данных между ПУ и процессором. Они рассматриваются в п.5.2 и главах 9-13.

Программная модель компьютера. Описание программно-доступных компонентов компьютера, предназначенное для программиста, работающего на нижних, «машинных» уровнях программирования, например, на языке ассемблера, принято называть *программной моделью компьютера* или *моделью компьютера для программиста*, в нашем случае – *программной моделью микроконтроллера*. Настоящая глава как раз и посвящена изложению программной модели микроконтроллеров AVR. Скрытые от программиста устройства, такие как регистр команд процессора, разного рода буферные регистры для временного хранения информации и прочие, не включаются в понятие программной модели МК и рассматриваться далее, в этой и последующих главах, не будут.

5.2. Программная модель

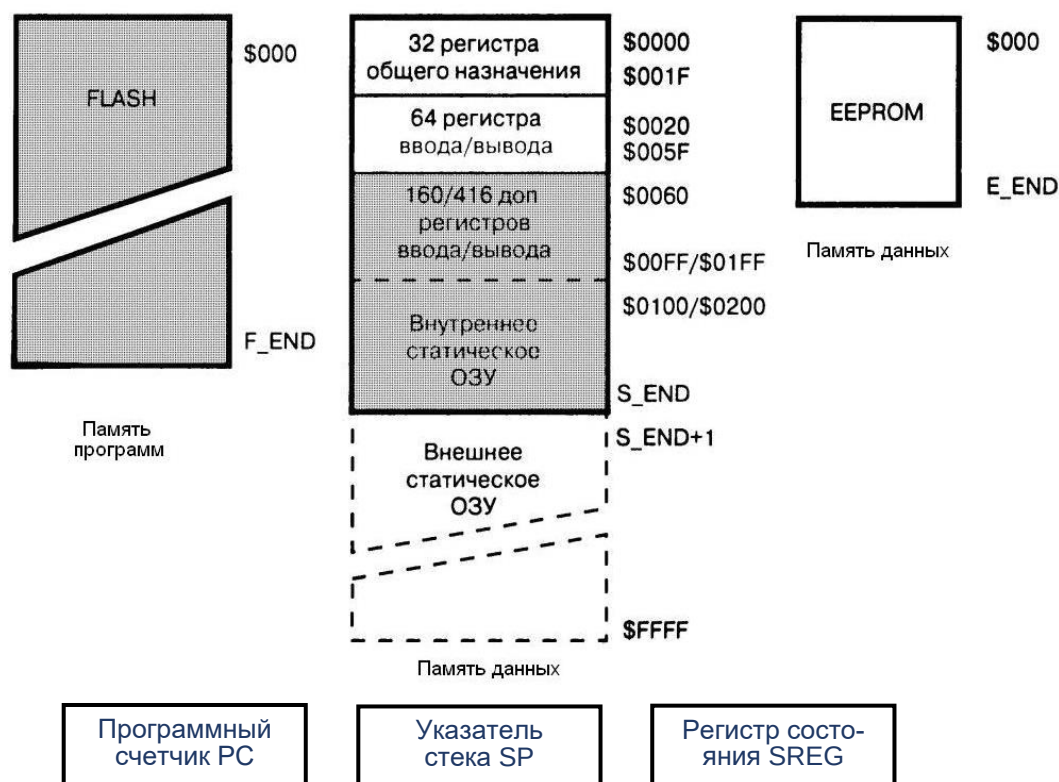
Обобщенная **карта памяти** * микроконтроллеров AVR семейства Mega приведена на рис.5.2, а на рис.5.3 – ее вариант для микроконтроллера ATmega16. Символ «\$» перед числом означает, что это число записано в шестнадцатеричной системе счисления. F_END, S_END (или RAMEND), E_END – нижние границы (максимальные адреса) памяти программ Flash-ROM и памяти данных: SRAM и EEPROM соответственно.

* *Карта памяти, карта распределения памяти* – это схема или описание размещения в адресном пространстве памяти компьютера собственно памяти (ОЗУ и ПЗУ), регистров процессора и периферийных устройств.

Компоненты программной модели. Согласно карте памяти на рис.5.2 и 5.3 программная модель микроконтроллера AVR разделяет все программно-доступные компоненты МК на

- память программ,

- память данных и
- программный счетчик.



Специальные регистры процессора. Из них SP и SREG отображаются на память данных – единое адресное пространство ОЗУ и ввода/вывода

Рисунок 5.2 – Карта памяти AVR-микроконтроллеров семейства Mega

Память программ представляет собой перепрограммируемое постоянное (энергонезависимое, nonvolatile) запоминающее устройство (ПЗУ, или ROM – Read Only Memory) типа Flash (Flash ROM). Она организована в виде последовательности 16-разрядных ячеек, так как команды AVR-микроконтроллера являются 16-разрядными (большинство команд) и 32-разрядными словами. Гарантируется не менее 10 000 циклов перезаписи. Память программ имеет размер от 2 до 256 Кбайт (от 1 до 128 К слов). Память программ контроллера ATmega16 составляет 16 К слов (16К×16), ее нижняя граница F_END = \$3FFF, диапазон адресов \$0000...\$3FFF (у ATmega16L – 8 К слов (8К×16)). [Воробьева Г.С.]

Память данных состоит из трех частей:

- регистровой памяти,
- оперативной (энергозависимой, volatile) памяти (статическое ОЗУ – СОЗУ, SRAM) и
- постоянной (энергонезависимой, nonvolatile) памяти с электрическим стиранием (ЭСПЗУ, EEPROM). [8]

Регистровая память подразделяется на

- регистры общего назначения (РОН) и

- служебные регистры ввода-вывода (PBB).

Регистры общего назначения представлены, как мы уже знаем, 32 регистрами процессора R0...R31, объединенными в файл. К *служебным регистрам ввода-вывода* отнесены регистры ввода-вывода периферийных устройств, а также специальные регистры процессора: указатель стека (SP – Stack Pointer) и регистр состояния (SREG – Status Register). SP 16-битный и является объединением двух 8-битных регистров SPL (Low) и SPH (High): SP=SPH:SPL. POH единообразны и предназначены для хранения любых операндов, а PBB уникальны и выполняют свои, особые, функции каждый.

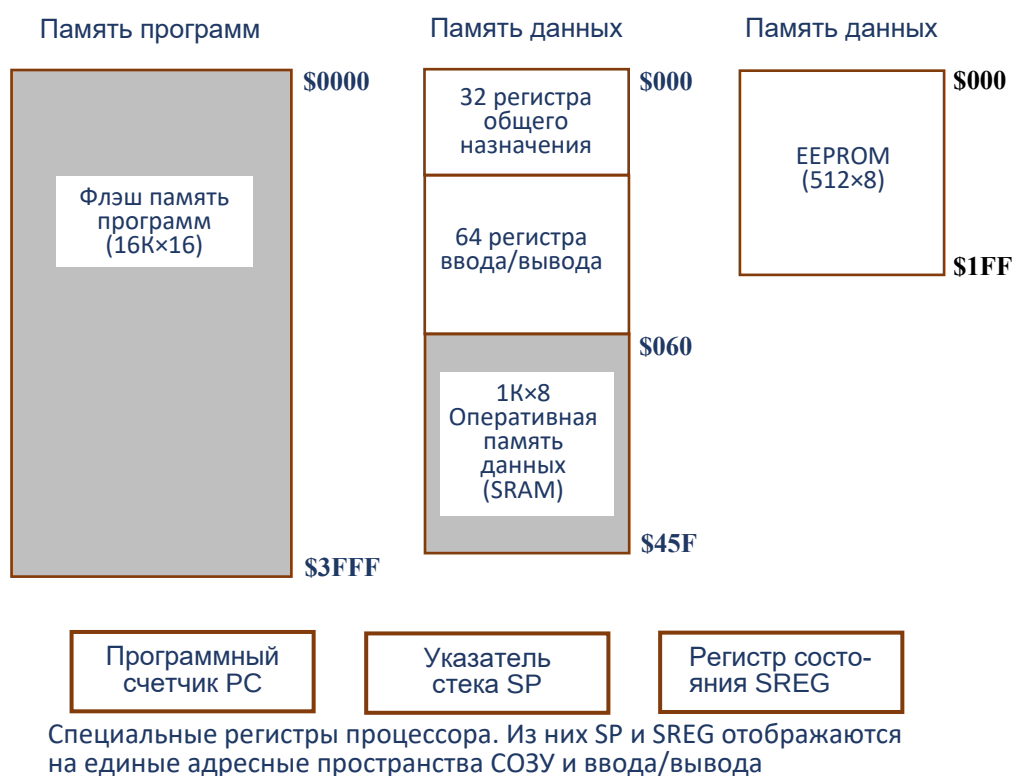


Рисунок 5.3 – Карта памяти ATmega16

Программный счетчик. Еще один специальный регистр процессора – это программный счетчик (PC – Program Counter). В зависимости от модели МК он имеет от 10 до 17 разрядов (округленно – 2 или 3 байта), например 14 у ATmega16 (у ATmega16L – 13 бит). Хотя он и является программно-доступным, но логически отделен от POH и PBB (не является ни POH, ни PBB).

Все POH и PBB (периферийные, SPL, SPH и SREG) 8-битные.

МК AVR Mega насчитывают от 64 до 480 PBB (ATmega16 – 64).

Поясним также, что под PBB понимаются не только регистры (запоминающие устройства), но и программно-доступные не запоминающие узлы периферийных устройств, такие, например, как каналы PIN цифровых портов ввода-вывода.

Обратите внимание, что память данных в программной модели включает в свой состав не только оперативную и постоянную память данных, но и регистровую память – набор

РОН и РВВ, несмотря на то, что эти регистры находятся территориально вне модулей основной памяти данных (см. рис.5.2, 5.3). Причина подобного включения состоит в том, что, как станет понятно из нижеизложенного, РОН и РВВ отображаются на расширение адресного пространства СОЗУ, т.е. логически, с точки зрения программиста, выглядят не только как регистры, но и как ячейки СОЗУ.

Регистровая память структурирована в программной модели тоже несколько иначе, чем она устроена физически. Она разделена не по аппаратной принадлежности: на программные регистры процессора и программные регистры периферийных устройств, а по функциональному назначению: на общие (РОН) и служебные (РВВ) регистры. Кроме того, процессор содержит специальный регистр – программный счетчик РС, не относящийся ни к РОН, ни к РВВ.

Оперативная память данных реализована на статическом оперативном запоминающем устройстве (СОЗУ, или SRAM – Static Random Access Memory, статическая память с произвольным доступом) и организована как последовательность 8-разрядных ячеек. Она предназначена для хранения основных объемов данных при выполнении программы. Оперативная память данных может быть внутренней (до 16 Кбайт) и внешней (до 64 Кбайт). Ее адресное пространство обособлено от пространства Flash ROM. Иными словами есть, например, ячейка Flash ROM с адресом \$0123 и ячейка SRAM с таким же адресом \$0123.

Сама память SRAM в зависимости от модели микроконтроллера занимает адреса на шине \$0060/\$0100/\$0200...\$S_END. Микроконтроллеры ATmega16 имеют SRAM объемом 1 Кб с адресами \$060...\$45F. По младшим адресам \$000-\$01F в ATmega16 можно обращаться к регистрам общего назначения процессора, а по близлежащим адресам \$020...\$05F – к регистрам ввода-вывода. Аналогично у других моделей (см. рис.5.2 и 5.3). (См. ниже «Единые адресные пространства СОЗУ и ввода-вывода».)

При обращении к SRAM используются различные режимы адресации: прямая (простая, со смещением, с предварительным декрементом и с постинкрементом) и косвенная. Указателем при косвенной адресации служат сдвоенные регистры (регистровые пары) X (R27:R26), Y (R29:R28) и Z (R31:R30). Используется и так называемая непосредственная адресация, при которой операнд находится в самой команде, встроен в ее код (находится в памяти программ, а не в памяти данных).

Увеличение оперативной памяти данных вплоть до нижней границы \$FFFF можно осуществлять за счет подключения внешнего запоминающего устройства ERAM (External RAM).

Постоянная (энергонезависимая) память данных представляет собой перепрограммируемое ПЗУ с электрическим стиранием (ЭСРПЗУ, или EEPROM – Electrically Erasable Programmable Read-Only Memory) (100 000 циклов перезаписи). Память EEPROM организована как последовательность 8-разрядных ячеек и имеет размер до 64 Кбайт. У ATmega16 она имеет размер 512 байт.

Память EEPROM тоже имеет изолированное (от Flash ROM и SRAM) адресное пространство. Данные в EEPROM могут быть записаны при программировании микроконтроллера. После выключения питания данные сохраняются.

Все три вида памяти AVR: Flash ROM, SRAM (вместе с регистровой памятью) и EEPROM, являются *линейными* и *непрерывными* – адреса их пробега-

ют линейный и последовательный (без пропусков) ряд значений 0, 1, 2, ..., max для каждого вида.

Единое адресное пространство оперативной памяти. Регистровая память и ячейки СОЗУ образуют так называемое *единое, общее адресное пространство оперативной памяти (СОЗУ)*. Под *адресным пространством* (англ. address space) понимают то или иное множество адресуемых, доступных программисту объектов вычислительной системы – ячеек памяти, регистров процессора и периферийных устройств, секторов диска, узлов сети и т.п. Зачастую *адресное пространство* определяют, для краткости, как множество допустимых адресов (а не адресуемых объектов) вычислительной системы.

Аппаратно регистровая память (РОН и РВВ) выполнена на запоминающих устройствах, не входящих в состав СОЗУ (SRAM). Несмотря на это РОН и РВВ логически, с целью достижения единообразия машинных кодов команд и характера обращения к регистрам и ячейкам, включены, вместе с СОЗУ, в единое адресное пространство. Это означает, что доступ к ним можно осуществлять не только как к регистрам с соответствующими именами, но и как к ячейкам СОЗУ. Двойной доступ к регистрам существенно увеличивает гибкость программ.

Существует две конфигурации единого адресного пространства памяти AVR-контроллеров (рис.5.4). В конфигурации **A** младшие 32 адреса (\$0000...\$001F) соответствуют РОН, следующие 64 адреса (\$0020...\$005F)

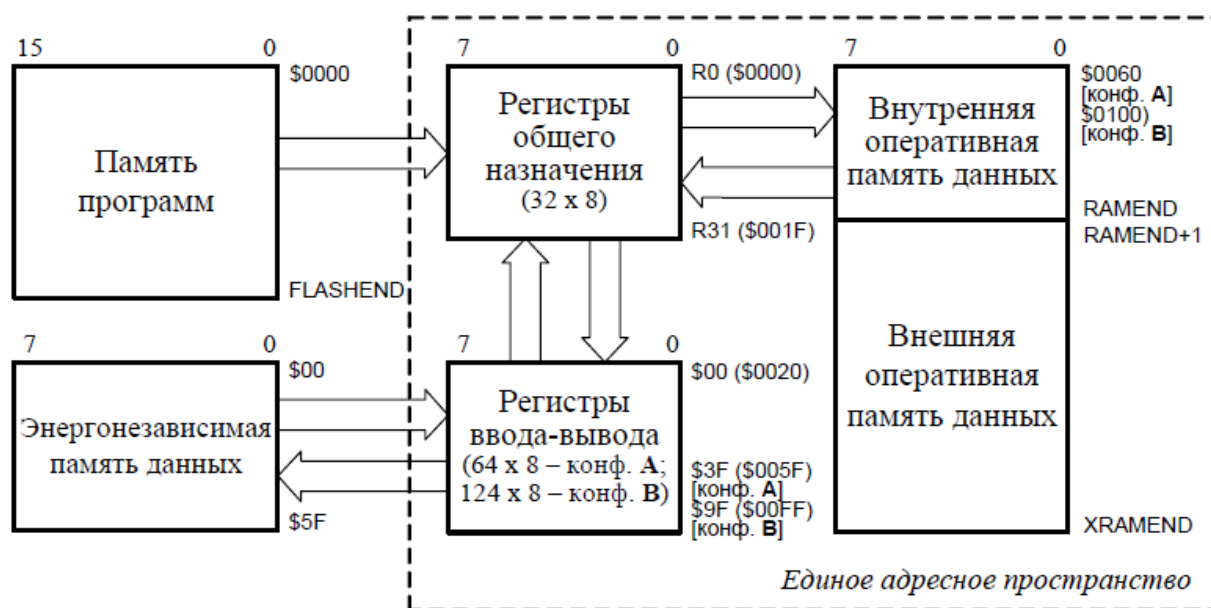


Рисунок 5.4 – Программная модель AVR-микроконтроллеров

занимают регистры ввода-вывода (периферийные регистры и специальные регистры процессора SPL, SPH, SREG), внутренняя оперативная память данных начинается с адреса \$0060. Т.е. записывая байт данных в ячейку памяти с адресом \$0000, вы на самом деле записываете его в регистр R0. Аналогично при чтении. В конфигурации **B** начиная с адреса \$0060 размещаются 160 дополни-

тельных регистров ввода-вывода; внутренняя оперативная память данных начинается с адреса \$0100. Конфигурация **A** используется в младших моделях микроконтроллеров (в частности, в МК ATmega16) и в некоторых старших моделях в режиме совместимости с моделями, снятыми с производства; конфигурация **B** – в старших моделях.

Для большей наглядности соответствие ячеек памяти и регистров для МК ATmega16 показано также на рис.5.5.

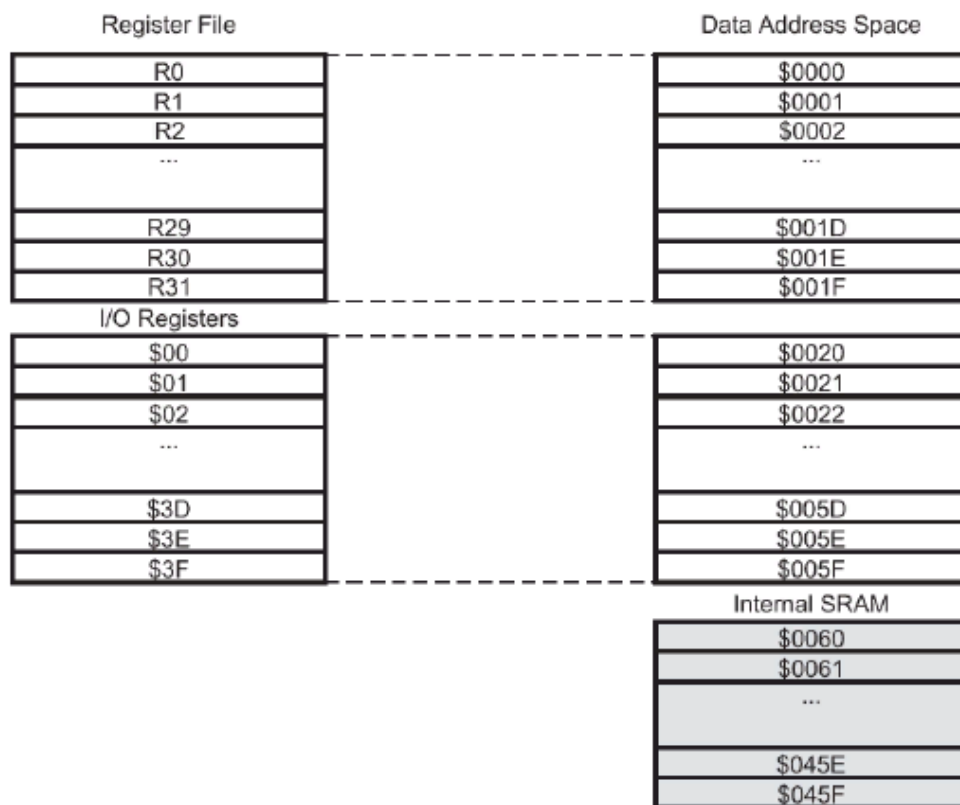


Рисунок 5.5 – Отображение регистровой памяти микроконтроллера ATmega16 на адресное пространство внутреннего СОЗУ данных (Internal SRAM)

Пространство ввода-вывода. Мы пояснили, что регистры ввода-вывода имеют штатные имена и, одновременно, адреса в СОЗУ. Наряду с этим для них организовано еще одно адресное пространство – *пространство ввода-вывода* (*пространство регистров ввода-вывода, пространство I/O – Input/Output*). Логически оно отделено от единого пространства СОЗУ и имеет собственные адреса в диапазоне \$00...\$3F (на \$20 меньше, чем адреса в пространстве СОЗУ) (см. рис.5.5). Различные ячейки (регистры ввода-вывода) этого пространства доступны через команды `in` и `out`, пересылающие данные между одним из регистров общего назначения и регистром ввода-вывода. К регистрам \$00...\$3F можно осуществить также побитовый доступ командами `cbi` и `sbi`. Значение отдельного бита этих регистров можно проверить командами ветвления `sbic` и `sbis`.

Итак, при использовании специальных команд `in`, `out`, `cbi`, `sbi`, `sbic` и `sbis` должны использоваться адреса \$00...\$3F или штатные имена

регистров ввода-вывода. А при доступе к ним как к ячейкам СОЗУ используются команды обмена с памятью данных, а к их адресам необходимо добавить \$20.

В последующих параграфах мы отдельно рассмотрим регистровую память и памяти программ и данных более подробно. Общие регистры (РОН), SREG и PC будут обсуждены в п.5.3, указатель стека SP и стек – в п.5.5, PWB (регистры ввода-вывода) – в главах 9-13.

5.3. Регистровая память

Все микроконтроллеры AVR, независимо от их вида, содержат, в соответствии с п.5.2, 32 регистра общего назначения процессорного ядра R0...R31 и от 64 до 480 (64 – у МК ATmega16) регистров ввода-вывода, представляющих периферийные устройства и процессорные регистры SPL, SPH и SREG. Кроме того, МК располагают программным счетчиком PC, который, в отличие от упомянутых регистров, не отображен на пространства СОЗУ и ввода-вывода. Он доступен лишь по записи и адресуется неявно посредством команд передачи управления.

Сдвоенные регистры общего назначения (регистровые пары), используемые для косвенной адресации СОЗУ, также имеют свои штатные имена X, Y, Z, при этом их младшие и старшие байты именуются как XL, XH, YL, YH и ZL, ZH соответственно (см. далее в этом параграфе). Указанные имена можно использовать при написании программ на языках программирования. Для этого в программу на языке ассемблера необходимо включить файл определений m16def.inc (с помощью директивы `.include "m16def.inc"`), а в программу на языке Си – файл 90s16.h.

Регистры общего назначения образуют так называемый регистровый файл, характеризующийся непосредственной связью с арифметико-логическим устройством МК и быстрым доступом. РОН предназначены для хранения обрабатываемых операндов: все арифметические и логические операции МК производятся только через РОН.

Большинство команд, использующих РОН, могут использовать любые регистры общего назначения. Исключение составляют пять команд, оперирующих с константами: `sbci`, `subi`, `cpi`, `andi`, `ori` и команда `ldi`, загружающая регистр константой. Эти команды работают только со второй половиной регистрового файла – R16...R31.

Кроме имен (которыми можно пользоваться в программах на ассемблере) каждому регистру, как мы сказали в п.5.2, присвоен адрес в пространстве данных, и эти адреса отображены на первые 32 адреса СОЗУ. Хотя регистровый файл физически размещен вне СОЗУ, подобная организация памяти дает гибкий доступ к регистрам.

Шесть из 32 регистров – R26...R31 можно использовать как три 16-разрядных адресных указателя в адресном пространстве данных. Один из адресных указателей (регистр Z) можно использовать для адресации таблиц в па-

мяти программ. Эти регистры обозначаются как X, Y, Z и определены согласно рис.5.6.

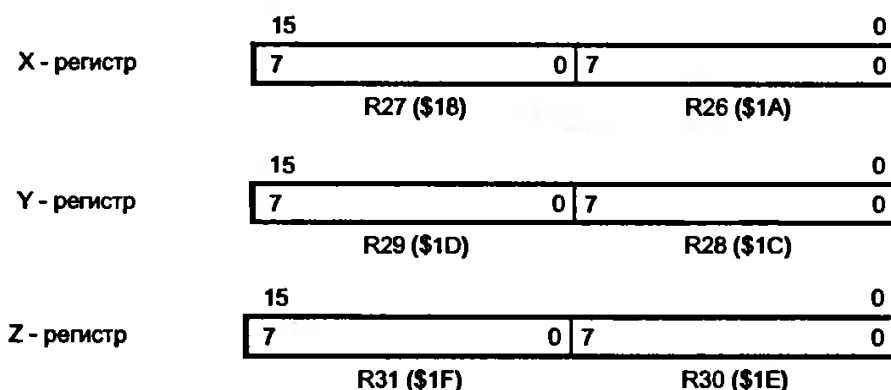


Рисунок 5.6 – Регистры X, Y, Z

При различных режимах адресации эти регистры могут использоваться для задания фиксированного адреса, для адресации с автоинкрементом или с автодекрементом.

Загрузка регистров X, Y, Z значениями 16-битных адресов осуществляется побайтно в младшую и старшую половины регистров. Для этого используются имена соответствующих половин (байтов): XL (X Low) и XH (X High) для X, YL и YH для Y, ZL и ZH для Z.

Таким образом, 16-битные регистры косвенной адресации X, Y, Z представляют собой регистровые пары – конкатенации (сочленения) 8-битных регистров:

$X = XH:XL = R27:R26;$

$Y = YH:YL = R29:R28;$

$Z = ZH:ZL = R31:R30.$

Регистры ввода-вывода обсуждаются в главах 9-13, а один из них (SREG) – ниже, другой (SP) – в п.5.5. В конце настоящего параграфа описывается также регистр процессора PC – программный счетчик (не относящийся, как уже указывалось, ни к POH, ни к PBB).

Регистр состояния (SREG – Status Register), называемый также *регистром флагов* (Flags), содержит слово состояния процессора (флаги). Он расположен по адресу \$3F пространства ввода-вывода (или \$5F пространства СОЗУ). Его формат представлен на рис.5.7.

Регистр определен следующим образом:

Бит 0 – C: флаг (признак) переноса (Carry). Этот флаг реагирует на перенос при сложении и на заём при вычитании.

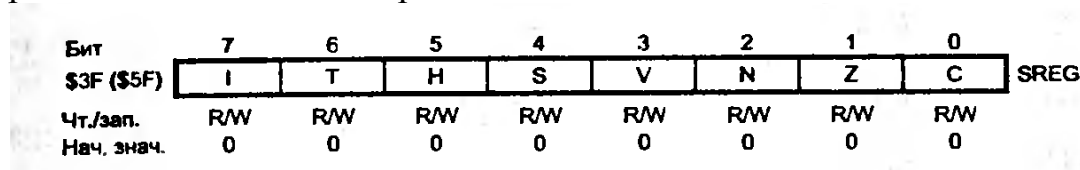


Рисунок 5.7 – Регистр состояния SREG

А именно, если при сложении чисел (происходящем по командам сложения `add` и др.) произошел перенос из старшего бита результата `d7` в следующий по старшинству, несуществующий бит `d8`, то признак переноса `C` устанавливается в состояние «1»: $C \leftarrow 1$. В противном случае, если переноса не возникло, признак `C` сбрасывается в ноль: $C \leftarrow 0$.

При вычитании чисел (происходящем по командам вычитания `sub` и др.) бит `C` сигнализирует не о переносе, а о заеме из несуществующего бита `d8` в старший бит `d7` уменьшаемого, т.е. устанавливается в этом случае в «1»: $C \leftarrow 1$. Если же заема не было, признак `C` сбрасывается в «0»: $C \leftarrow 0$. При вычитании флажок `C` служит, стало быть, признаком заема `Bv` (*Borrow*).

В целом флажок `C` реагирует на выход суммы/разности натуральных чисел из диапазона $0 \dots 255$. По этой причине его можно считать флажком переполнения в натуральной арифметике. Но так его не называют, чтобы не путать с флажком `V` переполнения в знаковой арифметике, а именуют, как уже сказано, флажком переноса/заема.

Флажок `C` фиксирует также значение выдвигаемого бита при сдвиге операнда с помощью команд сдвига.

Бит 1 – `Z`: флаг (признак) нулевого результата (*Zero*). Этот флаг индицирует нулевой результат различных арифметических и логических операций. Подробнее об этом можно прочитать в описании системы команд.

Бит 2 – `N`: флаг отрицательного результата (*Negative*). Этот флаг устанавливается в 1, если старший (знаковый) разряд (бит 7) равен 1, что означает, что результат отрицателен. Если, наоборот, результат положительный (т.е. знаковый бит результата 7 обратился в ноль), то `N` сбрасывается в 0. Иными словами, флаг `N` просто копирует значение старшего бита результата: $N \leftarrow d7$.

Флаг `N` дает правильный знак результата только в отсутствие арифметического переполнения ($V=0$). При этом в `Rd` образуется правильная сумма/разность. Если же переполнение есть ($V=1$), то в `N` ошибочный знак, а в `Rd` ошибочная сумма. Правильный знак всегда содержится во флаге `S`, а правильная сумма/разность – в `S_Rd` независимо от переполнения.

Следует отметить, что флаг `N` может быть установлен и в результате логических операций.

Бит 3 – `V`: флаг (признак) переполнения (*overflow*) в знаковой арифметике, «переполнения дополнительного кода». Он сигнализирует об арифметическом переполнении при сложении и вычитании целых чисел со знаком, записанных в дополнительном коде. Если арифметическое переполнение произошло, флаг `V` устанавливается в «1», если арифметического переполнения не было, `V` сбрасывается в «0».

Арифметическим переполнением называется выход суммы или разности целых знаковых чисел за пределы диапазона $-2^{n-1} \dots 2^{n-1} - 1$, где n – длина разрядной сетки в битах. Это максимальный диапазон чисел, представимых в n -разрядной сетке при помощи дополнительного кода. У МК AVR $n=8$. Поэтому арифметическое переполнение в AVR означает выход результата из диапазона

$-2^7 \dots 2^7 - 1$ (от -128 до $+127$), и об этом событии, как уже сказано, сигнализирует флаг V . (Дополнительные коды, а также арифметическое переполнение рассматриваются отдельно в Приложении А.)

Бит 4 – S : бит знака (Sign), $S = N \text{ XOR } V$. Бит S равен исключающему ИЛИ между флагами N (отрицательный результат) и V (арифметическое переполнение). Он содержит правильный знак результата сложения или вычитания 8-битных знаковых чисел в дополнительном коде независимо от наличия или отсутствия арифметического переполнения, а 9-битная конкатенация S_Rd , где Rd – 8-битный регистр-приемник выполнявшейся операции, дает всегда правильный результат этой операции. Именно, если после сложения или вычитания чисел флаг переполнения V устанавливается в 1, то результатом этой операции является 9-битное число в S_Rd , но не 8-битное в Rd (которое не выражает правильного результата). Старшим – знаковым битом результата в этом случае будет флаг S , а остальные 8 битов в Rd – цифровые.

Если, наоборот, $V=0$, то в S_Rd опять-таки содержится правильная сумма/разность, как и в случае $V=1$. Но при этом одновременно и в Rd находится правильная сумма/разность, т.е. результат уместается и в 8 битов приемника Rd . В этом случае старший бит $Rd(7)$ является знаковым и содержит правильный знак результата, а S просто совпадает с $Rd(7)$, служит его расширением. Остальные 7 битов результата $Rd(6-0)$ при $V=0$ являются цифровыми.

Коротко: в 9-битном получателе S_Rd всегда образуется правильная сумма/разность знаковых чисел в дополнительном коде независимо от наличия или отсутствия переполнения (независимо от значения флажка V), а в 8-битном получателе Rd правильная сумма/разность получается только при отсутствии переполнения (только при $V=0$).

Бит 5 – H : флаг половинного переноса (Half). Этот флаг индицирует перенос из младшей половины байта (из бита 3 в бит 4) при некоторых арифметических операциях. Подробнее об этом можно прочесть в описании системы команд.

Бит 6 – T : хранение копируемого бита (biT). Пользовательский флаг, применяемый в своих целях. Бит из регистра регистрового файла может быть скопирован в T командой `bst`, бит T может быть скопирован в бит регистрового файла командой `bld`.

Бит 7 – I : общее, или глобальное, разрешение прерываний (Interrupt). Для разрешения прерываний этот бит (флаг) должен быть установлен в единицу командой `sei`. Управление отдельными прерываниями производится регистрами маски прерываний – $GIMSK$ и $TIMSK$. Если бит $I=0$, независимо от состояния $GIMSK/TIMSK$ прерывания запрещены. Бит I обнуляется аппаратно после входа в прерывание или командой `cli` и восстанавливается в состояние 1 командой возврата из подпрограммы обработки прерывания `reti` для разрешения обработки последующих прерываний (или командой `sei`).

Программный счетчик (PC – Program Counter). В зависимости от модели МК регистр PC имеет от 10 до 17 разрядов, например 14 у $ATmega16$. PC содержит всегда адрес следующей по порядку, еще не исполненной команды про-

граммы, которую процессор будет выбирать из памяти программ и выполнять вслед за текущей, обрабатываемой в данный момент командой. Коротко: после исполнения очередной команды РС *содержит всегда адрес следующей команды для исполнения.*

Данное свойство достигается с помощью простого логического механизма, встроенного в аппаратуру процессора и поясняемого диаграммой на рис.5.8. Он сводится к трем следующим случаям модификации программного счетчика:

1) *При пуске* микроконтроллера (осуществляемого по внешнему или внутреннему сбросу; п.4.3) в РС заносится нулевой адрес:

$$PC \leftarrow 0$$

Поэтому процессорное ядро микроконтроллера запускается всегда с нулевой ячейки памяти программ.

2) *Естественное продвижение по программе.* После пуска процессор приступает к выборке и выполнению команд программы из памяти программ. После выборки каждого, 16-битного, слова команды содержимое РС автоматически и немедленно (еще до выполнения внутренних операций процессора в этом цикле выборки) увеличивается на 1:

$$PC \leftarrow PC + 1$$

Поэтому РС снова будет указывать на очередное слово программной памяти, подлежащее считыванию из памяти в следующем цикле чтения команды*, а после выборки всей команды – на следующую команду. Такое изменение РС и, равным образом, обусловленное им продвижение процессора по программе, называется *естественным*.

* Понятие цикла (машинного цикла, шинного цикла) см. в Приложении Б.

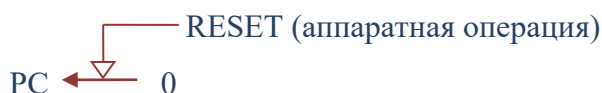
3) *Переходы по программе.* По командам передачи управления («перехода») в РС загружается адрес перехода, задаваемый этими командами:

$$PC \leftarrow \text{address},$$

т.е. «естественный» адрес в РС, продвинутый после выборки команды перехода, стирается и на его место записывается новое значение, задающее местоположение следующей команды для исполнения. Насильственная загрузка нового адреса в РС производится также во время прерывания программы (и, как уже было сказано, при сбросе). В результате выбирается не команда из ближайшей по ходу памяти ячейки, а команда, размещенная по адресу перехода, т.е. происходит «скачок», переход в новую точку программной памяти. Такое изменение РС и продвижение процессора по программе называется *принудительным*.

При отработке команд вызова подпрограмм (call и др.) «естественный» адрес из РС, получившийся после выборки call (и др.) из программной памяти и указывающий на соседнюю команду, расположенную сразу после call, предварительно запоминается в стеке в качестве адреса возврата из подпрограммы, и только после этого в РС заносится адрес перехода (см. также стек в п.5.5 и команды вызова в п.6.3). Аналогично обрабатываются прерывания (раздел 8).

- 1) При пуске (сбросе) микроконтроллера происходит обнуление РС:



- 2) При естественном продвижении по программе (при считывании очередного слова команды) содержимое программного счетчика увеличивается на 1:



- 3) При принудительном продвижении по программе в программный счетчик загружается адрес перехода:

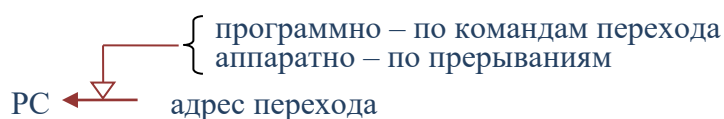


Рисунок 5.8 – Три типа операций, выполняемых с программным счетчиком

Из нашего изложения видно, что программный счетчик РС программно доступен, т.е. в него можно загружать новые значения с помощью машинных команд – команд передачи управления. Адресация программного счетчика в этих командах осуществляется неявно. Это означает, что имя РС не фигурирует в команде, а обращение к РС обеспечивается самим кодом операции команды. Исключение составляют записи команд на ассемблере вида `rjmp PC+k`, по которым к текущему значению РС прибавляется константа k (см. систему команд в главе 6). Но и в этом случае машинный код команды не содержит номера РС, а вовлекает РС в операцию по умолчанию.

5.4. Память программ и память данных

Память программ МК AVR (Flash ROM) объемом от 2 до 256 Кбайт (16 Кб у ATmega16) состоит из 16-битных ячеек. Она предназначена для хранения команд, управляющих работой микроконтроллера, а также часто используется для хранения таблиц констант, не меняющихся во время работы программы. Для пересылки байта из памяти программ в регистр общего назначения существует специальная команда – `lpm` (п.6.2). При использовании команды `lpm` адрес, по которому производится чтение, определяется содержимым индексного регистра Z. При этом старшие 15 битов содержимого регистра будут определять адрес слова (0...32 К), младший бит будет определять, какой из байтов будет прочитан: 0 — младший байт, 1 — старший байт.

В подавляющем большинстве моделей микроконтроллеров семейства Mega память программ логически разделена на две неравные части: область прикладной программы и область загрузчика. В последней может располагать-

ся специальная программа (загрузчик), позволяющая микроконтроллеру самостоятельно управлять загрузкой и выгрузкой прикладных программ. Если же возможность самопрограммирования микроконтроллера не используется, прикладная программа может располагаться и в области загрузчика.

После инициализации (*сброса*) микроконтроллера выполнение программы начинается с адреса \$0000. По этому адресу размещается команда перехода к инициализационной части программы. Эта команда называется *вектором сброса*. Начиная с адреса \$001 (модели с памятью программ 8 Кбайт и меньше) или \$0002 (остальные модели) памяти программ располагается таблица *векторов прерываний* (глава 8). Размер этой области зависит от модели микроконтроллера. Если прерывания не используются, то с пусковой ячейки \$0000 может начинаться сама программа (Приложение В).

Доступ к памяти программ осуществляется следующим образом: во время выполнения одной команды следующая команда (благодаря раздельным шинам памяти программ и данных) выбирается из памяти программ. Т.е. время выполнения текущей команды накладывается на время выборки следующей команды. Это дает возможность выполнять по одной команде за каждый машинный цикл. Это не относится, правда, к командам перехода (требующим от 2 до 5 циклов на выборку и выполнение), а также к командам сложения *adiw* и вычитания *sbiw* (2 цикла) (глава 6).

При помощи команд абсолютных и относительных переходов и вызова подпрограмм осуществляется доступ ко всему адресному пространству. Большая часть команд микроконтроллеров AVR имеет размер 16 разрядов – одно слово, остальные – 32 разряда – два слова. Так что каждый адрес в памяти программ содержит одну 16- или 32-разрядную команду.

При обработке прерываний и вызове подпрограмм адрес возврата запоминается в стеке. Стек размещается в оперативной памяти данных общего назначения (SRAM), его размер ограничен только размером доступной памяти SRAM и ее использованием в программе. Все программы пользователя должны инициализировать указатель стека (SP) сразу после запуска микроконтроллера, до того как вызываются подпрограммы и разрешаются прерывания. Исключения составляют микроконтроллеры, не имеющие оперативной памяти данных (SRAM), например AT90S1200. У этих микроконтроллеров реализован аппаратный стек глубиной 3. Это обязательно надо учитывать при написании для них программ.

Модуль прерываний имеет собственный управляющий регистр в пространстве ввода-вывода GICR, расположенный по адресу \$3B (\$5B), а также флаг глобального разрешения прерываний в регистре состояния SREG (адрес \$3F (\$5F)). Каждому прерыванию назначен свой вектор в начальной области памяти программ. Различные прерывания имеют приоритет в соответствии с расположением их векторов. По младшим адресам расположены вектора с большим приоритетом.

Программный счетчик, как было сказано, имеет ширину от 10 до 17 разрядов, что позволяет адресовать до \$20000 (128 К) слов памяти программ.

Способы занесения информации (т.е. программ) во флэш-память микроконтроллера рассмотрены в [2, 3, 7, 10, 24].

Память данных. Напомним, что программная модель памяти МК AVR трактует память данных как совокупность основной (оперативной и постоянной) памяти данных и регистровой памяти, к которым может добавляться еще внешняя память данных. Регистровая память и оперативная часть памяти данных исчерпывающе описаны в п.4.2, 4.3. Здесь мы обсудим детальнее постоянную (энергонезависимую) память данных.

Энергонезависимая память данных EEPROM. Для долговременного хранения данных, которые могут изменяться в процессе работы микроконтроллера, используют энергонезависимую память (EEPROM) объемом 512 байт (у ATmega16). Память EEPROM имеет обособленное адресное пространство, каждая ячейка содержит 8 разрядов. Данные в EEPROM могут быть записаны при программировании микроконтроллера. После выключения питания данные сохраняются.

Энергонезависимая память данных EEPROM предназначена для сохранения данных при отключении питания. К этой памяти может обращаться программа, считывая или записывая какие-либо данные. Кроме того, данные в эту память можно занести с помощью специального устройства – программатора, на этапе изготовления и программирования конструкции. Ее удобно использовать для хранения каких-либо констант.

Доступ к ячейкам EEPROM по адресам от \$000 до \$1FF осуществляется через регистры адреса EEARH:EEARL и регистр данных EEDR, управление через регистр EECR. Заметим, что при записи и чтении используется один и тот же регистр данных. Перед обращением к ячейке EEPROM необходимо в регистре EECR указать режим обращения (запись или чтение). При записи используются два бита управления EEMWE и EEWЕ (разрешение и сигнал записи), при чтении – бит разрешения EERE.

Для записи необходимо выполнить следующую последовательность действий:

- 1) записать адрес в регистр адреса;
- 2) записать байт данных в регистр данных;
- 3) записать в регистр управления EECR бит EEMWE=1, при этом бит EEWЕ должен быть равен 0. Бит EEMWE сохраняет единичное значение в течение 4-х тактов после установки, после чего аппаратно сбрасывается в 0;
- 4) инициировать цикл записи, установив бит EEWЕ в 1. Время записи зависит от напряжения питания и составляет 2,5-4 мс. После завершения записи бит EEWЕ аппаратно сбрасывается в 0.

Для чтения данных из ячейки EEPROM необходимо:

- 1) записать адрес в регистр адреса;
- 2) записать в регистр управления бит EERE=1.

Цикл чтения завершается поступлением байта данных в регистр EEDR. Бит EERE аппаратно сбрасывается в 0.

Ниже приведены примеры программирования операций записи и чтения из ячеек EEPROM.

Пример 1. Запись и чтение данных в отдельную ячейку EEPROM

EEWrite:

```
sbic  EECR,EEWE    ;ожидание завершения предыдущей
                    ; записи

rjmp  EEWwrite
ldi   temp,0x01    ; запись в указанную ячейку
out   EEARH,temp
ldi   temp,0xFE
out   EEARL,temp
ldi   temp,r0      ; байт данных из регистра r0
out   EEDR,temp
sbi   EECR,EEMWE   ; разрешение записи
sbi   EECR,EEWE    ; начать запись
```

EERead:

```
sbic  EECR,EEWE    ;ожидание завершения предыдущей
                    ; записи

rjmp  EERead
ldi   temp,0x01    ; чтение из ячейки 0x1FE
out   EEARH,temp
ldi   temp,0xFE
out   EEARL,temp
sbi   EECR,EERE
in    r1,EEDR      ; сохранение данных в регистре r1
```

Пример 2. Для записи/чтения массива перед началом цикла обращения необходимо получить старый адрес, хранимый в регистрах EEARH:EEARL, увеличив его на единицу, затем вернуть в те же регистры адреса новое значение и выполнить цикл обращения (из файла avr100.asm папки Appnotes AVR Studio 4):

```
.def  EEtmp  = r24   ; временный адрес младший байт
.def  EEtmph = r25   ; временный адрес старший байт
.def  EEedwr_s = r18 ; данные для записи
.def  EEedrd_s = r0  ; читаемые данные

EEWrite_seg:
sbic  EECR,EEWE
rjmp  EEWwrite_seg
in    EEtmp,EEARL
in    EEtmph,EEARH
```

```

    adiv    EEttmp, 0x01
    out     EEARL, EEttmp
    out     EEARH, EEttmp
    out     EEDR, EEdwr_s      ; данные для записи
    sbi     EECR, EEMWE
    sbi     EECR, EEWE

EERead_seg:
    sbic    EECR, EEWE
    rjmp    EERead_seg
    in      EEttmp, EEARL
    in      EEttmp, EEARH
    adiv    EEttmp, 0x01
    out     EEARL, EEttmp
    out     EEARH, EEttmp
    sbi     EECR, EERE
    in      EEdrd_s, EEDR      ; чтение данных

```

Для контроля памяти EEPROM при отладке программы нужно открыть окно Memory, выполнив команды меню View\Memory, и выбрать из выпадающего списка строку Eeprom. При необходимости данные в EEPROM при отладке программы можно загрузить ручным способом, выбрав нужную ячейку и двойным щелчком по ее значению открыв окно для записи нового значения.

Напомним, что все пространства памяти AVR (программ, оперативных данных и энергонезависимых данных) являются **линейными и непрерывными**: адреса пробегают линейный и последовательный (без пропусков) ряд значений 0, 1, 2, ..., max для каждого пространства.

5.5. Стек. Подпрограммы

Стек – это часть оперативной памяти (часть ОЗУ – оперативного запоминающего устройства), функционирующая по принципу: последний записанный в нее элемент считывается первым – «последним зашел, первым вышел», «last input, first output» (LIFO). То есть, последнее сохраненное в стеке значение будет первым значением, которое вы получите при чтении из стека. Стек можно представить себе как некий ящик без крышки, в который могут помещаться однотипные объекты (рис.5.9) [22]. Он «растет» снизу вверх (в сторону младших адресов) при записи в него и «укорачивается» сверху вниз (в сторону старших адресов) при чтении из него. Стек необходим для запоминания адресов возврата из подпрограмм и прерываний. Его также можно использовать для временного хранения данных – промежуточных операндов.

Некоторые микроконтроллеры не имеют оперативной памяти данных (SRAM), например AT90S1200. У этих микроконтроллеров реализован аппа-

ратный стек глубиной 3. Это обязательно надо учитывать при написании для них программ.

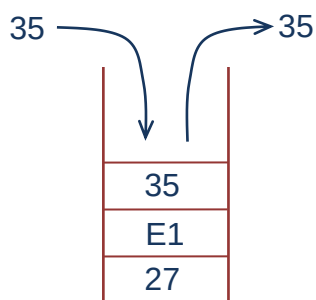


Рисунок 5.9 – Принцип действия стека

В микроконтроллерах AVR стек располагается в памяти данных SRAM (рис.5.10). Он должен задаваться программой до вызова подпрограмм и обработки разрешенных прерываний, и до обмена с ним переменными.

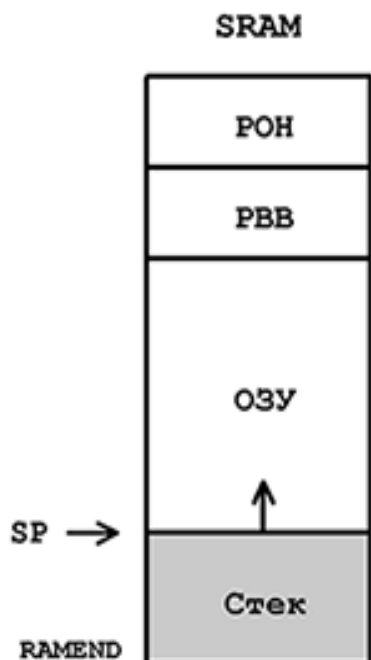


Рисунок 5.10 – Расположение стека в памяти данных SRAM.

Стек растет снизу вверх при записи в него и укорачивается сверху вниз при чтении из него.

POH – регистры общего назначения;

PBB – регистры ввода-вывода;

ОЗУ – оперативное запоминающее устройство;

RAMEND – максимальный адрес («конец»)

ОЗУ

Для адресации стека служит специальный регистр – **указатель стека SP** (Stack Pointer). Микроконтроллеры AVR оснащены 16-разрядным указателем стека, размещенным на двух 8-разрядных регистрах пространства I/O. Это регистры SPL (Stack Pointer Low) – младший байт указателя стека и SPH (Stack Pointer High) – старший байт указателя стека. Они находятся по адресам \$3D и \$3E пространства в/в (а также по адресам \$5D и \$5E памяти данных) соответственно (рис.5.11). Поскольку микроконтроллеры ATmega16 и выше поддерживают объем SRAM от 640 байт и до 64 Кбайт, то используются все 16 разрядов указателя стека. В младших моделях задействован только регистр SPL, способный адресовать до 256 байт стека.

Регистр SP постоянно указывает на первую свободную ячейку стека, содержит ее адрес (как показано на рис.5.10). Эта ячейка именуется **вершиной**

стека TOS (Top Of Stack). Она располагается над последней занятой ячейкой стека (имеющей адрес на 1 больший, чем вершина стека, т.е. SP+1).

Биты		7	6	5	4	3	2	1	0	
Адреса	\$3D (\$5D)	SP7	SP6	SP5	SP4	SP3	SP2	SP1	SP0	SPL
	\$3E (\$5E)	SP15	SP14	SP13	SP12	SP11	SP10	SP9	SP8	SPH
		15	14	13	12	11	10	9	8	
Чтение/Запись		R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
		R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Начальное состояние		0	0	0	0	0	0	0	0	
		0	0	0	0	0	0	0	0	

Рисунок 5.11 – Указатель стека SP

Со стеком разрешены **три операции**: 1) инициализация (начальная установка), 2) запись в стек и 3) чтение из стека. Рассмотрим сначала запись и чтение в/из стека, а потом инициализацию.

Запись в стек всегда происходит в вершину стека, т.е. по адресу, берущемуся из SP. После записи указатель стека автоматически декрементируется (уменьшает свое содержимое) на 1: $SP \leftarrow SP-1$ (рис.5.12). В результате SP снова указывает на вершину стека, но уже поднявшуюся вверх, в сторону младших адресов.

Чтение из стека всегда происходит из ячейки, расположенной под вершиной стека, т.е. имеющей адрес на 1 больший, чем содержащийся в SP. Для этого перед чтением SP автоматически инкрементируется на 1: $SP \leftarrow SP+1$, и потом производится чтение байта (рис.5.12). И опять-таки после чтения регистр SP будет указывать на вершину (свободную ячейку) стека, опустившуюся на этот раз вниз в направлении старших адресов.

Замечание. Смысл содержимого регистра SP у микроконтроллеров AVR и процессоров IA-32, AMD64 несколько отличается. У последних регистр SP (и его расширения ESP, RSP) указывает не на первую свободную, а на последнюю занятую ячейку стека, которая и называется вершиной стека. Из нее производится чтение. А первая свободная, куда производится запись байта, находится над вершиной стека по адресу SP-1.

Как понятно из рис.5.12, запись в стек AVR производится командами `push Rs` (вталкивание в стек, запись операнда) и `rcall/call/ecall/icall/eicall` (вызов подпрограммы с предварительным сохранением в стеке адреса возврата из нее), а также при прерываниях (аппаратно сохраняющих в стеке адрес возврата). Чтение из стека – командами `pop Rd` (выталкивание из стека, чтение операнда) и `ret` и `reti` – (возврат из подпрограммы и из прерывания, т.е. извлечение из стека адрес возврата).

Именно: когда микропроцессор встречается одну из инструкций вызовов `rcall/ call/ecall/icall/eicall`, то под управлением этой инструкции адрес следующего за ней слова – адрес возврата (содержащийся на тот мо-

мент в программном счетчике PC после выборки инструкции) автоматически копируется в стек. В момент выхода из подпрограммы по команде `ret` или `reti` адрес возврата восстанавливается из стека в программный счетчик, при этом `reti` восстанавливает еще в «1» флажок разрешения прерываний I. В моделях с объемом памяти программ 128 и 256 к/слов для сохранения PC в стеке потребуется 3 байта, для всех остальных – 2 байта. При сохранении каждого байта содержимое SP уменьшается на единицу, а при восстановлении, соответственно увеличивается. (См. также п.6.3.)

Запись в стек

Действия процессора
по команде `push Rn`:

$m(SP) \leftarrow Rn$
 $SP \leftarrow SP-1$

Действия процессора
при вызове подпрограмм
по командам группы `call *`
и прерываниям:

$m(SP) \leftarrow PC_{CH}$
 $SP \leftarrow SP-1$
 $m(SP) \leftarrow PC_{CL}$
 $SP \leftarrow SP-1$
 $PC \leftarrow \text{адрес перехода}$

Rn – общий регистр процессора.

$m(SP)$ – ячейка памяти данных (темогу), адрес которой задается регистром SP.

PC_{CL} и PC_{CH} – младший и старший байты программного счетчика PC: $PC = PC_{CH}:PC_{CL}$, образовавшиеся после выборки команды.

* Группа команд вызова подпрограмм: `rcall/call/ecall/icall/eicall`.

При выполнении команд группы `call` непосредственно перед занесением в PC адреса перехода PC содержит адрес возврата (адрес команды после `call`), запоминаемый в стеке. Аналогично при прерываниях.

** По команде `reti` еще устанавливается в «1» флажок I

Чтение из стека

Действия процессора
по команде `pop Rn`:

$SP \leftarrow SP+1$
 $Rn \leftarrow m(SP)$

Действия процессора
при возврате из подпрограмм
и прерываний по командам
`ret` и `reti**`:

$SP \leftarrow SP+1$
 $PC_{CL} \leftarrow m(SP)$
 $SP \leftarrow SP+1$
 $PC_{CH} \leftarrow m(SP)$
 $PC = \text{адрес возврата}$

Рисунок 5.12 – Операции со стеком AVR

Стек AVR, помимо сохранения адресов возврата, имеет, как мы сказали, еще одно очень важное предназначение. Он позволяет сохранять любые данные специально предназначенными для этого командами `push Rs` (загрузка в стек) и `pop Rd` (выгрузка из стека). Каждый раз при выполнении `push Rs` содержимое Rs копируется в стек, после чего SP уменьшается на единицу. При выполнении `pop Rd` значение SP увеличивается на единицу и затем содержимое ячейки стека, на которую стал указывать SP, восстанавливается в Rd.

В частности, через стек очень просто можно обменять содержимое регистров местами: втолкнуть в стек содержимое одного регистра командой `push R1`, заменить его на второе значение: `mov R1, R2` и вытолкнуть первое значение из стека командой `pop R2`.

Инициализация стека. Программист должен самостоятельно инициализировать стек (определить его местоположение, объявить, назначить, задать стек) перед его использованием. Для этого в регистры SPL и SPH необходимо загрузить адрес вершины стека. Стек состоит из смежных ячеек памяти и в принципе может располагаться в любом месте SRAM. Но с точки зрения максимальной его глубины, вершину стека нужно поместить в самом конце SRAM, как это изображено на рис.5.10. Это означает, что в качестве адреса вершины стека нужно взять максимальный адрес ОЗУ. Этот адрес хранится в константе RAMEND, которая определена в подключаемом файле для данной модели микроконтроллера. Для микроконтроллера ATmega16 – в файле m16def.inc (который надо подключить к программе с помощью директивы `.include «m16def.inc»`).

Младший байт константы RAMEND надо загрузить в регистр SPL, а старший – в регистр SPH. Это установит указатель стека на конец ОЗУ. Сделать это можно, например, так:

```
.def temp = r16
```

```
ldi temp, low(RAMEND) ;Указатель стека указывает
out SPL, temp        ;на последний адрес ОЗУ
ldi temp, high(RAMEND)
out SPH, temp        ;!!! Не для всех моделей МК
```

Обратите внимание на последнюю строку. Поскольку регистр SPH используется не во всех моделях серии AVR, то в некоторых случаях (например, для ATtiny13A) это приведёт к ошибке компиляции (в моделях с объёмом ОЗУ до 128 байт - ATtiny13A имеет память 64 байта).

Напоминаем, что этот код надо разместить до вызова каких-либо подпрограмм и команд `push` и `pop`.

В диапазоне адресов SRAM между регистрами ввода-вывода (PBB) и текущим положением вершины стека (TOS) размещаются переменные прикладной программы. Поэтому очень важно предварительно оценить максимальный размер стека (глубину стека). Может случиться так, что вершина стека поднимется слишком высоко и начнет «затирать» пользовательские данные, а это одна из самых сложно-выявляемых ошибок!

После инициализации стека можно обращаться к нему – производить запись и чтение из него данных.

Роль стека. Использование стека позволяет простыми аппаратными средствами осуществить:

- а) удобное временное хранение промежуточных результатов счета, не задумываясь об адресах хранения;
- б) переход к подпрограммам и возврат к основной программе (в т.ч. в случае нескольких вложенных подпрограмм);
- в) переход к прерывающей подпрограмме и возврат к прерванной программе (раздел 8).

Итак:

Инициализация стека производится командами `ldi` и `out`.

Запись в стек производится командами `push`, `call`, `rcall`, `icall` и в момент прерывания.

Чтение из стека производится командами `pop`, `ret` и `reti`.

Подпрограммы. Вызов и возврат из подпрограмм. *Подпрограммой* (англ. subroutine) называют поименованную последовательность команд, которую можно вызывать – запускать на счет (с помощью команд `call`, ...) из разных мест программы, при этом возврат из подпрограммы происходит всегда в ту точку программы, на ту ее команду, которая следует за командой последнего вызова («на после `call`»). Для возврата служит команда возврата `ret` (return) (для прерываний – `reti`), которую помещают в конце подпрограммы (рис.5.13).

Подпрограмма имеет имя (метку, заканчивающуюся двоеточием), которым, на языке ассемблера, помечается точка входа * в подпрограмму – команда, с которой подпрограмма запускается. Имя (без двоеточия) надо указывать в адресной части команды `call` при вызове подпрограммы:

`call <имя>`.

* *Точка входа* – это адрес оперативной памяти, с которого начинается выполнение программы, а также и сама команда, расположенная по этому адресу.

Основная программа

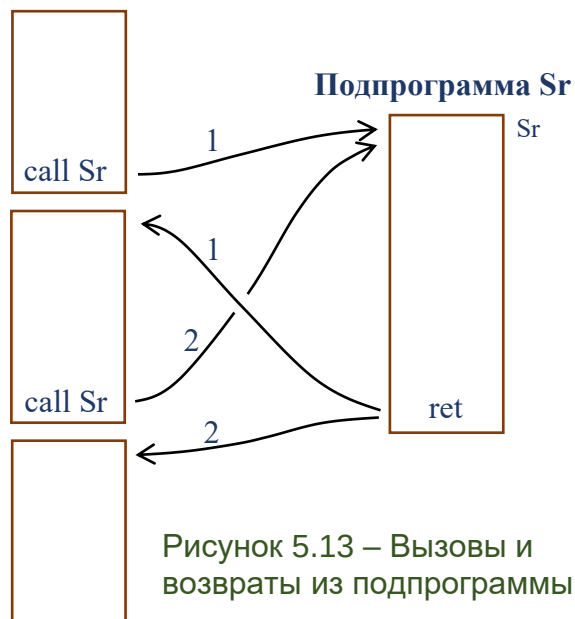


Рисунок 5.13 – Вызовы и возвраты из подпрограммы

По отношению к подпрограмме остальную часть программы принято называть основной программой. Подпрограммы служат для многократного выполнения однотипных действий, они значительно экономят память, так как позволяют описать эти действия однократно, а затем вызывать их по мере необходимости.

Подпрограмму можно размещать (в основной программе), в принципе, где угодно. Но при этом надо понимать, что сама по себе подпрограмма не должна выполняться, а может выполняться лишь тогда, когда к ней обратятся. Поэтому размещать ее надо так, чтобы на нее случайно не попало управление. Если же подпрограмму просто вставить в общий поток команд, то микропроцессор будет воспринимать команды подпрограммы как часть этого потока. Учитывая это обстоятельство, подпрограммы размещают обычно либо в конце сегмента команд, либо в самом начале этого сегмента – перед той командой, с которой должно начинаться выполнение программы. В больших программах (например, для процессоров IA-32 и AMD64) подпрограммы нередко размеща-

ют в отдельном сегменте команд. (В МК AVR сегмент команд единственный.) Если имеется несколько подпрограмм, то их обычно размещают рядом.

Второй вопрос – как описывать подпрограмму? В общем случае группу команд, образующих подпрограмму, можно никак не выделять в тексте программы. Однако в языках ассемблера (как и в языках программирования высокого уровня) подпрограммы принято оформлять специальным образом – в виде процедур или в виде функций.

Подпрограммы-функции и подпрограммы-процедуры. *Функция* – это подпрограмма, способная возвращать некоторое значение, т.е. ее вызов является, с точки зрения языка программирования, выражением. Поэтому он может использоваться только в других выражениях или в качестве правой части присваивания. *Процедура* – это независимая подпрограмма, которая осуществляет управление какими-то устройствами или выполняет какие-либо вычисления. Если производятся вычисления, то результат помещается в глобальную переменную, чтобы этим результатом могла воспользоваться другая подпрограмма или основная программа.

Пример использования функции вычисления синуса угла x :

$y := \sin(x);$

Пример вызова процедуры поиска максимального из чисел a и b :

Poisk(a,b,res);

Никак не оформленная подпрограмма имеет вид

; M – подпрограмма с именем M.

M: ...

ret

Ее вызов осуществляется командой

call M

Для описания последовательности команд в виде процедуры в языке ассемблера (процессоров IA-32/64, но не микроконтроллеров AVR) используются две директивы: PROC (procedure) и ENDP (end of procedure). Описание подпрограммы в виде процедуры выглядит так:

<имя процедуры> PROC <параметр (тип процедуры)>

<тело процедуры>

<имя процедуры> ENDP

Как видно, перед телом процедуры (ее командами) ставится директива PROC, а за ним – директива ENDP. В обеих этих директивах указывается одно и то же имя – имя, которое мы дали подпрограмме-процедуре. Имя процедуры считается меткой, считается, что оно метит первую команду процедуры (хотя за ним и не ставится двоеточие). Например, имя процедуры можно указать в команде вызова, и тогда будет осуществлен переход на первую команду процедуры.

Макрокоманда. *Макрокомандой* (макросом, macro command) называют поименованную группу команд. Для именования служит парная директива MACRO – ENDM в Ассемблере процессоров IA-32/64 и .MACRO – .ENDMMACRO в Ассемблере AVR. Ниже приведена структура макроса с именем ABC:

ABC MACRO <формальные параметры через запятую>

<тело макроса>

ENDM

После объявления макроса его имя (в нашем примере ABC) можно записывать в разных точках программы любое число раз:

ABC

...

ABC

...

ABC

Каждый раз транслятор вставляет вместо имени всю группу команд макроса (тело макроса).

Макрокоманда в сравнении с подпрограммой. Макрокоманда похожа на подпрограмму, но при трансляции ее тело вставляется в программу столько раз, сколько раз макрокоманда вызывается, а подпрограмма содержится в программной памяти в единственном экземпляре. Поэтому использование подпрограмм экономит место в памяти. Однако вызов и возврат из подпрограмм занимает дополнительное время процессора, затрачиваемое на выполнение команд `call` и `ret`, которых нет в макрокомандах. Коротко: подпрограмма экономит место, а макрокоманда – время.

Глава 6. Система команд AVR

Система команд микроконтроллеров Atmel семейства AVR насчитывает 133 ассемблерных мнемонических команд (которым соответствует 76 машинных команд). Одной из отличительных особенностей микроконтроллеров AVR является то, что почти все команды выполняются за 1 тактовый цикл. Исключение составляют команды передачи управления (от 2 до 5 циклов), а также команды сложения `adiw` и вычитания `sbiw` (2 цикла). Это существенно увеличивает производительность микроконтроллера даже при относительно невысокой тактовой частоте. [1, 6, 10, 17, 21]

6.1. Команды процессора

Классификация команд. Все множество команд микроконтроллеров AVR, как и любых МК и процессоров, можно разбить на три категории:

- команды обработки данных:
 - пересылки данных,
 - арифметических операций,
 - логических операций,
 - сдвигов,
 - операций с битами;
- команды передачи управления:
 - перехода абсолютного и относительного,
 - безусловного и условного (ветвления),
 - вызова и возврата из подпрограмм,
 - вызова и возврата из прерываний;
- команды изменения состояния системы (управления системой).

Понятие команды процессора. Напомним (п.1.2, 2.1.), что *машинной командой* (*машинной инструкцией*) называется приказ для процессора на выполнение какой-нибудь простой операции, например: сложить 2 числа; проверить, было ли переполнение при сложении; передать порцию данных из регистра процессора в ОЗУ. Более сложные действия производятся процессором посредством выполнения группы команд, т.е. *программы*.

Структура команды любого процессора, включая AVR, показана в общем виде на рис.6.1. [16] Команда содержит код выполняемой ею операции (КОП – Код ОПерации) и информацию о данных, участвующих в операции. КОП сообщает процессору, что нужно делать: переслать информацию, сложить два числа, осуществить переход. Поэтому КОП называют часто командой, как и полный формат команды. Информация о данных определяет операнды или адреса операндов в памяти или в регистрах периферийных устройств или номера рабочих регистров процессора, содержащих операнды (для команд обработки данных и изменения состояния системы); или адреса следующих для исполнения команд – адреса переходов (они же – адреса передачи управления) (для ко-

манд передачи управления). Команда назначает также (явно или неявно) адрес, по которому помещается результат операции.



Рисунок 6.1 – Общая структура команды

Часть команды, содержащая КОП, называется *операционной*, а часть, содержащая операнды или адреса – *адресной*. Операционная часть обязательна для любой команды, а адресная имеется не у всех команд. Например, команда `sleep` не имеет адресной части по определению, так как она не оперирует ни с какими операндами, а останавливает работу процессора, переводя его при этом в режим пониженного энергопотребления. Команда возврата `ret`, хотя и оперирует с операндами, но адресует их неявно (при помощи регистра указателя стека), поэтому тоже не имеет адресной части, т.е. тоже является безадресной.

Команды обработки данных и изменения состояния системы не меняют естественный порядок выполнения команд – после них по умолчанию, в соответствии с их кодом операции, выполняется всегда команда из очередной, соседней с ними ячейки программной памяти с бóльшим адресом. Команды передачи управления, наоборот, изменяют порядок выполнения команд. Они заставляют процессор перейти к отработке любой другой команды, адрес которой (адрес перехода, адрес передачи управления) ими задается (явно или неявно).

Числа, входящие в адресную часть команд, – непосредственные операнды, адреса операндов или адреса переходов – это есть то, что мы называем *константами*. А числа, располагающиеся вне команд, вне программы (вне программной памяти), называются *переменными*. Они находятся в памяти данных.

Размер команд AVR. Большинство команд микроконтроллеров AVR занимает 1 ячейку памяти программ (16 бит), остальные – 2 ячейки (32 бита) – это команды, у которых одним из операндов является 16-битный адрес.

Содержание команды. Мы показали, что каждая команда сообщает процессору:

- какую операцию он должен выполнить;
- какие данные (операнды) потребуются (и потребуются ли) для этого, где они находятся и как их вызвать;
- где искать следующую команду.

Адресация команд и данных. Поэтому возникает необходимость *адресации* и операндов и команд, т.е. указания в команде операндов или адресов операндов или команд.

Режимы (способы) адресации. Способ задания операнда в команде или адреса следующей команды для исполнения (адреса перехода в команде пере-

дачи управления) называется *режимом адресации* (операнда или перехода соответственно). В зависимости от наличия или отсутствия у команды адресной части режимы адресации бывают *явные* и *неявные*, а в зависимости от ранга (от степени вложенности) адреса – *непосредственные*, *прямые* и *косвенные*.

Здесь мы рассмотрим только режимы адресации операндов (для команд обработки данных и изменения состояния системы), а адресацию следующих команд для исполнения (для команд передачи управления) обсудим в п.6.3.

Явная и неявная адресация. Если команда в своей адресной части содержит буквальные сведения об операнде или его адресе или адресе результата операции или о месте, где хранится адрес, (об адресе адреса), то такая адресация называется *явной*. Если же адресное поле для операнда или адреса в команде отсутствует, а операнд/адрес подразумевается, т.е. предопределен, заранее известен, принимается по умолчанию, то такая адресация называется *неявной*. При неявной адресации операнд/адрес фактически задается кодом операции команды.

Например, команда `ldi R16, 5` – «загрузить 5 в регистр R16» использует явную адресацию R16 и явное задание непосредственного операнда (числа пять). Команда `inc R1` – «инкрементировать (увеличить на единицу) содержимое R1» адресует регистр R1 явно, а единицу – неявно, по умолчанию. В команде `ret` адресация перехода (через стек) неявная, так как в адресной части команды нет указания об адресе перехода (как и вообще нет самой адресной части).

Непосредственная, прямая и косвенная адресация. Адресация называется *непосредственной* («ссылкой, адресацией 0-го ранга»), если адресная часть команды содержит сам операнд (при явной адресации); или этот операнд подразумевается, принимается по умолчанию (при неявной адресации). Соответственно такой операнд называется *непосредственным операндом*. При непосредственной адресации не требуется обращения к памяти для выборки операнда и ячейки памяти для его хранения. Это способствует уменьшению времени выполнения программы и занимаемого ею объема памяти. Непосредственная адресация удобна для хранения различного рода констант.

В иных случаях адресная часть содержит не операнд, а его адрес; или этот адрес подразумевается кодом операции при неявной адресации. Такая адресация называется *прямой* («ссылкой 1-го ранга»).

И, наконец, если код команды определяет (явно или неявно) адрес ячейки памяти или имя регистра процессора, в котором находится не операнд, а его адрес (называемый *указателем*), то такая адресация называется *косвенной* («ссылкой 2-го ранга»). При косвенной адресации задается, как мы видим, не адрес операнда, а *где* его взять, в *каком* регистре или ячейке памяти он находится, т.е. задается *адрес адреса* («адрес 2-го ранга»).

Примеры.

`ldi R20, 7` – «занести число 7 в регистр R20». В данной команде задается сам операнд-источник – число 7, являющийся, следовательно, непосредственным операндом, а способ его задания – *непосредственной адресацией*.

`lds R20, 100` – «передать в регистр R20 содержимое 100-й ячейки памяти из ОЗУ». Здесь операнд-источник адресуется прямо – указанием его адреса в памяти (числа 100). Это – *прямая адресация* операнда.

`ld R20, X` – «прочитать в R20 содержимое ячейки памяти данных, адрес которой содержится в регистре X». Эта команда адресует источник (ячейку памяти) косвенно: она указывает, где взять адрес ячейки – в регистре X. Тут мы имеем дело с *косвенной адресацией* операнда-источника. Регистр X (ну или его содержимое) является указателем на операнд. Если предварительно занести в X какое-нибудь число, например 100 (двумя командами `ldi`), то будет прочитана 100-я ячейка ОЗУ, как и в предыдущем примере.

И во всех приведенных примерах операнд-приемник – регистр R20 адресуется прямо – указанием его имени (*прямая адресация* операнда-приемника).

Косвенная адресация широко применяется в ЭВМ, имеющих короткое машинное слово, (например, в микроконтроллерах) для преодоления ограничений короткого формата команд (обычно используется косвенная регистровая адресация, при которой адрес задается регистром процессора). Кроме того, при помощи косвенной адресации легко организовывать обработку больших массивов данных. Занеся один раз начальный адрес массива (адрес его первого элемента) в регистр-указатель, можно переходить к следующим элементам, инкрементируя этот адрес. Инкрементация производится в цикле программы с числом повторений, равным числу элементов массива. Это дает возможность обрабатывать все элементы с помощью короткого программного цикла. Если же применять прямую адресацию элементов, то приходилось бы записывать в программе столько команд обращения к памяти, сколько элементов содержится в массиве, указывая каждый раз адрес нового элемента. Если массив содержит, к примеру, 1000 элементов, то в программе пришлось бы записать 1000 команд с тысячами разными адресами. Без применения косвенной, «динамической» адресации обработать такой длинный массив практически невозможно. Прямая, «статическая» адресация не годится для этой цели.

Далее приведен набор команд процессоров AVR, более детальное описание их можно найти в описании ассемблера на официальном сайте (на английском языке, информация обновляется) [25].

6.2. Команды обработки данных

Команды обработки данных представлены в таблицах 6.1 – 6.4, где приводится мнемоника команд, их название, действие (с помощью формул – так называемых «псевдокодов») и реакция флажков. Аналогично описываются и другие категории команд в п.6.3, 6.4.

Таблица 6.1 – Команды пересылки данных

Мнемоника	Название	Действие	Флаги
MOV Rd, Rs	Пересылка между РОН	$Rd \leftarrow Rs$	-
MOVW Rd, Rs	Пересылка 2-байтных значений	$Rd+1:Rd \leftarrow Rs+1:Rs$	-
LDI Rd, K *	Загрузка константы в РОН	$Rd \leftarrow K$	-
LD Rd, X	Косвенное чтение из ОЗУ	$Rd \leftarrow [X]$	-
LD Rd, X+	Косвенное чтение с постинкрементом	$Rd \leftarrow [X], X \leftarrow X+1$	-
LD Rd, -X	Косвенное чтение с преддекрементом	$X \leftarrow X - 1, Rd \leftarrow [X]$	-
LD Rd, Y	Косвенное чтение из ОЗУ	$Rd \leftarrow [Y]$	-
LD Rd, Y+	Косвенное чтение с постинкрементом	$Rd \leftarrow [Y], Y \leftarrow Y+1$	-
LD Rd, -Y	Косвенное чтение с преддекрементом	$Y \leftarrow Y - 1, Rd \leftarrow [Y]$	-
LDD Rd, Y+q	Косвенное относительное чтение	$Rd \leftarrow [Y + q]$	-
LD Rd, Z	Косвенное чтение из ОЗУ	$Rd \leftarrow [Z]$	-
LD Rd, Z+	Косвенное чтение с постинкрементом	$Rd \leftarrow [Z], Z \leftarrow Z+1$	-
LD Rd, -Z	Косвенное чтение с преддекрементом	$Z \leftarrow Z - 1, Rd \leftarrow [Z]$	-
LDD Rd, Z+q	Косвенное относительное чтение	$Rd \leftarrow [Z + q]$	-
LDS Rd, k	Прямое чтение из ОЗУ	$Rd \leftarrow [k]$	-
ST X, Rs	Косвенная запись в ОЗУ	$[X] \leftarrow Rs$	-
ST X+, Rs	Косвенная запись с постинкрементом	$[X] \leftarrow Rs, X \leftarrow X+1$	-
ST -X, Rs	Косвенная запись с преддекрементом	$X \leftarrow X - 1, [X] \leftarrow Rs$	-

Продолжение табл.6.2.1. Команды пересылки данных

Мнемоника	Название	Действие	Флаги
ST Y, Rs	Косвенная запись в ОЗУ	$[Y] \leftarrow R_s$	-
ST Y+, Rs	Косвенная запись с постинкрементом	$[Y] \leftarrow R_s, Y \leftarrow Y+1$	-
ST -Y, Rs	Косвенная запись с преддекрементом	$Y \leftarrow Y-1, [X] \leftarrow R_s$	-
STD Y+q, Rs	Косвенная относительная запись	$[Y + q] \leftarrow R_s$	-
ST Z, Rs	Косвенная запись в ОЗУ	$[Z] \leftarrow R_s$	-
ST Z+, Rs	Косвенная запись с постинкрементом	$[Z] \leftarrow R_s, Z \leftarrow Z+1$	-
ST -Z, Rs	Косвенная запись с преддекрементом	$Z \leftarrow Z - 1, [Z] \leftarrow R_s$	-
STD Z+q, Rs	Косвенная относительная запись	$[Z+q] \leftarrow R_s$	-
STS k, Rs	Прямая запись в ОЗУ	$[k] \leftarrow R_s$	-
LMP	Загрузка данных из памяти программ	$R_0 \leftarrow \{Z\}$	-
LMP Rd, Z	Загрузка данных из памяти программ	$R_d \leftarrow \{Z\}$	-
LMP Rd, Z+	Загрузка данных из памяти программ	$R_d \leftarrow \{Z\}, Z \leftarrow Z+1$	-
SMP	Запись в память программ	$\{Z\} \leftarrow R_1:R_0$	-
IN Rd, P	Пересылка из PBB в РОН	$R_d \leftarrow P$	-
OUT P, Rs	Пересылка из РОН в PBB	$P \leftarrow R_s$	-
PUSH Rs	Сохранение байта в стеке	$Stack \leftarrow R_s$	-
POP Rd	Извлечение байта из стека	$R_d \leftarrow Stack$	-
$0 \leq d \leq 31, 0 \leq s \leq 31, 0 \leq K \leq 255, 0 \leq k \leq 65535, 0 \leq q \leq 63, 0 \leq P \leq 63$; * В команде LDI: $16 \leq d \leq 31$; Rd – регистр-назначение, регистр-приемник (destination register); Rs – регистр-источник (source register); $\leftarrow$ – присвоить, переслать; K – константа, непосредственный операнд, например число (number) или строка (string); k – имя переменной или адрес ячейки памяти данных; [k] – значение переменной k, содержимое ячейки k; X, Y, Z – 16-битные регистры процессора, составленные из двух РОН; [X] – ячейка памяти данных, адрес которой находится в регистре X, также содержащее, значение этой ячейки. Аналогично для регистров Y и Z; {Z} – ячейка памяти команд, адресуемая регистром Z (также содержимое, значение этой ячейки); q – число; P – имя или номер регистра (порта, Port) ввода-вывода			

Таблица 6.2 – Арифметические и сдвиговые команды

Мнемоника	Название	Действие	Флаги
ADD Rd, Rs	Сложение двух РОН	$Rd \leftarrow Rd + Rs$	Z,C,N,V,H,S
ADC Rd, Rs	Сложение двух РОН с переносом	$Rd \leftarrow Rd + Rs + C$	Z,C,N,V,H,S
SUB Rd, Rs	Вычитание двух РОН	$Rd \leftarrow Rd - Rs$	Z,C,N,V,H,S
SUBI Rd, K *	Вычитание константы из РОН	$Rd \leftarrow Rd - K$	Z,C,N,V,H,S
SBC Rd, Rs	Вычитание двух РОН с заемом	$Rd \leftarrow Rd - Rs - C$	Z,C,N,V,H,S
SBCI Rd, K *	Вычитание константы из РОН с заемом	$Rd \leftarrow Rd - K - C$	Z,C,N,V,H,S
NEG Rd	Изменение знака числа в дополнительном коде	$Rd \leftarrow \$100 - Rd = \$FF - Rd + 1$	Z,C,N,V,H,S
INC Rd	Инкрементирование РОН	$Rd \leftarrow Rd + 1$	Z,N,V,S
DEC Rd	Декрементирование РОН	$Rd \leftarrow Rd - 1$	Z,N,V,S
ADIW Rd, K **	Сложение регистровой пары с константой	$Rd+1:Rd \leftarrow Rd+1:Rd+K$	Z,C,N,V,S
SBIW Rd, K **	Вычитание константы из регистровой пары	$Rd+1:Rd \leftarrow Rd+1:Rd-K$	Z,C,N,V,S
ASR Rd	Арифметический сдвиг вправо	$Rd(n) \leftarrow Rd(n+1), n = 0..6$	Z,C,N,V,S
LSL Rd	Логический сдвиг влево	$Rd(n+1) \leftarrow Rd(n), Rd(0) \leftarrow 0$	Z,C,N,V,H,S
LSR Rd	Логический сдвиг вправо	$Rd(n) \leftarrow Rd(n+1), Rd(7) \leftarrow 0$	Z,C,N,V,S
ROL Rd	Циклический сдвиг влево через C	$Rd(0) \leftarrow C, Rd(n+1) \leftarrow Rd(n), C \leftarrow Rd(7)$	Z,C,N,V,H,S
ROR Rd	Циклический сдвиг вправо через C	$Rd(7) \leftarrow C, Rd(n) \leftarrow Rd(n+1), C \leftarrow Rd(0)$	Z,C,N,V,S
MUL Rd, Rs	Умножение беззнаковых чисел	$R1:R0 \leftarrow Rd * Rs$	Z,C
MULS Rd, Rs *	Умножение чисел со знаком	$R1:R0 \leftarrow Rd * Rs$	Z,C
MULSU Rd, Rs ***	Умножение беззнакового числа на число со знаком	$R1:R0 \leftarrow Rd * Rs$	Z,C

Продолжение таблицы 6.2 – Арифметические и сдвиговые команды

Мнемоника	Название	Действие	Флаги
FMUL Rd, Rs ***	Умножение дробных беззнаковых чисел	$R1:R0 \leftarrow (Rd * Rs) \ll 1$	Z,C
FMULS Rd, Rs ***	Умножение дробных чисел со знаком	$R1:R0 \leftarrow (Rd * Rs) \ll 1$	Z,C
FMULSU Rd, Rs ***	Умножение дробного беззнакового числа и дробного числа со знаком	$R1:R0 \leftarrow (Rd * Rs) \ll 1$	Z,C
$0 \leq d \leq 31, 0 \leq s \leq 31, 0 \leq K \leq 255;$ * в командах SUBI, SBCI, MULS: $16 \leq d \leq 31, 16 \leq s \leq 31;$ ** в командах ADIW, SBIW: $d = \{24, 26, 28, 30\}, 0 \leq K \leq 63;$ *** в командах MULSU, FMUL, FMULS, FMULSU: $16 \leq d \leq 23, 16 \leq s \leq 23$			

Детальное описание арифметических команд.

Числа с фиксированной запятой: целые и дробные. *Целые числа* это – множество натуральных чисел (1, 2, 3, 4, ..., n), чисел, противоположных натуральным, (-1, -2, -3, -4, ..., -n) и ноль (0). Иными словами, это числа, не имеющие дробной части. Например: 43; -79; 0; -685; 8730 – все целые (натуральные только 43 и 8730). 352,25; 0,87; 835,3448 – все нецелые (натуральных нет). Целое число называют также числом с фиксированной точкой (или запятой), мысленно помещаемой правее младшего разряда числа.

Дробное число – это число, между двумя заранее определёнными разрядами которого ставится запятая. Например, целое число 1234 после вставки запятой превращается в дробное 12,34. Иными словами, дробное число – это тоже число с фиксированной запятой, но помещаемой левее младшего разряда.

Арифметику с фиксированной запятой часто применяют в тех областях, когда использование чисел с плавающей запятой затратно или невозможно ввиду используемой архитектуры процессоров.

Арифметические операции над целыми и дробными числами.

Операции сложения и вычитания целочисленной арифметики выполняются всеми процессорами над двоичными операндами как над натуральными числами (целыми числами без знака). Это означает, что старшие, знаковые биты операндов обрабатываются так же, как остальные, цифровые – складываются и вычитаются. Аналогичным образом умножаются и делятся целые, а также дробные числа. Многие процессоры выполняют, кроме того, умножение знаковых, целых и дробных, чисел, записанных в дополнительном коде, или и умножение, и деление.

Сложение и вычитание на примере команд ADD и SUB.

Формат * и действие команды ADD:

add d, s; $C_d \leftarrow d + s$

d и s – 8-битные операнды, находящиеся в регистрах общего назначения; C (Carry) – флажок переноса/заема; C_d – конкатенация (сочленение) бита C и байта d , т.е. C_d – это 9-битный получатель результата операции.

* *Форматом команды* или вообще *цифрового данного* называется полное описание структуры команды / данного с указанием назначения каждого элемента структуры, каждого бита команды и данного.

Формат и действие команды SUB:

sub d, s ; $C_d \leftarrow 1_d - s$, $C \leftarrow \neg C$

$\neg$ (или $\sim$) обозначает побитную инверсию, в данном случае – инверсию бита C .

Действие команды SUB записано выше с точки зрения программиста. Процессор вычитает 8-битное натуральное число s из 9-битного натурального 1_d (где «1» дописана в 9-й, несуществующий разряд d) и заносит полученную 9-битную разность в C_d , после чего инвертирует флажок C .

Аппаратно вычитание осуществляется иначе, поскольку в процессорах нет вычитателя, и вычитание производится, эквивалентным образом, путем сложения уменьшаемого с дополнением вычитаемого:

$C_d \leftarrow d + \text{ДОП}(s)$, $C \leftarrow \neg C$

где $\text{ДОП}(s) = 2^8 - s = \neg s + 1$.

Доказательство: $1_d - s = 1^\circ 0000^\circ 0000 + d - s = 2^8 + d - s = d + 2^8 - s = d + \text{ДОП}(s)$.

Коротко: Сложение дает сумму, которая заносится в C_d . Вычитание заменяется на сложение с дополнением, и сумма заносится в C_d , после чего C инвертируется.

Интерпретация результата.

1. Если d и s – натуральные числа, то их правильная и натуральная сумма/разность получается в d тогда и только тогда, когда $C=0$. В этом случае не произошло арифметического переполнения в натуральной арифметике – результат операции не вышел из числового диапазона от 0 до 2^8-1 ($0 \dots 255$): сумма $d+s \leq 255$, разность $d-s \geq 0$ ($d \geq s$).

Независимо от значения C ($C=0$ – переполнения нет; или $C=1$ – переполнение есть: $d+s > 255$ или $d < s$) правильная сумма/разность находится в C_d . При этом сумма положительная и представлена в натуральном коде, а разность положительная в натуральном коде при $C=0$ ($d \geq s$), и отрицательная в дополнительном коде при $C=1$ ($d < s$).

2. Если d и s – дополнительные коды чисел со знаком, то при сложении / вычитании этих кодов правильный дополнительный код суммы/разности чисел получается в d при условии, что $V=0$, т.е. переполнения не было. V (oVerflow) – флажок переполнения. Это означает, что в знаковой арифметике результат не вышел за пределы диапазона чисел $-2^7 \dots 2^7-1$ ($-128 \dots 127$).

Наличие флажка знака S (Sign), равного $S = N \text{ XOR } V$, расширяет знаковую арифметику: правильная сумма/разность знаковых чисел (в дополнительных кодах) всегда содержится в S_d независимо от наличия или отсутствия переполнения. N (Negative) – это флаг отрицательного результата.

Другие команды сложения и вычитания (adc, sbc, subi, sbci и др.) выполняются аналогично. Сложение дает сумму, которая заносится в C_d .

Вычитание заменяется на сложение с дополнением, и сумма заносится в C_d , после чего C инвертируется. Интерпретация полученных результатов та же, что и изложенная выше.

Выводы. Команды сложения и вычитания (add , sub и др.) задают арифметические действия над натуральными числами и только над ними. По этой причине процессор как бы «думает», что все числа натуральные, он не различает знаковых и беззнаковых чисел, для него все числа беззнаковые. Но благодаря дополнительным кодам указанные команды можно напрямую использовать для выполнения арифметических операций и над числами со знаком. Таким образом, одни и те же команды можно привлекать как в натуральной, так и в знаковой арифметике. Какую из них применять, решает программист в соответствии со стоящей перед ним задачей. Если он реализует натуральную арифметику, то в качестве исходных операндов принимаются натуральные числа и результат операции тоже трактуется как натуральное число. Если знаковую арифметику – то исходные операнды (числа со знаком) записываются в дополнительном коде и результат тоже трактуется как знаковое число в дополнительном коде. Например, числовой код $0xFF=1111^o1111$ можно трактовать одновременно как натуральное число 255 и как знаковое число -1.

В любой из арифметик надо проверять правильность результата операции. Он будет правильным, если не было переполнения, и неправильным, если было. О переполнении, т.е. о неправильном результате сигнализируют флажки: в натуральной арифметике – «флажок переноса/заема» C , в знаковой – «флажок переполнения» V . Флажок $C=1$ сигнализирует о выходе суммы/разности натуральных чисел за диапазон $0...255$, $V=1$ – о выходе суммы/разности знаковых чисел за диапазон $-128...127$ (оба диапазона соответствуют 8-битной разрядной сетке). В этом случае результат в d неправильный. Если же $C=0$ или $V=0$, то переполнения не было, результат в d правильный.

Сказанное означает, что в отличие от процессора, оперирующего со всеми числами при сложении и вычитании как с натуральными, флажок переполнения V реагирует не на натуральную, а на знаковую арифметику, воспринимая исходные операнды и результат операции как числа со знаком, записанные в дополнительном коде. Это означает, что если, скажем, x и y – исходные операнды, числа со знаком, а $x_{доп}$ и $y_{доп}$ – их дополнительные коды (представляющие эти операнды в общих регистрах процессора), то при сложении операндов по команде $add\ x, y$ процессор сложит не числа x и y , а их дополнительные коды: $x_{доп} + y_{доп}$. При этом, однако, флажок V отреагирует не на сумму кодов $x_{доп} + y_{доп}$, а на сумму самих чисел $x + y$. И когда сумма $x + y$ выйдет за диапазон $-128...127$, флажок V установится в «1»: $V=1$, когда нет – сбросится в «0»: $V=0$. Аналогично при вычитании флажок V отслеживает выход за указанный диапазон разности чисел $x - y$, а не разности кодов $x_{доп} - y_{доп}$.

В то же время флажок C трактует операнды, будь то натуральные числа или числа со знаком, записанные в дополнительных кодах, как натуральные числа и после их сложения/вычитания сигнализирует о выходе суммы/разности за

диапазон 0...255. В частности, если процессор складывает знаковые числа, то C устанавливается в «1» при $x_{\text{доп}} + y_{\text{доп}} > 255$, а если вычитает, то $C = 1$ при $x_{\text{доп}} - y_{\text{доп}} < 0$ (отражая перенос/заем при сложении/вычитании дополнительных кодов).

Пример

Код	Код
ldi R21, 255	ldi R21, 128
ldi R21, 2	ldi R21, 129
add R21, R22	add R21, R22
Сумма	Сумма
C_R21=1_00000001=257	C_R21=1_00000001=257
Флажки	Флажки
C=1 V=0 Для флажка C операнды – это натуральные числа 255 и 2. Их сумма 255+2=257. Она выходит за диапазон 0...255. Поэтому C=1. Для флажка V операнды – это знаковые числа -1 и 2. Их сумма равна -1+2=1. Она входит в диапазон -128...127. Поэтому V=0.	C=1 V=1 Для флажка C операнды – это натуральные числа 128 и 129. Их сумма 128+129=257. Она выходит за диапазон 0...255. Поэтому C=1. Для флажка V операнды – это знаковые числа -128 и -127. Их сумма равна -128+(-127)=-255. Она выходит за диапазон -128...127. Поэтому V=1.

В случае переполнения ($C=1$ или $V=1$ – неправильный результат) программу надо организовать так, чтобы она сообщала пользователю об ошибке по переполнению, либо переключалась на другой алгоритм, преодолевающий переполнение. Последний может использовать, например, более широкую разрядную сетку, которая уже не будет переполняться. Это годится для знаковой и строго натуральной арифметики. Если же разность натуральных чисел получилась отрицательной ($d-s < 0$), то расширение сетки не поможет. Тогда надо использовать знаковые числа и реализовывать комбинированный алгоритм.

Независимо от наличия или отсутствия переполнения по флажку V правильная сумма/разность знаковых чисел (в дополнительных кодах) всегда содержится в S_d .

В заключение заметим, что переполнение в натуральной арифметике именуется переносом или заемом, а термины «арифметическое переполнение», «переполнение» закреплены за переполнением в знаковой арифметике дополнительных кодов. В данном изложении мы такого разделения терминологии не делали.

Умножение дробных чисел. Пусть выражение (N.Q) обозначает дробное число с N двоичными цифрами слева от точки и Q двоичными цифрами справа от точки. Умножение между двумя числами в форматах (N1.Q1) и (N2.Q2) даст результат в формате ((N1 + N2). (Q1 + Q2)).

Формат и действие команд FMUL / FMULS / FMULSU:

fmul/fmuls/fmul su Rd, Rs; R1:R0 $\leftarrow$ (Rd×Rs)<<1

где выражение << обозначает сдвиг влево на 1 бит.

Описание действия: Осуществляют умножение дробных чисел, находящихся в регистрах Rd и Rs. Формат чисел – 1.7 (старший разряд – целая часть, 7 младших – дробная). Результат умножения (формат результата – 2.14) сдвигается влево на один разряд для приведения к формату 1.15 и заносится в регистровую пару R1:R0.

В команде `fmul` регистры Rd, Rs содержат беззнаковые дробные числа; в `fmuls` – дробные числа со знаком, представленные в дополнительном коде;

в `fmulsu` Rd содержит дробное число со знаком в дополнительном коде, Rs – дробное число без знака.

C устанавливается, если бит15 результата установлен до сдвига влево; очищается в ином случае.

Z устанавливается, если результат равен нулю; очищается в ином случае.

Таблица 6.3 – Логические команды

Мнемоника	Название	Действие	Флаги
AND Rd, Rs	«Логическое И» двух РОН	$Rd \leftarrow Rd \cdot Rs$	Z,N,V
ANDI Rd, K *	«Логическое И» РОН и константы	$Rd \leftarrow Rd \cdot K$	Z,N,V
OR Rd, Rs	«Логическое ИЛИ» двух РОН	$Rd \leftarrow Rd \sqcup Rs$	Z,N,V
ORI Rd, K *	«Логическое ИЛИ» РОН и константы	$Rd \leftarrow Rd \sqcup K$	Z,N,V
EOR Rd, Rs	«Исключающее ИЛИ» двух РОН	$Rd \leftarrow Rd \oplus Rs$	Z,N,V
COM Rd	Побитная инверсия	$Rd \leftarrow \$FF - Rd$	Z,C,N,V,S
CLR Rd	Сброс всех битов РОН	$Rd \leftarrow Rd \sqcap Rd$	Z,N,V
SER Rd *	Установка всех битов РОН	$Rd \leftarrow \$FF$	-
TST Rd	Проверка РОН на отрицательное или нулевое значение	$Rd \cdot Rd$	Z,N,V
SWAP Rd	Обмен местами полубайтов в РОН	$Rd(3...0) \leftarrow Rd(7...4),$ $Rd(7...4) \leftarrow Rd(3...0)$	-
$0 \leq d \leq 31, 0 \leq s \leq 31, 0 \leq K \leq 255;$ • – операция побитного логического И; $\sqcup$ – операция побитного логического ИЛИ; * в командах ANDI, ORI и SER: $16 \leq d \leq 31$			

Таблица 6.4 – Команды битовых операций

Мнемоника	Название	Действие	Флаги
CBR Rd, K *	Сброс бита(ов) POH	$Rd \leftarrow Rd \cdot (\$FF - K)$	Z, N, V
SBR Rd, K *	Установка бита(ов) POH	$Rd \leftarrow Rd \square K$	Z, N, V
CBI P, b **	Сброс бита PBB	$P.b \leftarrow 0$	-
SBI P, b **	Установка бита PBB	$P.b \leftarrow 1$	-
BCLR f	Сброс флага	$SREG.f \leftarrow 0$	SREG.f
BSET f	Установка флага	$SREG.f \leftarrow 1$	SREG.f
BLD Rd, b	Загрузка бита POH из флага T (SREG)	$Rd.b \leftarrow T$	-
BST Rs, b	Запись бита POH в флаг T (SREG)	$T \leftarrow Rs.b$	T
CLC	Сброс флага переноса	$C \leftarrow 0$	C
SEC	Установка флага переноса	$C \leftarrow 1$	C
CLN	Сброс флага отрицательного числа	$N \leftarrow 0$	N
SEN	Установка флага отрицательного числа	$N \leftarrow 1$	N
CLZ	Сброс флага нуля	$Z \leftarrow 0$	Z
SEZ	Установка флага нуля	$Z \leftarrow 1$	Z
CLI	Общее запрещение прерываний	$I \leftarrow 0$	I
SEI	Общее разрешение прерываний	$I \leftarrow 1$	I
CLS	Сброс флага знака	$S \leftarrow 0$	S
SES	Установка флага знака	$S \leftarrow 1$	S
CLV	Сброс флага переполнения дополнительного кода	$V \leftarrow 0$	V
SEV	Установка флага переполнения дополн. кода	$V \leftarrow 1$	V
CLT	Сброс флага T	$T \leftarrow 0$	T
SET	Установка флага T	$T \leftarrow 1$	T
CLH	Сброс флага половинного переноса	$H \leftarrow 0$	H
SEH	Установка флага половинного переноса	$H \leftarrow 1$	H
$0 \leq d \leq 31, 0 \leq s \leq 31, 0 \leq b \leq 7, 0 \leq f \leq 7, 0 \leq K \leq 255;$ * в командах CBR, SBR: $16 \leq d \leq 31$ ** в командах CBI и SBI: $0 \leq P \leq 31, 0 \leq b \leq 7$			

6.3. Команды передачи управления

Назначение команд передачи управления. При выполнении линейных программных фрагментов процессор выбирает команды из программной памяти и интерпретирует их в соответствии с семантикой команд. При этом соответственно корректируется (увеличивается на 1) программный счетчик РС: $PC \leftarrow PC + 1$. В результате РС указывает на следующее командное слово, подлежащее выборке в следующем шинном цикле. [10, 11, 16]

Такие действия соответствуют *естественному, линейному* порядку выполнения команд, при котором команды выбираются из памяти и выполняются в том порядке, в каком они записаны в программе – от ячейки к ячейке. Такой порядок осуществляется при отработке *команд обработки данных*: пересылки, арифметических и логических операций и т.д., а также *команд изменения состояния системы*.

На практике обойтись только линейными программами невозможно. В разветвляющихся и циклических программах, а также при организации подпрограмм необходимо выполнять не следующую по порядку команду, а команду, находящуюся в другой ячейке программной памяти. Специальные команды, которые изменяют естественное продвижение процессора по программе и заставляют его перейти на любую другую ячейку программной памяти, где хранится нужная команда, называются *командами передачи управления* или *командами перехода* *.

* Командами перехода называют чаще всего те, которые изменяют содержимое регистра РС, но не оперируют со стеком, т.е. команды группы `jmp` и `sbxx`, `brxx`. Команды же вызова (`call`) и возврата из подпрограмм (`ret`) и возврата из прерываний (`reti`) оперируют со стеком и обычно именуются общим термином «команды передачи управления», но допускается также и термин «команды перехода».

Команда, на которую нужно перейти по программе, задаётся её адресом – адресом ячейки программной памяти, в которой она хранится. Этот адрес называется *адресом перехода* или *адресом передачи управления*. Адрес перехода известен команде перехода, и для осуществления перехода она заносит его в регистр-указатель программной памяти РС:

$PC \leftarrow \text{Адрес перехода.}$

Это заставляет процессор перейти по указанному адресу и продолжить выполнение программы уже с новой, заданной этим адресом ячейки. Продвижение процессора по программе с помощью команд передачи управления называется *принудительным*.

Отметим, что команды передачи управления не изменяют состояний флажков, за исключением команды возврата из прерываний `reti`.

Таким образом, когда процессор выбирает из программной памяти команду передачи управления, его действия становятся иными, чем при выполнении команд обработки данных. По указке этой команды процессор загружает в

регистр PC новое значение, определяющее адрес перехода. После загрузки PC будет содержать *новый адрес программной памяти – адрес перехода*. В следующем цикле шины процессор выставит этот адрес в шину адреса и выберет из программной памяти команду, находящуюся в адресованной ячейке. Это и означает, что процессор переходит, «перескакивает» на новую ячейку программной памяти. Представленный механизм обеспечивает любые переходы по программе, в пределах всего адресного пространства программной памяти.

Абсолютная и относительная адресация переходов. Команды `jmp`, `ijmp`, `eijmp`, `call`, `ret` и `reti` используют *абсолютную адресацию* точки перехода, т.е. они предоставляют процессору собственно адрес перехода – его законченное готовое значение, хранящееся в самой команде или в известном месте компьютера, и подлежащее загрузке в PC:

$PC \leftarrow \text{Готовый адрес перехода.}$

В зависимости от команды готовый адрес хранится либо в ее адресной части (прямой адрес) либо в регистре процессора Z или SP (косвенный адрес). Для некоторых процессоров (но не для AVR) адрес перехода (косвенный) может находиться в ячейке памяти данных.

Остальные команды передачи управления (`rjmp`, `rcall`, `sbxx` и `brxx`), задающие прямой адрес перехода, осуществляют *относительную адресацию* перехода. Это такая адресация, при которой процессору сообщается не сам адрес перехода, а расстояние до него в 16-битных ячейках (словах) от данной точки программы. А именно, указывается, на сколько ячеек вперед или назад надо «скакнуть» по программной памяти от текущей ячейки, чтобы попасть в целевую ячейку. Адрес перехода (целевой адрес) вычисляется в этом случае процессором путем прибавления знакового смещения, встроенного в адресную часть команды передачи управления, к текущему адресу:

$PC \leftarrow PC + k$

– «целевой адрес равен текущему плюс k».

Здесь k – знаковое целое, записанное в дополнительном коде. Перед прибавлением знак k расширяется до размера PC (если длина k короче) *. Значение PC в правой части выражения определяет текущий адрес программной памяти, т.е. адрес команды, находящейся в программной памяти сразу за командой передачи управления (так как после ее выборки он продвинулся именно до этого значения). Если коротко, то PC справа – это адрес команды за `rjmp` (`rcall`, ...). Таким образом, исходным, «текущим» состоянием программного счетчика PC при его модификации по команде передачи управления считается адрес команды, находящейся после команды передачи управления.

* 1) PC AVR имеет от 10 до 17 разрядов, например 14 у ATmega16.

2) Для команд `sbxx` $k=1$.

При вычислении нового адреса команды по формуле $PC+k$ возникающий перенос из старшего бита PC в несуществующий бит слева игнорируется, нигде не запоминается, т.е. сумма $PC + k$ обрезается слева до разрядности программ-

ного счетчика PC. Поэтому полученный адрес остаётся в пределах пространства программной памяти.

Отметим, что знаковое целое k формируется самим ассемблером (транслятором языка), который вычисляет его как разность между значением адреса перехода и значением адреса команды, следующей за `rjmp` (или другими подобными командами).

Замечание. В литературе по системе команд AVR под символом PC в правой части выражения выше иногда понимают адрес самой команды относительной передачи управления, и это выражение записывают так:

$$PC \leftarrow PC + k + 1,$$

где PC в правой части – это адрес команды относительного перехода, а 1 – длина команды в словах, что означает, что после ее выборки PC увеличился на 1 и показывает на следующую за переходом команду.

Далее для описания работы команд условного перехода мы будем придерживаться формулы $PC \leftarrow PC + k$, как и будет показано в табл.6.5.

Прямая и косвенная адресация переходов. *Прямая адресация перехода* – это задание в команде численного значения адреса перехода (абсолютного или относительного), который хранится в адресной части команды. На языке ассемблера в этом случае в поле команды вместо численного адреса записывается его символическое имя – метка (играющая роль мнемонического адреса перехода, мнемoadреса). Некоторые ассемблеры (включая AVR) допускают запись и численного адреса перехода в команде. *Косвенная адресация перехода* означает, что в команде указывается не сам адрес, а место, где он «лежит» – например, регистр процессора или ячейка памяти. Косвенный адрес всегда абсолютный. В AVR он может лежать только в регистрах Z и SP, но не в ячейке памяти.

Безусловные и условные переходы (ветвления). *Безусловный переход* — это переход, который выполняется всегда. У команд безусловного перехода один операнд, который может быть непосредственным адресом (меткой), регистром или ячейкой памяти, содержащей адрес. В AVR безусловный переход осуществляется с помощью всех тех команд, которые используют абсолютную адресацию точки перехода: `jmp`, `ijmp`, `eijmp`, `call`, `ret` и `reti`, а также команд с относительной адресацией перехода `rjmp` и `rcall`. Переход задается ими, в зависимости от команды, непосредственным адресом (меткой) или, как уже говорилось, регистрами Z и SP.

Условный переход, или *ветвление* осуществляется, если выполняется определённое условие, заданное флагами процессора. Состояние флагов изменяется после выполнения арифметических, логических и некоторых других команд. Если условие не выполняется, то управление переходит к следующей команде. В AVR таковыми являются команды с общей мнемоникой `sbxx` (от Skip if Bit ... – перепрыгнуть, пропустить, если бит ...) и `brxx` (от Branch – ветка, ветвь; Branching – ветвление). Все они используют относительную адресацию точки перехода.

Таблица 6.5 – Команды передачи управления

Мнемоника	Название	Действие	Флаги
RJMP k	Относительный безусловный переход	$PC \leftarrow PC + k$	-
IJMP	Косвенный безусловный переход	$PC \leftarrow Z$	-
JMP k	Абсолютный переход	$PC \leftarrow k$	-
RCALL k	Относительный вызов подпрограммы	$Stack \leftarrow PC,$ $SP \leftarrow SP - 2 \text{ or } 3^*,$ $PC \leftarrow PC + k$	-
ICALL	Косвенный вызов подпрограммы	$Stack \leftarrow PC,$ $SP \leftarrow SP - 2 \text{ or } 3,$ $PC \leftarrow Z$	-
CALL k	Абсолютный вызов подпрограммы	$Stack \leftarrow PC,$ $SP \leftarrow SP - 2 \text{ or } 3,$ $PC \leftarrow k$	-
RET	Возврат из подпрограммы	$SP \leftarrow SP + 2 \text{ or } 3,$ $PC \leftarrow Stack$	-
RETI	Возврат из п/п обработки прерывания	$SP \leftarrow SP + 2 \text{ or } 3,$ $PC \leftarrow Stack, I \leftarrow 1$	I
CP Rd, Rs *	Сравнение РОН	$Rd - Rs$	Z,N,V,C,H,S
CPC Rd, Rs	Сравнение РОН с учетом переноса	$Rd - Rs - C$	Z,N,V,C,H,S
CPI Rd, K	Сравнение РОН с константой	$Rd - K$	Z,N,V,C,H,S
CPSE Rd, Rs	Сравнение и пропуск следующей команды при равенстве	Если $Rd = Rs$, то $PC \leftarrow PC + 1 \text{ or } 2$	-
SBRC Rs, b	Пропуск следующей команды, если бит РОН сброшен	Если $Rs.b = 0$, то $PC \leftarrow PC + 1 \text{ or } 2$	-
SBR Rs, b	Пропуск следующей команды, если бит РОН установлен	Если $Rs.b = 1$, то $PC \leftarrow PC + 1 \text{ or } 2$	-
SBIC P, b	Пропуск следующей команды, если бит ПВВ сброшен	Если $P.b = 0$, то $PC \leftarrow PC + 1 \text{ or } 2$	-
SBIS P, b	Пропуск следующей команды, если бит ПВВ установлен	Если $P.b = 1$, то $PC \leftarrow PC + 1 \text{ or } 2$	-
* Команды CP[x] производят вычитание операндов без запоминания разности. Воздействуют на все арифметические флажки.			

Продолжение таблицы 6.5 – Команды передачи управления

Мнемоника	Название	Действие	Флаги
BRBC f, k	Переход, если флаг f регистра SREG сброшен	Если $SREG.f = 0$, то $PC \leftarrow PC + k$	-
BRBS f, k	Переход, если флаг f регистра SREG установлен	Если $SREG.f = 1$, то $PC \leftarrow PC + k$	-
BRCS k	Переход по переносу	Если $C = 1$, то $PC \leftarrow PC + k$	-
BRCC k	Переход, если нет переноса	Если $C = 0$, то $PC \leftarrow PC + k$	-
BREQ k	Переход по «равно»	Если $Z = 1$, то $PC \leftarrow PC + k$	-
BRNE k	Переход по «не равно»	Если $Z = 0$, то $PC \leftarrow PC + k$	-
BRSH k	Переход по «больше или равно»	Если $C = 0$, то $PC \leftarrow PC + k$	-
BRLO k	Переход по «меньше»	Если $C = 1$, то $PC \leftarrow PC + k$	-
BRMI k	Переход по «отрицательное значение»	Если $N = 1$, то $PC \leftarrow PC + k$	-
BRPL k	Переход по «положительное значение»	Если $N = 0$, то $PC \leftarrow PC + k$	-
BRGE k	Переход по «больше или равно» (числа со знаком)	Если $(N \oplus V) = 0$, то $PC \leftarrow PC + k$	-
BRLT k	Переход по «меньше нуля» (числа со знаком)	Если $(N \oplus V) = 1$, то $PC \leftarrow PC + k$	-
BRHS k	Переход по половинному переносу	Если $H = 1$, то $PC \leftarrow PC + k$	-
BRHC k	Переход, если нет половинного переноса	Если $H = 0$, то $PC \leftarrow PC + k$	-
BRTS k	Переход, если флаг T установлен	Если $T = 1$, то $PC \leftarrow PC + k$	-
BRTC k	Переход, если флаг T сброшен	Если $T = 0$, то $PC \leftarrow PC + k$	-

Продолжение таблицы 6.5 – Команды передачи управления

Мнемоника	Название	Действие	Флаги
BRVS k	Переход по переполнению (в дополнительном коде)	Если $V = 1$, то $PC \leftarrow PC + k$	-
BRVC k	Переход, если нет переполнения дополнительного кода	Если $V = 0$, то $PC \leftarrow PC + k$	-
BRID k	Переход, если прерывания запрещены	Если $I = 0$, то $PC \leftarrow PC + k$	-
BRIE k	Переход, если прерывания разрешены	Если $I = 1$, то $PC \leftarrow PC + k$	-

$0 \leq d \leq 31, 0 \leq s \leq 31, 0 \leq b \leq 7, 0 \leq f \leq 7, 0 \leq K \leq 255, 0 \leq P \leq 31$;
k – метка или адрес перехода в мнемонике команды, знаковая добавка в действии команды;
в командах RJMP, RCALL: $-2048 \leq k \leq +2047$. По этим командам к текущему значению PC прибавляется константа (знаковая добавка) k. Соответственно под текущим значением PC в правой части формулы $PC \leftarrow PC + k$ понимается адрес команды, находящейся в программной памяти сразу за командой перехода;
в командах JMP, CALL: $0 \leq k \leq 4M$;
в командах BRxx: $-64 \leq k \leq 63$;
в команде CPI: $16 \leq d \leq 31$;
CP, CPC, CPI не являются командами передачи управления. Они используются перед командами условного перехода для подготовки условий перехода;
* 2 or 3 – в зависимости от разрядности PC.

Вызовы и возвраты из подпрограмм.

Подпрограммой мы назвали (в п.5.5) поименованную последовательность команд, которую можно вызывать – запускать на счет (с помощью команды call и ей подобных) из разных мест программы, при этом возврат из подпрограммы происходит всегда в ту точку программы, на ту ее команду, которая следует за командой последнего вызова («на после call»). Для возврата служит команда возврата ret (а для прерываний – reti), которую помещают в конце подпрограммы (см. рис.5.13).

Формат и действие команды CALL.

call k; $Стек \leftarrow PC, PC \leftarrow k, k$ – пусковой адрес подпрограммы

Разложение команды CALL \$0500 по машинным циклам.* Код команды \$940E°0500. Команда расположена в ячейках \$0300, \$0301 и передает управление подпрограмме, начинающейся с ячейки \$0500, как показано на рис.6.2. Перед этим она запоминает в стеке адрес возврата \$0302. Следующим за call будет выполнен цикл чтения команды из ячейки \$0500.

* Понятие машинного и командного циклов см. в Приложении Б.

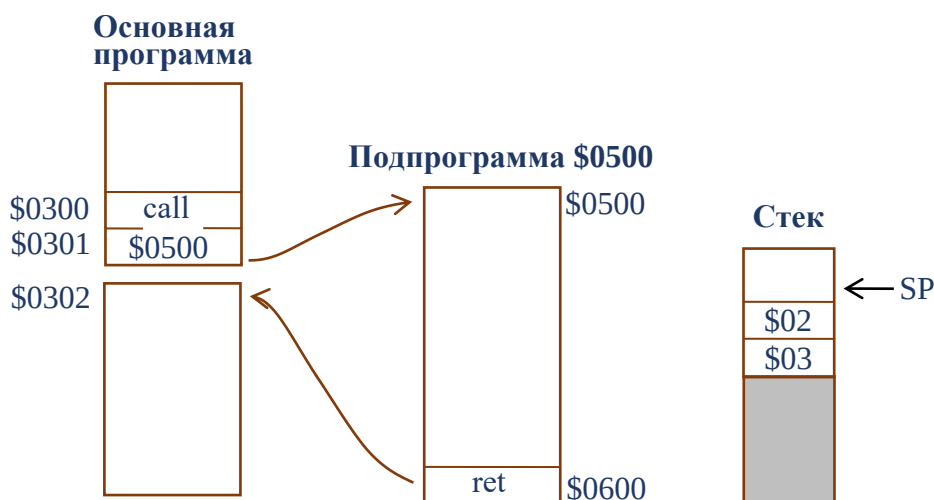


Рисунок 6.2 – Работа команд call и ret

Командный цикл call \$050

<u>Маш. цикл 1 "Чтение команды (кода операции – КОП)"</u> 1) $PC = \$0300 \rightarrow \text{ША}$, 2) $\text{Чт} \rightarrow \text{ШУ}$ 3) $\text{РгК} \leftarrow \text{ШД} \leftarrow m(PC) = \$940E$ $PC \leftarrow PC + 1 = \$0301$ 4) Дешифрация команды	<u>Маш. цикл 2 "Чтение команды (адреса перехода)"</u> 1) $PC = \$0301 \rightarrow \text{ША}$, 2) $\text{Чт} \rightarrow \text{ШУ}$ 3) $\text{РгК} \leftarrow \text{ШД} \leftarrow m(PC) = \$0500$ $PC \leftarrow PC + 1 = \$0302$ 4) $\text{PCстар} \leftarrow PC, PC \leftarrow \$0500 \text{ из РгК}$
<u>Маш. цикл 3 "Запись в стек РСН"</u> 1) $SP \rightarrow \text{ША}$ 2) $\text{Зп} \rightarrow \text{ШУ}$ 3) $\text{РСНстар} = \$03 \rightarrow \text{ШД} \rightarrow m(SP)$, $SP \leftarrow SP - 1$	<u>Маш. цикл 4 "Запись в стек PCL"</u> 1) $SP \rightarrow \text{ША}$ 2) $\text{Зп} \rightarrow \text{ШУ}$ 3) $\text{PCLстар} = \$02 \rightarrow \text{ШД} \rightarrow m(SP)$, $SP \leftarrow SP - 1$
ША, ШУ, ШД – шины адресов, управления, данных, отдельные для каждого вида памяти программ и данных; Чт, Зп – сигналы управления чтением, записью; РгК – регистр команд; $m(PC)$ – ячейка памяти программ (memory), адресуемая регистром PC; $m(SP)$ – аналогично, ячейка памяти данных, адресуемая регистром SP; РСстар – старое значение программного счетчика, подлежащее сохранению в стеке: PC строится в виде сдвоенного регистра, один из которых принимает адрес перехода, другой содержит старый адрес – адрес последующей смежной команды; РСНстар и PCLстар – старший и младший байты РСстар	

Формат и действие команды RET.

ret; $PC \leftarrow \text{Стек} = \text{адрес возврата из подпрограммы}$

Разложение команды RET по машинным циклам. Код команды \$9508. Команда ret расположена в ячейке \$0600 и возвращает управление

на команду после `call`, т.е. на адрес $\$0302$ (см. рис.6.2). Следующим за `ret` будет выполнен цикл чтения команды из адреса $\$0302$.

Командный цикл `ret`

<u>Маш. цикл 1 "Чтение команды (КОП)"</u> 1) $PC = \$0600 \rightarrow ША$, 2) $Чт \rightarrow ШУ$ 3) $РгК \leftarrow ШД \leftarrow m(PC) = \$9508$ $PC \leftarrow PC + 1 = \$0601$ 4) Дешифрация команды	<u>Маш. цикл 2 "Чтение РСНстар из стека"</u> 1) $SP \leftarrow SP + 1, SP \rightarrow ША$ 2) $Чт \rightarrow ШУ$ 3) $РСН \leftarrow ШД \leftarrow m(SP) = \$03$
<u>Маш. цикл 3 "Чтение PCLстар из стека"</u> 1) $SP \leftarrow SP + 1, SP \rightarrow ША$ 2) $Чт \rightarrow ШУ$ 3) $PCL \leftarrow ШД \leftarrow m(SP) = \$02$ $PC = \$0302$,	
РСН и PCL – старший и младший байты программного счетчика PC	

Выводы. Любая команда перехода на подпрограмму, например `call`, прежде чем передать управление команде, находящейся по *адресу перехода* (адрес $\$0500$ на рис.6.2), заносит в стек *адрес возврата* (адрес $\$0302$ на рис.6.2). Последний есть значение программного счетчика PC, которое образовалось после выборки команды `call` из памяти. Оно показывает на расположенную сразу за `call` команду (так как во время выборки каждой команды, в нашем случае `call`, значение PC прирастает на «1» на каждое выбранное, 16-битное, слово и, в частности, после выборки `call` из ячеек $\$0300, \$0301$ становится равным $\$0302$). Таким образом, *адрес возврата из подпрограммы – это адрес команды после `call`* (в нашем случае $\$0302$).

Всякая подпрограмма заканчивается командой возврата `ret` – return, извлекающей из стека адрес возврата и заносящей его обратно в PC. Благодаря этому возврат происходит всегда в то место основной программы, откуда в последний раз был произведен вызов, на ту команду, которая расположена непосредственно за `call` (см. рис.6.2 или 5.13). Поэтому можно осуществлять вызовы подпрограмм из разных мест основной программы, возврат будет гарантирован в эти же места.

Аналогично происходят вызовы (с участием команд `jmp`, `rjmp`, играющих роль векторов прерываний) и возвраты (по команде `reti` – return from an interruption) из подпрограмм прерываний: при вызове прерываний содержимое регистра PC (указывающее на команду, перед которой прерывание произошло) запоминается в стеке и сбрасывается флажок прерываний $I: I \leftarrow 0$, а по команде `reti` PC восстанавливается из стека и I устанавливается: $I \leftarrow 1$, после чего работа основной, прерванной программы возобновляется с команды, перед которой прерывание произошло. (Глава 8.)

Если стек может хранить несколько адресов (значений PC), то мы получаем возможность выполнять вложенные подпрограммы, т.е. из подпрограммы

можно сделать переход на другую подпрограмму или даже на саму себя – это называется рекурсией. Очевидно, что глубина вложений подпрограмм зависит от емкости стека.

6.4. Команды изменения состояния системы

В эту группу входят всего 3 безоперандных команды (табл.6.6):

NOP – пустая команда.

Описание действия: Команда отрабатывается без выполнения операции.

SLEEP – перевод микроконтроллера в режим пониженного энергопотребления.

Описание действия: Команда устанавливает схему в SLEEP режим, определяемый регистром управления центрального процессорного устройства (ЦПУ). При выборке команды из программной памяти программный счетчик продвигается на 1, указывая на следующую за SLEEP команду (см. таблицу ниже), но перехода к ее выполнению при этом не происходит. Когда прерывание выводит ЦПУ из режима SLEEP, команда, следующая за командой SLEEP, будет выполнена прежде, чем отработает обработчик прерывания. Для пробуждения микроконтроллера может использоваться не только прерывание, но и сброс.

Формальное описание.

	Синтаксис	Операнды:	Счетчик программ:
(i)	SLEEP	None	PC <-- PC + 1

16-разрядный код операции:

1001	0101	100X	1000
------	------	------	------

Булевы выражения регистра статуса (SREG)

I	T	H	S	V	N	Z	C
-	-	-	-	-	-	-	-

WDR – сброс сторожевого таймера.

Действие: Команда сбрасывает (перезапускает) сторожевой таймер (Watch Dog Timer). Команда может быть выполнена внутри заданного предделителем сторожевого таймера промежутка времени (см. аппаратные характеристики сторожевого таймера).

Команды NOP и WDR выполняются за один машинный цикл, а команда SLEEP – за четыре машинных цикла.

Таблица 6.6 – Команды изменения состояния системы (управления системой)

Мнемоника	Название	Действие	Флаги
NOP	Нет операции	Нет	-
SLEEP	Спать (уменьшить энергопотребление)	Смотрите описание выше	-
WDR	Сброс сторожевого таймера	Смотрите описание выше	-

6.5. Расшифровка мнемоник команд

Ниже приведена расшифровка названий наиболее часто используемых команд

Команды передачи данных	
MOV Rd, Rs	Move between Registers
LDI Rd, K	Load Immediate
LD Rd, X	Load Indirect
LD Rd, X+	Load Indirect and Post-inc.
LD Rd, -X	Load Indirect and Pre-dec.
CLR Rd	Clear Register
SER Rd	Set Register
ST X, Rs	Store Indirect
ST X+, Rs	Store Indirect and Post-inc.
IN Rd, P	Input Port
OUT P, Rs	Output Port
PUSH Rs	Push Register on Stack
POP Rd	Pop Register from Stack
Команды с битами	
ADD Rd, Rs	Add Two Registers
ADC Rd, Rs	Add with Carry Two Registers
SUBI Rd, K	Subtract Constant from Register
FMUL	Fractional Multiply
FMULS	Fractional Multiply Signed
FMULSU	Fractional Multiply Signed with Unsigned
LSL Rd	Logical Shift Left
ROR Rd	Rotate Right through Carry
ASR Rd	Arithmetic Shift Right
Команды передачи управления	
SBRC Rs, b	Skip if Bit in Register Cleared
SBRs Rs, b	Skip if Bit in Register is Set
SBIC P, b	Skip if Bit in I/O Register Cleared
SBIS P, b	Skip if Bit in I/O Register is Set
RJMP k	Relative Jump

IJMP	Indirect Jump to (Z)
RCALL k	Relative Subroutine Call
ICALL	Indirect Call to (Z)
RET	Subroutine Return
RETI	Interrupt Return
CPSE Rd, Rs	Compare, Skip if Equal
CP Rd, Rs	Compare
CPC Rd, Rs	Compare with Carry
BRBS f, k	Branch if Status Flag Set
BRBC f, k	Branch if Status Flag Cleared
BREQ k	Branch if Equal
BRNE k	Branch if Not Equal
Прочие, часто используемые команды	
SEI	Global Interrupt Enable
CLI	Global Interrupt Disable
NOP	No Operation

Глава 7. Ассемблер AVR

В состав AVR Studio входит компилятор с языка ассемблера. Компилятор транслирует исходные коды с языка ассемблера в объектный код, который не требует линковки и может быть непосредственно запрограммирован в микроконтроллеры AVR. Полученный объектный код можно использовать в симуляторе Atmel AVR Studio либо в эмуляторе Atmel AVR In-Circuit Emulator, а также в программном эмуляторе XP-VMLAB. Компилятор работает под Microsoft Windows 9x.

Начиная с AVR Studio 4.11, язык ассемблера для AVR дополнен новой частью AVR Assembler 2 (в дальнейшем AVRASM2). Получилась совместимая замена старого Ассемблера (в дальнейшем AVRASM) с новыми характеристиками. В данном учебном пособии отражены обновления Ассемблера до версии AVRASM2.1.9. [13]

7.1. Лексемы

Изучение языка ассемблера (ЯА) начнем с рассмотрения **лексем**. *Лексемы* – это простейшие, неделимые конструкции языка, к которым относятся *имена (идентификаторы), числа и строки*.

Идентификаторы. Идентификаторы (имена) нужны для обозначения различных объектов программы – переменных, меток, названий операций, названий регистров и т.п.

В языке ассемблера (ЯА) идентификатор – это последовательность из латинских букв (больших и малых), цифр и следующих знаков:

? . @ _ \$

причем на эту последовательность накладываются следующие ограничения:

- длина идентификатора может быть любой, но значащими являются только первые 31 символ (идентификаторы, отличающиеся только в 32-й и последующих позициях, считаются одинаковыми);
- идентификатор не должен начинаться с цифры (7A – ошибка);
- точка может быть только первым символом идентификатора (.A – можно, A. – нельзя);
- в идентификаторах одноименные большие и малые буквы считаются эквивалентными, ассемблер их не различает (AX, Ax, aX, ax – одно и то же).

Особо подчеркнем, что в идентификаторах нельзя использовать буквы русского алфавита.

Идентификаторы делятся на *служебные слова* и *пользовательские имена*. Служебные слова встроены в ассемблер, имеют заранее определенный смысл, они используются для обозначения таких объектов, как регистры (R0, R1, SPL, SPH, PORTA и т.п.), названия команд (ADD, NEG и т.п.) и т.п. Все остальные идентификаторы вводятся самим программистом и называются именами. Про-

граммист может пользоваться ими по своему усмотрению, обозначая ими переменные, метки и другие объекты программы.

В качестве имен вообще-то можно использовать и некоторые служебные слова, однако настоятельно не рекомендуется этого делать.

Целые числа. В ЯА имеются средства записи целых и вещественных чисел. Вещественные числа мы рассматривать не будем, остановимся только на целых.

Целые числа на ассемблере AVR могут быть записаны в десятичной, двоичной, восьмеричной и шестнадцатеричной системах счисления (другие системы не допускаются). Десятичные числа записываются как обычно, а вот при записи чисел в других системах в начале числа ставится префикс – символы, которые указывают, в какой системе записано число. В начале двоичного числа ставится цифра 0 и буква b (binary): 0b, восьмеричные начинаются с нуля: 0, в начале шестнадцатеричного числа – символы 0x (ноль, икс) или знак доллара \$.

Примеры:

десятичные числа (приняты по умолчанию): 25, -126, +4, -85

двоичные числа: 0b101, -0b11000

восьмеричные числа: 010, 077

шестнадцатеричные числа: 0xAF, 0xAF, -0xAf, \$57, \$ff

Как и в случае идентификаторов, в числах малые и большие буквы (в том числе буквы-префиксы) отождествляются: 0x1A, 0x1a – одно и то же число; 0b1011и 0B1011 – тоже одно и то же число.

Символьные данные: символы и строки. Символы (символы внешнего алфавита ЭВМ) заключаются либо в одинарные, либо в двойные кавычки: 'А' или "А". Естественно, левая и правая кавычки должны быть одинаковыми: 'А' или "В" – ошибка.

Строки (последовательности символов) также заключаются либо в одинарные, либо в двойные кавычки: 'A+a' или "A+B".

Символы транслируются ассемблером в двоичные представления в коде ASCII, т.е. вместо символов ассемблер ставит их двоичные ASCII-коды *.

* ASCII (англ. American standard code for information interchange, ['æ.s.ki]) — название таблицы (кодировки, набора), в которой некоторым распространённым печатным и непечатным символам сопоставлены числовые коды. Таблица была разработана и стандартизована в США, в 1963 году.

Если в качестве символа или внутри строки надо указать кавычку, то делается это так: если символ или строку мы заключаем в одинарные кавычки, то символ одинарной кавычки («апостроф») надо удваивать, а вот символ двойной кавычки не надо удваивать, и наоборот, если внешние кавычки двойные, то двойная кавычка должна удваиваться, а одинарная не удваивается. Все следующие примеры корректны и эквивалентны: 'don' 't', 'don"t', "don't", "don""t". Можно также просто указать ASCII код кавычек: 27h код апострофа «'», 22h код двойных кавычек «"».

Собственно ради того, чтобы не удваивать внутренние кавычки, в язык и введены два вида кавычек, ограничивающих символы и строки.

AVRASM2 понимает строки и символы так же, как AVRASM, кроме того, служебные символы, которые не поддерживаются AVRASM. Строка, заключенная в двойные кавычки ("), может быть использована только вместе с директивой .DB. Строка передается буквально, никакие служебные символы и NUL-окончания не распознаются. Символьные константы, заключенные в одиночные кавычки ('), могут использоваться везде, где допустимо целое выражение. Служебные символы в Си-стиле распознаются с тем же значением, как в Си (табл.7.1).

Таблица 7.1 – Служебные символы в Си-стиле

№	Служебные символы	Назначение
1	<code>\n</code>	Конец строки (ASCII LF 0x0a)
2	<code>\r</code>	Перевод строки (ASCII CR 0x0d)
3	<code>\a</code>	Звуковой сигнал Alert bell (ASCII BEL 0x07)
4	<code>\b</code>	Возврат каретки (ASCII BS 0x08)
5	<code>\f</code>	Подача формы (ASCII FF 0x0c)
6	<code>\t</code>	Горизонтальная таб. (ASCII HT 0x09)
7	<code>\v</code>	Вертикальная таб. (ASCII VT 0x0b)
8	<code>\\</code>	Обратная косая черта
9	<code>\0</code>	Нулевой символ (ASCII NUL)

Ассемблером также распознаются `\ooo` (`ooo` = восьмеричное число) и `\xhh` (`hh` = шестнадцатеричное число).

Примеры:

```
.DB "Hello\n"
```

// - эквивалент:

```
.DB 'H', 'e', 'l', 'l', 'o', '\\', 'n',
```

Для того чтобы создать эквивалент Си-строки "Привет, мир \n", делают следующим образом:

```
.DB "Hello, world", '\n', 0
```

7.2. Выражения

Компилятор позволяет использовать в программе **выражения**. *Выражение* на языке программирования – это совокупность операндов, соединенных знаками действий (операторами) с возможным использованием скобок. К действиям над операндами относятся арифметические сложение, вычитание и др.; логические операции И, ИЛИ и др.; символьные действия. Выражение имеет («возвращает») значение, вычисляемое ассемблером путем выполнения указанных в нем действий над операндами. Соответственно выполняемым действиям различают арифметические, логические (выражения отношения) и символьные выражения (над символами и строками). Простейшими выражениями являются одиночные операнды.

Например:

1 – простейшее арифметическое выражение;

1+2 – арифметическое выражение: `ldi r16, 1+2;`

`c1>c2` – логическое выражение (выражение отношения). Возвращает 1, если левое выражение (`c1`) больше, чем правое (`c2`), и 0 в противном случае: `ori r18, bitmask*(c1>c2)+1;`

`'Привет'. $name` – символьное выражение. Вычисляет строку – конкатенацию строк. Операция `'.'` – конкатенация строк.

Все выражения являются 32-битными в AVRASM и 64-битными в AVRASM2. Дополнительно к операндам AVRASM AVRASM2 поддерживает выражения с плавающей точкой.

Операнды.

Могут быть использованы следующие операнды:

- Метки, определённые пользователем (дают значение своего положения).
- Выражения, определённые директивой SET.
- Константы, определённые директивой EQU.
- Числа.
- Константы с плавающей точкой.
- PC – текущее значение программного счётчика (PC – Programm Counter).
- Значения функций.

Препроцессор AVRASM2 распознает все форматы целых чисел, в том числе `$abcd` и `0b011001` признаются препроцессором и могут быть использованы в выражениях в директивах `#if`.

Операторы (знаки операций).

Таблица 7.2 – Операторы ассемблера AVR

Приоритет	Символ	Описание	Пример
1	2	3	4
14	!	Логическое отрицание возвращает 1, если выражение равно 0, и наоборот	<code>ldi R16, ! 0xF0</code> ; В R16 загрузить 0x00
14	~	Побитное отрицание возвращает выражение, в котором все биты проинвертированы	<code>ldi R16, ~0xF0</code> ; В R16 загрузить 0x0F
14	-	Минус возвращает арифметическое отрицание выражения (унитарный минус)	<code>ldi R16, -2</code> ; Загрузить -2 (0xFE) в R16
13	*	Умножение возвращает результат умножения двух выражений	<code>ldi R30, label*2</code>
13	/	Деление возвращает целую часть результата деления левого выражения на правое	<code>ldi R30, label/2</code>
13	%	Деление по модулю	<code>ldi R30, label%2</code> ; label делится по модулю 2
12	+	Суммирование возвращает сумму двух выражений	<code>ldi R30, c1+c2</code>
12	-	Вычитание возвращает результат вычитания правого выражения из левого	<code>ldi R17, c1-c2</code>

11	<<	Сдвиг влево возвращает левое выражение, сдвинутое влево на число бит, указанное справа	ldi R17, 1<<bit ; В R17 загрузить 1, сдвинутую влево bit раз
11	>>	Сдвиг вправо возвращает левое выражение, сдвинутое вправо на число бит, указанное справа	ldi R17, c1>>c2 ; В R17 загрузить c1, сдвинутое вправо c2 раз
10	<	Меньше чем: возвращает 1, если левое выражение меньше, чем правое (учитывается знак), и 0 в противном случае	ori r18, bitmask*(c1<c2)+1
10	<=	Меньше или равно: возвращает 1, если левое выражение равно или меньше, чем правое (учитывается знак), и 0 в противном случае	ori r18, bitmask*(c1<=c2)+1
10	>	Больше чем: возвращает 1, если левое выражение больше, чем правое (учитывается знак), и 0 в противном случае	ori r18, bitmask*(c1>c2)+1
10	>=	Больше или равно: возвращает 1, если левое выражение равно или больше, чем правое (учитывается знак), и 0 в противном случае	ori r18, bitmask*(c1>=c2)+1
9	==	Равно: возвращает 1, если левое выражение равно правому (учитывается знак), и 0 в противном случае	andi r19, bitmask*(c1==c2)+1
9	!=	Не равно: возвращает 1, если левое выражение не равно правому (учитывается знак), и 0 в противном случае	.SET flag = (c1!=c2) ; Установить flag равным 1 или 0
8	&	Побитное И возвращает результат побитового И выражений	ldi r18, High(c1&c2)
7	^	Побитное исключающее ИЛИ возвращает результат побитового исключающего ИЛИ выражений	ldi r18, Low(c1^c2)
6		Побитное ИЛИ возвращает результат побитового ИЛИ выражений	ldi r18, Low(c1 c2)
5	&&	Логическое И возвращает 1, если оба выражения не равны нулю, и 0 в противном случае	ldi r18, Low(c1&& c2)
4		Логическое ИЛИ возвращает 1, если хотя бы одно выражение не равно нулю, и 0 в противном случае	ldi r18, Low(c1 c2)
3	?	Условный оператор <i>Синтаксис:</i> Условие ? выражение 1 : выражение 2	ldi r18, >b? a: b ; Загрузка в r18 большего числа из двух чисел a и b

Функции.

В Ассемблере определены следующие функции:

- LOW(выражение) – возвращает младший байт выражения;
 HIGH(выражение) – возвращает второй байт выражения;
 BYTE2(выражение) – то же, что и функция HIGH;
 BYTE3(выражение) – возвращает третий байт выражения;

BYTE4(выражение) – возвращает четвёртый байт выражения;
 LWRD(выражение) – возвращает биты 0-15 выражения;
 HWRD(выражение) – возвращает биты 16-31 выражения;
 PAGE(выражение) – возвращает биты 16-21 выражения;
 EXP2(выражение) – возвращает 2 в степени (выражение);
 LOG2(выражение) – возвращает целую часть $\log_2(\text{выражение})$.

Следующие функции определены только в AVRASM2:

INT (выражение) – преобразовывает выражение с плавающей точкой в целое (т.е. отбрасывает дробную часть);
 FRAC(выражение) – выделяет дробную часть выражения с плавающей точкой (т.е. отбрасывает целую часть);
 Q7(выражение) – преобразовывает выражение с плавающей точкой в форму пригодную для инструкций FMUL/FMULS/FMULSU (знак + 7-битовая дробная часть);
 Q15(выражение) – преобразовывает выражение с плавающей точкой в форму пригодную для инструкций FMUL/FMULS/FMULSU (знак + 15-битовая дробная часть);
 ABS() – возвращает абсолютную величину постоянного выражения;
 DEFINED(символ) – возвращает «истина», если символ прежде определен директивами .equ, .set или .def. Обычно используется вместе с директивами if (.if defined(foo)), но может быть использовано в любом контексте. В отличие от других функций DEFINED(символ) требует наличия круглых скобок вокруг своего аргумента.

Ассемблер AVRASM2 различает регистр символов (AVRASM не различает).

7.3. Структура программы. Предложения

Программа на языке ассемблера записывается в виде последовательности *предложений*, каждое из которых располагается в отдельной строке:

```

<предложение>
<предложение>
...
<предложение>

```

Предложения называют также *операторами* или *строками* ассемблера.

Рассмотрим **правила записи предложений**. Предложение (строка ассемблера) не должно быть длиннее 120 символов. Если в предложении более 120 символов, то (если оно не продолжено; см. далее) 121-й и все последующие символы игнорируются. Записывать в одной строке два предложения нельзя.

При записи предложений действуют следующие правила расстановки пробелов:

- пробел обязателен между рядом стоящими идентификаторами и/или числами (чтобы отделить их друг от друга);
- внутри идентификаторов и чисел пробелы недопустимы;
- в остальных местах пробелы можно ставить или не ставить;
- там, где допустим один пробел, можно ставить любое число пробелов.

Эти правила не относятся к пробелам внутри строк, где пробел – обычный значащий символ.

Виды предложений. Предложение (строка исходного текста, входная строка) может представлять инструкцию (команду процессору), директиву (сообщение ассемблеру), комментарий, метку или быть пустой и иметь одну из пяти форм:

[метка:] инструкция [операнды] [Комментарий]

[метка:] директива [операнды] [Комментарий]

Комментарий

метка:

Пустая строка (которая иногда считается комментарием).

Инструкции описаны в главе 6, а их синтаксис в п.7.4, директивы – в п.7.5, а метки и комментарии – ниже.

Метки. *Метка* (также – «адресная метка») – это символьное имя (идентификатор), обозначающее ячейку памяти, которая содержит некоторую команду или переменную. Ассемблер транслирует метку в численное значение адреса этой ячейки. Поэтому метка является адресом, записанным в виде буквенно-цифрового, например мнемонического, имени. Это символьный, мнемонический адрес, мнемонадрес команды или переменной. Метки используются для указания места (точки, команды в программе), в которое передается управление при переходах; или они служат в качестве имен переменных.

Любое предложение на Ассемблере может начинаться с метки, после которой ставится двоеточие. Если метка устанавливается перед командой (метит ее), то она указывается в соответствующей команде перехода. В этом случае метка – это мнемонический адрес перехода, мнемонадрес команды, на которую производится переход.

Если метка устанавливается перед директивой, то она задает имя переменной, объявляемой этой директивой; либо служит мнемонадресом перехода на следующую содержательную строку, представляющую команду.

Метка в строке может быть и единственной. Тогда она тоже указывает на следующую, ближайшую к ней команду. Эта возможность полезна, когда команду надо пометить двумя или более метками; или когда метка очень длинная и поэтому остальная часть команды слишком сильно сдвигается вправо, что плохо смотрится.

Пример:

Initialization:

Lab: add R3, R12

Бывают символьные имена, которые обозначают константы — числа, встраиваемые в команды. Они тоже могут называться *метками*. Такая (константовая) метка транслируется не в адрес, а в численное значение константы.

Компилятор работает с **исходным текстом программы** (который иногда именуют «исходным кодом» или «исходником»). После обработки исходного текста компилятор выдает **двоичный код программы**, представляющий собой последовательность двоичных кодов инструкций. Директивы в двоичном коде отсутствуют. Они необходимы только на стадии компиляции для сообщения компилятору информации об атрибутах программы: именах и параметрах переменных, точках перехода, используемых сегментах памяти и пр. После компиляции директивы исчезают из программы, не входят в ее двоичный код.

Компилятор не требует, чтобы метки, директивы, комментарии или инструкции находились в определенной позиции (колонке) строки.

Комментарии не влияют на смысл программы, при трансляции ассемблер игнорирует их. Они предназначены для людей, они поясняют смысл программы.

Комментарий представляет собой любую последовательность символов внешнего алфавита ЭВМ, начинающуюся с символа ";" («точка с запятой») либо пустую строку (точнее строку, в которой нет иных символов, кроме пробелов). Комментарий имеет следующую форму:

; [Текст]

Позиции в квадратных скобках необязательны. Текст после точки с запятой (;) и до конца строки игнорируется компилятором.

Примеры:

```
label: .EQU c1=10 ; Устанавливает c1 равным 10 (Это директива)
.EQU c2=200 ; Устанавливает c2 равным 200 (Это тоже директива)
test: rjmp test ; Бесконечный цикл (Это инструкция)
test2: ; Одиночная метка, указывающая на следующую инструкцию
rjmp test2 ; Та же инструкция перехода
; Строка с одним только комментарием
; Еще одна строка с комментарием
```

Дополнительно к классическим комментариям ассемблера, начинающимся с ";", AVRASM2 признает комментарии в Си-стиле:

```
..... ; Остальная часть строки является комментарием.
// Подобно ';' остальная часть строки является комментарием.
/* Блок комментариев может располагаться в нескольких строках.
Этот стиль комментариев не может быть вложенным */
```

Ассемблер распознает разделители комментария (;) в стиле AVRASM, а также комментарии Си-стиля. Однако ';' используется в синтаксисе языка Си, что может привести к конфликту при использовании ';' в качестве разделителя комментария, поэтому не рекомендуется использовать комментарии в стиле Ассемблера вместе с директивами препроцессора AVRASM2.

Составные инструкции в строке. AVRASM2 допускает составные инструкции (команды) и директивы в строке, но их использование не рекомендовано. Это нужно для того, чтобы поддерживать распаковку (расширение) многострочных макроопределений препроцессора.

Продолжение предложений. Если предложение длиннее 120 символов, то подобно Си оно может быть продолжено посредством '\' – обратной косой черты в конце строки. Это полезно для длинных макроопределений препроцессора и для длинных директив .db.

Пример:

```
.db 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 11,12, 21, 214,235,634, '\n', 0, 2, \
12, 3,"это продолжение верхней строки", '\n', 0, 3, 0
```

Шаблон для любой AVR программы выглядит так:

```
1. .include "m16def.inc" ; Используем ATmega16
2. ;= Start macro.inc =====
3.
4. ; Тут будут наши макросы
5.
6. ;= End macro.inc =====
7.
8.
9. ; RAM =====
10.         .DSEG ; Сегмент ОЗУ
11.
12.
13.; FLASH =====
14.         .CSEG ; Кодовый сегмент
15.
16.
17.; EEPROM =====
18.         .ESEG ; Сегмент EEPROM
```

Кода в этом шаблоне 0 байт. Только директивы, выделяющие отдельные части (сегменты) программы. Необходимость в сегментах связана с тем, что в микроконтроллере AVR имеется несколько отдельных видов памяти (раздел 5). И в нем используется гарвардская архитектура. Т.е. исполняемый код в одном месте, а переменные в другом. Причем это разные адресные пространства. Т.е. адрес 0000 в сегменте данных это не то же что и адрес 0000 в сегменте кода. Это дает большое преимущество в том, что данные не могут испортить код, но не дает писать полиморфные программы *. Такая архитектура является традиционной для микроконтроллеров.

* *Полиморфизм* в языках программирования и теории типов – способность функции обрабатывать данные разных типов. Примеры разных типов данных: целые числа, числа с плавающей точкой, массивы, списки и т.д.

У обычного персонального компьютера используется другая архитектура – неймановская. Там данные и код находятся в одном адресном пространстве. Т.е., скажем, с адреса 0000 по 0100 идет код, а с 100 до FFFF данные.

В наших же программах и код и данные располагаются в разных адресных пространствах, и чтобы компилятор понял, где у нас что, они выделяются директивами сегментации. Директива `.DSEG` выделяет сегмент данных, располагающийся в СОЗУ (SRAM), `.ESEG` – дополнительный сегмент данных, располагающийся в постоянной памяти ЭСППЗУ (EEPROM), `.CSEG` – сегмент кода, располагающийся в перепрограммируемой постоянной памяти флэш ПЗУ (Flash ROM).

Таким образом, с помощью директив сегментации исходный текст делится на три сегмента. В сегменте кода записывается сама программа на ЯА, а в сегментах данных – данные (переменные).

Кроме того, в начале нашего шаблона стоит директива `.include "m16def.inc"` (для случая МК АТmega16). Она подключает к программе файл описаний имен регистров РОН и РВВ, которые (имена) не входят в перечень служебных, понятных компилятору, слов.

Ниже приведен **пример программы** на ассемблере AVR сложения элементов однобайтного массива чисел.

; Программа суммирования элементов однобайтного массива Array

```
.DSEG
    .ORG    0x100
Array: .BYTE 0x20    ; Зарезервировать 32 байта
                        ; памяти под массив Array начиная с адреса 0x100
Sum:   .BYTE 2       ; Зарезервировать 2 байта для переменной SUM,
                        ; принимающей сумму элементов массива
    .DEF    AcL=R0    ; Назначить младший байт аккумулятора в R0
    .DEF    AcH=R1    ; Назначить старший байта аккумулятора в R1
    .DEF    Next=R2   ; Очередной элемент массива в R2
    .DEF    NextH=R3  ; Старший байт очередного элемента массива в R3
    .DEF    Count=R16 ; Счетчик элементов в R16

.CSEG
    jmp InputArray    ; Перепрыгнуть через векторы прерываний
    ...               ; на некоторую программу ввода массива
InputArray:
    ...               ; Программа ввода массива
    jmp Start        ; Запустить программу суммирования
    .ORG    0x0500
Start:                ; Инициализация
    eor AcL, AcL      ; Сбросить
    eor AcH, AcH      ; аккумулятор
    eor NextH, NextH   ; В старшем байте элемента всегда ноль
    ldi XL, 0x00       ; Адрес массива 0x0100
    ldi XH, 0x01       ; в регистре X
```

```

ldi Count, 0x20 ; Число элементов в Count
Summa:
ld Next, X+      ; Очередной элемент занести в регистр Next
                  ; и продвинуть его адрес
add AcL, Next    ; Прибавить элемент к аккумулятору
adc AcH, NextH   ; с учетом переноса
dec Count        ; Это последний элемент?
brne Summa       ; Нет, повторить прибавление
sts Sum, AcL     ; Да, запомнить сумму в
sts Sum+1, AcH   ; в переменной Sum

```

7.4. Команды

Предложения-команды – это символьная форма записи машинных команд (инструкций). Общий синтаксис этого типа предложений таков:

[метка:] инструкция (ее мнемокод) [операнды] [Комментарий]

Примеры:

```

Lab: add R0, R1 ; сложение двух РОН
      neg R31   ; изменение знака
      cbr R15, 0b1111000 ; сброс битов РОН

```

Метки рассмотрены в п.7.3.

Мнемокод (мнемонический код) является обязательной частью команды. Это служебное слово, обозначающее операцию, выполняемую командой. Сами мнемокоды можно найти в описании системы команд микропроцессора в главе 6.

Операнды, или «собственно данные», * если они есть, отделяются друг от друга запятыми и могут быть *названиями регистров процессора, именами переменных, константами* (числами, строками, в том числе поименованными числами и строками) и, вообще, *выражениями*.

* Данными (в расширенном смысле) называют еще и команды.

Регистры процессора описаны в главах 5, 6.

Операнды (собственно данные) делятся на переменные и константы (непосредственные операнды). **Переменные** – это данные, которые могут изменяться в ходе работы программы. Сами переменные (их значения) размещаются не в программе (машинной команде), а в памяти данных. В командах указываются их имена, являющиеся мнемоническим обозначением их адресов в памяти. Адрес служит указанием, где хранится переменная в памяти. Таким образом, **имя переменной** – это мнемонический адрес той ячейки памяти, в которой хранится переменная, это – мнемонадрес переменной, а содержимое по этому адресу – это **значение переменной**, сама переменная. По адресу / мнемонадresу можно обращаться к переменной в памяти: считывать ее из памяти, записывать в память или преобразовывать. Например:

sts 100, R20 ; передать в 100-ю ячейку ОЗУ число из регистра R20

Итак, у переменной есть имя (адрес) и есть значение (содержимое по этому адресу).

Константы (числа и строки), в отличие от переменных, непосредственно включаются в состав команд при трансляции в качестве непосредственных операндов. Константы не изменяются в ходе работы программы. Они тоже могут обозначаться именами. Константой является также и адрес (или мнемонадрес) переменной, встроенный в команду. Но имя константы – это ее значение, а не ее адрес (как в случае переменной). Имена констант (как и имена переменных) объявляются в программе при помощи соответствующих директив ассемблера, которые мы поясним дальше. Например:

ldi R16, \$7F ; загрузить в регистр R16 константу \$7F

7.5. Директивы

Помимо машинных команд в программе на ЯА надо указывать и другие вещи. Например, надо сообщать, какие константы и переменные используются в программе, и какие имена мы им дали. Это делается с помощью **предложений-директив**. *Директивы* – это сообщения (приказы) для самого ассемблера, определяющие некоторые параметры программных объектов. Они используются только во время трансляции программы – перевода ее в машинный код. После трансляции директивы исчезают из кода программы (не включаются в него ассемблером, не транслируются в двоичные коды). Итак, директивы не транслируются ни в какие машинные коды. Это отличает их от команд, которые транслируются в свои машинные (двоичные) коды, образующие машинный код всей программы.

Практика показывает, что в любом ассемблере наиболее интенсивно используется только порядка 10...20 директив. Все остальные либо не являются обязательными, либо отвечают за управление лишь незначительными свойствами компилятора. К «основным», характерным и для ассемблеров других процессоров, относятся директивы `.equ`, `.org`, `.def`, `.cseg`, `.dseg` и т.д. Ну, а такие директивы, как `.dq`, `.exit`, `.listmac` в реальных программах встречаются действительно очень редко.

Синтаксис директив следующий [13]:

[метка:] директива [операнды] [Комментарий]

Пример:

```
label: .EQU var1=100 ; Устанавливает var1 равным 100 (Это директива)
      .EQU var2=200 ; Устанавливает var2 равным 200
```

Названия директив, как и мнемокоды, – это служебные слова, которые в ассемблере AVR начинаются с точки. Они будут указываться по мере знакомства с директивами.

Остальные части директив (операнды и комментарии) записываются так же, как и в командах.

Директивы AVRASM. [13]

Список директив AVRASM приведён в табл.7.3, а с директивами AVRASM2 можно познакомиться в [13]. Директивы AVRASM и AVRASM2 версии 2.1.9 предваряются точкой. Директивы AVRASM2 версии ниже 2.1.9 предваряются #.

Таблица 7.3 – Директивы AVRASM

Директива	Описание
<code>.BYTE</code>	Зарезервировать байты в ОЗУ
<code>.CSEG</code>	Программный сегмент
<code>.CSEGSIZE</code>	Определяет размер памяти программ
<code>.DB</code>	Определить байты во FLASH или EEPROM
<code>.DEF</code>	Назначить регистру символическое имя
<code>.DEVICE</code>	Определить устройство, для которого компилируется программа
<code>.DSEG</code>	Сегмент данных
<code>.DW</code>	Определить слова во FLASH или EEPROM
<code>.ENDM</code> , <code>.ENDMACRO</code>	Конец макроса
<code>.EQU</code>	Установить постоянное выражение
<code>.ESEG</code>	Сегмент EEPROM
<code>.EXIT</code>	Выйти из файла
<code>.INCLUDE</code>	Вложить другой файл
<code>.LIST</code>	Включить генерацию листинга
<code>.LISTMAC</code>	Включить разворачивание макросов в листинге
<code>.MACRO</code>	Начало макроса
<code>.NOLIST</code>	Выключить генерацию листинга
<code>.ORG</code>	Установить положение в сегменте
<code>.SET</code>	Установить переменный символический эквивалент выражения

Далее приводится подробное описание директив.

.BYTE – зарезервировать байты в ОЗУ

Директива BYTE резервирует байты в ОЗУ. Если вы хотите иметь возможность ссылаться на выделенную область памяти, то директива BYTE должна быть предварена меткой. Эта метка служит в качестве имени переменной или массива, располагающегося в выделенной области (ее первого байта). Директива принимает один обязательный параметр, который указывает количество выделяемых байт. Эта директива может использоваться только в сегменте данных (смотреть директивы CSEG и DSEG). Выделенные байты не инициализируются.

Синтаксис:

МЕТКА: .BYTE выражение

Пример:

```
.DSEG
var1: .BYTE 1 ; Резервирует 1 байт для var1.
```

```
table: .BYTE tab_size ; Резервирует tab_size байт.
.CSEG
    ldi r30, low(var1) ; Загружает младший байт регистра Z.
    ldi r31, high(var1) ; Загружает старший байт регистра Z.
    ld r1, Z ; Загружает VAR1 в регистр 1.
```

.CSEG – программный сегмент

Директива CSEG определяет начало программного сегмента. Исходный файл может состоять из нескольких программных сегментов, которые объединяются в один программный сегмент при компиляции. Программный сегмент является сегментом по умолчанию. Программные сегменты имеют свои собственные счётчики положения, которые считают не побайтно, а пословно. Директива ORG может быть использована для размещения кода и констант в необходимом месте сегмента. Директива CSEG не имеет параметров.

Синтаксис:

.CSEG

Пример:

```
.DSEG ; Начало сегмента данных.
var1: .BYTE 4 ; Резервирует 4 байта в ОЗУ.
.CSEG ; Начало кодового сегмента.
const: .DW 2 ; Разместить константу 0x0002 в памяти программ.
    mov r1, r0 ; Выполнить действия.
```

.CSEGSIZE – размер памяти программ

Устройства AT94K имеют раздел памяти, которую пользователь может присоединить к памяти программ AVR или памяти данных. Программа и данные SRAM подразделены на три блока: 10К x 16 память программ, 4К x 8 память данных и 6К x 16 или 12К x 8 перестраиваемой SRAM, которые могут быть распределены между программной памятью (до 4-х разделов по 2К x 16) и памятью данных (до 8-ми разделов по 4Кx8). Эта директива используется, чтобы определять размер программного блока памяти.

Синтаксис:

.CSEGSIZE = 10 | 12 | 14 | 16

Пример:

```
.CSEGSIZE = 12 ; Определить размер памяти программ как 12К x 16.
```

.DB – определить байты во флэш-памяти или EEPROM

Директива DB резервирует необходимое количество байт в памяти программ или в EEPROM. Если вы хотите иметь возможность ссылаться на выделенную область памяти, то директива DB должна быть предварена меткой. Директива DB должна иметь хотя бы один параметр. Данная директива может быть размещена только в сегменте программ (CSEG) или в сегменте EEPROM (ESEG).

Параметры, передаваемые директиве, – это последовательность выражений, разделённых запятыми. Каждое выражение должно быть или числом в

диапазоне (-128...255), или в результате вычисления должно давать результат в этом же диапазоне, в противном случае число усекается до байта, причём **без** выдачи предупреждений.

Если директива получает более одного параметра и текущим является программный сегмент, то параметры упаковываются в слова (первый параметр – младший байт), и если число параметров нечётно, то последнее выражение будет усечено до байта и записано как слово со старшим байтом, равным нулю, даже если далее идет ещё одна директива DB.

Синтаксис:

МЕТКА: .DB список_выражений

Пример:

```
.CSEG
consts: .DB 0, 255, 0b01010101, -128, 0xaa
.ESEG
const2: .DB 1, 2, 3.
```

.DEF – назначить регистру символьное имя

Директива DEF позволяет ссылаться на регистр через некоторое символическое имя. Назначенное имя может использоваться во всей нижеследующей части программы для обращений к данному регистру. Регистр может иметь несколько различных имен. Символическое имя может быть переназначено позднее в программе.

Синтаксис:

.DEF Символическое_имя = Регистр

Пример:

```
.DEF temp=R16
.DEF ior=R0
.CSEG
ldi temp, 0xf0 ; Загрузить 0xf0 в регистр temp (R16).
in ior, 0x3f ; Прочитать SREG в регистр ior (R0).
eor temp, ior ; Регистры temp и ior складываются по исключающему ИЛИ.
```

.DEVICE – определить устройство, для которого компилируется программа

Директива DEVICE позволяет указать, для какого устройства компилируется программа. При использовании данной директивы компилятор выдаст предупреждение, если будет найдена инструкция, которую не поддерживает данный микроконтроллер. Также будет выдано предупреждение, если программный сегмент либо сегмент EEPROM превысят размер, допускаемый устройством. Если же директива не используется, то все инструкции считаются допустимыми и отсутствуют ограничения на размер сегментов.

Синтаксис:

.DEVICE <код устройства>

Кодом устройства могут быть: AT90S1200 | AT90S2313 | AT90S2323 | AT90S2333 | AT90S2343 | AT90S4414 | AT90S4433 | AT90S4434 |

AT90S8515 | AT90S8534 | AT90S8535 | ATtiny11 | ATtiny12 | ATtiny22 |
 ATmega163 | ATmega103 AT90CAN128 | AT86RF401 | AT94K и т.д.

Пример:

```
.DEVICE AT90S1200 ; Используется AT90S1200
```

```
.CSEG
```

```
push r30 ; Эта инструкция вызовет предупреждение, поскольку
          ; AT90S1200 её не имеет.
```

Примечание:

Директива `.DEVICE` больше не используется в AVRASM2. Она заменена множеством директив `#pragma`, описывающих свойства устройства. В AVRASM2 директива `.device` эквивалентна `#pragma AVRPART PART_NAME`.

.DSEG – сегмент данных

Директива `DSEG` определяет начало сегмента данных. Исходный файл может состоять из нескольких сегментов данных, которые объединяются в один сегмент при компиляции. Сегмент данных обычно состоит только из директив `BYTE` и меток. Сегменты данных имеют свои собственные побайтные счётчики положения. Директива `ORG` может быть использована для размещения переменных в необходимом месте ОЗУ. Директива не имеет параметров.

Синтаксис:

```
.DSEG
```

Пример:

```
.DSEG ; Начало сегмента данных.
```

```
var1: .BYTE 1 ; Резервировать 1 байт для var1.
```

```
table: .BYTE tab_size ; Резервировать tab_size байт.
```

```
.CSEG
```

```
ldi r30, low(var1) ; Загрузить младший байт регистра Z.
```

```
ldi r31, high(var1) ; Загрузить старший байт регистра Z.
```

```
ld r1, Z ; Загрузить var1 в регистр r1.
```

.DW – определить слова во флэш или EEPROM

Директива `DW` резервирует необходимое количество слов в памяти программ или в EEPROM. Если вы хотите иметь возможность ссылаться на выделенную область памяти, то директива `DW` должна быть предварена меткой. Директива `DW` должна иметь хотя бы один параметр. Данная директива может быть размещена только в сегменте программ (`CSEG`) или в сегменте EEPROM (`ESEG`).

Параметры, передаваемые директиве, – это последовательность выражений, разделённых запятыми. Каждое выражение должно быть или числом в диапазоне (-32768...65535), или в результате вычисления должно давать результат в этом же диапазоне, в противном случае число усекается до слова, причем БЕЗ выдачи предупреждений.

Синтаксис:

```
МЕТКА: .DW список_выражений
```

Пример:

```
.CSEG
varlist: .DW 0, 0xffff, 0b1001110001010101, -32768, 65535
.ESEG
eevarlst: .DW 0,0xffff,10
```

.ENDM

.ENDMACRO – конец макроса

Директива определяет конец макроопределения и не принимает никаких параметров. Для информации по определению макросов смотрите директиву MACRO.

Синтаксис:

```
.ENDMACRO
```

```
.ENDM
```

Пример:

```
.MACRO SUBI16 ; Начало определения макроса.
    subi r16,low(@0) ; Вычесть младший байт первого параметра.
    sbci r17,high(@0) ; Вычесть старший байт первого параметра.
.ENDMACRO ; Конец определения макроса
```

.EQU – установить постоянное выражение

Директива EQU присваивает метке – символьному имени значение. Это имя может позднее использоваться в выражениях. Имя, которому присвоено значение данной директивой, не может быть переназначено и его значение не может быть изменено.

Синтаксис:

```
.EQU метка = выражение
```

Пример:

```
.EQU io_offset = 0x23
.EQU porta = io_offset + 2
.CSEG ; Начало сегмента данных.
    clr r2 ; Очистить регистр r2.
    out porta,r2 ; Записать в порт A.
```

.ESEG – сегмент EEPROM

Директива ESEG определяет начало сегмента EEPROM. Исходный файл может состоять из нескольких сегментов EEPROM, которые объединяются в один сегмент при компиляции. Сегмент EEPROM обычно состоит только из директив DB, DW и меток. Сегменты EEPROM имеют свои собственные побайтные счётчики положения. Директива ORG может быть использована для размещения переменных в необходимом месте EEPROM. Директива не имеет параметров.

Синтаксис:

```
.ESEG
```

Пример:

```
.DSEG ; Начало сегмента данных.
var1: .BYTE 1 ; Зарезервировать 1 байт для var1.
table: .BYTE tab_size ; Зарезервировать tab_size байт.
.ESEG
eevar1: .DW 0xffff ; Проинициализировать 1 слово в EEPROM.
```

.EXIT – выйти из файла

Встретив директиву EXIT, компилятор прекращает компиляцию данного файла. Если директива использована во вложенном файле (см. директиву INCLUDE), то компиляция продолжается со строки, следующей после директивы INCLUDE. Если же файл не является вложенным, то компиляция прекращается.

Синтаксис:

```
.EXIT
```

Пример:

```
.EXIT ; Выйти из данного файла.
```

.INCLUDE – вложить другой файл

Встретив директиву INCLUDE, компилятор открывает указанный в ней файл, компилирует, его пока файл не закончится или не встретится директива EXIT, после этого продолжает компиляцию начального файла со строки, следующей за директивой INCLUDE. Вложенный файл может также содержать директивы INCLUDE.

Синтаксис:

```
.INCLUDE "имя_файла"
```

Пример:

```
; файл iodefs.asm:
```

```
.EQU sreg = 0x3f ; Регистр статуса.
.EQU sphigh = 0x3e ; Старший байт указателя стека.
.EQU splow = 0x3d ; Младший байт указателя стека.
; файл incdemo.asm
.INCLUDE iodefs.asm ; Вложить файл.
in r0, sreg ; Прочитать регистр статуса.
```

.LIST – включить генерацию листинга

Директива LIST указывает компилятору на необходимость создания листинга. Листинг представляет из себя комбинацию ассемблерного кода, адресов и кодов операций. По умолчанию генерация листинга включена, однако данная директива используется совместно с директивой NOLIST для получения листингов отдельных частей исходных файлов.

Синтаксис:

```
.LIST
```

Пример:

```
.NOLIST ; Отключить генерацию листинга.
.INCLUDE "macro.inc" ; Вложенные файлы не будут
```

`.INCLUDE "const.def"` ; отображены в листинге
`.LIST` ; Включить генерацию листинга.

.LISTMAC – включить разворачивание макросов в листинге

После директивы LISTMAC компилятор будет показывать в листинге содержимое макроса. По умолчанию в листинге показываются только вызов макроса и передаваемые параметры.

Синтаксис:

`.LISTMAC`

Пример:

```
.MACRO MACX      ; Определение макроса.
    add r0,@0    ; Тело макроса.
    eor r1,@1
.ENDMACRO        ; Конец макроопределения.
.LISTMAC         ; Включить разворачивание макросов.
MACX r2,r1       ; Вызов макроса (в листинге будет показано тело макроса).
```

.MACRO – начало макроса

С директивы MACRO начинается определение макроса. *Макрос (макроопределение, макрокоманда)* – это микропрограмма, состоящая из последовательности операторов, которые несколько раз встречаются в тексте программы. Эта последовательность операторов включается в тело макроса. Кроме этого, макрос имеет заголовок, состоящий из имени и списка параметров, и окончание. После того как макрос определен, повторяющиеся участки кода в тексте программы заменяются на его имя. Это сокращает текст программы. При встрече имени макроса позднее в тексте программы компилятор заменит его имя на операторы из тела макроса. Таким образом, исполнение макроса заключается в его «расширении».

Макрос в AVRASM может иметь до 10 параметров, к которым в его теле обращаются через @0 – @9. Макрос в AVRASM2 может иметь неограниченное число параметров.

При вызове параметры перечисляются через запятые. Определение макроса заканчивается директивой ENDMACRO.

По умолчанию в листинг включается только вызов макроса, для разворачивания макроса необходимо использовать директиву LISTMAC. Макрос в листинге показывается знаком +.

Синтаксис:

`.MACRO имя_макроса`

Пример:

```
.MACRO SUBI16    ; Начало макроопределения.
    subi @1,low(@0) ; Вычесть младший байт параметра 0 из параметра 1.
    sbci @2,high(@0) ; Вычесть старший байт параметра 0 из параметра 2.
.ENDMACRO        ; Конец макроопределения.
.CSEG           ; Начало программного сегмента.
SUBI16 0x1234,r16,r17 ; Вычесть 0x1234 из r17:r16.
```

.NOLIST – выключить генерацию листинга

Директива NOLIST указывает компилятору на необходимость прекращения генерации листинга. Листинг представляет собой комбинацию ассемблерного кода, адресов и кодов операций. По умолчанию генерация листинга включена, однако может быть отключена данной директивой. Кроме того, данная директива может быть использована совместно с директивой LIST для получения листингов отдельных частей исходных файлов.

Синтаксис:

.NOLIST

Пример:

```
.NOLIST ; Отключить генерацию листинга.
.INCLUDE "macro.inc" ; Вложенные файлы не будут
.INCLUDE "const.def" ; отображены в листинге.
.LIST ; Включить генерацию листинга.
```

.ORG – установить положение в сегменте

Директива .ORG содержит параметр – значение выражения (см. ниже синтаксис директивы), указывающий адрес памяти, с которого будет располагаться следующий за этой директивой участок программы или данных. То есть, при помощи этой директивы можно разместить команду или переменную в памяти (команд или данных) микроконтроллера по любому адресу. Служит для ручного распределения памяти во время записи программы.

Директива .ORG устанавливает счётчик положения (адресный счётчик программы, счётчик адреса СчА) в заданное значение – значение выражения, которое передаётся как параметр. Счётчик положения – это переменная транслятора, равная смещению конкретной команды / данного относительно начала сегмента кода / сегмента данных. Иначе говоря, СчА содержит внутрисегментный адрес ячейки памяти, в которую будет занесена в процессе компиляции следующая после ORG команда или переменная программы. Если директива .ORG не используется, то (по умолчанию) для первой обрабатываемой команды СчА равен нулю, а далее, по ходу обработки следующих команд транслятором, он увеличивается на длину этих команд. Аналогично для переменных. Директива же .ORG принудительно устанавливает СчА на значение выражения.

Для сегмента данных директива .ORG устанавливает счётчик положения в SRAM (CO3Y), для сегмента программ это положение во Flash ROM (флэш-ПЗУ), а для сегмента EEPROM это положение в EEPROM. Если указана <метка>, то она будет иметь адрес <выражение>. Перед началом компиляции счётчик Flash ROM и счётчик EEPROM равны нулю, а счётчик SRAM равен 0x60 (поскольку адреса 0-0x5F заняты регистрами). Поэтому если в тексте программы не встречается директива .ORG, то программа начинается с нулевого адреса кодового сегмента, а данные-переменные – с адреса 0x60 сегмента данных.

Коротко: директива .ORG сообщает ассемблеру, куда, по какому внутрисегментному адресу тот должен поместить следующее за ней данное – команду или переменную.

Синтаксис:

.ORG выражение

Пример:

```
.DSEG    ; Начало сегмента данных.
.ORG 0x37 ; Установить адрес SRAM равным 0x37.
variable: .BYTE 1 ; Зарезервировать байт по адресу 0x37H.
.CSEG
.ORG 0x10 ; Установить счетчик положения равным 0x10.
mov r0, r1 ; Данная команда будет размещена по адресу 0x10.
```

.SET – присвоить переменной значение выражения

Директива SET присваивает имени некоторое значение. Это имя позднее может быть использовано в выражениях. Причем в отличие от директивы EQU значение имени может быть изменено другой директивой SET.

Синтаксис:

.SET имя = выражение

Примеры:

1)

```
.SET io_offset = 0x23
.SET porta = io_offset + 2
.CSEG;    Начало кодового сегмента.
clr r2;   Очистить регистр 2.
out porta, r2 ; Записать в порт A.
```

2)

```
.SET FOO = 0x114 ; Set FOO to point to an SRAM location,
lds r0, FOO ; load location into r0,
.SET FOO = FOO + 1 ; increment (redefine) FOO.
; This would be illegal if using .EQU.
lds r1, FOO ; Load next location into r1.
```

Глава 8. Прерывания

8.1. Понятие прерываний

Общие сведения о прерываниях [15]. Во время выполнения процессором текущей программы в периферийных устройствах или во внешней среде, обменивающейся информацией с процессором через ПУ, а также внутри самого процессора могут возникать события, требующие немедленной реакции на них со стороны процессора. Например: получение неправильного кода команды, деление на нуль, поступление импульсов от системного таймера, нажатие кнопки «мыши» пользователем, приход сигнала об аварии на управляемом технологическом объекте и т.д.

Реакция состоит в том, что процессор прерывает обработку текущей программы и переходит к выполнению некоторой другой программы, специально предназначенной для данного события. По завершении этой программы процессор возвращается к выполнению прерванной программы.

Рассматриваемый процесс, называемый *прерыванием программы*, поясняется на рис.8.1. Принципиально важным является то, что моменты возникновения событий, требующих прерывания программ, заранее неизвестны и поэтому не могут быть учтены при программировании.

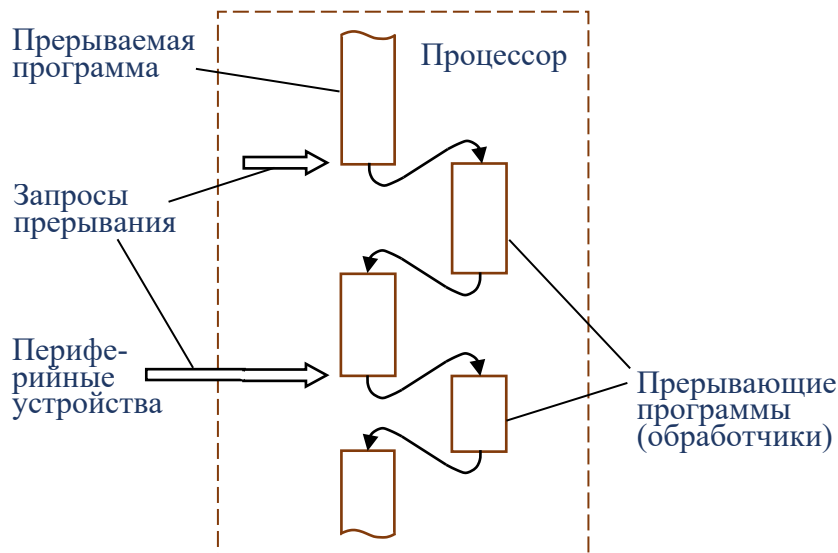


Рисунок 8.1 – Прерывания программы

Каждое событие, требующее прерывание, сопровождается сигналом, оповещающим процессор. Назовем эти сигналы *запросами прерывания*. Программу, затребованную запросом прерывания, называют *прерывающей программой* или (чаще) *подпрограммой* (так как она может запускаться многократно при повторных возникновениях прерывающих событий), противопоставляя ее *прерываемой программе*, выполнявшейся машиной до появления запроса.

Виды прерываний. Прерывания принято делить на *внешние (аппаратные, асинхронные)* и *внутренние (программные, в большинстве своем синхронные)* (рис.8.2) [16].

Внешними, аппаратными (Interruptions) называются прерывания, исходящие от периферийных устройств и из внешней среды при возникновении в них событий, требующих немедленной реакции процессора.

Внутренними, программными, или особыми случаями (исключениями) в программе (Exceptions) считаются прерывания, возбуждаемые изнутри процессора, а также так называемые «чисто программные прерывания». Прерывания изнутри процессора порождаются средствами его диагностики при возникновении аномальных ситуаций в процессоре или в программе – при делении на ноль, получении неправильного кода команды и пр. Их условно относят к программным прерываниям, поскольку они являются реакцией процессора на ошибки, зафиксированные при выполнении программы. Прерывания, названные чисто программными, в действительности, никакими прерываниями не являются. Они представляют собой просто вызовы подпрограмм из основной программы, запланированные программистом и осуществляемые при помощи специальных команд (например команд `int 3`, `into`, `int n` и `bound` для процессоров IA-32, AMD64). А называются они так по сходству с внешними и внутренними прерываниями в части использования одностипных с ними адресов переходов на подпрограммы.

Микроконтроллеры не снабжены внутренними прерываниями.

Прерывания

- **аппаратные (внешние)** – «истинные» прерывания (**Interruptions**). Вызываются внешними сигналами, поступающими на процессор от периферийных устройств или из внешней среды через периферийные устройства
- **программные (внутренние и чисто программные)** – особые случаи в программе (**Exceptions**)
 - внутренние прерывания, вызываемые средствами диагностики самого процессора при аномальных ситуациях в процессоре или программе
 - чисто программные прерывания, происходящие по специальным командам (`int3`, `into`, `int n` и `bound` для IA-32 / AMD64)

Рисунок 8.2 – Разновидности прерываний

Внешнее (аппаратное) прерывание служит мощным средством асинхронных переходов по программе. [16] Как правило, переходы по программе происходят под управлением самой программы. Однако, в любой момент времени у того или иного внешнего (по отношению к процессору) устройства может возникнуть внеочередная, не предусмотренная этой программой необходимость выполнения совместных операций с процессором. Она вызывается такими событиями, как:

- нажатие клавиши на клавиатуре или кнопки «мыши» пользователем для ввода символов в редактируемый текст или команд в операционную систему, аналогично отжатию клавиши/кнопки для завершения ввода;
- подключение «флэшки» к USB-порту;

- сообщение о возникновении ошибок на шине или в памяти, об аварийном выключении сети;
- отладка программы пользователем;
- тактирование системным таймером работы процессора для поочередного («параллельного») исполнения нескольких задач;
- тактирование функциональным таймером выдачи управляющих воздействий на технический объект в системах управления;
- приход сигналов от датчиков о возникновении аварийных и нештатных ситуаций на технологическом оборудовании;
- другие причины, требующие немедленной реакции процессора.

В этом случае внешнее устройство может прервать выполняемую, основную программу (послав на процессор специальный сигнал – запрос прерывания ЗПр, INTR – Interrupt, Interruption) и заставить процессор перейти на другую, нужную ему программу (подпрограмму), которая и осуществит необходимое обслуживание устройства: введет данные с клавиатуры или "мыши", организует обмен информацией с флэш-ПЗУ и т.п. При этом адрес передачи управления (*адрес перехода*) на программном счетчике процессора задается внешним устройством (путем отработки специальной автоматической процедуры взаимодействия с процессором; п.8.2). В результате процессор переходит по заданному адресу, начиная с которого располагается подпрограмма обслуживания внешнего устройства – *прерывающая подпрограмма* (также *подпрограмма прерывания*, *подпрограмма обслуживания прерывания*, *подпрограмма обработки прерывания*, *процедура прерывания*, *обработчик прерывания* – ПП, Interruption SubRoutine – ISR, Interrupt Handler).* По завершении подпрограммы процессор возвращается к выполнению прерванной программы (рис.8.3).

Описанный режим называется *аппаратным (внешним) прерыванием*.

* Subroutine [sʌbruːˈtiːn] с англ. – подпрограмма, функция, процедура. Handler – обработчик; дрессировщик, укротитель. Handle – ручка, рукоять, рукоятка; делать что-либо руками, регулировать.

Принципиально **построение подпрограмм обслуживания внешних прерываний** не отличается от любых других программ, выполняемых процессором. Отличие состоит лишь в том, как организован переход к этим подпрограммам. Этот переход происходит не по указанию выполняемой в данный момент программы, поскольку, как уже отмечалось, он не предусмотрен ею и может понадобиться в любой момент, в любой точке основной программы, на любой ее команде. Данный переход осуществляется по требованию внешнего устройства, посылающего на процессор специальный сигнал – запрос прерывания и сообщаящего процессору *адрес перехода* на подпрограмму прерывания.

В конце подпрограммы прерывания ставится команда "вернуться из прерывания". По ней процессор вновь переключается на исполнение основной, прерванной программы, возвращаясь в точку прерывания. Правильный возврат происходит благодаря тому, что в момент реакции на запрос прерывания адрес команды, перед которой прерывание произошло, находящийся в программном

счетчике процессора, (*адрес возврата*) автоматически запоминается в стеке. В конце ПП этот адрес, наоборот, программно, по команде возврата, извлекается из стека и заносится обратно в программный счетчик. Тем самым обеспечивается возврат в правильную точку – точку прерывания основной программы, на команду, перед которой произошло прерывание (на команду по адресу 0106 на рис.8.1.2). При этом текущая команда, на которую пришелся приход запроса прерывания, (расположенная по адресу 0104 на рис.8.1.2) выполняется до полного завершения.



Рисунок 8.3 – Переход на подпрограмму обслуживания прерывания (ПП).

Адрес перехода равен А. Адрес возврата из прерывания 0106.

Запрос прерывания ЗПр поступил на процессор в момент выполнения команды, расположенной в ячейках 0104, 0105.

Итак, **прерыванием (внешним, аппаратным прерыванием)** называется временное прекращение выполнения текущей ("основной") программы и переход процессора к другой программе – подпрограмме обслуживания прерывания (ПП, ISR), если этот переход не был предусмотрен текущей программой, а произошел асинхронно по отношению к ней, на произвольной, случайно подвернувшейся в данный момент команде и под управлением внешнего устройства, а не по указке программы.

Коротко, **прерывание** – это асинхронный аппаратный переход по программе.

Внутренние (программные) прерывания. Аналогичный режим – режим прерывания может быть вызван средствами диагностики самого процессора при обнаружении ошибки в его функционировании или функционировании программы. Такие прерывания называют исключениям в программе. Исключения происходят при возникновении в процессоре таких событий, как появление ошибки в работе его аппаратуры, поступление неправильного кода

команды, переполнение разрядной сетки, попытка деления на 0, выход из установленной для данной программы области памяти и др.

Адрес перехода на подпрограмму внутреннего прерывания задается изнутри процессора. Адрес возврата из внутреннего прерывания, как правило, устанавливается равный адресу команды, на которой поступил внутренний запрос при ошибке ее исполнения, а не адресу следующей за ней команды, как в случае внешних прерываний.

Переход на ПП может быть вызван, как мы сказали, и специальными командами прерывания (для процессоров IA-32, AMD64 – командами `int 3`, `into`, `int n` и `bound`), осуществляющими обычный вызов ПП и, по сути, ничего не прерывающими, а являющимися запланированными действиями программиста (чисто программные прерывания). Такие прерывания также условно считаются исключениями. В этом случае адрес перехода на ПП задает сама команда прерывания.

Подпрограммы обслуживания программных прерываний строятся аналогично с внешними, аппаратными, с записью в конце общей для всех видов прерываний команды "вернуться из прерывания".

Истинные прерывания (Interruptions) и особые случаи, или исключения, в программе (Exceptions). К прерываниям в собственном смысле слова следует отнести лишь их аппаратные разновидности: внешние (Interruptions) и собственно исключения (Exceptions) – внутренние, вызванные средствами диагностики самого процессора. В то же время так называемые «чисто программные прерывания» представляют собой банальные вызовы подпрограмм из основной программы с помощью специальных команд. Они именуются так лишь потому, что используют общие с аппаратными прерываниями вектора (адреса подпрограмм). При чисто программных прерываниях (как и при аппаратных) происходит также автоматическое запоминание не только адреса возврата, но и состояние некоторых наиболее важных регистров (например, флажков) в стеке (при обычных вызовах подпрограмм – по команде `call` – эти регистры не запоминаются).

Среди аппаратных внешние прерывания являются безусловно асинхронными, непредсказуемыми в отношении моментов их наступления. Внутренние прерывания не обладают таким свойством безоговорочно и по ряду свойств занимают промежуточное положение между внешними и чисто программными, имея признаки и тех и других. Многие внутренние прерывания могут быть выработаны в самых неожиданных местах программы: прерывания по ошибочному командному коду, нарушению правил доступа к сегментам памяти и т.д. Такие прерывания похожи на внешние в той части, что они, как и внешние, происходят асинхронно по отношению к выполняемой программе. Другие внутренние прерывания, наоборот, имеют сходство с программными, поскольку, как и программные, назначаются самим программистом: например, пошаговое прерывание, применяемое для отладки программы или проверки исправности оборудования; либо они, подобно программным, могут наступать только в определенных местах программы: прерывание по делению на «0» генерируется командами деления `div` и `idiv` при делении на нуль или малое число, если частное не помещается в разрядную сетку регистра-результата.

Тем не менее, общим для внутренних прерываний является то, что все они, в конечном итоге, порождаются программой, являются следствием программных событий. Запросы прерываний, поступающие от внутренних блоков процессора, свидетельствуют о возникновении исключительной ситуации в ходе работы программы: ошибках или затруднениях при выполнении команд, пошаговом прохождении программы, нарушении границ сегмента или правил защиты памяти и т.д. По этой причине внутренние прерывания и причисляют к программным, называя их *особыми случаями* или *исключениями* в программе (*exceptions*). Для удобства терминологии исключения именуют и собственно программные вызовы «подпрограмм прерываний», происходящие по командам `int` процессоров IA-32 и др.

В противоположность этому внешние прерывания не связаны с ходом работы программы. Они являются следствием исключительно аппаратных, внешних по отношению к процессору событий и происходят всегда в произвольные моменты времени, асинхронно к

процессу выполнения программы. Поэтому они являются прерываниями в полном смысле этого слова, *истинными прерываниями (interruptions)*.

Значения термина «прерывание». В технической литературе термин «прерывание» используется в нескольких родственных, но не вполне совпадающих значениях. Прерыванием называют, во-1-х, как было сказано, особый режим работы ЭВМ, заключающийся во внеочередном, не предусмотренном основной программой переходе процессора на другую программу – подпрограмму прерывания и последующем ее выполнении. Во-2-х, прерыванием называют саму подпрограмму обслуживания прерывания (ПП), или, как говорят, *процедуру прерывания, обработчик прерывания*, «прерывание». В-3-х, прерыванием или подтверждением прерывания называют первую фазу режима прерываний – автоматическую аппаратную реакцию процессора на сигнал запроса прерывания, не относя к ней исполнение собственно ПП. И, наконец, в-4-х, прерыванием именуют внешний сигнал, переводящий процессор в режим прерывания – сигнал запроса ЗПр, INTR (Interrupt).

Прерывания от нескольких устройств. Понятие аппаратного прерывания можно обобщить для случая нескольких периферийных устройств. Любое из устройств может прислать свой запрос на внимание. Обработка запросов следует по принципу обслуживания «первым вошел – первым вышел», либо по некоторой приоритетной схеме в соответствии с приоритетом запросов. Для развитых процессоров эта операция (*арбитраж*) выполняется с помощью специальных микросхем – блоков приоритетных прерываний, или контроллеров прерываний (п.8.2 и рис.8.4, 1.1). Для микроконтроллеров она осуществляется аналогичным устройством, встроенным в МК и называемым чаще схемой, модулем прерываний, а не контроллером прерываний.

Резюме. При всех случаях способность обрабатывать прерывания может оказаться эффективным средством во многих приложениях микропроцессоров и микроконтроллеров. Процессор переключается на обработку прерываний по адресу, аппаратно задаваемому внешним устройством или изнутри процессора (*адресу перехода*). Поскольку при прерываниях нужно сохранять информацию о текущем состоянии процессора, можно воспользоваться стеком, т.е. сохранить в нем внутреннее состояние процессора в момент поступления запроса, включая программный счетчик, т.е. адрес команды, перед которой или на которой запрос поступил (*адрес возврата из прерывания*).

Все МП, в том числе и процессоры МК, реагируют на запросы только после завершения текущей команды, так как проверяют состояние входа ЗПр в промежутке между командами. Поэтому исполнение текущей команды не обрывается где-то посередине, а доводится до конца.

8.2. Обработка прерываний

Обработка внешних прерываний. Процессоры могут реагировать на запросы прерываний по-разному. В каждой архитектуре реализуется своя оригина-

нальная система обслуживания прерываний. Однако общий порядок обработки запросов всегда содержит одни и те же действия процессора [16].

Внешнее устройство (ВУ), нуждающееся во внеочередном обслуживании процессором, посылает на него асинхронный сигнал запроса (требования) прерывания ЗПр (ТПр). Сигнал посылается по специальной линии, которая называется IRQ-линией (Interruption ReQuest или ReQuire – запрос или требование прерывания), причем каждому ВУ отвечает своя линия IRQ (рис.8.4). Эти линии заводятся на входы особого устройства – контроллера прерываний *. Контроллер запоминает запросы и формирует на их основе сигнал прерывания INTR (Interrupt, Interruption) на процессор, соответствующий источнику с наибольшим приоритетом в данный момент. Если поступает одновременно несколько запросов, контроллер пропускает на процессор запрос с бóльшим приоритетом. Остальные запросы запоминаются контроллером и обслуживаются позже. Приоритеты устанавливаются при включении компьютера по умолчанию в последовательности расположения линий IRQ – линиям с меньшим номером отдается бóльший приоритет; но могут быть и перепрограммированы.

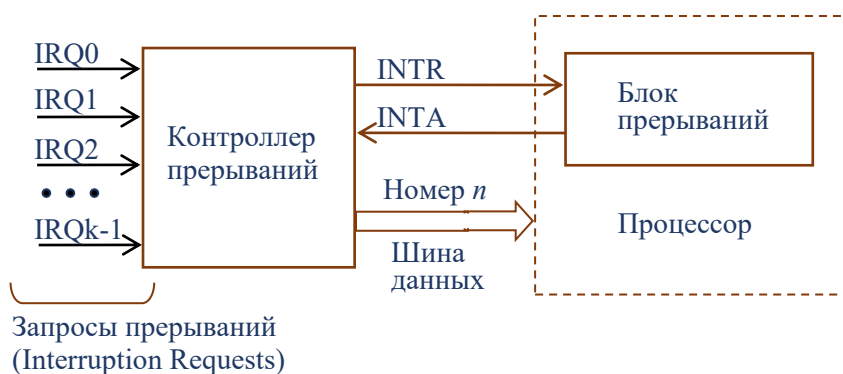


Рисунок 8.4 – Подача запросов прерывания в процессор.

Номер прерывания n служит для определения адреса перехода на подпрограмму прерывания

* *Контроллер прерываний* – микросхема или встроенный модуль процессора, отвечающий за возможность последовательной обработки запросов на прерывание от разных устройств (см. рис.8.4 и 1.1). Как правило, представляет собой электронное устройство, выполненное как часть самого процессора или же сложных микросхем его обрамления, входы которого присоединены электрически к соответствующим выходам внешних устройств.

Перед выдачей на процессор сигнала INTR контроллер идентифицирует прерывающее устройство и выявляет его индивидуальный числовой код n . Этот код называется *номером прерывания, типом прерывания*. (Для процессоров IA-32, AMD64, например, это 8-битное число.) Номер прерывания n присваивается каждому ВУ, способному вырабатывать запросы прерывания, служит его уникальным именем, идентификатором, с помощью которого его можно распознать среди множества других устройств. Присвоение номера происходит в соответствие со входом IRQ, используемым ВУ. За каждым входом закреплён

фиксированный номер прерывания (назначаемый при помощи внутреннего шифратора контроллера). Он то и присваивается устройству, подключенному к данному входу. После того, как схемная разводка запросов от ВУ по входам IRQ произведена, номера прерываний становятся строго закрепленными за устройствами. Поскольку контроллер знает, на какой вход поступил сигнал запроса, то он знает и номер n устройства – источника запроса и идентифицирует его по этому номеру n .

Следует отличать номер входа контроллера IRQ от приоритета прерывания, а также от номера, типа прерывания n , присваиваемого устройству и используемого как номер входа в таблицу векторов прерываний (понятие таблицы векторов прерываний см. ниже). Например, в IA-32 и AMD64 в реальном режиме работы процессора прерывание от стандартной клавиатуры использует вход контроллера №1 (IRQ1) и номер прерывания $n = 9$ (INT 9), прерывание от «мыши» – вход №12 (IRQ12) и номер прерывания $n = 74h$ (INT 74h).

Завершив идентификацию прерывающего устройства (определив его номер n), контроллер пропускает запрос прерывания на процессор в виде сигнала INTR. Реагируя на сигнал INTR, процессор заканчивает выполнение текущей команды и, при условии, что прерывания разрешены, прерывает свою работу по исполнению текущей программы и совершает далее автоматически следующие действия:

- а) отвечает контроллеру сигналом подтверждения прерывания ППр, или INTA (Interrupt Acknowledge), под действием которого контроллер сообщает процессору по шине данных номер прерывания n , идентифицирующий прерывающее устройство. На основании номера n процессор устанавливает местоположение (адрес) ячейки памяти, в которой хранится *адрес перехода* на подпрограмму, обслуживающую данное устройство.
- б) запоминает *адрес возврата* из подпрограммы – текущее содержимое программного счетчика РС и, возможно, другую полезную информацию о своем состоянии, например флажки, в стеке.
- в) считывает из ячейки памяти, найденной при помощи номера n , *адрес перехода* на подпрограмму прерывания для данного прерывающего устройства и заносит его в программный счетчик РС.

Таким образом, в результате операций «а» и «в» процессор узнаёт, куда ему надо прыгнуть по памяти, чтобы попасть на ПП. А в результате операции «б» процессор запоминает, откуда он прыгнул, чтобы потом, после завершения ПП, вернуться в исходное место и продолжить выполнение прерванной программы.

Отметим, что операция «а» реализуется в форме одного или нескольких (в зависимости от типа микропроцессора) машинных циклов типа «Прерывание» («Подтверждение прерывания»). Операция «б» – в форме одного или нескольких циклов «Запись в стек». Операция «в» – в форме одного или нескольких циклов «Чтение памяти». *

* *Машинные циклы* см. в Приложении Б.

После завершения указанных действий процессор переходит по адресу, занесенному в РС, на подпрограмму прерывания и выполняет ее. В конце подпрограммы ставится команда "вернуться из прерывания", по которой процессор

возобновляет исполнение прерванной, фоновой программы, возвращаясь в точку прерывания – на ту команду фоновой программы, перед которой прерывание произошло. Это происходит за счет того, что адрес возврата и информация о состоянии извлекается процессором из стека и заносится обратно в РС и соответствующие регистры. Для процессоров IA-32, AMD64 возврат производится по команде `iret` (Interrupt Return), для микроконтроллеров AVR – по команде `reti` (Return from Interruption).

Кратко: при обработке процессором сигнала запроса прерывания:

- адрес перехода на ПП загружается в регистр программный счетчик РС, а
- адрес возврата и информация о состоянии процессора – в стек.

Далее процессор переходит на ПП и выполняет ее.

По завершении ПП следует возврат из нее, происходящий по команде возврата, установленной в конце ПП, и выполнение прерванной программы возобновляется.

Векторы прерываний и их таблица. Адрес перехода на ПП называется *вектором прерывания* (ВП). Он как бы указывает направление на подпрограмму (отсюда его название). ВП определяет *пусковой адрес* подпрограммы – адрес ее стартовой команды. Как правило, все векторы системы прерываний компьютера * хранятся в начальной области памяти в виде одномерного непрерывного массива данных, называемого *таблицей прерываний* или *таблицей векторов прерываний*. Номером входа в эту таблицу как раз и служит упомянутый выше номер прерывания *n*, привязанный к прерывающему устройству. Таблица ВП для микроконтроллера AVR ATmega16 будет рассмотрена в п.8.3 (табл.8.1).

* *Системой прерываний* называют совокупность всех возможных прерываний компьютера.

Прерывание – аппаратно-программный режим работы процессора. Мы пояснили, что сигнал прерывания ЗПр (INTR) принуждает процессор выполнить predetermined операции «а», «б» и «в». Эти операции реализуются в форме соответствующих машинных циклов и не требуют управления со стороны программы. Они «встроены» в электронную схему процессора, генерируются ею автоматически в ответ на появление активного уровня напряжения на входе процессора ЗПр. Эти операции (циклы) не могут быть изменены пользователем никакими программными и аппаратными средствами. Они обязательны, фиксированы для процессора данного типа и не зависят ни от работы программы, ни от действий внешних устройств. Они представляют собой, так сказать, «рефлексивную» реакцию процессора на сигнал прерывания.

Указанные операции составляют первую фазу режима прерывания. Эта фаза реализуется исключительно аппаратными средствами процессора и только в ответ на сигнал прерывания, а не по указке машинных команд.

После завершения первой фазы процессор переходит ко второй: он выполняет подпрограмму обслуживания периферийного устройства, запросившего прерывание. Все действия процессора в этой фазе происходят под управлением программы (подпрограммы прерывания).

Таким образом, прерывание представляет собой аппаратно-программный режим работы процессора, состоящий из двух последовательных фаз – аппаратной и программной.

Разрешение и запрещение внешних прерываний. Процессор воспринимает сигнал прерывания ЗПр (INTR) не при любых условиях, а только в том случае, если прерывание разрешено – при этом говорят, что процессор *не замаскирован* от прерываний. В противном случае, если прерывание запрещено, сигнал ЗПр не проходит в процессор – процессор «замаскирован» от прерываний. Прохождением запросов прерываний в процессор управляет специальный внутренний триггер процессора, называемый *триггером* или *флагом разрешения прерывания* IF или I (Interruption Flag). У некоторых микропроцессоров этот триггер называется триггером приоритета. У МП IA-32, AMD64 флаг IF входит в состав регистра флагов и может блокировать сигнал ЗПр (INTR) (рис.8.5). У МК AVR флаг I глобального разрешения прерываний тоже входит в состав регистра флагов – регистра состояния SREG. Обычно, в т.ч. у МП IA-32, AMD64 и МК AVR, в состоянии логической «1» данный триггер разрешает прохождение запросов в процессор, а в состоянии логического «0» – блокирует запросы (см. рис.8.5). Для ряда процессоров значения состояния триггера – триггера приоритета интерпретируются наоборот: при единичном состоянии триггера процессор не реагирует на сигнал прерывания, т.е. имеет высокий приоритет; при нулевом состоянии – воспринимает этот сигнал – низкий приоритет процессора.

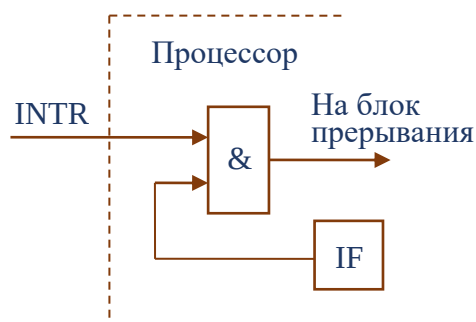


Рисунок 8.5 – Прохождение запросов в процессор

Посредством специальных машинных команд триггер разрешения прерывания доступен программисту, и программист может изменять его нужным образом, разрешая или запрещая прерывания (команды `cli` и `sti` процессоров IA-32, AMD64; `cli` и `sei` МК AVR, глава 6). Кроме того, для многих процессоров (в том числе для IA-32 и AVR) этот триггер после восприятия очередного запроса автоматически переходит в состояние, соответствующее блокированию дальнейших запросов, выход из которого возможен посредством команды, разрешающей прерывание. Для этих процессоров данный триггер блокирует прерывания также сразу после пуска по сигналу сброса Сбр (RESET). Следовательно, сразу после пуска процессора прерывания запрещены.

Запросы прерываний могут быть также дополнительно заблокированы локально, внутри самих периферийных устройств путем сброса определенных триггеров их регистров в 0; и разблокированы установкой в 1.

При использовании конкретного процессора необходимо внимательно анализировать все ситуации, которые приводят к маскированию и демаскированию прерываний.

Обработка внутренних и чисто программных прерываний. Прерывания от источников внутри процессора обрабатываются в том же порядке, что и внешние прерывания, но не требуют обращения процессора к периферийному устройству (контроллеру прерываний) для считывания номера прерывания; этот номер задается внутренней схемой диагностики самого процессора. Поэтому у внутренних прерываний отсутствуют машинные циклы типа «Прерывание», служащие для считывания номера прерывания n из контроллера.

Внутренние прерывания воспринимаются процессором безусловно, т.е. их обработку нельзя запретить. И они имеют, как правило, более высокий приоритет, чем внешние, т.е. при возникновении одновременно внутреннего и внешнего запросов будет обслужен внутренний. В остальном реакция процессора на внутренние прерывания аналогична таковой на внешние.

«Чисто программные прерывания» вызываются специальными машинными командами, помещаемыми в нужные места программы. Это коренным образом отличает их от аппаратных прерываний, поскольку делает их плановыми, наступающими в заранее намеченных точках программы. Они предусмотрены самим программистом для лучшей организации работы программы. Понятно также, что их нельзя запретить, как нельзя запретить выполнение ни одной команды в программе.

В случае чисто программных прерываний (как и внутренних исключений) тоже отсутствуют машинные циклы типа «Прерывание», предназначенные для считывания номера прерывания из контроллера, поскольку адрес перехода на ПП задается командой, а не периферийным устройством. Вместо циклов прерывания выполняются циклы выборки команды.

8.3. Прерывания микроконтроллеров AVR

Прерывания в микроконтроллерах. Как и в любых компьютерах прерывания в микроконтроллерах прекращают работу основной, текущей программы для того, чтобы процессорное ядро («процессор») могло выполнить более приоритетную программу, определяемую какими-либо важными внешними или внутренними событиями. Эту, приоритетную, программу мы назвали ранее *подпрограммой прерывания* (ПП) (т.к. она может запускаться многократно при повторных возникновениях прерывающих событий). Она является *прерывающей* по отношению к основной, фоновой, *прерываемой программой*.

Для уведомления процессора об упомянутого рода событиях, требующих его немедленной реакции, служат сигналы запроса прерывания. Если события внешние, запросы подаются на внешние ножки МК. Если внутренние, вырабатываются по внутренним цепям МК периферийными устройствами (но не изнутри процессора). Каждый запрос поступает на схему (модуль, «контроллер») прерываний МК (см. рис.4.1), с помощью которой процессор идентифицирует прерывающее устройство и определяет адрес перехода на подпрограмму, затребованную этим запросом. Далее он переходит по этому адресу и исполняет подпрограмму. Если выработано одновременно несколько запросов, схема прерываний выбирает более приоритетный из них, который и обслуживается процессором. Остальные запросы дожидаются своей очереди.

После завершения прерывающей подпрограммы процессор возвращается к прерванной программе, выполнявшейся до появления запроса, в точку, где

она была прервана. Возврат из ПП происходит не самопроизвольно, а инициируется специальной командой, которую надо поставить в конце ПП. Для МК AVR – это команда `reti` (Return from Interruption).

Виды прерываний в микроконтроллерах. В МК реализуются только аппаратные прерывания – от периферийных устройств и из внешней среды, а программные прерывания – изнутри процессора и чисто программные отсутствуют. Поэтому, согласно общей терминологии, все прерывания в МК внешние (для процессора). Но для микроконтроллеров они рассматриваются не по отношению к процессору, а по отношению ко всему МК и делятся соответственно на *внешние* (EXINT) и *внутренние* (ININT). Внешние срабатывают при поступлении определённого напряжения на некоторые выходы микросхемы (для AVR ATmega16 – на ножки INT0, INT1, INT2; первая ножка совпадает с ножкой порта PD2, вторая – с PD3, третья – с PB2). Внутренние возбуждаются периферийными устройствами, такими, как таймер-счётчик при переполнении, сторожевой таймер при срабатывании, последовательный приемо-передатчик USART по завершении приема/передачи, аналоговый компаратор в случае превышения первого входного уровня напряжения над вторым, АЦП по завершении преобразования, прочей периферией.

Источники прерываний в AVR. Источники прерываний – это те объекты, которые вырабатывают запросы прерываний. Источниками внешних прерываний (поступающих на ножки INT0, INT1, INT2 для ATmega16), могут быть кнопки, переключатели, датчики возникновения аварийных и нештатных ситуаций на технологическом оборудовании и др. Внутренние прерывания вырабатываются периферийными устройствами, при этом у каждого ПУ, что входит в состав AVR микроконтроллеров, есть как минимум один источник прерывания (Interruption Source). К прерываниям следует причислить и прерывание сброса (Reset Interruption), которое описано в п.4.3.

Большим плюсом микроконтроллеров AVR является то, что они имеют сравнительно большое количество прерываний. Оно составляет (не считая события сброса) от 3 до 35 в зависимости от типа МК. Например, ATmega16 имеет 20 источников: 3 внешних и 17 от внутренней периферии. Все они представлены в табл.8.1. Используя прерывания, можно мониторить множество параллельных процессов и достигать своего рода многозадачности, когда параллельно выполняется несколько процессов.

Векторы прерываний. Подпрограммы прерываний МК AVR размещаются в памяти программ. Возникает вопрос: как процессор узнаёт, куда ему перепрыгивать, чтобы, когда прерывание наступило, попасть на нужную подпрограмму? Он узнаёт это по *вектору прерывания*. Вектор – это адрес перехода. У каждого прерывания есть свой вектор (ВП). Он хранится в определенном месте (ячейке) памяти программ. Эти ячейки строго закреплены за каждым источником прерывания. Например, за внешним прерыванием INT0 закреплена ячейка \$0002, и только она (см. табл.8.1). Когда прерывание наступает, схема прерываний распознаёт его и сообщает процессору, в какой ячейке хранится вектор это-

го прерывания. Процессор считывает из указанной ячейки вектор и переходит по нему на подпрограмму.

Таблица 8.1 – Источники и векторы прерываний МК ATmega16

Но- мер ВП	Адрес ВП	ВП	Источник	Содержание прерывания
1	2	3	4	5
0	\$0000	rjmp k0	RESET	Сброс
1	\$0002	rjmp k1	INT0	Внешнее прерывание 0
2	\$0004	rjmp k2	INT1	Внешнее прерывание 1
3	\$0006	rjmp k3	TIMER2 COMP	Совпадение таймера/счетчика T2
4	\$0008	...	TIMER2 OVF	Переполнение таймера/счетчика T2
5	\$000A		TIMER1 CAPT	Захват таймера/счетчика T1
6	\$000C		TIMER1 COMPA	Совпадение «А» таймера/счетчика T1
7	\$000E		TIMER1 COMPB	Совпадение «В» таймера/счетчика T1
8	\$0010		TIMER1 OVF	Переполнение таймера/счетчика T1
9	\$0012		TIMER0 OVF	Переполнение таймера/счетчика T0
10	\$0014		SPI, STC	Передача по SPI завершена
11	\$0016		USART,RXC	USART, прием завершен
12	\$0018		USART,UDRE	Регистр данных USART пуст
13	\$001A		USART,TXC	USART, передача завершена
14	\$001C		ADC	Преобразование АЦП завершено
15	\$001E		EE_RDY	EEPROM, готово
16	\$0020		ANA_COMP	Аналоговый компаратор
17	\$0022		TWI	Прерывание от модуля TWI
18	\$0024		INT2	Внешнее прерывание 2
19	\$0026		TIMER0 COMP	Совпадение таймера/счетчика T0
20	\$0028	rjmp k20	SPM_RDY	Готовность SPM
ВП – вектор прерывания				

В случае МК AVR «вектор прерывания» – адрес перехода на ПП хранится в памяти не самостоятельно, а в составе команды перехода `rjmp`, которая, собственно, и выполняет функцию *вектора прерывания AVR*. Ее общий формат на языке ассемблера такой: `rjmp k`, где `rjmp` – название команды, а `k` – адрес перехода на подпрограмму прерывания (на языке ассемблера `k` это мнемонадрес, имя подпрограммы). На машинном языке команда `rjmp k` имеет код `Sxxxh` и длину 16 бит, где `xxx` – относительный адрес перехода. Она занимает одну ячейку программной памяти.

Например, команда `rjmp k2` в табл.8.1 – это команда перехода на подпрограмму обслуживания внешнего прерывания INT1 по адресу `k2`, или, по нашей терминологии, это вектор прерывания для INT1. Он хранится в ячейке \$0004.

Имена подпрограмм `k0`, `k1`, ..., `k20`, указанные в табл.8.1, условны. В действительности программист волен давать подпрограммам какие угодно имена на ассемблере.

Наряду с командой относительного перехода `rjmp` (16-битной) в качестве векторов прерываний может применяться и команда абсолютного перехода

`jmp` (32-битная, с кодом 940С xxxxh), занимающая две ячейки программной памяти.

Итак, *вектором прерывания* МК AVR служит команда `rjmp` относительный_адрес или `jmp` абсолютный_адрес. В первом случае он занимает в памяти 1 слово (ячейку), во втором – 2 слова (ячейки). Каждому прерыванию отводится своя, строго фиксированная ячейка, содержащая его вектор (ВП). Адрес этой ячейки называется *адресом вектора прерывания* (АВП).

В моделях AVR с объемом Flash ≤ 8 Кбайт для векторов отводится по 1 слову памяти программ, как раз для инструкций `rjmp`.

Таблица векторов прерываний. Все векторы прерываний размещаются в самом начале памяти программ в виде одномерного непрерывного массива, который называется *таблицей векторов прерываний* (Interruption Vector Table). Таблица векторов МК ATmega16 представлена в табл.8.1 столбцом 3 и содержит векторы – команды `rjmp kn`, $n = 1, 2, \dots, 20$. Каждый вектор размещен в клетке таблицы размером в две (16-битных) ячейки памяти. Таблица начинается с адреса \$0002 и занимает парные ячейки \$0002, \$0004, ..., \$0028. Адрес каждого последующего вектора больше предыдущего на 2. Адреса \$0002, \$0004 и \$0024 отведены для внешних входов запросов на прерывания. В таблицу прерываний включен, кроме того, вектор прерывания от сброса («вектор сброса») – команда `rjmp k0`, располагающаяся по адресу \$0000. Прерывание Reset, в отличие от всех остальных, нельзя запретить. Такие прерывания еще называют немаскируемыми (non-maskable interrupts).

Приоритет прерывания зависит от его расположения в таблице: чем меньше номер прерывания n (и адрес вектора), тем выше приоритет.

МК ATmega16 допускает обработку 20 прерываний, соответствующих 20-ти источникам и пронумерованных числами 1, 2, 3, ..., 20 (см. табл.8.1).

Таблица векторов прерываний, кроме вектора Reset, может быть перемещена в начало Boot раздела Flash памяти, путем установки бит IVSEL в регистре GICR. Вектор сброса также может быть перемещен в начало Boot раздела Flash памяти, путем программирования Fuse (фьюз) бита – BOOTrST. Fuse бит определяет адрес, с которого будет начато исполнение программы после сброса: если бит установлен, то начало программы будет не с адреса 0000h (как обычно), а с адреса области загрузчика (Boot Loader) (см. [7, 10, 13, 24]).

8.4. Обработка прерываний микроконтроллерами AVR

Порядок обработки прерываний. Внешний объект или периферийное устройство, преследующее цель немедленного переключения процессора на выполнение необходимых внеочередных действий, не предусмотренных текущей программой, вырабатывает сигнал запроса прерывания для процессора. Сигналы от внешних объектов, как уже сказано, заводятся на ножки INT0, INT1 и INT2 микроконтроллера, а от периферийных устройств идут внутри МК. Эти сигналы могут появиться в произвольные моменты времени.

Если прерывания от данного устройства не заблокированы пользователем (местно, в регистрах устройства), то сигнал запроса попадает на схему (модуль)

прерываний МК. Та идентифицирует устройство – источник запроса и пропускает запрос на процессор.

Получив запрос, процессор заканчивает выполнение текущей инструкции и при условии, что прерывания разрешены (флажок I в статусном регистре SREG установлен в 1), прерывает выполнение текущей программы и совершает далее автоматически следующие действия:

- а) получает от схемы информацию о местоположении ячейки, в которой хранится вектор прерывания поступившего запроса – команда `rjmp k`. Адрес этой ячейки связан с номером прерывания n из таблицы векторов соотношением $ABП=2 \times n$;
- б) включает в стек содержимое регистра PC (указывающее на команду, расположенную вслед за текущей), запрещает глобальные прерывания (сбрасывает флажок I в ноль: $I=0$) и
- в) считывает вектор прерывания – команду `rjmp k` из заданной ячейки (с адресом АВП).

После считывания процессор декодирует вектор – команду `rjmp k` и исполняет ее: переходит по указываемому ею адресу k – *адресу перехода* на начало подпрограммы прерывания. Далее следует программная стадия прерывания – выполнение подпрограммы.

В конце ПП ставится команда `reti`, возобновляющая выполнение прерванной программы, начиная с той команды, перед которой прерывание произошло. Это происходит вследствие того, что `reti` извлекает из стека ранее включенное туда содержимое PC (*адрес возврата* из ПП) и заносит его обратно в PC. Также команда `reti` устанавливает бит I в регистре статуса SREG, разрешая прерывания.

Подпрограмма прерывания может установить бит I в единицу: $I=1$ командой `sei`, разрешив вложенные прерывания.

Запрещение и разрешение прерываний в AVR. Пользователю системы всегда предоставляется возможность отключить (блокировать, маскировать) отдельные источники прерываний (локальный запрет). Для этого в схемах предусматриваются специальные программируемые регистры маски, где каждому источнику прерываний отводится один бит. При локальном запрете запрос прерывания не будет пропущен на схему прерываний. Для глобального запрета прерываний, служит упомянутый бит I регистра состояния SREG, в который надо занести 0 с помощью команды `cli`. Это запрещает все прерывания. Для глобального разрешения (демаскирования) прерываний надо занести в I единицу 1 командой `sei`. Если прерывания разрешены, то при появлении любого запроса бит I в регистре статуса SREG автоматически сбрасывается и все дальнейшие прерывания запрещаются. Программа пользователя может вновь установить этот бит, разрешив вложенные прерывания. Инструкция возвращения из подпрограммы обслуживания прерывания `reti` также устанавливает бит I в регистре статуса SREG, разрешая прерывания.

Только при разблокировании указанного двухступенчатого запрета процессор воспримет запрос от ПУ.

Прерывания от нескольких устройств. При большом количестве источников прерываний порядок их обслуживания определяется приоритетом источника. Приоритет устанавливается в зависимости от важности решаемой этим устройством задачи. Полагается, что при одновременном поступлении нескольких запросов будет обслужен наиболее приоритетный из них. Самым простым является фиксированное распределение приоритетов. При этом каждому из источников запросов присваивается постоянный приоритет, соответствующий его порядковому номеру n в таблице прерываний. Приоритет уменьшается с возрастанием номеров векторов. И, наоборот, чем меньше номер вектора в таблице, тем выше приоритет прерывания. То есть, самый высокий приоритет имеет прерывание сброса (Reset Interrupt), которое располагается первым в таблице и соответственно в памяти программ. Внешнее прерывание INT0, идущее следом за прерыванием Reset в таблице векторов прерываний, имеет приоритет меньше, чем у Reset, но выше, чем у всех остальных прерываний и т.д.

А что будет с остальными, отклоненными запросами, которые поступили одновременно с обслуженным приоритетным? Они потеряются? Нет, не потеряются – у них взведется флаг, и как только завершится первый обработчик, автоматом произойдет переход к отложенному прерыванию. Единственное, что мы можем потерять количество одинаковых прерываний. Т.е. если обрабатывается, например, INT0 и прерывания запрещены, а в это время придет три раза INT1, то на выходе INT1 обработается только один раз.

Структура переходов по программной памяти при прерываниях показана на рис.8.6 на примере внешнего прерывания INT1.

Программа для подготовки таблицы внешних прерываний AVR. Целиком заполненная векторами таблица, подобно табл.8.1, никогда не применяется. На практике обычно используется только 1-4 прерывания, в этом случае не используемые адреса векторов остаются обычно не инициализированными. Здесь мы сформируем таблицу для внешних прерываний. Внешние запросы на прерывания INT0, INT1 и INT2 имеют векторы, расположенные по адресам 0x2, 0x4 и 0x24 (табл.8.2). Типовая программа подготовки этих векторов выглядит следующим образом:

```
.cseg
.org 0x0
; в качестве векторов прерываний
; используем команду абсолютно перехода jmp :
jmp main ; переход к программе пользователя
jmp ext_int0 ; переход к подпрограмме обработки INT0, n=1
jmp ext_int1 ; переход к подпрограмме обработки INT1, n=2
.org 0x24
jmp ext_int2 ; переход к подпрограмме обработки INT2, n=18
...
main: ; Начало программы пользователя
...
ext_int0: ; Начало подпрограммы обработки INT0
```

...
 ext_int1: ; Начало подпрограммы обработки INT1
 ...
 ext_int2: ; Начало подпрограммы обработки INT2
 ...

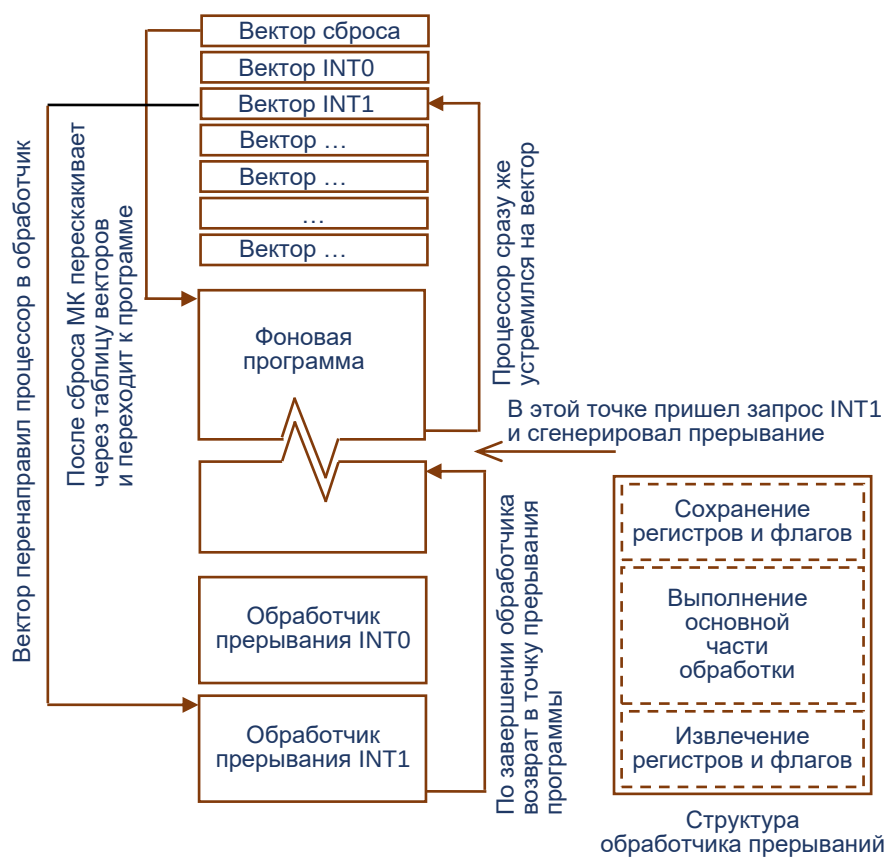


Рисунок 8.6 – Обработка внешнего запроса прерывания INT1 микроконтроллером AVR

Таблица 8.2 – Векторы внешних прерываний контроллера ATmega16

№ прерыв.	Адрес ВП	ВП	Источник	Содержание прерывания
1	\$0002	jmp ext_int0	INT0	Внешнее прерывание 0 (External Interrupt Request 0)
2	\$0004	jmp ext_int1	INT1	Внешнее прерывание 1 (External Interrupt Request 1)
18	\$0024	jmp ext_int2	INT2	Внешнее прерывание 2 (External Interrupt Request 2)

Теперь, если во время выполнения основной программы main поступит запрос внешнего прерывания INT0, INT1 или INT2, процессор прервет main и прочитает из памяти программ команду `jmp ext_intm`. Здесь m – номер сигнала INT_m , $m = 0, 1, 2$. Расшифровав команду `jmp ext_intm`, процессор

выполнит ее, т.е. перейдет на подпрограмму обслуживания поступившего запроса.

Для выхода из подпрограммы в системе команд микроконтроллера ATmega16 предусмотрена, как указывалось, специальная команда `reti` (Return from Interruption). Для правильного функционирования системы прерывания в начале `main` обязательно должен быть загружен указатель стека.

8.5. Регистры системы внешних прерываний

Список регистров. Внешние прерывания вызываются с контактов INT0, INT1 и INT2. Для их обработки и программирования в контроллере задействованы отдельные биты четырех регистров:

- регистра управления прерываниями GICR (General Interrupt Control Register);
- регистра управления процессорным ядром MCUCR (Microcontroller Unit (MCU) Control Register);
- регистра управления и состояния процессорного ядра MCUCSR (MCU Control and Status Register);
- регистра флагов прерываний GIFR (General Interrupt Flags Register).

Для **разрешения или запрещения внешних прерываний** предназначен управляющий регистр GICR.

GICR – \$3B (\$5B)

– General Interrupt Control Register

Bit	7	6	5	4	3	2	1	0	
	INT1	INT0	INT2	-	-	-	IVSEL	IVCE	GICR
Read/Write	R/W	R/W	R/W	R	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 7:6:5 – INT1:0:2: External Interrupt Request 1:0:2 Enable** – Маска внешнего прерывания INT1:0:2
- **Bit 1 – IVSEL: Interrupt Vector Select** – Выбор местоположения векторов прерываний
- **Bit 0 – IVCE: Interrupt Vector Change Enable** – Изменение местоположения векторов прерываний разрешено

Установка битов INT1, INT0 или INT2 разрешает прерывания при возникновении события на соответствующей ножке микроконтроллера AVR, а сброс – запрещает, маскирует.

Естественно, нужно установить еще и флаг глобального разрешения прерываний – I, который расположен в регистре SREG. Без него вообще ни одно прерывание вызываться не будет.

Биты IVSEL и IVCE определяют местоположение таблицы векторов прерываний. Если IVSEL сброшен, вектора прерываний располагаются в начале Flash-памяти, если установлен – в начале загрузочной секции Boot Loader.

Адрес начала загрузочной секции определяется перемычками BOOTSZ.

Чтобы изменить расположение векторов прерываний, надо установить бит IVCE, а затем в течение четырех циклов — бит IVSEL. IVCE сбрасывается

аппаратно по прошествии четырех циклов после своей установки или при установке IVSEL. При выполнении этой последовательности действий прерывания запрещаются автоматически, но на бит I регистра SREG это не влияет.

Условия генерации прерываний.

Внешнее прерывание может происходить по одному из следующих событий:

- по низкому уровню на выводах INT0, INT1;
- по любому изменению логического уровня на выводах INT0, INT1;
- по спадающему фронту сигнала на выводах INT0, INT1, INT2;
- по нарастающему фронту на выводах INT0, INT1, INT2.

Эти условия устанавливаются с помощью конфигурационных регистров. Для INT0, INT1 – это регистр MCUCR. Для INT2 – MCUCSR.

MCUCR – \$35 (\$55)

– Microcontroller Unit (MCU) Control Register

Bit	7	6	5	4	3	2	1	0	
	SM2	SE	SM1	SM0	ISC11	ISC10	ISC01	ISC00	MCUCR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 3:0 – ISC11:10:01:00: The Interrupt Sense Control** – Управление восприятием прерываний INT1:0

MCUCSR – \$34 (\$54)

– MCU Control and Status Register

Bit	7	6	5	4	3	2	1	0	
	JTD	ISC2	-	JTRF	WDRF	BORF	EXTRF	PORF	MCUCSR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0						

- **Bit 6 – ISC2: Interrupt Sense Control 2** – Управление восприятием прерываний INT2

В таблице ниже приведены возможные значения разрядов ISC11, ISC10 (ISC01, ISC00) и соответствующие им условия генерации внешнего прерывания INT1 (INT0).

Таблица 8.3 – Активизация параметров входа INT1 (INT0)

ISC11 (ISC01)	ISC10 (ISC00)	Условие генерации внешнего прерывания INT1 (INT0)
0	0	По низкому уровню на ножке INT1 (INT0)
0	1	По любому изменению логического уровня на INT1 (INT0)
1	0	По спадающему фронту импульса на INT1 (INT0)
1	1	По нарастающему фронту на INT1 (INT0)

Прерывание INT2 может происходить только по фронтам сигнала, поэтому для установки условий используется всего один бит – это бит ISC2 регистра MCUCSR.

Таблица 8.4 – Активизация параметров входа INT2

ISC2	Условие генерации внешнего прерывания INT2
0	По спадающему фронту на выводе INT2
1	По нарастающему фронту на INT2

Заметим, что при смене значения бита ISC2 может быть сгенерировано прерывание INT2. Чтобы этого не происходило, нужно производить модификацию бита ISC2 так: запретить внешнее прерывание, поменять бит ISC2, сбросить флаг прерывания – INTF2 (смотри выше) и опять разрешить прерывание INT2.

Обнаружение фронтов сигналов на выводах INT0/INT1 осуществляется синхронно, то есть по сигналу тактового генератора. Минимальная длительность входного импульса, гарантирующая генерацию прерывания, составляет один период тактового сигнала микроконтроллера AVR.

Внешние прерывания INT0/INT1 сконфигурированные на срабатывание по низкому уровню обрабатываются асинхронно. Для генерации прерывания, уровень должен удерживаться до окончания выполнения текущей команды. Если после обработки прерывания уровень еще удерживается, прерывание будет вызвано снова.

Обнаружение перепадов сигнала на выводе INT2 тоже осуществляется асинхронно. Минимальная длительность импульса, гарантирующая генерацию прерывания, составляет 50 нс.

Внешние прерывания, обнаруживаемые асинхронно, могут быть использованы для пробуждения микроконтроллера, находящегося в любом из шести режимов пониженного энергопотребления.

Флаги внешних прерываний.

Последний регистр, имеющий отношение к внешним прерываниям, – это статусный регистр GIFR (General Interrupt Flag Register). В нем содержатся *флаги прерываний*.

GIFR – \$3A (\$5A)

– General Interrupt Flag Register

Bit	7	6	5	4	3	2	1	0	
	INTF1	INTF0	INTF2	-	-	-	-	-	GIFR
Read/Write	R/W	R/W	R/W	R	R	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 7:6:5 – INTF1:0:2: External Interrupt Flag 1:0:2** – флаг внешнего прерывания INT1:0:2

Событие, вызывающее прерывание может быть кратковременным (например – фронт логического сигнала). Для того, чтобы не потерять информацию о таком событии, если оно произошло в то время, когда прерывание было запрещено, существуют *флаги прерываний*. Каждому источнику прерывания соответствует свой флаг, который устанавливается при возникновении события. Сброс флага прерывания происходит при вызове подпрограммы его обслуживания. Как правило, флаги прерываний могут устанавливаться и сбрасываться программно, что позволяет имитировать те или иные прерывания, например, для целей отладки, либо отменять обработку каких-либо событий. В микроконтроллерах AVR возможно только сбрасывать флаги прерываний, причем, для сброса необходимо записать в соответствующий бит единицу. Аналогично флагам маскирования, флаги прерывания могут

группироваться в специальные регистры, либо (в том числе – в AVR) могут быть разбросаны по различным регистрам.

В статусном регистре GIFR содержатся флаги внешних прерываний. При поступлении запросов на входы соответствующие биты этого регистра устанавливаются, а когда вызываются обработчики прерываний (когда счетчик команд микроконтроллера устанавливается на конкретный вектор прерывания), соответствующий бит в регистре GIFR аппаратно сбрасывается. Также флаги можно сбросить программно, записав в соответствующий бит регистра логическую единицу. Причем сброс нужно производить перезаписью регистра GIFR, но не операцией побитового ИЛИ.

Остальные биты зарезервированы.

Глава 9. Периферийные компоненты и пространство ввода-вывода

9.1. Состав периферийных компонентов AVR

МК AVR семейства Mega содержат богатый набор периферийных компонентов, которые подразделяются на порты ввода-вывода (ПВВ) и периферийные устройства (ПУ). Порты предназначены для обмена цифровыми данными МК с внешней средой, ПУ – для выполнения дополнительных, специальных операций по вводу-выводу и обработке данных, а также обслуживанию МК. Порты в/в зачастую причисляют к ПУ (п.3.1). В свою очередь, ПУ (включая ПВВ) называют еще устройствами ввода-вывода, поскольку процессорное ядро вводит из них и выводит в них данные (п.3.1).

Порты ввода-вывода:

- 8-разрядные цифровые порты ввода/вывода. МК AVR Mega имеют от 23 до 86 линий ввода/вывода, которые объединяются в 8-разрядные порты ввода/вывода, причем отдельные линии могут быть запрограммированы как входные или как выходные независимо друг от друга. (п.4.2). Все линии в/в имеют, кроме того, дополнительные, программно устанавливаемые функции, относящиеся в том числе к последовательным портам в/в (интерфейсам), перечисляемым ниже. Каждая линия имеет входной буфер с триггером Шмита и индивидуально отключаемый внутренний подтягивающий резистор сопротивлением 20...50 кОм;
- последовательный синхронный интерфейс SPI;
- последовательный двухпроводный интерфейс TWI (полный аналог интерфейса I²C);
- от одного до четырех полнодуплексных универсальных синхронных / асинхронных приемо-передатчиков (USART), которые в ряде моделей могут использоваться в качестве ведущего устройства шины SPI;
- универсальный последовательный интерфейс USI, который может использоваться в качестве интерфейса SPI или I²C, а также полудуплексного UART или 4/12-битного счетчика.

Периферийные устройства:

- один или два 8-битных таймера/счетчика, во всех моделях с двумя 8-битными таймерами/счетчиками один из них может работать в качестве часов реального времени (в асинхронном режиме);
- от одного до четырех 16-битных таймеров/счетчиков;
- одно- и двухканальные генераторы 8-битного ШИМ-сигнала (один из режимов работы 8-битных таймеров/счетчиков);
- двух- и трехканальные генераторы ШИМ-сигнала регулируемой разрядности (один из режимов работы 16-битных таймеров/счетчиков), разрешение формируемого сигнала может составлять от 1 до 16 бит;

- аналоговый компаратор;
- многоканальный 10-битный АЦП последовательного приближения, имеющий как несимметричные, так и дифференциальные входы;

9.2. Ввод-вывод

Ввод и вывод. *Вводом* называется передача данных из периферийных устройств или внешней среды в процессорное ядро микроконтроллера («процессор»), а *выводом* – обратная передача – из процессора в периферийную часть и далее, возможно, во внешнюю среду (п.2.3).

Контроллеры ввода-вывода в МК. Мы упоминали (в п.2.3), что ввод-вывод данных осуществляется не напрямую между ПУ/внешней средой и процессором, а через *контроллеры ввода-вывода* (КВВ). Конструктивно КВВ микроконтроллера – это буферное устройство, которое непосредственно подключено к внутренней системной шине ввода-вывода МК и разделяет процессор и периферийное устройство или процессор и внешнюю среду (см. рис.1.1, 2.11, 2.12). Оно осуществляет управление передачей вводимой и выводимой информации и ее временное хранение в процессе передачи. Те КВВ, которые сопрягают МК с внешней средой, мы называли выше, в п.9.1, *портами ввода-вывода* *. Те же КВВ, что сопрягают процессор и периферийные устройства, портами в/в не называют, а оставляют за ними общее наименование «контроллеров ввода-вывода». В микроконтроллерах они встроены в ПУ, считаются их частью, скрыты от пользователя и отдельно не обсуждаются. Описываются лишь их программно-доступные регистры – регистры ввода-вывода.

* Здесь термин «порт» понимается в широком смысле, как «вход» в компьютер, как контроллер, а не регистр ввода-вывода (см. п.2.3).

Таким образом, *порт ввода-вывода* – это *вакантный контроллер ввода-вывода*, контроллер, к которому снаружи компьютера/микроконтроллера изначально не подключено никакое внешнее устройство, а последнее можно подключать по мере необходимости. Порт в/в называется также *интерфейсным, соединительным портом* или просто «портом». В микроконтроллерах AVR Mega используются параллельные и последовательные порты (интерфейсы), перечисленные в п.8.1.

Регистры ввода вывода. Для временного хранения вводимой и выводимой информации каждый КВВ (в частности ПВВ) снабжен одним или несколькими регистрами, называемыми *регистрами ввода-вывода*. Местоположение на шине каждого регистра определяется его уникальным номером – *адресом* регистра. В МК AVR Mega для адресации регистров в/в используется 16-разрядный код, изменяющийся в пределах \$0020 ... \$00FF/01FF. Поэтому существует от 64 до 224/480 регистров в/в (глава 5.) Вся работа процессора с периферийными компонентами производится только через регистры в/в – адресуемые регистры КВВ.

Порты ввода-вывода МК соединяют микроконтроллер с внешними устройствами – переключателями, матричными или 8-сегментными индикаторами, другими дисплеями, светодиодами, электродвигателями, персональными компьютерами и т.д. В главе 10 рассматриваются параллельные порты ввода-вывода микроконтроллеров AVR ATmega16.

Периферийные устройства. Некоторые периферийные устройства микроконтроллера AVR ATmega16 обсуждаются в главах 10-13.

9.3. Пространство ввода-вывода AVR

Состав пространства ввода-вывода AVR. *Пространством ввода-вывода, адресным пространством ввода-вывода* называется множество адресуемых регистров ввода-вывода, принадлежащих периферийным компонентам компьютера или микроконтроллера – портам и периферийным устройствам. Через эти регистры процессор общается с внешней средой и ПУ. Пространство в/в МК AVR представлено в табл.9.1. [25]

Таблица 9.1 – Пространство ввода-вывода AVR ATmega16

Register Summary

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
\$3F (\$5F)	SREG	I	T	H	S	V	N	Z	C
\$3E (\$5E)	SPH	–	–	–	–	–	SP10	SP9	SP8
\$3D (\$5D)	SPL	SP7	SP6	SP5	SP4	SP3	SP2	SP1	SP0
\$3C (\$5C)	OCR0	Timer/Counter0 Output Compare Register							
\$3B (\$5B)	GICR	INT1	INT0	INT2	–	–	–	IVSEL	IVCE
\$3A (\$5A)	GIFR	INTF1	INTF0	INTF2	–	–	–	–	–
\$39 (\$59)	TIMSK	OCIE2	TOIE2	TICIE1	OCIE1A	OCIE1B	TOIE1	OCIE0	TOIE0
\$38 (\$58)	TIFR	OCF2	TOV2	ICF1	OCF1A	OCF1B	TOV1	OCF0	TOV0
\$37 (\$57)	SPMCR	SPMIE	RWWSB	–	RWWSRE	BLBSET	PGWRT	PGERS	SPMEN
\$36 (\$56)	TWCR	TWINT	TWEA	TWSTA	TWSTO	TWWC	TWEN	–	TWIE
\$35 (\$55)	MCUCR	SM2	SE	SM1	SM0	ISC11	ISC10	ISC01	ISC00
\$34 (\$54)	MCUCSR	JTD	ISC2	–	JTRF	WDRF	BORF	EXTRF	PORF
\$33 (\$53)	TCCR0	FOC0	WGM00	COM01	COM00	WGM01	CS02	CS01	CS00
\$32 (\$52)	TCNT0	Timer/Counter0 (8 Bits)							
\$31 ⁽¹⁾ (\$51) ⁽¹⁾	OSCCAL	Oscillator Calibration Register							
	OCDR	On-Chip Debug Register							
\$30 (\$50)	SFIOR	ADTS2	ADTS1	ADTS0	–	ACME	PUD	PSR2	PSR10
\$2F (\$4F)	TCCR1A	COM1A1	COM1A0	COM1B1	COM1B0	FOC1A	FOC1B	WGM11	WGM10
\$2E (\$4E)	TCCR1B	ICNC1	ICES1	–	WGM13	WGM12	CS12	CS11	CS10
\$2D (\$4D)	TCNT1H	Timer/Counter1 – Counter Register High Byte							
\$2C (\$4C)	TCNT1L	Timer/Counter1 – Counter Register Low Byte							
\$2B (\$4B)	OCR1AH	Timer/Counter1 – Output Compare Register A High Byte							
\$2A (\$4A)	OCR1AL	Timer/Counter1 – Output Compare Register A Low Byte							
\$29 (\$49)	OCR1BH	Timer/Counter1 – Output Compare Register B High Byte							
\$28 (\$48)	OCR1BL	Timer/Counter1 – Output Compare Register B Low Byte							
\$27 (\$47)	ICR1H	Timer/Counter1 – Input Capture Register High Byte							
\$26 (\$46)	ICR1L	Timer/Counter1 – Input Capture Register Low Byte							
\$25 (\$45)	TCCR2	FOC2	WGM20	COM21	COM20	WGM21	CS22	CS21	CS20
\$24 (\$44)	TCNT2	Timer/Counter2 (8 Bits)							
\$23 (\$43)	OCR2	Timer/Counter2 Output Compare Register							
\$22 (\$42)	ASSR	–	–	–	–	AS2	TCN2UB	OCR2UB	TCR2UB
\$21 (\$41)	WDTCR	–	–	–	WDTOE	WDE	WDP2	WDP1	WDP0
\$20 ⁽²⁾ (\$40) ⁽²⁾	UBRRH	URSEL	–	–	–	UBRR[11:8]			
	UCSRC	URSEL	UMSEL	UPM1	UPM0	USBS	UCSZ1	UCSZ0	UCPOL

\$1F (\$3F)	EEARH	–	–	–	–	–	–	–	EEAR8
\$1E (\$3E)	EEARL	EEPROM Address Register Low Byte							
\$1D (\$3D)	EEDR	EEPROM Data Register							
\$1C (\$3C)	EECR	–	–	–	–	EERIE	EEMWE	EERE	EERE
\$1B (\$3B)	PORTA	PORTA7	PORTA6	PORTA5	PORTA4	PORTA3	PORTA2	PORTA1	PORTA0
\$1A (\$3A)	DDRA	DDA7	DDA6	DDA5	DDA4	DDA3	DDA2	DDA1	DDA0
\$19 (\$39)	PINA	PINA7	PINA6	PINA5	PINA4	PINA3	PINA2	PINA1	PINA0
\$18 (\$38)	PORTB	PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0
\$17 (\$37)	DDRB	ddb7	ddb6	ddb5	ddb4	ddb3	ddb2	ddb1	ddb0
\$16 (\$36)	PINB	PINB7	PINB6	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0
\$15 (\$35)	PORTC	PORTC7	PORTC6	PORTC5	PORTC4	PORTC3	PORTC2	PORTC1	PORTC0
\$14 (\$34)	DDRC	ddc7	ddc6	ddc5	ddc4	ddc3	ddc2	ddc1	ddc0
\$13 (\$33)	PINC	PINC7	PINC6	PINC5	PINC4	PINC3	PINC2	PINC1	PINC0
\$12 (\$32)	PORTD	PORTD7	PORTD6	PORTD5	PORTD4	PORTD3	PORTD2	PORTD1	PORTD0
\$11 (\$31)	DDRD	ddd7	ddd6	ddd5	ddd4	ddd3	ddd2	ddd1	ddd0
\$10 (\$30)	PIND	PIND7	PIND6	PIND5	PIND4	PIND3	PIND2	PIND1	PIND0
\$0F (\$2F)	SPDR	SPI Data Register							
\$0E (\$2E)	SPSR	SPIF	WCOL	–	–	–	–	–	SPI2X
\$0D (\$2D)	SPCR	SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0
\$0C (\$2C)	UDR	USART I/O Data Register							
\$0B (\$2B)	UCSRA	RXC	TXC	UDRE	FE	DOR	PE	U2X	MPCM
\$0A (\$2A)	UCSRB	RXCIE	TXCIE	UDRIE	RXEN	TXEN	UCSZ2	RXB8	TXB8
\$09 (\$29)	UBRRL	USART Baud Rate Register Low Byte							
\$08 (\$28)	ACSR	ACD	ACBG	ACO	ACI	ACIE	ACIC	ACIS1	ACIS0
\$07 (\$27)	ADMUX	REFS1	REFS0	ADLAR	MUX4	MUX3	MUX2	MUX1	MUX0
\$06 (\$26)	ADCSRA	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0
\$05 (\$25)	ADCH	ADC Data Register High Byte							
\$04 (\$24)	ADCL	ADC Data Register Low Byte							
\$03 (\$23)	TWDR	Two-wire Serial Interface Data Register							
\$02 (\$22)	TWAR	TWA6	TWA5	TWA4	TWA3	TWA2	TWA1	TWA0	TWGCe
Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
\$01 (\$21)	TWSR	TWS7	TWS6	TWS5	TWS4	TWS3	–	TWPS1	TWPS0
\$00 (\$20)	TWBR	Two-wire Serial Interface Bit Rate Register							

Расположение регистров ввода-вывода в общем пространстве памяти данных. Все периферийные регистры (регистры ввода-вывода) находятся, как следует из табл.9.1, по определенным адресам, т.е. к ним можно обращаться не только по их именам, но и по этим адресам. Они образуют пространство ввода-вывода (ПВВ) и одновременно, в программной модели, находятся в общем пространстве памяти данных, т.е. разделяют общее адресное пространство памяти данных (ОЗУ) с рабочими регистрами процессора и собственно ячейками памяти (см. п.5.2). В частности, для Atmega16 это пространство имеет размер \$0000-\$045F (460h байт), и в этом пространстве рабочие регистры занимают адреса \$0000-\$001F (0-31), а ПВВ – адреса \$0020-\$005F (см. рис.5.5).

Первый адрес в табл.9.1 – это адрес ПВВ в ПВВ, второй (в скобках) – адрес ПВВ в ОЗУ. Для обращения к ПВВ в ПВВ служат команды ввода IN и вывода OUT и команды битовых операций CBI – сброс бита регистра в/в, SBI – установка бита регистра в/в. Для обращения к ПВВ как к ячейкам памяти ОЗУ можно использовать команды работы с памятью (ОЗУ).

Глава 10. Порты ввода-вывода

10.1. Характеристика портов ввода-вывода

Функции портов. С внешним миром микроконтроллер общается через порты ввода-вывода (I/O порты – Input-Output Ports). Здесь мы рассмотрим параллельные порты. Они пересылают за раз одно слово, состоящее из нескольких бит: каждый бит – по своей сигнальной линии. Для этого порт снабжен определенным числом выводов (ножек, контактов, «пинов» *) – металлических штырьков, торчащих из корпуса микрочипа контроллера и предназначенных для подсоединения к микрочипу внешних устройств. По этим выводам микроконтроллер и осуществляет прием (ввод) и выдачу (вывод) на внешние устройства цифровых сигналов. Направление передачи сигналов по любому выводу может быть задано программно в любой момент времени.

* От англ. pin – иголка, булавка или вообще любой продолговатый предмет для соединения чего-л., например, вывод на микрочипе.

От превышенных и отрицательных значений напряжения порт защищён внутренними диодами. Эти диоды слабы и защищают порт от перепадов всего в 1-2 В. Большие перепады способны привести к выгоранию порта.

Параллельные порты (далее просто «порты») микроконтроллера AVR обычно имеют имена A, B и иногда C или даже D и имеют по 8 (некоторые по 3) разрядов, привязанных по отдельности к определенным ножкам корпуса.

На рис.4.2 была показана **цоколёвка** («распиновка», разводка выводов) **микроконтроллера ATmega16**. Порты ввода-вывода обозначены символом P_x, где x – имя порта, а их выводы (ножки, контакты на микросхеме) – P_{xn}, где n – номер вывода (бита) в этом порту. Например, вывод PD0 является нулевым битом порта D.

Отличительной особенностью портов МК AVR является возможность выполнять операции над любым отдельно взятым выводом (ножкой) МК (с помощью команд SBI – установка бита регистра в/в и CBI – сброс бита регистра в/в), не влияя на другие выводы порта.

МК AVR различных моделей семейства Mega имеют различное количество портов и соответственно ножек ввода/вывода. Например:

- ATmega8x имеют три порта ввода/вывода: порт B (8-разрядный), порт C (7-разрядный) и порт D (8-разрядный). Всего контактов ввода/вывода 23;
- ATmega16x имеют четыре 8-разрядных порта ввода/вывода (порты A, B, C, D). Всего контактов ввода/вывода 32;
- ATmega161x и ATmega162x имеют четыре 8-разрядных порта ввода/вывода (порты A, B, C, D) и один 3-разрядный порт ввода/вывода E. Всего контактов ввода/вывода 35;

- ATmega163x, ATmega32x и ATmega323x имеют четыре 8-разрядных порта ввода/вывода (порты A, B, C, D). Всего контактов ввода/вывода 32;
- ATmega64x и ATmega128x имеют шесть 8-разрядных порта ввода/вывода (порты A, B, C, D, E, F) и один 5-разрядный порт ввода/вывода G. Всего контактов ввода/вывода 53.

Регистры портов в/в. Все порты в/в (любого компьютера, в частности микроконтроллера) снабжены программно доступными регистрами, которые непосредственно отвечают за их работу. Эти регистры называются *регистрами конфигурации порта* или (как и все прочие программные регистры периферийных устройств) *регистрами ввода вывода*. Одни из регистров конфигурации служат для настройки и слежения за портами (*регистры управления/состояния*), другие – собственно для передачи данных (*регистры данных*), включая их буферирование (временное хранение) при вводе и выводе. Каждый параллельный I/O порт микроконтроллера AVR снабжен 3-я такими регистрами. Это: PORTx – регистр данных, DDRx – регистр направления и PINx – регистр состояния ножек, где x – имя порта. Например, для микроконтроллера ATmega16 имя x = A, B, D или C, поскольку у него имеется четыре порта с именами A, B, D и C, и все эти порты, как и обслуживающие их регистры, 8-разрядные. Скажем, за порт «B» отвечают три восьмиразрядных регистра PORTB, PINB, DDRB, каждый разряд в которых отвечает за соответствующую ножку PBn.

Ножки (выводы) портов. Каждая ножка порта Pxn может быть как портом ввода, так и портом вывода. То есть, мы можем либо управлять чем-либо (например, зажигать светодиод), либо принимать данные с чего-либо (например, проверять нажатие кнопки). Режим работы ножки и значение передаваемых по ней однобитных данных задаются при помощи соответственных битов (разрядов) конфигурационных регистров PORTx, DDRx, PINx.

Также почти каждая ножка имеет **дополнительные функции**. На распиновке они подписаны в скобках (см. рис.10.1). Это могут быть ножки приемо-передатчиков, разные последовательные интерфейсы, аналоговые входы, выходы ШИМ генераторов и т.п. По умолчанию все эти функции отключены, а ножка управляется исключительно парой DDR и PORT, но если включить какую-либо дополнительную функцию, то тут уже управление может полностью или частично перейти под контроль периферийного устройства и тогда даже при записи в DDR/PORT ничего не изменится. До тех пор пока не выключить периферию, занимающую эти ножки. Например, приемник USART. Стоит только выставить бит разрешения приема RXEN по ножке RXD, то, как бы она ни была настроена до этого, эта ножка переходит в режим приема (входа) в USART.

10.2. Регистры портов

PINx.

PINx (от англ. Pin – штырь, вывод, пин, нога, ножка в компьют. технике и Input – *компьют. ввод информации*; информация на входе) – это регистр чте-

ния. Из него можно только читать. В регистре PINx содержится информация о реальном текущем логическом уровне на ножках порта вне зависимости от настроек порта. Так что если хотим узнать что у нас на ножке Pxn – читаем соответствующий бит регистра PINx.

Заметим, что в действительности PINx регистром не является, так как не запоминает значение сигналов на ножках, а лишь передает их текущие мгновенные значения по запросам программы. PINx – это прямой канал ввода информации с ножек.

DDRx.

DDRx – это регистр направления порта (Data Direction Register). Порт в конкретный момент времени может быть либо входом, либо выходом (но для состояния битов PINx это значения не имеет – читать из PINx реальное значение можно всегда):

- DDRxn=0 – ножка Pxn работает как ВХОД (в регистр PINx).
- DDRxn=1 – ножка Pxn работает на ВЫХОД (из регистра PORTx).

При DDRxn=1 мы все равно можем ввести из регистра PINx состояние ножек, которое в этом случае совпадает с состоянием регистра PORTx (если, конечно, на ножках нет сигналов от внешних устройств, накладываемых на сигналы от PORTx и спутывающих их).

PORTx.

PORTx – это регистр данных. Управляет состоянием ножек порта x (если соответствующий разряд настроен как выход), или подключением внутреннего Pull-Up резистора (если соответствующий разряд настроен как вход).

А именно, когда ножка настроена на выход (DDRxn=1), значение соответствующего бита (бита n) в регистре PORTx задает состояние ножки: если PORTxn=1, то на ножке логическая 1, если PORTxn=0, то на ножке логический 0.

Когда мы настраиваем ножку на вход (DDRxn=0), то от PORTxn зависит тип входа – Hi-Z или Pull Up. Если PORTxn=0, то ножка в режиме Hi-Z. Если PORTxn=1, то ножка в режиме Pull Up с подтяжкой резистором в 100к до питания*.

* Pull Up – от англ. pull – тяга, дерганье; натяжение; тянуть, тащить и Up – вверх; pull up – вытягивать, выдергивать вверх.

Подтягивающий резистор, Pull резистор – это резистор, включённый между проводником, по которому распространяется электрический сигнал, и питанием (*pull-up resistor* – подтягивающий вверх резистор), либо между проводником и землёй (*pull-down resistor* – подтягивающий вниз резистор).

Тип входа – Hi-Z или Pull Up.

Когда ножка настроена на вход (DDRxn=0), то в зависимости от значения PORTxn возможны 2 варианта состояния:

1) Вход Pull Up — вход с подтяжкой (PORTxn=1).

При такой конфигурации замыкается ключ подтяжки и к линии подключается резистор 100 кОм, что приводит к возникновению на ножке логической 1. Если эту ножку замкнуть на землю (например, кнопкой), то резистор не бу-

дет уже удерживать напряжение на линии на уровне логической единицы и на ножке установится ноль.

Для неиспользуемых выводов рекомендуется устанавливать этот режим (это приведёт к снижению энергопотребления и повышению надёжности).

P.S. Есть еще бит **PUD** (Pull Up Disable *) в регистре **SFIOR**. Он запрещает включение подтяжки сразу для всех портов. По умолчанию он равен 0, то есть подтяжка разрешена

* disable – делать неспособным, непригодным.

2) Вход Hi-Z – режим высокоимпендансного входа – High Z (PORTxn=0).

Здесь Z – это общепринятое обозначение импеданса – комплексного сопротивления цепи.

При такой конфигурации ножка вообще никуда не подключена и ни на что не влияет. Но при этом она постоянно отправляет свое состояние в регистр **PIN** и всегда можно узнать, что на этом входе — единица или ноль. Этот режим подходит для прослушивания какой-либо шины данных, так как он не оказывает на неё никакого влияния. Если ножка никуда не подключена и просто "висит в воздухе", то напряжение на ней будет колебаться от действия внешних наводок, электромагнитных полей и т.д.

Этот режим стоит по умолчанию.

Регистры портов имеют следующие адреса на шине:

PORTA	DDRA	PINA	PORTB	DDRB	PINB
\$1B(\$3B)	\$1A(\$3A)	\$19(\$39)	\$18(\$38)	\$17(\$37)	\$16(\$36)

PORTC	DDRC	PINC	PORTD	DDRD	PIND
\$15(\$35)	\$14(\$34)	\$13(\$33)	\$12(\$32)	\$11(\$31)	\$10(\$30)

Логическая схема порта ввода-вывода. Все описанные функции портов ввода-вывода демонстрирует также схема, изображенная на рис.10.1.

10.3. Примеры

Примеры для DDRx. Пример кода на C (Настройка ножки PD4 на вывод):

```
int main(void)
{
    DDRD|=(1<<PD4);
    return 0;
}
```

Пример кода на ASM (Настройка ножки PD4 на вывод):

```
Start:
sbi DDRD,4
```

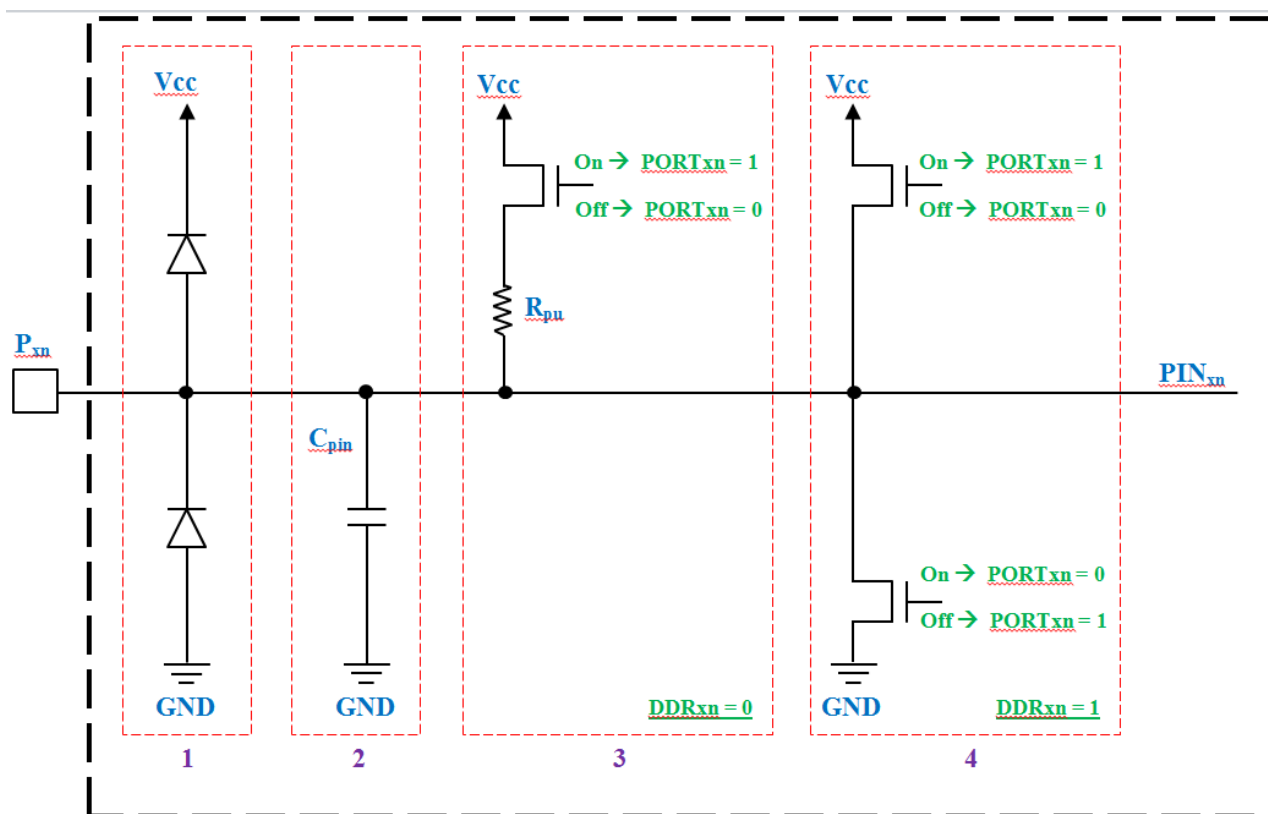


Рисунок 10.1 – Логическая схема порта в/в.

P_{xn} — имя ножки порта микроконтроллера, где x буква порта (A, B, C или D), n номер разряда порта (7... 0).

C_{pin} — паразитная емкость порта.

V_{cc} — напряжение питания.

R_{pu} — отключаемый нагрузочный верхний резистор (pull-up).

$PORT_{xn}$ — бит n регистра $PORTx$.

PIN_{xn} — бит n регистра $PINx$.

DDR_{xn} — бит n регистра $DDRx$

Примеры для $PORTx$.

При настройке соответствующей ножки на вывод, регистр $PORTx$ управляет состоянием этой ножки. То есть, установив 1 в соответствующем бите регистра, мы подадим логическую 1 на ножку контроллера (около +5 вольт), тем самым мы, например, зажигаем светодиод, подав на него эти +5v. Установив 0 в соответствующем бите регистра $PORTx$, мы установим логический 0 на ножке контроллера (около 0 вольт), тем самым, например, мы гасим светодиод, подав на него около 0v, что прекратит протекание электрического тока через него.

Пример кода на C (установим ножку PB3 в состояние вывода и подадим на нее логическую единицу):

```
int main(void)
{
    DDRB|=(1<<PB3);
    PORTB|=(1<<PB3);
    return 0;
}
```

Пример кода на Ассемблере (установим ножку PB3 в состояние вывода и подадим на нее логическую единицу):

Start:

```
sbi DDRB,3
sbi PORTB,3
```

Примеры для PINx.

Каждый бит регистра PINx, показывает текущее состояние соответствующей ножки контроллера, при её настройке в состояние ввода. Данный регистр, в отличие от двух предыдущих работает только в режиме чтения. То есть, записать что-либо в него мы не можем, однако считать состояние текущей ножки контроллера не составит труда! При подаче логической единицы на соответствующую ножку контроллера (которая, разумеется, работает в режиме ввода), в соответствующий бит данного регистра будет записана единица. Наоборот, при подаче логического нуля, в соответствующий бит будет записан 0.

Схема управления светодиодом.

Следующий пример поможет Вам лучше понять описанное выше. Соберем следующую цепь (рис.10.2)

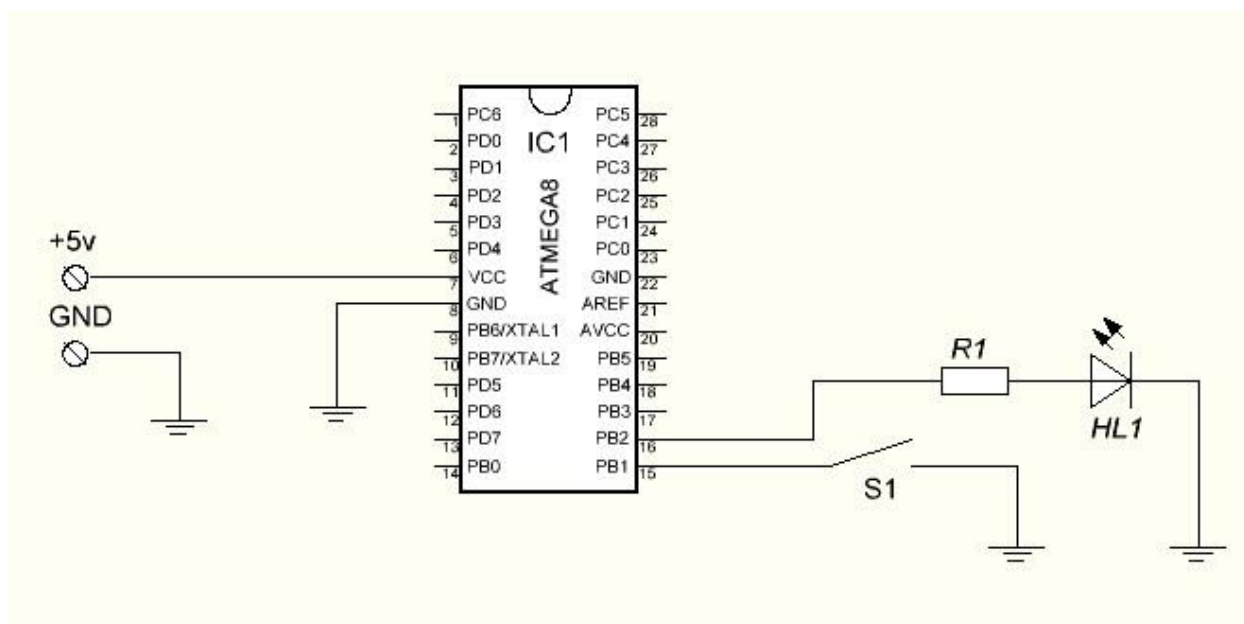


Рисунок 10.2 – Схема управления светодиодом с МК АТмега8.
Номинал резистора R1 можно рассчитать самостоятельно

Создадим в Atmel Studio новый проект на удобном Вам языке.

С:

```
#include <avr/io.h> // подключаем заголовочные файлы
int main(void)
{
    DDRB|=(1<<PB2); //Настраиваем ножку PB2 в режим выхода
    DDRB&=~(1<<PB1); //Настраиваем ножку PB1 в режим входа
```

```

    PORTB|=(1<<PB1); //Устанавливаем pull-up режим ножки PB1
while(1) //бесконечный цикл
{
    if((PINB&(1<<PB1))==0)//Если на PB1 логический ноль
    {
        PORTB|=(1<<PB2);//Зажигаем светодиод
    }
    else// если на PB1 логическая единица
    {
        PORTB&=~(1<<PB2);//Гасим светодиод
    }
}
}

```

ASSEMBLER:

```

.include "m8def.inc" //подключаем заголовочные файлы
start:
    sbi DDRB,PB2    //Настраиваем ножку PB2 в режим выхода
    cbi DDRB,PB1    //Настраиваем ножку PB1 в режим входа
    sbi PORTB,PB2   //Устанавливаем pull-up режим ножки PB2
main_loop: //Бесконечный цикл
    sbic PINB,PB1   //Пропустить следующую команду если бит PB1
регистра PINB = 0
        cbi PORTB,PB2 //Гасим светодиод
    sbis PINB,PB1   //Пропустить следующую команду если бит PB1
регистра PINB = 1
        sbi PORTB,PB2 //Зажигаем светодиод
    rjmp main_loop //прыжок на метку бесконечного цикла

```

При нажатии кнопки S1, на ножку PB1 контроллера AtMega8, подается логический ноль. В бесконечном цикле мы проверяем состояние регистра PINB, и если бит PB1 равен 0, то зажигаем светодиод. В противном случае, выключаем светодиод HL1.

Выводы [AVR. Учебный курс. <http://easyelectronics.ru/avr-uchebnyj-kurs-ustrojstvo-i-rabota-portov-vvoda-vyvoda.html>]:

- Самый безопасный для МК и схемы, ни на что не влияющий режим это Hi-Z (High-Z). Очевидно, что этот режим и должен быть по умолчанию.
- Значения большинства портов I/O при включении питания/сбросе = 0x00, PORT и DDR не исключение.
- Соответственно когда DDR=0 и PORT=0 это Hi-Z – самый безопасный режим, оптимальный при старте.
- Hi-Z это вход, значит при DDR=0 нога настроена на вход. Запомнили.

- Однако, если $DDR=0$ – вход, то что будет если PORT переключить в 1? Очевидно, что будет другой режим входа. Какой? Pull Up, другого не дано! Логично? Логично. Запомнили.
- Раз дефолтный режим был входом и одновременно в регистрах нуль, то для того, чтобы настроить ножку на выход надо в DDR записать 1.
- Ну, а состояние выхода уже соответствует регистру PORT — высокий это 1, низкий это 0.
- Читаем же из регистра PIN.

Есть еще один способ, мнемонический:

«1» похожа на стрелку. Стрелка, выходящая из МК, – выход. Значит $DDR=1$ это выход! 0 похож на гнездо, дырку – вход! Резистор подтяжки дает в висящем порту единичку, значит, PORT в режиме Pull Up должен быть в единичке!

Глава 11. Таймер T0

11.1. Таймеры

Назначение таймера.

Таймер (англ. timer от time – отмечать время) – контрольно-регулирующий прибор, который по истечении заданного промежутка времени выдает определенный сигнал либо включает (отключает) какое-либо оборудование.

По назначению таймеры делятся на две группы:

- ☐ таймеры общего назначения (таймеры/счетчики, Timer/Counters);
- ☐ сторожевой таймер (Watchdog Timer, букв. «сторожевой пёс»).

Функции таймера достигаются путем подсчета им тактовых импульсов фиксированной частоты («тиков», «clocks») либо подсчета любых внешних импульсных сигналов. Считается, что в первом случае устройство выполняет функцию *таймера*, во втором – *счетчика*. Если выполняются обе функции, то таймеры называют *таймерами-счетчиками*.

В современных микроконтроллерах, кроме простейших функций

- подсчета импульсов,

на таймеры/счетчики возлагаются обычно дополнительные функции:

- захвата (Capture);
- сравнения (Compare);
- широтно-импульсной модуляции (ШИМ, PWM – Pulse-Width Modulation).

Широтно-импульсный модулятор генерирует последовательность импульсов со скважностью, пропорциональной уровню сигнала на его входе. Скважность есть отношение периода следования (повторения) импульсов к длительности импульса;

- часов реального времени (RTC – Real Time Clock).

В режиме захвата содержимое таймера/счетчика в момент времени, задаваемый каким-либо внешним событием на входе захвата, запоминается в специальном регистре захвата и становится доступным для процессорного ядра. Одновременно формируется запрос на прерывание, сообщающий программе о готовности данных.

В режиме сравнения содержимое таймера/счетчика сравнивается с некоторым фиксированным числом, хранящимся в специальном регистре сравнения микроконтроллера. В момент равенства данных формируется сигнал запроса на прерывание и сигнал на одном из контактов микросхемы.

В режиме широтно-импульсной модуляции таймер/счетчик формирует на одном из выходов микросхемы последовательность импульсов определенной частоты, в которой длительность импульсов может быть изменена программно.

В режиме реального времени таймер/счетчик производит отсчет времени в общепринятых физических единицах, секундах, минутах, часах.

Количество таймеров/счетчиков, интегрируемых на кристалл микроконтроллера, может быть различно. Микроконтроллеры серии AVR содержат в

своем составе от 1 до 4 таймеров/счетчиков и 1 сторожевой таймер (его назначение пояснено в п.2.1). Таймеры/счетчики обозначаются символами T0, T1, T2 и T3, сторожевой таймер – символом WDT (WatchDog Timer). Таймер T3 присутствует только в моделях ATmega162х и ATmega64х/128х. Микроконтроллер ATmega16 имеет три универсальных таймера/счетчика: два 8-битных (Timer/Counter0 и Timer/Counter2) и один 16-битный (Timer/Counter1).

Для регулирования частоты входного сигнала все таймеры снабжаются предварительными делителями («предделителями», prescalers). Таймер/счетчик 0 и таймер/счетчик 1 имеют общий предделитель, обеспечивающий изменение частоты на входе каждого счетчика в диапазоне от 1/8 до 1/1024 входной тактовой частоты CLK (Clock).

Каждый таймер/счетчик использует один или более выводов (ножек) микроконтроллера. Эти ножки могут быть либо линиями портов ввода-вывода с альтернативной функцией, либо выделенными ножками микроконтроллера.

11.2. Таймер-счетчик T0 ATmega16 (8 бит)

Функции таймера/счетчика T0:

- отсчет и измерение временных интервалов или подсчет внешних событий с ножки T0. При переполнении восьми бит выставляется флажок TOV0 в регистре TIFR и внешний сигнал на ножку OC0;
- сравнение с заданным значением и при совпадении выставление флажка OCF0 в регистре TIFR и внешнего сигнала на ножку OC0;
- формирование выходного ШИМ сигнала на ножке OC0.

Для их выполнения таймер T0 поддерживает 4 различных режима работы, отвечающих той или иной функции.

Используемые выводы. Таймер/счетчик T0 использует два вывода (ножки) микроконтроллера ATmega16 (рис.11.1 и 11.2). Ножка T0 (PB0) – это вход внешнего тактового сигнала. Она может применяться, например, для подсчета импульсов. Ножка OC0 (PB3) – это выход схемы сравнения таймера-счетчика. На этой ножке таймер может формировать меандр * или ШИМ сигнал. Также она может просто менять свое состояние при срабатывании схемы сравнения. Ножки T0 и OC0 задействуются только при соответствующих настройках таймера, в обычном состоянии это ножки общего назначения PB0 и PB3.

* Меандр – прямоугольный импульс, длительность которого равна длительности паузы

В состав таймера/счетчика T0 входит набор программно-доступных регистров TCNT0, OCR0, TCCR0 и схема управления (Control Logic) (рис.11.2). Кроме этого, в T0 используется два разряда регистра флагов прерываний TIFR и два разряда регистра маски прерываний TIMSK. [24]

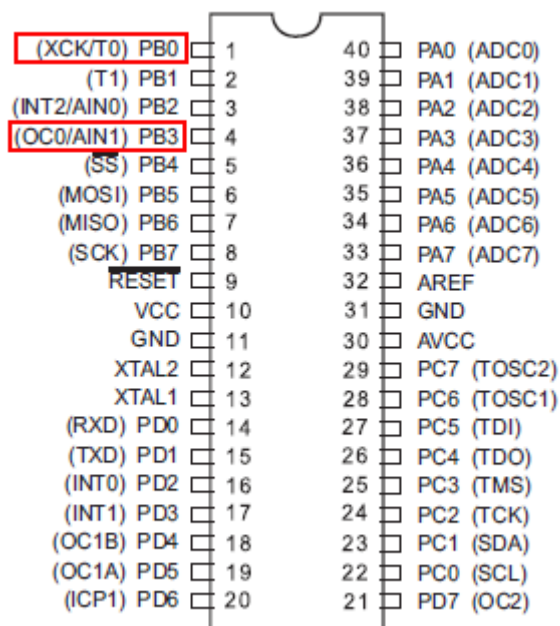
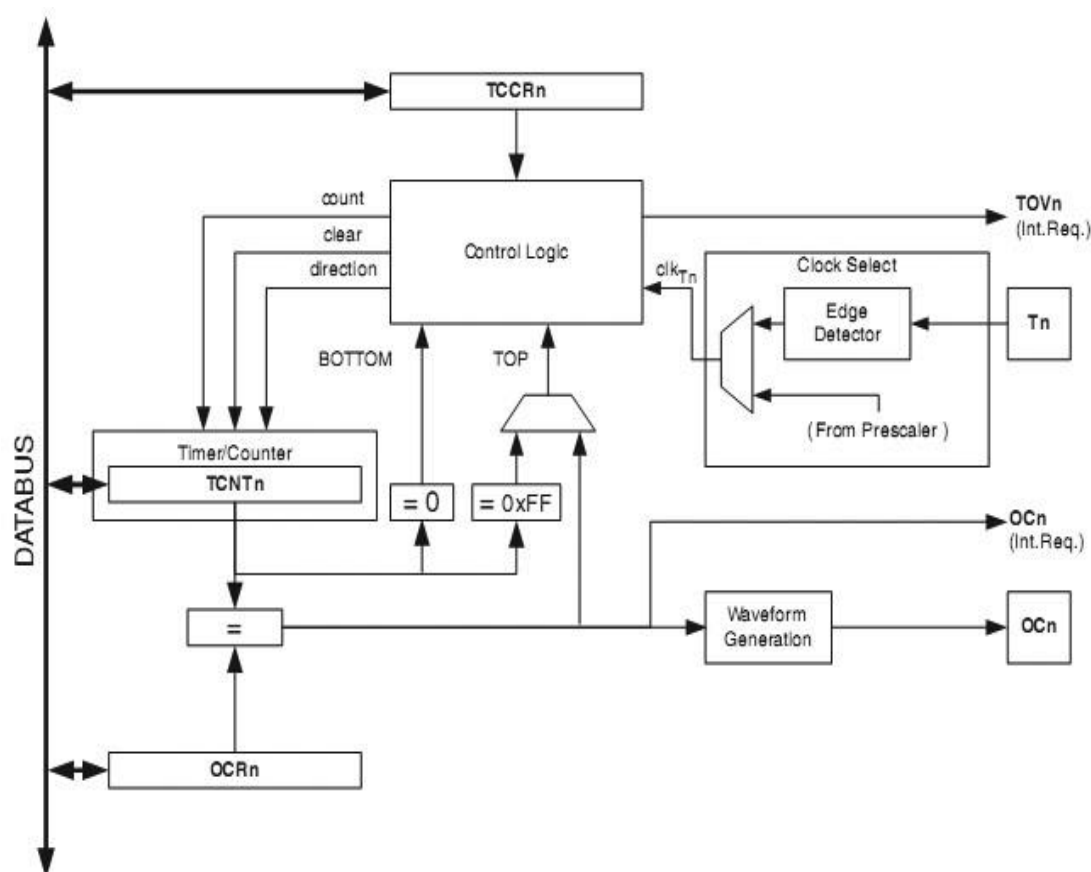


Рисунок 11.1 – Выводы таймера 0 ATmega16.

Рисунок 11.2 – Схема 8-битного таймера/счетчика.
n=0 – таймер T0, n=2 – T2

Источники тактового сигнала. Таймер/счетчик T0 тактируется внутренним источником тактовых импульсов фиксированной частоты процессорного ядра CLK (Clock) через предварительный делитель частоты («предделитель»,

«prescaler»), или внешним источником сигналов через входную ножку T0. В первом случае он является таймером, во втором – счетчиком внешних событий.

Выбор источника и частоты внутренних сигналов, а также запуск и останов таймера/счетчика осуществляется с помощью битов CS00 – CS02 (Clock Select) регистра управления таймером TCCR0 в соответствии с табл.11.1. Как видим, таймер-счетчик может быть остановлен (комбинацией 000 в этих битах) и запущен другими комбинациями, которые также определяют тактирование таймера от заданной внутренней частоты или от входного сигнала на ножке T0.

Таблица 11.1 – Коммутация входа таймера/счетчика 0

CS02	CS012	CS00	Источник сигнала тактирования СК
0	0	0	Источника тактирования нет. Остановка таймера
0	0	1	CLK – тактовая частота МК
0	1	0	CLK/8
0	1	1	CLK/64
1	0	0	CLK/256
1	0	1	CLK/1024
1	1	0	Внешний контакт T0, задний фронт
1	1	1	Внешний контакт T0, передний фронт

11.3. Регистры таймера-счетчика T0

Регистры таймера служат для его программирования на выполнение требуемых функций. Таймер Т0 имеет в своем составе три регистра (см. рис.11.2):

- счетный регистр TCNT0 (Timer/Counter Register),
- регистр сравнения с выходом (результата сравнения на ножку OC0), или просто «регистр сравнения» OCR0 (Output Compare Register),
- управляющий регистр TCCR0 (Timer/Counter Control Register).

Кроме этих есть еще три регистра, относящиеся ко всем трем таймерам ATmega16:

- регистр маски прерываний, масочный регистр, TIMSK (Timer/Counter Interrupt Mask Register),
- регистр флагов прерываний TIFR (Timer/Counter Interrupt Flag Register).
- регистр специальных функций ввода-вывода SFIOR (Special Function IO Register)

Описание регистров таймера-счетчика Т0.

Начнем с самого простого.

TCNT0 – \$32 (\$52)

– **Timer/Counter Register**, счетный регистр

Bit	7	6	5	4	3	2	1	0	
	TCNT0[7:0]								TCNT0
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Это 8-ми разрядный счетный регистр. Когда таймер работает, по каждому импульсу тактового сигнала (тику) СК, или clk_{T0} , (CLK , $\text{CLK}/8$, ..., $T0$ задний фронт, $T0$ передний фронт) значение TCNT0 изменяется на единицу. В зависимости от режима работы таймера, счетный регистр может или увеличиваться, или уменьшаться.

При переполнении счетчика, т.е. при переходе TCNT0 из состояния $\$FF$ в состояние $\$00$, устанавливается в 1 флаг переполнения TOV0 (Timer Overflow) в регистре TIFR и генерируется запрос на прерывание. Разрешение прерывания выполняется установкой в 1 разряда TOIE0 (Timer Overflow Interrupt Enable) регистра маски TIMSK . Естественно, что флаг общего разрешения прерывания I регистра состояния SREG микроконтроллера также должен быть установлен в 1. Соответствующий вектор прерывания равен $\$10$. Флаг переполнения TOV0 может быть сброшен пользователем, либо он будет сброшен автоматически при переходе процессора на вектор прерывания.

Регистр TCNT0 доступен в любой момент времени как для чтения, так и для записи. Последнее используется, когда требуется задать его начальное значение (начиная с которого будет происходить счет). Когда таймер работает, изменять содержимое TCNT0 не рекомендуется, так как это блокирует схему сравнения на один такт.

OCR0 – \$3C (\$5C)

– Output Compare Register, регистр сравнения

Bit	7	6	5	4	3	2	1	0	
	OCR0[7:0]								OCR0
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Это 8-ми разрядный регистр сравнения. Его значение постоянно. В каждый тактовый момент оно сравнивается со счетным регистром TCNT0 , и в случае совпадения флажок OCF0 в регистре TIFR устанавливается в 1, формируя сигнал запроса на прерывание (этот флажок может считываться и программно), и возбуждается сигнал на контакте OC0 микросхемы.

Значение OCR0 можно как читать, так и записывать.

TCCR0 – \$33 (\$53)

– Timer/Counter Control Register, регистр управления

Bit	7	6	5	4	3	2	1	0	
	FOC0	WGM00	COM01	COM00	WGM01	CS02	CS01	CS00	TCCR0
Read/Write	W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Это управляющий регистр таймера-счетчика $T0$, он определяет источник тактирования таймера, коэффициент предделителя, режим работы таймера и

поведение вывода OC0. С его помощью осуществляется также запуск и останов таймера/счетчика путем записи в биты **CS02-CS00** (Clock Select) нужных комбинаций: 000 – останов, остальные комбинации – запуск. Все возможные состояния описаны в табл.11.1. По сути, самый важный регистр.

Режимы таймера-счетчика TCNT0.

Всего их четыре – нормальный режим (normal), сброс таймера при совпадении (CTC) и два режима широтно-импульсной модуляции (Fast PWM – Pulse Width Modulation и Phase Correct PWM). Режим назначается битами **WGM10, WGM00** (Wave Generator Mode). Все возможные значения описаны в таблице ниже.

Таблица 11.2 – Режимы работы таймера/счетчика 0

WGM01	WGM00	Режим работы таймера/счетчика
0	0	Normal
0	1	PWM, Phase Correct
1	0	CTC
1	1	Fast PWM

Нормальный режим (Normal Mode).

При WGM01:0 = 0:0 включается нормальный режим. Это наиболее простой режим работы таймеров/счетчиков. В режиме Normal счетный регистр функционирует как обычный суммирующий счетчик. По каждому импульсу тактового сигнала СК (CLK, CLK/8, ..., внешний сигнал с контакта T0) осуществляется инкрементирование счетного регистра. При переходе через значение \$FF возникает переполнение, и счет продолжается со значения \$00. В том же такте сигнала СК, в котором обнуляется регистр TCNT0, флаг прерывания по переполнению TOV0 устанавливается в 1.

Табл.11.3 показывает поведение сигнала выхода сравнения на ножке OC0 в режимах Normal и CTC в зависимости от битов **COM01-COM00** (Compare Match Output Mode).

Таблица 11.3 – Управление выходом сравнения OC0 в режимах Normal и CTC

COM01	COM00	Результат
0	0	Выход отключен
0	1	Переключение выхода OC0 при установке флага
1	0	Очистка выхода OC0 при установке флага
1	1	Установка выхода OC0 при установке флага

Режим очистки таймера по совпадению (CTC Mode).

При $WGM01:0 = 1:0$ включается режим очистки таймера по совпадению CTC (Clear Timer on Compare Match Mode). При этом счетчик TCNT0 обнуляется, когда его состояние совпадает с состоянием регистра сравнения OCR0.

Каждый раз, когда значение счетчика TCNT0 достигает OCR0, устанавливается OCF0 флаг в регистре TIFR. Если прерывание разрешено, то обработчик прерывания можно использовать для обновления содержимого регистра сравнения OCR0.

Для генерации выходного сигнала в режиме CTC на выходе сравнения OC0 можно установить переключение его логического уровня в соответствии с табл.11.3 при каждой установке флага OCIE0.

Частота сигнала на выходе OC0 определяется следующей формулой:

$$f_{OC0} = \frac{CLK}{2N(1 + OCR0)}$$

где N – коэффициент деления делителя (1, 8, 64, 256 или 1024).

Быстрый режим ШИМ (Fast PWM Mode).

Быстрая ШИМ используется в случаях когда нам не важна фаза импульсов, например, для управления скоростью вращения двигателя или яркостью свечения светодиода.

Быстрый режим широтно-импульсной модуляции (Fast PWM Mode) включается при $WGM01:0 = 1:1$. При этом счетчик TCNT0 считает до переполнения. Каждый раз при сравнении устанавливается флаг OCF0 и при переполнении – флаг TOV0.

Если прерывание разрешено, то обработчик прерываний может быть использован для обновления значения в регистре сравнения OCR0.

Режим Fast PWM позволяет сформировать ШИМ сигнал на контакте OC0. Установка битов COM01 и COM00 в регистре TCCR0 в соответствии с табл.11.4 позволяет получить неинвертированный или инвертированный ШИМ-сигнал.

Таблица 11.4 – Управление выходом сравнения OC0 в режимах Fast PWM и Correct PWM

COM01	COM00	Результат
0	0	Выход отключен
0	1	Резерв
1	0	Сброс OC0 при сравнении, установка OC0 при обнулении, (<i>non-inverting mode</i>)
1	1	Установка OC0 при сравнении, сброс OC0 при обнулении, (<i>inverting mode</i>)

Частота ШИМ для вывода может быть рассчитана по следующей формуле

$$f_{OC0} = \frac{CLK}{256 \cdot N},$$

где N – коэффициент деления предделителя (1, 8, 64, 256 или 1024).

Режим ШИМ с коррекцией фазы (Phase Correct PWM Mode).

Режим Phase Correct PWM включается при $WGM01:0 = 0:1$. В этом режиме счетчик считает от 0 до максимального значения $MAX = 255$, а далее в обратном порядке от MAX до 0.

Настройка битов $COM01$ и $COM00$ в соответствии с табл.10.4 позволяет получить инвертированный или неинвертированный сигнал ШИМ на выходе $OC0$.

Частота ШИМ для вывода $OC0$ может быть рассчитана по формуле:

$$f_{OC0} = \frac{CLK}{510 \cdot N},$$

где N – коэффициент деления предделителя (1, 8, 64, 256 или 1024).

И последний бит регистра $TCCR0$ – это бит **FOC0** (Force Output Compare). Этот бит предназначен для принудительного изменения состояния вывода $OC0$. Он работает только для режимов Normal и CTC. При установке бита $FOC0$ в единицу состояние вывода меняется соответственно значениям битов $COM01$, $COM00$. $FOC0$ бит не вызывает прерывания и не сбрасывает таймер в CTC режиме.

Контакт микросхемы $OC0$ (Output Compare 0) предназначен для использования в различных режимах с использованием регистра сравнения $OCR0$. Его поведение зависит от битов **COM01**, **COM00** (Compare Match Output Mode) в соответствии с табл.11.4. Если хоть один из этих битов установлен в 1, то вывод $OC0$ перестает функционировать как обычный вывод общего назначения $PB3$ и подключается к схеме сравнения таймера счетчика $T0$. После этого схема сравнения начинает выдавать по нему сигнал при каждом переполнении счетного регистра (при установке флага переполнения $TOV0$) – в режиме Normal или при совпадении сравнения (при установке флага сравнения $OCF0$) – в режимах CTC, Fast PWM и Phase Correct PWM. Однако при этом он должен быть еще настроен как выход с помощью регистра $DDRB$. Из табл.11.4 мы также видим, что в режимах Normal и CTC вывод $OC0$ ведет себя одинаково, а вот в режимах широтно-импульсной модуляции его поведение отличается прямыми и инверсными значениями и частотой (см. формулы для частоты ШИМ выше).

11.4. Регистры, разделяемые таймерами-счетчиками

Следующие регистры используются всеми таймерами/счетчиками ATmega16.

TIMSK – \$39 (\$59)

– **Timer/Counter Interrupt Mask Register, регистр маски прерываний**

Bit	7	6	5	4	3	2	1	0	
	OCIE2	TOIE2	TICIE1	OCIE1A	OCIE1B	TOIE1	OCIE0	TOIE0	TIMSK
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 0 – TOIE0: Timer/Counter0 Overflow Interrupt Enable**, 0-е значение бита запрещает прерывание по событию переполнение, 1 – разрешает.
- **Bit 1 – OCIE0: Timer/Counter0 Output Compare Match Interrupt Enable**, 0-е значение запрещает прерывания по событию совпадения, а 1 разрешает.

Общий регистр для всех трех таймеров ATmega16, он содержит флаги разрешения прерываний. Таймер T0 может вызывать прерывания при переполнении счетного регистра TCNT0 и при совпадении счетного регистра с регистром сравнения OCR0. Соответственно для таймера T0 в регистре TIMSK зарезервированы два бита – это TOIE0 и OCIE0. Остальные биты относятся к другим таймерам.

Естественно, прерывания будут вызываться, только если установлен бит глобального разрешения прерываний – бит I регистра SREG.

TIFR – \$38 (\$58)

– **Timer/Counter Interrupt Flag Register**, регистр флагов прерываний

Bit	7	6	5	4	3	2	1	0	
	OCF2	TOV2	ICF1	OCF1A	OCF1B	TOV1	OCF0	TOV0	TIFR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 0 – TOV0: Timer/Counter0 Overflow Flag**, устанавливается в 1 при переполнении счетного регистра.
- **Bit 1 – OCF0: Output Compare Flag 0**, устанавливается в 1 при совпадении счетного регистра с регистром сравнения

Общий для всех трех таймеров-счетчиков регистр. Содержит статусные флаги, которые устанавливаются при возникновении событий. Для таймера T0 – это переполнение счетного регистра TCNT0 и совпадение счетного регистра с регистром сравнения OCR0. Если в эти моменты в регистре TIMSK разрешены прерывания и установлен бит I, то микроконтроллер вызовет соответствующий обработчик.

Флаги автоматически очищаются при запуске обработчика прерывания. Также это можно сделать программно, записав 1 в соответствующий флаг.

SFIOR – \$30 (\$50)

– **Special Function IO Register**, регистр специальных функций ввода-вывода

Bit	7	6	5	4	3	2	1	0	
	ADTS2	ADTS1	ADTS0	—	ACME	PUD	PSR2	PSR10	SFIOR
Read/Write	R/W	R/W	R/W	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 0 – PSR10: Prescaler Reset Timer/Counter1 and Timer/Counter0**, при установке бита в 1 осуществляется сброс 10-ти разрядного двоичного счетчика, который делит входную частоту для таймера T0 и таймера T1.

Начинающему про этот регистр в принципе можно и не знать, один из его разрядов сбрасывает 10-ти разрядный двоичный счетчик, который делит входную частоту для таймера T0 и таймера T1. Сброс осуществляется при установке бита **PSR10 (Prescaler Reset Timer/Counter1 and Timer/Counter0)** в единицу.

Примечание. 8-битный таймер/счетчик 2 отличается от T0 лишь тем, что выполняет еще одну функцию. Если T0 (а также T1) считает либо тактовые импульсы от встроенного тактового генератора, либо со счетного входа, то таймер/счетчик 2 способен работать в асинхронном режиме независимо от тактовых импульсов CLK. То есть T2 считает не тактовые импульсы процессора, не входящие импульсы на ножки, а импульсы своего собственного генератора, работающего от отдельного кварца. Для этого у T2 есть входы TOSC1 и TOSC2, на которые можно повесить кварцевый резонатор. Соответственно для управления счетом с этих входов T2 обладает дополнительным асинхронным регистром состояния ASSR.

Зачем это вообще надо? Да хотя бы организовать часы реального времени. Повесил на них часовой кварц на 32768 Гц да считай время — за секунду произойдет 128 переполнений (т.к. T2 восьми разрядный). Так что одно переполнение это 1/128 секунды. Причем на время обработки прерывания по переполнению таймер не останавливается, он также продолжает считать. Так что часы сделать плевое дело!

В других отношениях T0 и T2 функционально одинаковы.

Глава 12. Таймер T1

11.2. Таймер-счетчик T1 ATmega16 (16 бит)

Таймеры-счетчики – это такие устройства или модули в микроконтроллере, которые, как видно из названия, постоянно что-то считают. Считают они либо до заданной величины, либо до предельной величины, какую допускает их битность. Считают они постоянно с одной скоростью – со скоростью тактовой частоты микроконтроллера, поправленной делителями частоты, которые мы будем конфигурировать в определенных регистрах. И вот эти таймеры-счетчики постоянно считают, если мы их инициализируем.

Микроконтроллеры серии AVR содержат в своем составе от 1 до 4 таймеров/счетчиков и 1 сторожевой таймер. Таймеры/счетчики обозначаются символами T0, T1, T2 и T3, сторожевой таймер – символом WDT (WatchDog Timer, букв. «сторожевой пёс»). Таймер T3 присутствует только в моделях ATmega162x и ATmega64x/128x.

Схема. Упрощенная блок-схема T1 показана на рис.12.1.

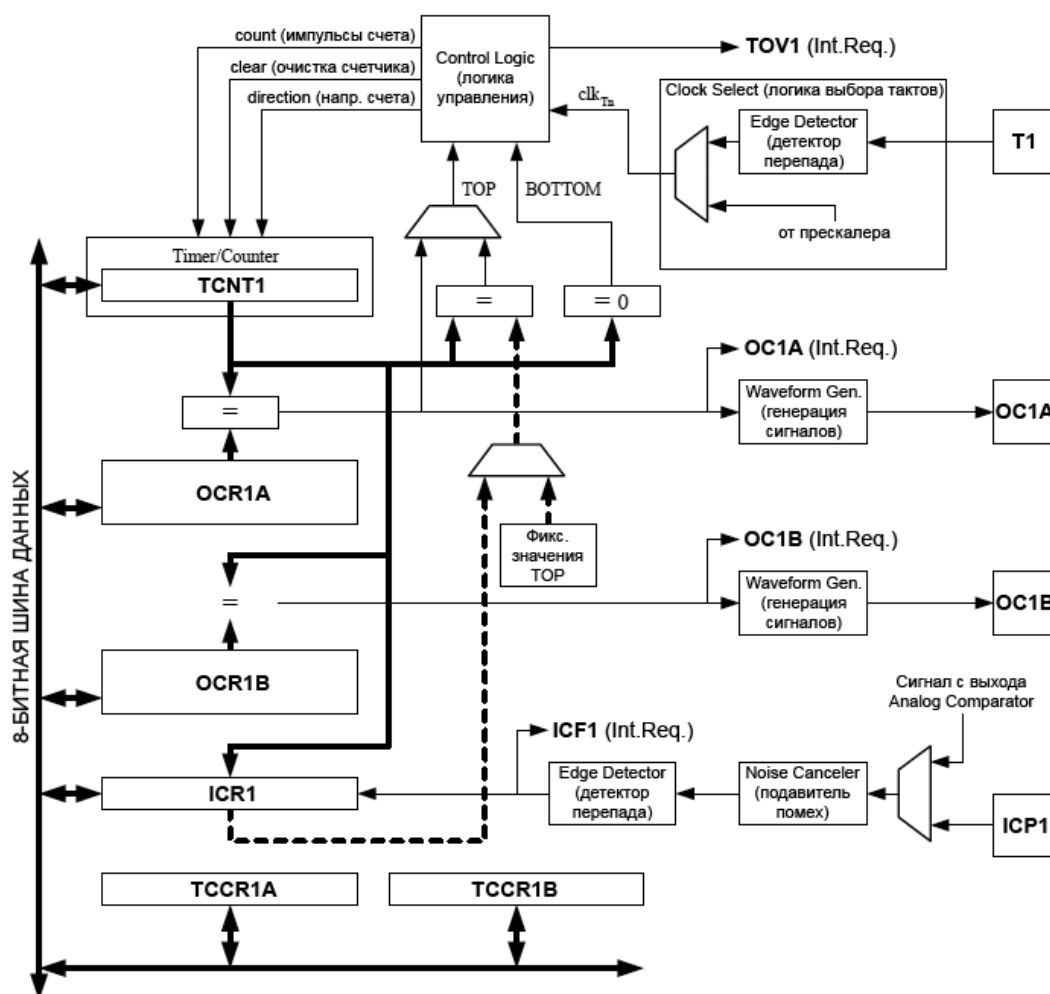


Рисунок 12.1 – Блок-схема 16-bit Timer/Counter 1

Функции таймера/счетчика T1:

- подсчет счетчиком TCNT1 (напрямую или через предделитель) тактовых импульсов процессорного ядра CLK или импульсов со входа T1 микроконтроллера. При переполнении счетчика формируется флаг запроса на прерывание TOV1;
- сравнение значения счетного регистра с заданным значением в регистрах OCR1A/B. При совпадении устанавливается флаг прерываний OCF1A/B в регистре TIFR и изменяется состояние ножки OC1A/B микроконтроллера;
- широтно-импульсная модуляция выходного сигнала на ножке OC1A/B;
- захват значения счетного регистра по сигналу, поступающему на ножку ICP1.

Для выполнения этих функций Таймер T1 поддерживает различные режимы работы (п.12.2).

Характеристики таймера/счетчика T1:

- Истинная 16-битная модель (True 16-bit Design)
- Два независимых выхода OC1A и OC1B (Two Independent Output Compare Units)
- Двойной буферный регистр сравнения (Double Buffered Output Compare Registers)
- Таймер сброса при совпадении (Clear Timer on Compare Match)
- Один вход захвата (One Input Capture Unit)
- Блок шумоподавления входа захвата (Input Capture Noise Canceller)
- Изменяемый период ШИМ сигнала (Variable PWM Period)
- Фазовый корректор ШИМ сигнала (Phase Correct PWM)
- Тактовый генератор (Frequency Generator)
- Счетчик внешних событий (External Event Counter)
- Четыре независимых источника прерывания (TOV1, OCF1A, OCF1B and ICF1 in register TIFR)

Регистры таймера/счетчика T1:

- TCNT1 (Timer/Counter1) – счетный регистр таймера/счетчика T1 (16 бит)
- OCR1A (Output Compare Register 1 A) – регистр сравнения A (16 бит)
- OCR1B (Output Compare Register 1 B) – регистр сравнения B (16 бит)
- ICR1 (Input Capture Register) – регистр захвата входа (16 бит)
- TCCR1A (Timer/Counter1 Control Register A) – регистр управления A (8 бит)
- TCCR1B (Timer/Counter1 Control Register B) – регистр управления B (8 бит)

Общие для таймеров/счетчиков регистры:

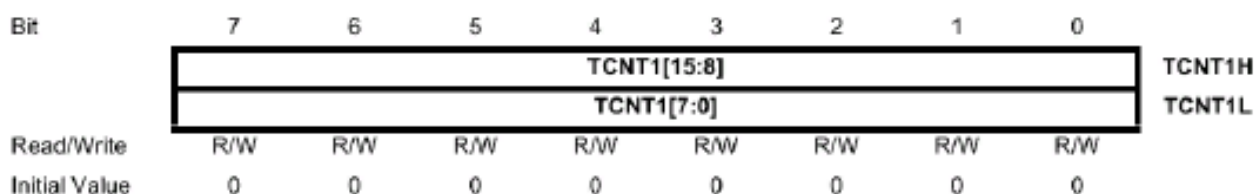
- TIMSK (Timer/Counter Interrupt Mask Register) – регистр маски прерываний, общий для всех таймеров/счетчиков (8 бит)
- TIFR (Timer/Counter Interrupt Flag Register) – регистр флагов прерываний, общий для всех таймеров/счетчиков (8 бит)

Используемые выводы. Таймер-счетчик T1 использует четыре вывода (ножки) микроконтроллера ATmega16 (см. рис.12.1). Ножка T1 – это вход внешнего тактового сигнала. Она может применяться, например, для подсчета импульсов. Ножки OC1A/B – это выходы схемы сравнения таймера-счетчика. На этих ножках таймер может формировать меандр или ШИМ сигнал. Также они могут просто менять свое состояние при срабатывании схемы сравнения. Ножка ICP1 служит входом захвата таймера счетчика T1. Все ножки задействуются только при соответствующих настройках таймера, в обычном состоянии это ножки общего назначения PB1, PD5/4 и PD6.

Источники тактового сигнала. Таймер-счетчик может тактироваться внутренне – тактовым сигналом CLK (Clock), используемым для всего микроконтроллера с подключением предделителя, или без предделителя; либо внешне – тактовым сигналом, поступающим на ножку T1. Блок выбора источника тактирования позволяет выбрать тактовый источник и фронт, по которому будет изменяться состояние таймера-счетчика. Возможные варианты назначаются битами CS12, CS11, CS10 (Control Select) регистра TCCR1B (п.11.2, табл.12.3). Если тактовый источник не задан (комбинация CS12:0=000), то таймер-счетчик стоит. Другие комбинации запускают таймер. Сигнал на выходе блока выбора тактового источника является тактовым сигналом таймера СК, или clk_{T1} (см. рис.12.1).

12.2. Регистры таймера-счетчика T1

TCNT1: TCNT1H – \$2D (\$4D) and TCNT1L – \$2C (\$4C)
– Timer/Counter1, счетный регистр (16 бит)



Пара 8-битных регистров TCNT1H и TCNT1L вместе образует 16 битный регистр TCNT1. Данный регистр открыт как для записи, так и для чтения. При работе таймера 1, значение данного регистра при каждом счете СК изменяется на единицу. То есть в регистре TCNT1 записано число тактов, которые сосчитал таймер. Так же мы можем записать сюда любое число в диапазоне от 0 до 2¹⁶ в 16 степени. В таком случае отсчет тактов будет вестись не с 0, а с записанного нами числа.

При переполнении счетного регистра TCNT1 (при переходе его значения из \$FFFF в \$0000) устанавливается в 1 флажок переполнения таймера TOV1 (Timer/Counter0 Overflow Flag), находящийся в регистре TIFR.

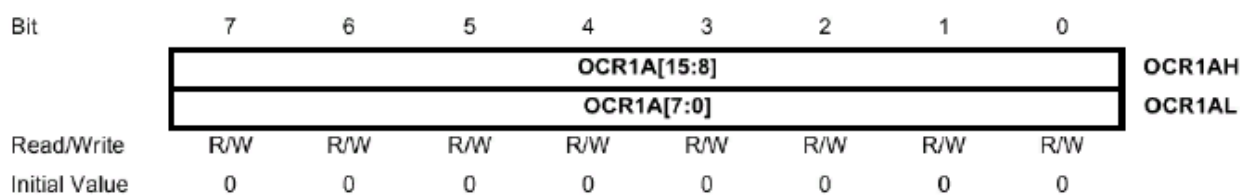
Физически таймер TCNT1, как указано, состоит из двух частей TCNT1H и TCNT1L. В связи с тем, что установка обоих счетчиков в начальное состояние

или чтение их текущего состояния должны осуществляться одновременно, а запись или чтение их происходит побайтно, то для согласования этих процедур введен регистр временного хранения, который программно недоступен. Соответственно при инициализации счетчика первым должен записываться старший байт в регистр TCNT1H, затем младший в регистр TCNT1L. А при чтении его текущего состояния первым считывается младший байт (регистр TCNT1L), а затем старший — регистр TCNT1H.

OCR1A и OCR1B – регистры сравнения

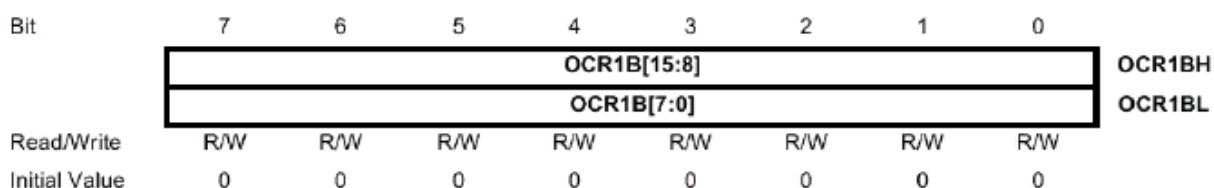
OCR1A: OCR1AH – \$2B (\$4B) and OCR1AL – \$2A (\$4A)

– Output Compare Register 1 A, регистр сравнения 1 A (16 бит)



OCR1B: OCR1BH – \$29 (\$49) and OCR1BL – \$2A (\$4A)

– Output Compare Register 1 B, регистр сравнения 1 B (16 бит)



Регистры сравнения OCR1A и OCR1B содержат 16-битную величину (каждый свою, значение которых постоянно), и эта величина непрерывно, в каждый тактовый момент, сравнивается со счетной величиной TCNT1. При совпадении устанавливается соответствующий флаг прерываний (OCF1A/B в регистре TIFR), который в свою очередь может служить источником прерывания (если оно разрешено). Кроме того, при наступлении этого события может изменяться состояние ножки OC1A/B микроконтроллера. Чтобы таймер/счетчик мог управлять состоянием этой ножки, она должна быть сконфигурирована как выход (соответствующий бит регистра DDRD должен быть установлен в 1).

1) Изменение состояния вывода OC1A/B при совпадении с регистром сравнения. Настроенный на выход, вывод OC1A/B может изменять своё состояние в момент совпадения значений счётного регистра и регистра сравнения. Характер этих изменений определяется комбинацией битов COM1A1, COM1A0, COM1B1 и COM1B0 (Compare Output Mode for Channel A/B) в регистре управления TCCR1A согласно табл.11.1 (см. описание регистра TCCR1A в п.11.2).

Назначая на каждом такте значение регистра сравнения, пропорциональное текущему аналоговому сигналу с датчика (принимаемому через АЦП), мы получаем на выходе ножки OC1A/B широтно-модулированный сигнал, длительность которого пропорциональна аналоговому сигналу. В этом случае таймер T1 работает как ШИМ.

2) Разрешается управление выводом OC1A/B установкой бита FOC1A/B в TCCR1A (или FOC3A в TCCR3A), однако при этом блокируется вызов прерывания по совпадению.

Регистры сравнения 16-битные. Чтобы быть уверенным, что оба старший и младший байты записываются одновременно, когда процессор (CPU) записывает в эти регистры, доступ к ним осуществляется с использованием скрытого 8-битного временного регистра High Byte Register (TEMP). Этот временный регистр разделяется всеми другими 16-битными регистрами.

Шестнадцатиразрядные таймеры в большинстве типов микроконтроллеров имеют возможность самостоятельно сбрасывать счётный регистр сразу после его совпадения с регистром сравнения, после чего счёт автоматически продолжается с нулевого значения. Такой режим называется режимом сброса при совпадении (CTC – Clear Timer on Compare Match Mode) и может быть использован, в частности, для генерации сигналов фиксированной частоты. Для перевода таймера в данный режим необходимо установить биты WGM12 и WGM13 (Wave Generator Mode) регистра TCCR1B (п.12.2, табл.12.2).

ICR1: ICR1H – \$27 (\$47) and ICR1L – \$26 (\$46)

– Input Capture Register 1, регистр захвата входа (16 бит)

Bit	7	6	5	4	3	2	1	0	
	ICR1[15:8]								ICR1H
	ICR1[7:0]								ICR1L
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Регистры ICR1H и ICR1L (Timer/Counter1 Input Capture Register High and Low Register) образуют 16-битный регистр входа захвата ICR1, доступный только для чтения.

При обнаружении на входе захвата ICP нарастающего или падающего фронта сигнала (определяемого установкой бита ICES1) текущее состояние таймера/счетчика 1 пересылается в регистр захвата ICR1. Одновременно устанавливается в состояние 1 флаг захвата входа ICF1 (Input Capture Flag 1) в регистре флагов прерывания TIFR.

При чтении всего 16-разрядного регистра операцию чтения необходимо начинать с младшего байта ICR1L.

Регистры управления TCCR1A и TCCR1B.

Служат для установки режима работы таймера/счетчика T1.

TCCR1A – \$2F (\$4F)

– Timer/Counter1 Control Register A, регистр управления 1 A (8 бит)

Bit	7	6	5	4	3	2	1	0	
	COM1A1	COM1A0	COM1B1	COM1B0	FOC1A	FOC1B	WGM11	WGM10	TCCR1A
Read/Write	R/W	R/W	R/W	R/W	W	W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Регистр управления TCCR1A управляет работой таймера/счетчика 1:

- **Bit 7:6 – COM1A1:0: Compare Output Mode for Channel A**
- **Bit 5:4 – COM1B1:0: Compare Output Mode for Channel B**

Биты управления для выходов OC1A и OC1B. Определяют характер выходного сигнала при сравнении (совпадении) содержимого таймера/счетчика 1 с регистрами OCR1A и OCR1B. Конфигурирование выходов представлено в табл.12.1.

Таблица 12.1 – Конфигурирование выходов OC1A и OC1B

COM1X1	COM1X0	Описание
0	0	Таймер/счетчик1 отключен от вывода OC1X
0	1	Переключение выходной линии OC1X
1	0	Очистка выходной линии OC1X (на линии низкий уровень)
1	1	Установка выходной линии OC1X (на линии высокий уровень)

Примечание: X = A или B

- **Bit 3 – FOC1A: Force Output Compare for Channel A**
- **Bit 2 – FOC1B: Force Output Compare for Channel B**

Прямое воздействие на выходы OC1A и OC1B. Запись логической единицы в эти биты приводит к изменениям на выводах OC1A и OC1B относительно значений, установленных битами COM1A1 и COM1A0; бит может использоваться, чтобы изменить состояние выходов, при этом никакое прерывание не генерируется и таймер не очищается. [Водовозов]

- **Bit 1:0 – WGM11:0: Waveform Generation Mode**

В сочетании с битами WGM13:2, размещенными в регистре TCCR1B, задают режим работы таймера/счетчика и максимальное (TOP) значение счетчика (табл.11.2). [6].

Поддерживаются режимы: нормальный (Counter, Normal Mode), очистки таймера по совпадению (CTC Mode – Clear Timer on Compare Match Mode) и три типа ШИМ режима (PWM Mode – Pulse Width Modulation Mode).

Режимы работы таймера/счетчика T1

Нормальный режим (Normal Mode)

Нормальный режим включается при WGM13:0 = 0 (табл.12.2). В этом режиме счетчик считает на сложение до максимального 16-битного значения (MAX = 0xFFFF) и сбрасывается в 0. При сбросе устанавливается флаг переполнения (TOV1). Установка флага переполнения приводит к формированию запроса на прерывание TIMER1_OVF (Timer/Counter1 Overflow) с вектором прерывания 0x10. Бит разрешения прерывания TOIE1 размещен в регистре TIMSK. При переходе на вектор прерывания флаг TOV1 автоматически сбрасывается.

Состояние счетчика можно изменить программно в любое время записью нового значения в регистр TCNT1H и TCNT1L. При 16-разрядных операциях записи обращение к старшему байту (TCNT1H) должно выполняться первым. При использовании таймера/ счетчика 1 в качестве 8-разрядного таймера достаточно производить запись только младшего байта.

При появлении сигнала на входе захвата ICP содержимое таймера счетчика копируется в регистры входа захвата ICR1H и ICR1L. Сигналом может

быть нарастающий или спадающий фронт импульса на входе ICP в зависимости от значения бита ICES1 (Input Capture1 Edge Select) в регистре управления TCCR1B. Одновременно в регистре флагов TIFR устанавливается флаг ICF1 и формируется запрос на прерывание TIMER1 CAPT (Timer/Counter1 Capture Event) с вектором прерывания 0x0A. Бит разрешения прерывания TICIE1 размещен в регистре TIMSK. При переходе на вектор прерывания флаг ICF1 автоматически сбрасывается.

Таблица 12.2 – Режимы работы таймера/счетчика T1

Mode	WGM13	WGM12 (CTC1)	WGM11 (PWM11)	WGM10 (PWM10)	Timer/Counter Mode of Operation	TOP	Update of OCR1X	TOV1 Flag Set on
0	0	0	0	0	Normal	0xFFFF	Immediate	MAX
1	0	0	0	1	PWM, Phase Correct, 8-bit	0x00FF	TOP	BOTTOM
2	0	0	1	0	PWM, Phase Correct, 9-bit	0x01FF	TOP	BOTTOM
3	0	0	1	1	PWM, Phase Correct, 10-bit	0x03FF	TOP	BOTTOM
4	0	1	0	0	CTC	OCR1A	Immediate	MAX
5	0	1	0	1	Fast PWM, 8-bit	0x00FF	TOP	TOP
6	0	1	1	0	Fast PWM, 9-bit	0x01FF	TOP	TOP
7	0	1	1	1	Fast PWM, 10-bit	0x03FF	TOP	TOP
8	1	0	0	0	PWM, Phase and Frequency Correct	ICR1	BOTTOM	BOTTOM
9	1	0	0	1	PWM, Phase and Frequency Correct	OCR1A	BOTTOM	BOTTOM
10	1	0	1	0	PWM, Phase Correct	ICR1	TOP	BOTTOM
11	1	0	1	1	PWM, Phase Correct	OCR1A	TOP	BOTTOM
12	1	1	0	0	CTC	ICR1	Immediate	MAX
13	1	1	0	1	Reserved	–	–	–
14	1	1	1	0	Fast PWM	ICR1	TOP	TOP
15	1	1	1	1	Fast PWM	OCR1A	TOP	TOP

Обозначения CTC1 и PWM11:0 устаревшие. Используйте для этих битов имена WGM12:0. Между тем, функционирование и локализация этих битов сходны с предыдущими версиями таймера.

Режим очистки таймера по совпадению (CTC Mode)

Режим очистки таймера по совпадению (Clear Timer on Compare Match (CTC) Mode) включается при WGM13:0 = 4 или 12. В режиме CTC счетчик обнуляется, когда значение счетчика TCNT1 соответствует либо значению регистра OCR1A (WGM13:0 = 4) или регистра ICR1 (WGM13: 0 = 12).

Регистры OCR1A или ICR1 определяют наибольшее значение в счетчике.

Запрос на прерывание может быть сформирован либо с помощью OCF1A или ICF1. Если соответствующее прерывание разрешено, то при переходе к подпрограмме произойдет сброс установленного флага запроса.

Для генерации выходного сигнала в режиме CTC, выход OC1A/B с помощью битов COM1A/B1:0 = 1 (табл. 12.1) может быть установлен в режим переключения его логического уровня при каждом запросе.

Частота сигнала на выходе OC1A/B определяется следующим уравнением:

$$f_{OC1A/B} = \frac{CLK}{2N(1 + OCR1A / B)}$$

где N – коэффициент деления предделителя (1, 8, 64, 256 или 1024).

Быстрый режим ШИМ (Fast PWM Mode)

Быстрый режим ШИМ включается при $WGM13:0 = 5, 6, 7, 14$ или 15 . В этом режиме счетчик считает на сложение до максимального (TOP) значения, а затем сбрасывается в 0.

Режим Fast PWM позволяет сформировать ШИМ сигнал на контактах OC1A и OC1A 0. Установка битов COM1A1:0 и COM1B1:0 в регистре TCCR1A в соответствии с табл.12.1 позволяет получить неинвертированный или инвертированный ШИМ-сигнал.

Режим ШИМ с корректной фазой (Phase Correct PWM Mode)

В режиме ШИМ с корректной фазой (Phase Correct PWM Mode) счетчик считает на сложение от 0 до максимального TOP значения и, далее, на вычитание от TOP до нуля.

Значения TOP могут быть фиксированными 8-, 9-, или 10 битов, определяться содержимым регистра захвата ICR1 или регистра сравнения OCR1A (см. табл.12.2).

Минимальное значение составляет 2 бита (ICR1 или OCR1A), максимальное – 16 бит (ICR1 или набор OCR1A).

Установка битов COM1x1:0 = 2 приводит к формированию неинвертированной ШИМ, а инвертированная ШИМ формируется при COM1x1:0 = 3 (см. табл.12.1).

В неинвертированной ШИМ при счете на сложение сигнал на выходе OC1x сбрасывается при равенстве TCNT1 и OCR1x, а при счете на сложение - устанавливается. Если сравниваемое значение в регистре OCR1x превышает значение TOP, то выходной сигнал OCR1x изменяется на противоположный при каждом достижении TCNT1 TOP значения.

В этом режиме частота выходного ШИМ сигнала в 2 раза меньше, чем в режиме Fast PWM, однако из-за симметричного характера этот способ предпочтителен для применений в задачах управления электродвигателями.

Флаг сравнения OC1x всегда устанавливается при равенстве TCNT1 и OCR1x.

Флаг переполнения TOV1 устанавливается каждый раз, когда счетчик обнуляется. Когда регистры OCR1A или ICR1 используются для задания значения TOP, флаги OC1A или ICF1 устанавливаются в процессе выполнения операций сравнения или захвата.

Режим ШИМ с корректной фазой и частотой (Phase and Frequency Correct PWM Mode)

Режим Phase and Frequency Correct PWM включается при ($WGM13:0 = 8$ или 9). В этом режиме, как и в предыдущем, счетчик считает на сложение от 0 до максимального TOP значения и, далее, на вычитание от TOP до нуля.

Значения TOP задаются содержимым регистра захвата ICR1 ($WGM13:0 = 8$) или регистра сравнения OCR1A ($WGM13:0 = 9$).

Флаги запросов на прерывания OC1A или ICF1 устанавливаются, когда TCNT1 достигает значения TOP. Они могут быть использованы, чтобы генерировать прерывание каждый раз при достижении счетчиком верхнего или нижнего значения.

Если TOP значение меньше, чем содержимое любого регистра сравнения, сравнение не происходит и сигнал на выходе не меняется. Поэтому выходной сигнал, в отличие от режима Phase Correct PWM Mode, генерируется симметрично во все периоды работы счетчика.

Неинвертированный ШИМ сигнал на выходе OC1x формируется при COMx1:0 = 2, инвертированный – при COMx1:0 = 3 (см. табл.12.1).

TCCR1B – \$2E (\$4E)

– Timer/Counter1 Control Register B, регистр управления 1 B (8 бит)

Bit	7	6	5	4	3	2	1	0	
	ICNC1	ICES1	–	WGM13	WGM12	CS12	CS11	CS10	TCCR1B
Read/Write	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Регистр TCCR1B отвечает за конфигурацию таймера 1:

- **Bit 7 – ICNC1: Input Capture Noise Canceler**

Управляет схемой подавления шума на входе захвата 1 (0 – выключена / 1 – включена). При сброшенном в 0 бите ICNC1 функция подавления шума входного триггера захвата запрещена. Вход захвата в этом случае реагирует на первый же импульс, поступивший на контакт входа захвата IC1. При установленном в 1 бите ICNC1 импульс, поступивший на вход захвата IC1, подвергается серьезной проверке – состояние входа IC1 опрашивается последовательно четыре раза. Все четыре выборки должны иметь одинаковый (высокий/низкий), определяемый битом ICES1, уровень. Частота опроса соответствует частоте синхронизации процессорного ядра.

- **Bit 6 – ICES1: Input Capture Edge Select**

Бит ICES1 регистра TCCR1B выбирает активный фронт срабатывания на входе захвата 1: при ICES1=0 содержимое таймера/счетчика 1 пересылается в регистр захвата входа ICR1 по падающему фронту на входе захвата IC1, при ICES1=1 – по нарастающему фронту на входе захвата IC1.

- **Bit 5 – Reserved Bit**

Резервный бит.

- **Bit 4:3 – WGM13:2: Waveform Generation Mode**

Задают режим работы счетчика совместно с битами WGM11:0 регистра TCCR1B согласно табл.11.2.

- **Bit 2:0 – CS12:0: Clock Select**

Битами CS10-CS12 мы выбираем источник тактовых сигналов – внутренний, с делителем тактовой частоты или без, CLK, CLK/8, ... или внешний – с контакта T1 и фронт внешнего сигнала, по которому будет изменяться состояние таймера-счетчика, в соответствие с табл.12.3.

Таблица 12.3 – Выбор тактовых источников таймера/счетчика T1

CS12	CS11	CS10	Источник сигнала тактирования СК
------	------	------	----------------------------------

0	0	0	Источник тактирования нет. Таймер остановлен
0	0	1	CLK – тактовая частота МК
0	1	0	CLK/8
0	1	1	CLK/64
1	0	0	CLK/256
1	0	1	CLK/1024
1	1	0	Внешний контакт T1, задний фронт импульса
1	1	1	Внешний контакт T1, передний фронт импульса

12.3. Использование разделяемых регистров

TIMSK – \$39 (\$59)

– Timer/Counter Interrupt Mask Register,

регистр маски прерываний таймеров/счетчиков (8 бит)

Bit	7	6	5	4	3	2	1	0	
	OCIE2	TOIE2	TICIE1	OCIE1A	OCIE1B	TOIE1	OCIE0	TOIE0	TIMSK
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Регистр TIMSK отвечает за прерывания, генерируемые при работе таймеров микроконтроллера. Любое прерывание микроконтроллера может быть разрешено или запрещено. При возникновении разрешенного прерывания ход основной программы прерывается, и происходит обработка данного сигнала. При возникновении запрещенного прерывания ход программы не прерывается, а прерывание игнорируется. Если в биты данного регистра записать 1 в данные биты, прерывание разрешено. Если в эти биты записать 0, прерывания от таймера/счетчика будут запрещены.

Бит TICIE1 (5) разрешает прерывание по захвату, биты OCIE1A (4) и OCIE1B (3) разрешают прерывания при совпадении с A и B, бит TOIE1 (2) разрешает прерывание по переполнению:

- **Bit 5 – TICIE1: Timer/Counter1, Input Capture Interrupt Enable**

Разрешает прерывание по захвату.

- **Bit 4 – OCIE1A: Timer/Counter1, Output Compare A Match Interrupt Enable**

Разрешает прерывание при совпадении содержимого счетчика и содержимого регистра сравнения OCR1A.

- **Bit 3 – OCIE1B: Timer/Counter1, Output Compare B Match Interrupt Enable**

Разрешение прерывание при совпадении содержимого счетчика и содержимого регистра сравнения OCR1B.

- **Bit 2 – TOIE1: Timer/Counter1, Overflow Interrupt Enable**

Разрешает прерывание при переполнении таймера/счетчика 1.

TIFR – \$38 (\$58)

– Timer/Counter Interrupt Flag Register,

регистр флагов прерываний (8 бит)

Bit	7	6	5	4	3	2	1	0	TIFR
	OCF2	TOV2	ICF1	OCF1A	OCF1B	TOV1	OCF0	TOV0	
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Регистр флагов прерываний TIFR показывает, какое прерывание поступило от таймера/счетчика. Биты ICF1 (5), OCF1A (4), OCF1B (3) и TOV1 (2) устанавливаются в 1 в зависимости от того, какое прерывание поступило – захват, совпадение с А, В или переполнение счетчика TCNT1:

- **Bit 5 – ICF1: Input Capture Flag 1 – Флаг 1 захвата входа**

Бит ICF1 устанавливается в случае захвата входа, показывающего, что состояние таймера/счетчика1 переслано в входной регистр захвата ICR1. Бит очищается аппаратно при обработке соответствующего вектора прерывания. Возможна очистка бита записью во флаг логической 1.

- **Bit 4 – OCF1A: Output Compare Flag 1A – Флаг 1A совпадения выхода**

Бит OCF1A устанавливается при совпадении состояния таймера/счетчика1 и содержимого регистра OCR1A (Output Compare Register 1A). Бит OCF1A аппаратно очищается при обработке соответствующего вектора прерывания. Возможна очистка бита записью во флаг логической 1. При установленном бите I в регистре SREG, установленных OCIE1A (Timer/Counter1 Compare Interrupt Enable) и OCF1A выполняется прерывание по совпадению выхода таймера/счетчика1.

- **Bit 3 – OCF1B: Output Compare Flag 1B – Флаг 1B совпадения выхода**

Бит OCF1B устанавливается при совпадении состояния таймера/счетчика1 и содержимого регистра OCR1B (Output Compare Register 1B). Бит OCF1B аппаратно очищается при обработке соответствующего вектора прерывания. Возможна очистка бита записью во флаг логической 1. При установленном бите I в регистре SREG, установленных OCIE1B (Timer/Counter1 Compare Interrupt B Enable) и OCF1B выполняется прерывание по совпадению выхода таймера/счетчика1.

- **Bit 2 – TOV1: Timer /Counter1 Overflow Flag – Флаг переполнения таймера/счетчика1**

Бит TOV1 устанавливается при переполнении таймера/счетчика1. Он аппаратно очищается при обработке соответствующего вектора прерывания. Возможна очистка бита записью во флаг логической 1. При установленном бите I в регистре SREG, установленных TOIE1 (Timer/Counter1 Overflow Interrupt Enable) и TOV1 выполняется прерывание по переполнению таймера/счетчика1. В режиме PWM этот бит устанавливается при смене таймером/счетчиком1 направления счета при \$0000.

Глава 13. Аналого-цифровой преобразователь

13.1. Общее описание АЦП

Аналого-цифровой преобразователь (АЦП, ADC – Analog to Digital Converter) выполняет преобразование электрического напряжения с аналогового датчика в цифровой двоичный код некоторой разрядности. Современные АЦП обеспечивают выход, совместимый с типичными цифровыми микросхемами. Типичным диапазоном входного сигнала являются значения от 0 до 5 В или от 0 до 10 В или от -5 до 5 В. Типичный АЦП имеет аналоговый вход, цифровой выход, вход запуска преобразования информации, после завершения которого формируется признак готовности (ПГ) (рис.13.1 слева). Сигнал на выводе АЦП «запуск» инициирует преобразование, он может быть сформирован с помощью команды вывода out по адресу, присвоенному данному входу. ПГ принимает значение 1, когда преобразование завершено, и может анализироваться процессором или использоваться в качестве запроса прерывания (ЗПр).

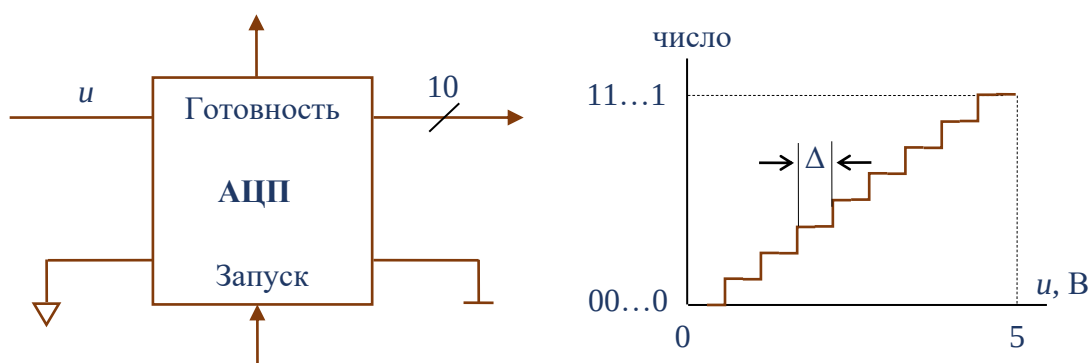


Рисунок 13.1 – Типичный 10-разрядный АЦП и его характеристика преобразования (ХП)

Зависимость выходного двоичного числа АЦП от входного аналогового напряжения u показана на рис.13.1 справа. Она представляет собой кусочно-постоянную возрастающую функцию напряжения, состоящую из 2^N ступеней – **уровней квантования**, где N – разрядность АЦП. Эта зависимость называется **характеристикой преобразования АЦП**.

Разрешение, или **шаг квантования** АЦП есть такое минимальное изменение величины входного аналогового сигнала, которое может быть зафиксировано данным АЦП. Оно равно разности напряжений, соответствующих максимальному и минимальному выходному коду, делённой на количество выходных дискретных значений (уровней квантования). Например, для диапазона входного напряжения 0...5 В и разрядности 10 разрешение есть

$$\Delta = \frac{U_{\max} - U_{\min}}{ADC_{\max}} = \frac{5 \text{ В} - 0 \text{ В}}{2^{10}} = \frac{5 \text{ В}}{1024} = 4,9 \text{ мВ}.$$

13.2. АЦП микроконтроллера AVR

Схема. Микроконтроллер ATmega16 оснащен 10-разрядным АЦП последовательного приближения. Аналоговый вход АЦП образован внешними выводами микросхемы ADC0...ADC7 и AGND (Analog Ground – аналоговая «земля»), соединенными через 8-канальный аналоговый мультиплексор (коммутатор) и дифференциальный усилитель со схемой управления (рис.13.2). На эти выводы подается аналоговое электрическое напряжение, подлежащего преобразованию в двоичное число. Цифровым выходом служит регистр данных ADCH:ADCL, выдающий результат преобразования. Для управления модулем АЦП используются регистры ADCSR (другое название ADCSRA), ADMUX и разделяемый другими модулями ATmega16 регистр SFIOR (п.11.3).

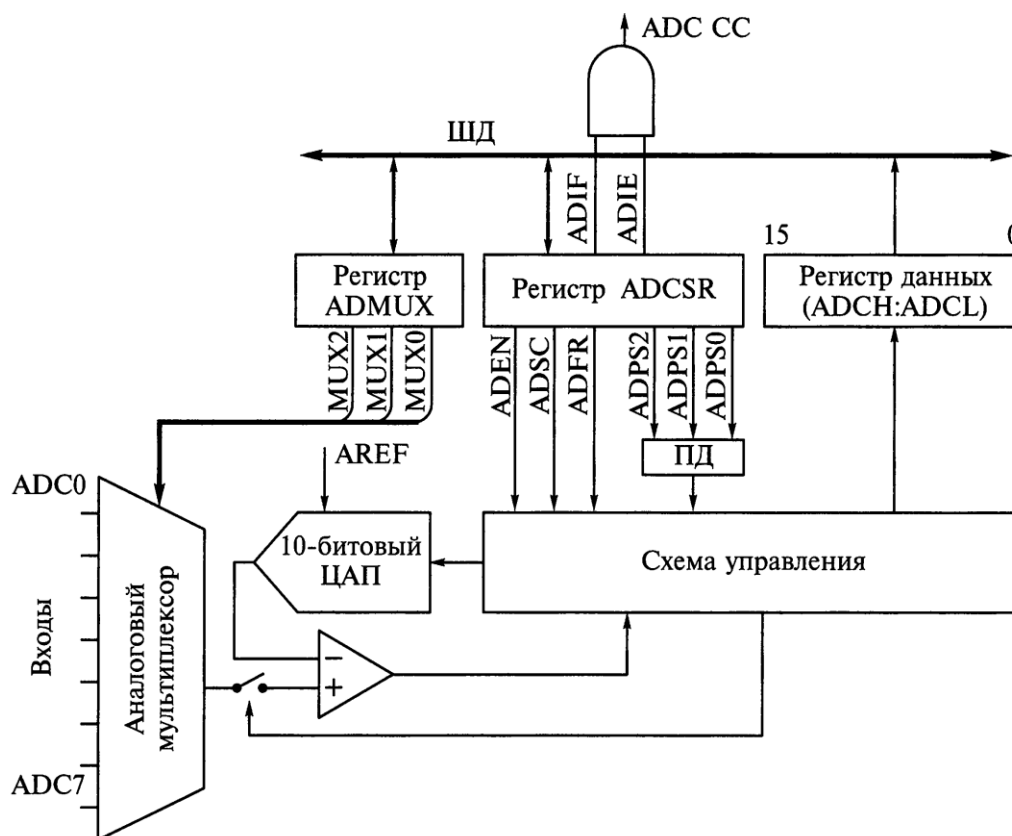


Рисунок 13.2 – Схема аналого-цифрового преобразователя AVR

Входные каналы. Аналого-цифровой преобразователь преобразует напряжение аналогового входного сигнала с одного из выводов ADC0...ADC7 в 10-разрядное цифровое значение (так называемый режим «одиного входа» АЦП, «одиного канала», single channel).

Входы ADC0...ADC7 могут объединяться попарно для формирования каналов с дифференциальным входом, при котором в цифровое значение преобразуется разность напряжений этой пары («дифференциальный вход», «дифференциальный канал», differential channel). [8] В этом случае в некоторых каналах имеется возможность 10- и 200-кратного предварительного усиления входного сигнала. При коэффициентах усиления в 1 раз и 10 раз разрешающая способность АЦП по этим каналам составляет 8 бит, а при коэффициентах усиления в 200 раз – 7 бит.

Параметры. Минимальное значение входного напряжения равно напряжению на контакте AGND – напряжению аналоговой «земли» V_{AGND} , максимальное значение не должно превышать напряжение на контакте AREF – опорное напряжение V_{REF} (reference voltage).

В качестве источника опорного напряжения V_{REF} для АЦП может использоваться как напряжение питания микроконтроллера, так и внутренний или внешний источник опорного напряжения. [8]

Преобразование «аналог-число» во всех случаях производится методом последовательных приближений на основе использования встроенного в модуль цифро-аналогового преобразователя (ЦАП) (см. рис.13.2).

Работа преобразователя выполняется на частоте от 50 до 200 кГц. Для получения этой частоты используется делитель тактовой частоты с заданным коэффициентом деления. Значение коэффициента деления задают с помощью трех разрядов ADPS2, ADPS1, ADPS0 регистра управления ADCSRA (п.13.3).

13.3. Характеристика преобразования

Преобразователь может выдать на выходе фиксированное количество дискретных значений – **уровней квантования**. У 10-битного АЦП ATmega16 оно равно $2^{10}=1024$.

Для **одиночного входа** результат преобразования в виде 10-битного двоичного числа D (с отбрасыванием всех последующих, более младших битов) есть [22, 23]

$$D = \frac{V_{IN}}{V_{REF}} \cdot 2^{10},$$

где $0 \leq V_{IN} < V_{REF}$ – входное напряжение на выбранной ножке (относительно аналоговой «земли»), V_{REF} – опорное напряжение преобразователя.

Минимальный результат равен 0x000 при $V_{IN} = 0$ (более точно: при $0 \leq V_{IN} < \Delta$), максимальный – $2^{10} - 1 = 0x3FF = 0b1111111111$ (при $V_{REF} - \Delta \leq V_{IN} < V_{REF}$). Характеристика преобразования для одиночного входа показана на рис.13.3.

Для **дифференциального входа** (разности двух сигналов на паре ножек)

$$D = \frac{(V_{POS} - V_{NEG}) \cdot GAIN}{V_{REF}} \cdot 2^9,$$

где V_{POS} – напряжение на положительном входе, V_{NEG} – напряжение на отрицательном входе, $GAIN$ – выбранный коэффициент усиления.

Результат преобразования представляется при этом в дополнительном коде, значение которого лежит в диапазоне от 0x200 (-512) до 0x1FF (+511). Характеристика преобразования для этого случая показана на рис.12.4.

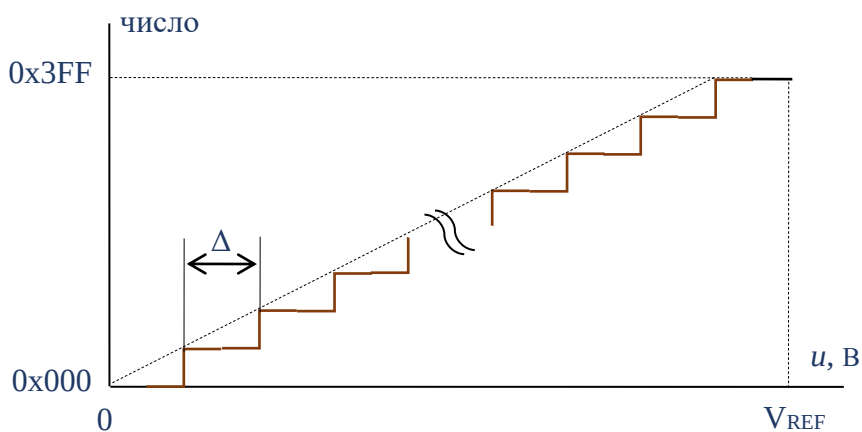


Рисунок 13.3 – Характеристика преобразования АЦП ATmega16 для одиночного входа

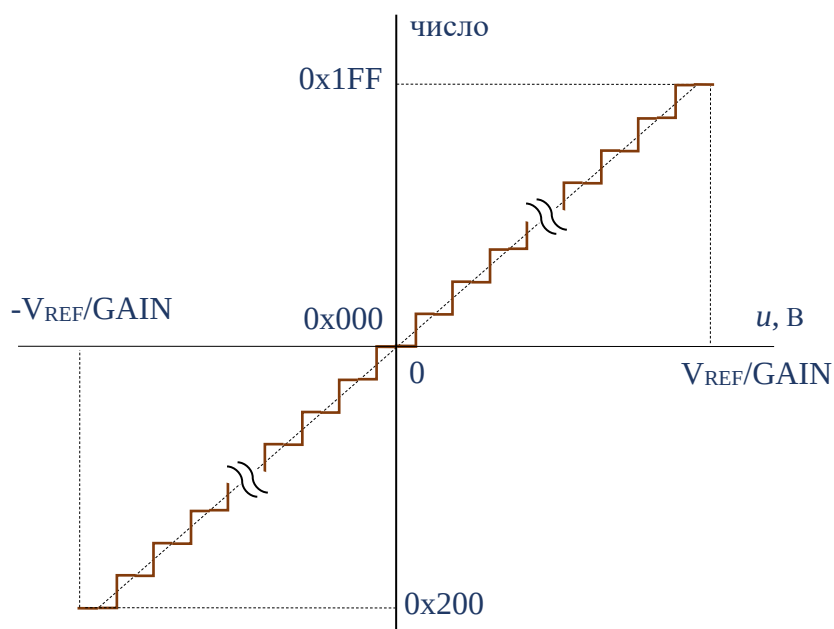


Рисунок 13.4 – Характеристика преобразования АЦП ATmega16 для дифференциального входа

Выбор одиночного и дифференциального входов производится в регистре ADMUX (п.13.3).

Источники опорного напряжения. В качестве источника опорного напряжения преобразователя V_{REF} можно использовать внешний сигнал с вывода AREF, внутренний источник 2,56 В, либо напряжение питания аналоговой части микроконтроллера с вывода AVCC. Напряжение на выводе AVCC не должно отличаться от напряжения питания V_{CC} более, чем на $\pm 0,3$ В. [7]

13.4. Регистры

Для управления аналого-цифровым преобразователем в микроконтроллере ATmega16 используются регистры:

- ADMUX – регистр управления мультиплексором;
- ADCSRA – регистр управления и состояния АЦ преобразователя;
- ADCL и ADCH – регистры данных.

ADMUX – \$07 (\$27)

– ADC Multiplexer Selection Register,
регистр управления мультиплексором

Bit	7	6	5	4	3	2	1	0	
	REFS1	REFS0	ADLAR	MUX4	MUX3	MUX2	MUX1	MUX0	ADMUX
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 7:6 – REFS1:0: Reference Selection Bits**, обеспечивают выбор эталонного напряжения на входе AREF АЦП. Выбор производится в соответствии с табл.13.1. Внутренние источники напряжения не могут быть использованы, если к контакту AREF приложено внешнее напряжение.

Таблица 13.1 – Выбор источника опорного напряжения

REFS1	REFS0	Выбор источника напряжения
0	0	AREF, внутреннее напряжение Vref отключено
0	1	AVCC с внешним конденсатором на контакте AREF
1	0	Резерв
1	1	Внутренний источник 2.56В с внешним конденсатором на AREF

- **Bit 5 – ADLAR: ADC Left Adjust Result**, воздействует на запись результата в регистры ADCL ADCH. При ADLAR=0 можно использовать упрощенное 8-битное преобразование.
- **Bit 4:0 – MUX4:0: Analog Channel and Gain Selection Bits**, коммутируют сигнал на вход преобразователя. Выбор осуществляется в соответствии с табл.13.2.

Таблица 13.2 – Выбор входного канала и коэффициента предварительного усиления GAIN с помощью битов MUX4...MUX0 регистра ADMUX

MUX4:0	Одиночный вход	Дифференциальный вход «POS»	Дифференциальный вход «NEG»	Коэффициент усиления
00000	ADC0			
00001	ADC1			
00010	ADC2			
00011	ADC3			
00100	ADC4			
00101	ADC5			
00110	ADC6			
00111	ADC7			
01000		ADC0	ADC0	10x
01001		ADC1	ADC0	10x
01010		ADC0	ADC0	200x
01011		ADC1	ADC0	200x
01100		ADC2	ADC2	10x
01101		ADC3	ADC2	10x
01110		ADC2	ADC2	200x
01111		ADC3	ADC2	200x
10000		ADC0	ADC1	1x
10001		ADC1	ADC1	1x
10010		ADC2	ADC1	1x
10011		ADC3	ADC1	1x
10100		ADC4	ADC1	1x
10101		ADC5	ADC1	1x
10110		ADC6	ADC1	1x
10111		ADC7	ADC1	1x
11000		ADC0	ADC2	1x
11001		ADC1	ADC2	1x
11010		ADC2	ADC2	1x
11011		ADC3	ADC2	1x
111100		ADC4	ADC2	1x
111101		ADC5	ADC2	1x
11110	1,22B			
11111	0 (GND)			

ADCSR – \$06 (\$26)
– ADC Control and Status Register,
регистр управления и состояния АЦП

Bit	7	6	5	4	3	2	1	0	
	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0	ADCSRA
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 7 – ADEN: ADC Enable**, разрешает работу аналого-цифрового преобразователя.

ADLAR=1

Bit	15	14	13	12	11	10	9	8	
	ADC9	ADC8	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADCH
	ADC1	ADC0	–	–	–	–	–	–	ADCL
	7	6	5	4	3	2	1	0	
Read/Write	R	R	R	R	R	R	R	R	
	R	R	R	R	R	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	

SFIOR – \$30 (\$50)– **Special Function IO Register,****регистр специальных функций ввода-вывода**

Bit	7	6	5	4	3	2	1	0	
	ADTS2	ADTS1	ADTS0	–	ACME	PUD	PSR2	PSR10	SFIOR
Read/Write	R/W	R/W	R/W	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Bit 7:5 – ADTS2:0: ADC Auto Trigger Source, определяют источник, инициирующий запуск АЦП при установленных битах ADEN и ADATE в регистре ADCSR (табл.13.4). Преобразование будет запущено по фронту выбранного флага прерывания.

Bit 4 – Res: Reserved Bit, зарезервирован

Таблица 13.4 – Выбор источника запуска АЦП

ADTS2	ADTS1	ADTS0	Источник запуска
0	0	0	Свободный запуск
0	0	1	Аналоговый компаратор
0	1	0	Внешнее прерывание 0
0	1	1	Таймер/счетчик 0 сравнение
1	0	0	Таймер/счетчик 0 переполнение
1	0	1	Таймер/счетчик 1 сравнение В
1	1	0	Таймер/счетчик 1 переполнение
1	1	1	Таймер/счетчик 1 захват

Порядок работы. Преобразование начинается после установки в 1 разряда ADSC регистра ADCSR. После завершения преобразования устанавливается в 1 бит ADIF регистра ADCSR, используемый для формирования запроса прерывания ADIF при разрешающем значении 1 бита ADIE регистра ADCSR. При переходе к прерывающей программе бит ADIF аппаратно сбрасывается в нулевое состояние. Программно этот бит можно сбросить в 0 путем установки 1 в данный разряд. Аналого-цифровой преобразователь имеет свое собственное прерывание ADC (вектор 0x1C).

Результат преобразования, представляющий собой 10-разрядный двоичный код, размещается в старшей и младшей половинах регистра данных ADCH:ADCL – при ADLAR=0 (бит 5 регистра ADMUX) младшие восемь раз-

рядов располагаются в ADCL, оставшиеся два старших занимают младшие биты ADCH, а при ADLAR=1 наоборот – старшие восемь разрядов в ADCH, два младших в старших битах ADCL (см. формат регистра ADCH:ADCL ниже). Считывание результата выполняется с помощью двух операций ввода. Сначала считывается младший байт ADCL, затем старший ADCH. Такой порядок обеспечивает принадлежность читаемых данных одному и тому же результату преобразования. Если после считывания состояния ADCL, но до считывания ADCH, будет завершено следующее преобразование, ни один из регистров не будет обновлен и записанный ранее результат не будет искажен. Обращение аналого-цифрового преобразователя к регистрам ADCH и ADCL разрешается по завершении считывания содержимого регистра ADCH.

Преобразователь может работать в однократном режиме либо в циклическом. Выбор режима осуществляется с помощью бита ADATE регистра ADCSR: при ADATE = 0 выполняется однократный режим преобразования, при ADATE = 1 – циклический. В обоих случаях преобразование начинается после установки в 1 бита ADSC. В однократном режиме после завершения преобразования для выполнения следующего необходимо снова установить бит ADSC. В циклическом режиме следующее преобразование начинается автоматически после завершения предыдущего и прекращается после сброса бита ADATE (ADFR). В обоих случаях время, затрачиваемое на первое преобразование, увеличивается для инициализации преобразователя.

Для уменьшения помех, вызываемых работой процессора, предусмотрена возможность преобразования с переводом микроконтроллера в режим холостого хода. Для этого биты управления устанавливаются в состояния: ADEN = 1, ADSC = 0, ADATE = 0, ADIE = 1. Далее контроллер переводится в режим холостого хода, при этом запускается АЦП. После выполнения преобразования формируется запрос прерывания, контроллер выходит из режима холостого хода и выполняет прерывающую программу.

В модуле АЦП могут использоваться различные источники опорного напряжения (ИОН). Выбор конкретного ИОН осуществляется посредством битов REFS1:REFS0 регистра ADMUX. Поскольку внутренний ИОН соединяется с выводом AREF, то при его использовании к выводу AREF желательно подключить внешний фильтрующий конденсатор для повышения помехозащищенности. Кроме того, при использовании канала с дифференциальным входом первое измерение после смены источника опорного напряжения следует производить не ранее чем через 125 мкс.

Список литературы

1. Бальзамов А.Ю. Программирование на ассемблере для AVR-микроконтроллеров: Лаб. практикум по основам микропроцессорной техники / А.Ю. Бальзамов. – Саранск: Изд-во Мордовского ун-та, 2012. – 108 с.
2. Баранов В.Н. Применение микроконтроллеров AVR: схемы, алгоритмы, программы / В.Н. Баранов. – М.: Издательский дом «Додека-XXI», 2004. – 288 с.
3. Белов А.В. Микроконтроллеры AVR: от азов программирования до создания практических устройств / А.В. Белов. – СПб.: Наука и техника, 2016. – 544 с.
4. Белов А.В. Разработка устройств на микроконтроллерах AVR: шагаем от чайников до профи / А.В. Белов. – СПб.: Наука и техника, 2013. – 528 с.
5. Брусенцов Н.П. Микрокомпьютеры / Н.П. Брусенцов. – М.: Наука, 1985. – 208 с.
6. Васильев А.С. Основы программирования микроконтроллеров [Электронный ресурс]: учебно-методическое пособие / А.С. Васильев, О.Ю. Лашманов, А.В. Пантюшин. – Электрон. дан. — Санкт-Петербург: НИУ ИТМО, 2016. — 95 с. – Режим доступа: <https://e.lanbook.com/book/91371>.
7. Водовозов А.М. Микроконтроллеры для систем автоматики [Электронный ресурс]: учебное пособие / Водовозов А.М. – Вологда: Инфра-Инженерия, 2016. – 164 с.: ISBN 978-5-9729-0138-8 – Режим доступа: <http://znanium.com/catalog/product/760122>.
8. Воробьева Г.С. Микроконтроллеры семейства AVR. Лабораторный практикум / Г.С. Воробьева, М.В. Журавлев, Д.П. Упадышев, А.М. Штейн. – Томск: Изд-во Томского политехнического университета, 2009. – 90 с.
9. Глазков В.В. Принципы работы микроконтроллеров на основе стенда EasyAVR 6 [Электронный ресурс]: учебное пособие / В.В. Глазков. — Электрон. дан. – Москва: МГТУ им. Н.Э. Баумана, 2017. – 96 с. – Режим доступа: <https://e.lanbook.com/book/103436>.
10. Голубцов М.С. Микроконтроллеры AVR: от простого к сложному / М.С. Голубцов. – М.: СОЛОН-Пресс, 2003. – 288 с. Серия «Библиотека инженера».
11. Григорьев В.Л. Программирование однокристальных микропроцессоров. – М.: Энергоатомиздат, 1987. – 288 с.
12. Григорьев В.Л. Микропроцессор i486. Архитектура и программирование / В.Л. Григорьев. – В 4-х кн. – М.: Гранал, 1993.
13. Зубарев А.А. Ассемблер для микроконтроллеров AVR: Учебное пособие. – Омск: Изд-во СибАДИ, 2007. – 112 с.
14. Кабанов А.А., Осадченко А.Е. Программирование микроконтроллеров AVR. Лабораторный практикум: учебное пособие / А.А. Кабанов, А.Е. Осадченко. – Севастополь: СевГУ, 2017. – 206 с.

15. Каган Б.М. Электронные вычислительные машины и системы: Учеб. пособие для вузов / Б.М. Каган. – М.: Энергоатомиздат, 1991. – 592 с.
16. Козырев В. Г. Архитектура компьютерных систем : учебное пособие / В. Г. Козырев. – Севастополь : СевГУ, 2017. – 252 с. — URL: <https://lib.sevsu.ru/xmlui/handle/123456789/8311>. — Режим доступа: Библиотека Севастопольского государственного университета: [сайт], раздел «Репозиторий eSevSUIR». — Текст : электронный.
17. Кочегаров И.И., Трусов В.А. Микроконтроллеры AVR. Лабораторный практикум: учеб. пособие / И.И. Кочегаров, В.А. Трусов. – Пенза: Изд-во ПГУ, 2012. – 122 с.
18. Коффрон Дж. Расширение микропроцессорных систем / Пер. с англ.; Под ред. П.В. Нестерова / Дж. Коффрон, В. Лонг. – М.: Машиностроение, 1987. – 320 с.
19. Микро-ЭВМ. / Пер. с англ. под ред. А. Дирксена. – М.: Энергоиздат, 1982. – 328 с.
20. Мортон Дж. Микроконтроллеры AVR. Вводный курс. Пер. с англ. / Дж. Мортон. – М.: Издательский дом «Додека-XXI», 2006. – 272 с.
21. Ревич Ю.В. Практическое программирование микроконтроллеров AtmelAVR на языке ассемблера / Ю.В. Ревич. – СПб.: БХВ-Петербург, 2008. – 384 с.
22. Степанов А.Н. Архитектура вычислительных систем и компьютерных сетей: Учебное пособие / А.Н. Степанов. – М. и др.: Питер, 2007. – 509 с.
23. Таненбаум Э. Архитектура компьютера: пер. с англ. / Э. Таненбаум; Ред. А. В. Гордеев. – 6-е изд. – М. и др.: Питер, 2013. – 816 с.
24. Хартов В.Я. Микроконтроллеры AVR. Практикум для начинающих: учеб. пособие. 2-е изд., испр. и доп. / В.Я. Хартов. – М.: Изд-во МГТУ им. Н. Э. Баумана, 2012. – 280 с.
25. 8-bit AVR Microcontroller ATmega16. – Atmel Corporation, 2003. – 348 р.
26. URL: <http://www.atmel.com/products/microcontrollers/avr/default.aspx>

Приложения

А. Арифметические операции над двоичными числами

Целочисленные арифметические операции и команды. Микроконтроллер AVR выполняет только целочисленные операции, а развитые процессоры, например архитектуры IA-32, AMD64 могут выполнять целочисленные операции и операции с плавающей точкой. Для этого в их архитектуре есть два отдельных блока:

устройство для выполнения целочисленных операций;

устройство для выполнения операций с плавающей точкой.

Каждое из этих устройств имеет свою систему команд. В принципе, целочисленное устройство может взять на себя многие функции устройства с плавающей точкой, но это потребует больших вычислительных затрат. Устройство с плавающей точкой мы рассматривать не будем. Поясим лишь, что в случае IA-32/AMD64 оно выполняет арифметические операции над числами, хранящимися в регистрах R0-R7/FPR0-FPR7 в формате, показанном на рис.А.1, а названия всех его команд на языке ассемблера начинаются с буквы F (Floating): fadd, fsub, fmul, fdiv и т.д.

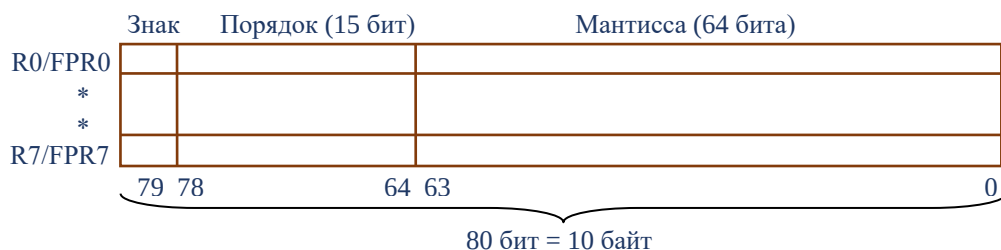


Рисунок А.1 – Форматы двоичных чисел с плавающей запятой

Далее будем обсуждать только целочисленную арифметику.

Целые двоичные числа. Целочисленное устройство (например АЛУ МК AVR) выполняет операции над целыми двоичными числами. *Целое двоичное число* – это целое число, закодированное в двоичной системе счисления. Целое число называют также числом с фиксированной точкой (или запятой) (мысленно помещаемой обычно правее младшего разряда числа). Размерность целого двоичного числа, обрабатываемого одной командой МК AVR составляет 8 бит.

Форматы целых двоичных чисел. Для представления целых чисел в компьютерах применяют натуральный, прямой, обратный и дополнительный коды. Обратный код мы рассматривать не будем ввиду его редкого использования.

Для наглядности и без ограничения общности будем рассматривать коды для случая однобайтной (8-битной) разрядной сетки (рис.А.2).

Определения натурального, прямого и дополнительного кодов.

Натуральный код – это просто двоичная запись целого неотрицательного числа без знака – *натурального числа* на всю ширину разрядной сетки (см. рис.А.2) *.

* В математике натуральными называются числа 1, 2, 3... . В информатике к ним причисляют еще и число 0.

Максимальный диапазон чисел, представимых 8-битным натуральным кодом, составляет $0 \div 2^8 - 1$, или $0 \div 255$.

Прямой код служит для записи чисел со знаком. Для обозначения знака отводят старший разряд (разряд 7 для 8-разрядной сетки; см. рис.А.2). В него заносят 0, если число неотрицательное, и 1 – если отрицательное. В остальные, цифровые, разряды (разряды 6-0) помещают само число – его модуль в двоичном представлении.

Максимальный диапазон чисел в 8-битном прямом коде составляет от $-(2^7 - 1) = -127$ до $2^7 - 1 = 127$.

Форматы целых двоичных чисел

Целое без знака x – так называемый натуральный код числа:

$$x = \begin{array}{|c|c|} \hline 7 & 0 \\ \hline \hline & x \\ \hline \hline \end{array}$$

все разряды 7 – 0 цифровые,
диапазон представления чисел
 $0 \leq x \leq 2^8 - 1 = 255$

Целое со знаком x :

$$x_{\text{код}} = \begin{array}{|c|c|} \hline 7 & 0 \\ \hline \hline \text{Зн} & \text{Цифры} \\ \hline \hline \end{array}$$

разряд 7 знаковый: 0, если $x \geq 0$, и 1, если $x < 0$; разряды 6 – 0 цифровые

Целое со знаком x в прямом коде:

$$x_{\text{пр}} = \begin{cases} \begin{array}{|c|c|} \hline 7 & 0 \\ \hline \hline 0 & x \\ \hline \hline \end{array}, & x \geq 0 \\ \begin{array}{|c|c|} \hline 7 & 0 \\ \hline \hline 1 & |x| \\ \hline \hline \end{array}, & x < 0 \end{cases}$$

$$-127 = -(2^7 - 1) \leq x \leq 2^7 - 1 = 127$$

Целое со знаком x в дополнительном коде:

$$x_{\text{доп}} = \begin{cases} \begin{array}{|c|c|} \hline 7 & 0 \\ \hline \hline 0 & x \\ \hline \hline \end{array} = x_{\text{пр}}, & x \geq 0 \\ \begin{array}{|c|c|} \hline 7 & 0 \\ \hline \hline 1 & 2^7 - |x| \\ \hline \hline \end{array} = \begin{array}{|c|c|} \hline 7 & 0 \\ \hline \hline & 2^8 - |x| \\ \hline \hline \end{array} = \\ = \begin{array}{|c|c|} \hline 7 & 0 \\ \hline \hline & \lceil |x| + 1 \\ \hline \hline \end{array} = \begin{array}{|c|c|} \hline 7 & 0 \\ \hline \hline \bar{\xi} \dots \bar{\xi} 1 [0 \dots 0] \\ \hline \hline \end{array}, & x < 0 \end{cases}$$

Здесь $2^7 - |x|$ – дополнение модуля x до 2^7 ; $2^8 - |x|$ – дополнение модуля x до 2^8 ;
 $\lceil$ (или $\sim$) обозначает поразрядную инверсию числа, например: $\lceil 11010010 = 00101101$; $\bar{\xi} \dots \bar{\xi} 1 [0 \dots 0]$ – побитная инверсия числа $|x| = \xi \dots \xi 1 [0 \dots 0]$, не затрагивающая самой младшей единицы и всех нулей справа от нее (здесь $[0 \dots 0]$ – необязательные нули).

$$-128 = -2^7 \leq x \leq 2^7 - 1 = 127$$

Рис.А.1. Форматы целых двоичных чисел
(на примере 8-битной разрядной сетки)

Дополнительный код (англ. two's complement, иногда twos-complement) – наиболее распространённый способ представления знаковых чисел в компьютерах. Он позволяет заменить операцию вычитания на операцию сложения и сделать операции сложения и вычитания одинаковыми для знаковых и беззнаковых чисел, чем упрощает архитектуру ЭВМ.

Дадим его *определение*. Пусть x – целое число со знаком. Тогда если число неотрицательное: $x \geq 0$, то его дополнительный код $x_{\text{доп}}$ принимается равным прямому коду: в знаковом бите (7) записывается нуль, а в цифровых (6-0) – само число в двоичном представлении:

$$x_{\text{доп}} = x.$$

Такое x лежит в диапазоне $0 \leq x \leq 2^7 - 1 = 127$ (см. рис.А.2). Иными словами, специального дополнительного кода для положительных чисел нет, а в качестве него используется прямой код, т.е. попросту говоря само число, к которому дописывается старший (знаковый) разряд, равный 0 (в 8-битном формате – разряд 7). По этому поводу говорят иногда, что положительные числа записываются не в дополнительном, а в прямом коде. Тем не менее, для единства терминологии мы будем придерживаться нашего определения, т.е. говорить о дополнительном коде как отрицательных, так и положительных чисел, понимая под последним прямой код.

Если же число отрицательное: $x < 0$, то его дополнительный 8-битный код $x_{\text{доп}}$ содержит в знаковом бите (бите 7) единицу, а в цифровых битах (6-0) – дополнение модуля числа $|x|$ до 2^7 , т.е. разность $2^7 - |x|$:

$$x_{\text{доп}} = 1_{-}(2^7 - |x|), \quad (\text{A.1})$$

причем последняя обязана оставаться неотрицательной: $2^7 - |x| \geq 0$ (см. рис.А.2). Выражение $1_{-}(2^7 - |x|)$ обозначает конкатенацию 1 (единицы) и 7-битного двоичного числа $2^7 - |x|$, т.е. 8-битное число, образованное путем сочленения цифровых записей этих чисел в одно число.

Конкатенация (англ. concatenate – сцеплять) – операция сочленения, склеивания объектов линейной структуры, обычно строк, добавление одной строки в конец другой. Например, конкатенация слов «микро» и «мир» дает слово «микромир», конкатенация чисел 5 и 2 дает число 52.

Поскольку в (1) $2^7 - |x| \geq 0$, или $|x| \leq 2^7$, то закодированное по формуле (1) отрицательное число x находится в диапазоне $-128 = -2^7 \leq x < 0$.

Знаковую единицу (в бите 7, формула (1)) можно трактовать как число 2^7 . Поэтому 8-битный дополнительный код отрицательного числа можно получить также вычитанием модуля числа из $2^7 + 2^7 = 2^8$ (дополнением до 2^8):

$$x_{\text{доп}} = 2^8 - |x| = \text{ДОП}(|x|), \quad (\text{A.2})$$

где по-прежнему $-2^7 \leq x < 0$, а выражение $\text{ДОП}(|x|)$ обозначает дополнение модуля числа до 2^8 , т.е. разность $2^8 - |x|$.

Соответственно, весь диапазон чисел, представимых 8-битным дополнительным кодом, составляет от $-2^7 = -128$ до $2^7 - 1 = 127$.

В общем случае, когда разрядная сетка состоит из n двоичных разрядов, всюду в приведенных здесь и ниже выражениях 8 разрядов надо заменить на n разрядов, а 7 на $n - 1$. Числа будут тогда лежать в диапазоне от -2^{n-1} до $2^{n-1} - 1$.

Итак, n -битный дополнительный код положительного (неотрицательного) числа совпадает с прямым кодом, а n -битный дополнительный код отрицательного числа можно получить вычитанием модуля числа из 2^n : так называемое «дополнение до 2^n » (отсюда название этого кода: «дополнительный»), или, что эквивалентно, вычитанием модуля из нуля с заемом единицы из несуществующего, воображаемого, $n + 1$ -го разряда (разряда номер n).

Получим более простые записи для формул (1) и (2), не требующие вычитания из 2^n . Начнем с определения (дефиниции) дополнения.

Определение дополнения. Пусть s – некоторый 8-разрядный двоичный код. Тогда дополнение s до 2^8 по определению есть 8-битная разность

$$\text{ДОП}(s) = 2^8 - s, \quad (\text{A.3})$$

где код s интерпретируется как натуральное число (число без знака).

При этом дополнение нуля ($s = 0$) равно нулю:

$$\text{ДОП}(0) = 0,$$

так как $2^8 - 0 = 1^{\circ}0000^{\circ}0000$, но после вычисления 9-й по счету бит (бит d8), содержащий 1, отбрасывается (сетка 8-битная!). В результате $\text{ДОП}(0) = 0000^{\circ}0000$.

Простые способы вычисления дополнения (теоремы).

Способ 1. Дополнение легко вычисляется путем побитной инверсии кода с последующим прибавлением 1:

$$\text{ДОП}(s) = 2^8 - s = \overline{}s + 1, \quad (\text{A.4})$$

где символ $\overline{}$ (или $\sim$ или $^-$ – надчеркивание, черта сверху) обозначает инверсию всех битов числа s . Например, если $s = 1001^{\circ}1000$, то $\overline{}s = \overline{}1001^{\circ}1000 = 0110^{\circ}0111$.

$$\begin{aligned} \text{Докажем (4): } 2^8 - s &= 2^8 - 1 - s + 1 = 11111111 - s_7s_6s_5s_4s_3s_2s_1s_0 + 1 = \\ &= \overline{s_7}\overline{s_6}\overline{s_5}\overline{s_4}\overline{s_3}\overline{s_2}\overline{s_1}\overline{s_0} + 1 = \overline{}s + 1 \end{aligned}$$

(здесь $s_7s_6s_5s_4s_3s_2s_1s_0$ – развернутая двоичная запись кода s , а $\overline{s_i}$ – инверсия бита s_i).

Отдельно для $s = 0$ по формуле (4) имеем: $\overline{}0000^{\circ}0000 + 1 = 1111^{\circ}1111 + 1 = 1^{\circ}0000^{\circ}0000$, и $\text{ДОП}(0) = 0000^{\circ}0000$, так как единица из 9-го бита отбрасывается.

Формула (4) доказана.

Способ 2 (еще один простой способ вычисления дополнения). Из (4) вытекает, что поскольку любое ненулевое число ($s \neq 0$) всегда имеет один из двух видов $s = \sigma \dots \sigma 10 \dots 0$ или $s = \sigma \dots \sigma 1$, то его дополнение можно найти так:

$$\text{ДОП}(s) = \overline{\sigma} \dots \overline{\sigma} 10 \dots 0 \text{ или } \text{ДОП}(s) = \overline{\sigma} \dots \overline{\sigma} 1, \quad (\text{A.5})$$

где $\overline{\sigma} = \overline{}\sigma$ – инверсия бита σ .

$$\text{В самом деле, } \text{ДОП}(s) = \overline{}\sigma \dots \sigma 10 \dots 0 + 1 = \overline{\sigma} \dots \overline{\sigma} 01 \dots 1 + 1 = \overline{\sigma} \dots \overline{\sigma} 10 \dots 0,$$

т.е. дополнение можно получить путем инверсии всех битов кода, находящихся слева от самой младшей единицы.

Из (5) опять-таки имеем: $\text{ДОП}(0) = 0$, так как инвертировать (слева от единицы) нечего (поскольку нет самой единицы).

Мы показали, что согласно формулам (3)-(5) дополнение 8-битного кода s до 2^8 можно получить любым из 3-х следующих способов:

$$\text{ДОП}(s) = 2^8 - s = \overline{}s + 1 = \overline{\sigma} \dots \overline{\sigma} 1[0 \dots 0], \quad (\text{A.6})$$

где $s = \sigma \dots \sigma 1[0 \dots 0]$, $[0 \dots 0]$ – необязательные нули,

т.е., дополнение можно найти,

вычитая код s из 2^8 ;

или инвертируя код побитно с последующим прибавлением 1;

или инвертируя все биты кода, расположенные слева от младшей единицы.

Согласно формулам (1), (2) и (6) получаем

четыре эквивалентных определения дополнительного кода отрицательного числа $-2^7 \leq x < 0$ (см. рис.А.2):

$$x_{\text{доп}} = 1_-(2^7 - |x|) = 2^8 - |x| = \overline{}|x| + 1 = \overline{\xi} \dots \overline{\xi}[10 \dots 0], \quad (\text{A.7})$$

где $|x| = \xi \dots \xi[10 \dots 0]$.

Итак, для нахождения дополнительного кода отрицательного числа надо:

в знаковый бит (7) записать 1, а в цифровые (6-0) – разность $2^7 - |x|$;

или во все восемь бит (7-0) записать разность $2^8 - |x|$;

или инвертировать (8-битный) модуль числа побитно и прибавить единицу;

или инвертировать все биты (8-битного) модуля числа, находящиеся слева от младшей единицы.

Таблица представлений знаковых чисел в прямом и дополнительном кодах

Десятичное представление	Двоичное представление (8 бит)	
	прямой	дополнительный
127	01111111	01111111
* * *		
5	00000101	00000101
4	00000100	00000100
3	00000011	00000011
2	00000010	00000010
1	00000001	00000001
0	00000000	00000000
-0	10000000	- - -
-1	10000001	11111111
-2	10000010	11111110
-3	10000011	11111101
-4	10000100	11111100
-5	10000101	11111011
-6	10000110	11111010
* * *		
-125	11111101	10000011
-126	11111110	10000010
-127	11111111	10000001
-128	- - -	10000000

Операция расширения знака (легко следует из определения дополнительного кода). При необходимости увеличения разрядности числа (например, с 8 до 16 разрядов) выполняется операция расширения знака. *Расширение знака* (англ. Sign extension) – это операция над двоичным числом, которая позволяет увеличить разрядность числа с сохранением знака и значения. Выполняется добавлением цифр со стороны старшего значащего разряда. Применительно к дополнительному коду, добавляются нули, если число положительное (старший разряд равен 0), и единицы, если число отрицательное (старший разряд равен 1).

Пример

Десятичное число	Двоичное число в дополнительном коде (8 разрядов)	Двоичное число в дополнительном коде (16 разрядов)
10	0000 1010	0000 0000 0000 1010
-15	1111 0001	1111 1111 1111 0001

Арифметические операции над целыми двоичными числами. Все целочисленные операции (команды) сложения и вычитания выполняются любым процессором над двоичными операндами как над целыми числами *без знака* (натуральными числами). Это означает, что процессор обрабатывает все разряды операндов единообразно – знаковый (старший) разряд как остальные, цифровые разряды, т.е. трактует этот разряд как цифровой, а не знаковый.

Указанное обстоятельство затрудняет сложение и вычитание чисел со знаком в прямом коде. Однако оно органично вписывается в природу *дополнительного кода*. Последний обладает двумя полезными свойствами, сводящими знаковую арифметику к натуральной, а вычитание из нуля к инверсии. Эти свойства как раз и побудили разработчиков использовать до-

полнительные коды для записи знаковых чисел и ограничиться, таким образом, аппаратной реализацией лишь натуральной арифметики в компьютерах и к тому же без применения вычитателей. Тем самым было достигнуто значительное упрощение схем арифметико-логических устройств.

Для описания упомянутых свойств дополнительного кода нам потребуется предварительно вспомнить понятие арифметического переполнения (данного в п.5.3).

Определение: арифметическое переполнение. Пусть x и y – целые знаковые числа, лежащие в диапазоне от -2^{n-1} до $2^{n-1} - 1$, где n – длина разрядной сетки в битах, отводимой для записи чисел. Тогда *арифметическим переполнением разрядной сетки* (или просто «арифметическим переполнением», «переполнением») называется выход суммы $(x + y)$ или разности $(x - y)$ этих чисел за пределы указанного диапазона. Поскольку указанный диапазон соответствует представлению чисел в дополнительном коде, данное событие иногда называют *переполнением дополнительного кода*.

У МК AVR $n = 8$. Поэтому арифметическое переполнение в AVR означает выход результата за диапазон $-2^7 \dots 2^7 - 1$.

Свойства дополнительного кода (теоремы).

Свойство 1: сложение и вычитание чисел в дополнительном коде.

Сложение. Пусть x и y – целые числа со знаком, лежащие в диапазоне от -2^7 до $2^7 - 1$, а $x_{\text{доп}}$ и $y_{\text{доп}}$ – их 8-битные дополнительные коды. Тогда при условии отсутствия арифметического переполнения

$$(x + y)_{\text{доп}} = x_{\text{доп}} + y_{\text{доп}}, \quad (\text{A.8})$$

где $(x + y)_{\text{доп}}$ – 8-битный дополнительный код суммы чисел; $x_{\text{доп}} + y_{\text{доп}}$ – 8-битная сумма дополнительных кодов, складываемых как натуральные числа, а возможный перенос из старшего, 8-го бита (бита номер 7) в несуществующий 9-й бит (бит 8), если он возник, отбрасывается.

Если же арифметическое переполнение имеет место, то равенство (8) не верно. В этом случае для записи дополнительного кода суммы $(x + y)_{\text{доп}}$ требуется брать 9 бит суммы $x_{\text{доп}} + y_{\text{доп}}$. Ее старший бит – бит 8 будет знаковым, остальные биты 7-0 – цифровыми. Другими словами, возможный перенос из бита 7 в бит 8 в этом случае не игнорируется, сохраняется в бите 8.

Коротко, сумма двух 8-битных кодов в общем случае 9-битная:

$$d8 \ d7 \dots d0 \leftarrow x_{\text{доп}} + y_{\text{доп}}.$$

В отсутствие переполнения

$$(x + y)_{\text{доп}} = d7 \dots d0,$$

в присутствии переполнения

$$(x + y)_{\text{доп}} = d8 \ d7 \dots d0.$$

Итак, если нет переполнения, надо пользоваться 8-битным результатом (а не 9-битным, который может быть ошибочным), если переполнение есть – 9-битным (а не 8-битным, ошибочным).

Вычитание. Аналогичное свойство справедливо и для разности двух целых знаковых чисел x и y . А именно, в отсутствие переполнения

$$(x - y)_{\text{доп}} = x_{\text{доп}} - y_{\text{доп}}, \quad (\text{A.9})$$

где $(x - y)_{\text{доп}}$ – 8-битный дополнительный код разности чисел; $x_{\text{доп}} - y_{\text{доп}}$ – 8-битная разность дополнительных кодов, вычитаемых как натуральные числа. Если $x_{\text{доп}} < y_{\text{доп}}$, вычисление разности $x_{\text{доп}} - y_{\text{доп}}$ требует заема единицы («1») из несуществующего 9-го бита (би-

та 8), т.е. фактически вычисляется разность $2^8 + x_{\text{доп}} - y = 1_x_{\text{доп}} - y_{\text{доп}}$, где $1_x_{\text{доп}}$ – конкатенация (сочленение) 1 и $x_{\text{доп}}$, и в этой разности берутся только разряды 7-0, а разряд 8 игнорируется, отбрасывается.

При наличии переполнения формула (9) не справедлива. В этом случае дополнительный код разности $(x - y)_{\text{доп}}$ должен быть 9-битным, и он получается приписыванием слева к 8-битной разности $x_{\text{доп}} - y_{\text{доп}}$ нуля («0»), если заем из бита 8 был, и единицы («1») – если не был. В обоих случаях правильный 9-битный дополнительный код разности получается вычитанием $1_x_{\text{доп}} - y_{\text{доп}}$.

Коротко. Вычитание проводится по формуле

$$d8\ d7 \dots d0 \leftarrow 1_x_{\text{доп}} - y_{\text{доп}}, \text{ где } 1_x_{\text{доп}} - \text{конкатенация } 1 \text{ и } x_{\text{доп}}.$$

Тогда в отсутствие переполнения

$$(x - y)_{\text{доп}} = d7 \dots d0,$$

в присутствии переполнения

$$(x - y)_{\text{доп}} = d8\ d7 \dots d0.$$

Таким образом, если переполнения нет, в качестве дополнительного кода разности берем 8-битный результат (а не 9-битный, который может оказаться ошибочным), если переполнение есть – 9-битный (а не 8-битный – ошибочный).

Резюмируя, получаем:

Сумма/разность двух дополнительных кодов целых чисел со знаком (складываемых/вычитаемых как натуральные числа в пределах принятой разрядной сетки) равна правильному дополнительному коду их суммы/разности при условии, что не произошло арифметического переполнения.

Если же переполнение имело место, то правильный дополнительный код суммы/разности представляется суммой/разностью дополнительных кодов, расширенной битом переноса/инверсии заема.

Представленное свойство позволяет вместо того, чтобы складывать/вычитать знаковые числа, складывать/вычитать их дополнительные коды как натуральные числа, числа без знака.

Заметим, что применительно к операциям умножения и деления аналогичное свойство дополнительных кодов не имеет места.

Второе полезное свойство дополнительного кода.

Свойство 2: изменение знака числа в дополнительном коде. Пусть x – целое число со знаком, находящееся в диапазоне $-2^7 \leq x \leq 2^7 - 1$, а $x_{\text{доп}}$ – его 8-битный дополнительный код. Тогда в отсутствие арифметического переполнения

$$(-x)_{\text{доп}} = \text{ДОП}(x_{\text{доп}}), \quad (\text{A.10})$$

где $\text{ДОП}(x_{\text{доп}})$ – 8-битное дополнение $x_{\text{доп}}$ до 2^8 ,

т.е. *дополнительный код числа $-x$ («минус» x) равен дополнению дополнительного кода $x_{\text{доп}}$ самого числа x при условии, что нет арифметического переполнения; или, иначе: для изменения знака числа в дополнительном коде надо взять дополнение его дополнительного кода.*

Арифметическое переполнение, т.е. неправильный результат, наступает здесь в единственном случае $x = -128$, так как у противоположного числа $-x = +128$ нет 8-битного дополнительного кода. В этом случае правильный, 9-битный, код получается расширением 8-битного дополнения $\text{ДОП}(x_{\text{доп}})$ нулевым значением бита 8:

$$\text{при } x = -128 \quad (-x)_{\text{доп}} = 0_ \text{ДОП}(x_{\text{доп}}) = 0^\circ 1000^\circ 0000. \quad (\text{A.11})$$

Для $x = 0$ имеем: $x_{\text{доп}} = 0$, $(-x)_{\text{доп}} = (-0)_{\text{доп}} = \text{ДОП}(x_{\text{доп}}) = \text{ДОП}(0) = 2^8 - 0 = 2^8 = 1^{\circ}0000^{\circ}0000$, и далее надо брать в этом результате опять-таки только 8 младших разрядов, отбрасывая 9-й разряд, содержащий единицу: $(-0)_{\text{доп}} = 0000^{\circ}0000$.

Для вычисления дополнения $\text{ДОП}(x_{\text{доп}})$, а значит, для изменения знака числа, можно воспользоваться формулами (6). В результате получаем, что $(-x)_{\text{доп}}$ (за исключением $x = -128$) определяется любым из 3-х следующих способов:

$$(-x)_{\text{доп}} = \text{ДОП}(x_{\text{доп}}) = 2^8 - x_{\text{доп}} = \bar{1}x_{\text{доп}} + 1 = \bar{\xi} \dots \bar{\xi} 1 [0 \dots 0], \quad (\text{A.12})$$

где $x_{\text{доп}} = \xi \dots \xi 1 [0 \dots 0]$.

Итак, для изменения знака числа, записанного в дополнительном коде, надо:

вычесть этот дополнительный код из 2^8 ;

или инвертировать этот код побитно и прибавить единицу;

или инвертировать все биты этого кода, находящиеся слева от младшей единицы,

за исключением числа $x = -2^7 = -128$, дающего переполнение в этой операции. В этом случае противоположное к -128 число, т.е. (знаковое) число $+128$ не помещается в 8-битную разрядную сетку. Оно представляется 9-битным дополнительным кодом (11).

Следствие из свойств 1 и 2: замена операции вычитания на операцию сложения. Свойства 1 и 2 дополнительного кода *позволяют производить вычитание двух чисел путем сложения уменьшаемого с дополнением вычитаемого*:

$$x - y = x + (-y) = x + \text{ДОП}(y),$$

поскольку взятие дополнения эквивалентно изменению знака числа.

Приведем строгую формулировку данного свойства. Пусть x и y – целые числа со знаком, а $x_{\text{доп}}$ и $y_{\text{доп}}$ – их дополнительные коды. Тогда согласно свойствам 1 и 2 в отсутствие арифметического переполнения справедливы следующие соотношения

$$(x - y)_{\text{доп}} = (x + (-y))_{\text{доп}} = x_{\text{доп}} + (-y)_{\text{доп}} = x_{\text{доп}} + \text{ДОП}(y_{\text{доп}}),$$

или, коротко,

$$(x - y)_{\text{доп}} = x_{\text{доп}} + \text{ДОП}(y_{\text{доп}}) \quad (\text{A.13})$$

– дополнительный код разности двух чисел равен сумме дополнительного кода уменьшаемого и дополнения дополнительного кода вычитаемого.

Такая операция дает правильный ответ – разность двух знаковых чисел x и y в 8-битном дополнительном коде, если не произошло арифметического переполнения.

Если же переполнение произошло, то правильный дополнительный код суммы является 9-битным, равным сумме кодов $x_{\text{доп}} + \text{ДОП}(y_{\text{доп}})$, в которой надо учесть бит номер 8.

Коротко:

$$d8 \, d7 \dots d0 \leftarrow x_{\text{доп}} + \text{ДОП}(y_{\text{доп}}).$$

В отсутствие переполнения

$$(x - y)_{\text{доп}} = d7 \dots d0,$$

в присутствии переполнения

$$(x - y)_{\text{доп}} = d8 \, d7 \dots d0.$$

Благодаря возможности заменить вычитание сложением в процессорах не делают устройств - вычитателей, а ограничиваются сумматорами и схемами побитной инверсии. А поскольку, к тому же, знаковые разряды дополнительных кодов обрабатываются подобно обычным, цифровым, достаточно иметь сумматоры только для натуральных чисел. Эти два обстоятельства существенно упрощают аппаратную реализацию арифметических блоков цифровых приборов.

Итак, при выполнении арифметических операций сложения и вычитания чисел со знаком надо просто складывать и вычитать их дополнительные коды, включая значения знако-

вых разрядов, и результат всегда получится правильным – в виде правильного дополнительного кода суммы или разности чисел со знаком, если не было переполнения.

Напомним, что факт арифметического переполнения фиксируется процессором с помощью **флажка OF** (Overflow Flag) для процессоров IA-32 и **флажка V** (oVerflow flag) для микроконтроллеров AVR. Этот флажок устанавливается в «1», если арифметическое переполнение произошло, и сбрасывается в «0», если переполнения не было (п.5.3).

Критерий арифметического переполнения (теорема). Можно доказать (доказательство мы не приводим) следующий **критерий арифметического переполнения**, удобный для аппаратной реализации. При сложении двух чисел переполнение (а значит, установка OF/V в «1») происходит, когда переносы в старший, знаковый разряд – разряд $n-1$ ($n = 8, 16, 32$) и из него (т.е. в разряд n) не совпадают. В противном случае, когда они совпадают, т.е. имеются оба переноса или нет ни того, ни другого, переполнения нет (производится сброс OF/V в «0»). Говоря условно, если число указанных переносов (в и из знакового бита) нечетное, то переполнение есть (OF/V=1), а если четное – нет (OF/V=0).

Иными словами, значение флага OF/V при сложении в процессоре определяется логическим выражением $OF / V = C_{n-1} \oplus C_n$, где C_{n-1} – перенос в знаковый разряд, C_n – перенос из знакового разряда.

При вычитании критерий аналогичен, но надо считать не переносы, а заемы в и из знакового разряда.

В процессоре, как мы указали, нет вычитателя, а вычитание производится путем сложения уменьшаемого с дополнением вычитаемого по формуле (13). В такого рода операции надо опять-таки следить за переносами (в и из знакового бита), а не заемами (которых в данном случае просто нет). При этом переносы надо фиксировать в обеих фазах операции: как при суммировании собственно по формуле (13), так и при прибавлении «1» во время вычисления дополнения $ДОП(u_{доп})$ по формуле (6). Если в этих двух суммированиях общее число переносов нечетное, то переполнение есть, а если четное – нет. Поясним, что общее число переносов здесь может быть 0, 1 или 2. Действительно, перенос в (6) будет только когда $u = -128$ ($u_{доп} = 1000\ 0000$). После этого в (13) может возникнуть еще только один перенос (из d7). И всего в обоих суммированиях не более 2-х переносов. Если же $u \neq -128$, то возможны только переносы в (13). Их снова будет 0, 1 или 2.

Указанный критерий встроен в электронную схему процессора, обеспечивая правильную автоматическую реакцию флажка OF/V.

Б. Обмен информацией в компьютере

Б.1. Структура аппаратуры и потоки информации компьютера

На рис.Б.1 представлена **структурная схема и потоки информации вычислительной системы**. [19] Как видно из рисунка, для ввода данных в компьютер могут понадобиться обычные периферийные устройства: диск, клавиатура, флэш-карты и пр. Данные от процесса или объекта управления могут быть введены непосредственно. Если данные имеют аналоговую форму, например, для управления температурой на промышленном предприятии, они сначала должны быть преобразованы в цифровую форму, так как компьютер может работать только по двоичной системе. Это преобразование осуществляется с помощью аналого-цифрового преобразователя (АЦП). И обратно, данные для управления процессом, полученные на выходе компьютера, могут быть преобразованы в аналоговую форму с помощью цифро-аналогового преобразователя (ЦАП).

Б.2. Базовые машинные операции

Компоненты программного процесса. Работа процессора (Пр, центрального процессора, ЦП) заключается в выполнении *машинной программы*. Процесс выполнения любой программы состоит из последовательного выполнения отдельных ее команд. Обработку процессором одной команды, а также интервал времени, требующийся для этого, называют *командным циклом, циклом команды*. [16]

Цикл любой команды сводится к одному или нескольким обращениям процессора к устройствам шины – памяти и «периферии». Обращаться к шине процессору требуется, например, для чтения команд, а также для чтения, записи, ввода или вывода данных выполняемой команды. Всякое такое обращение есть один *машинный цикл (цикл шины, цикл магистралей)*. При этом операции, следующие сразу после обращения процессора к шине и до нового такого обращения, происходящие внутри процессора, также входят в состав данного машинного цикла. К ним относятся декодирование считанной команды, сложение операндов, сдвиг операнда и др. Машинный цикл содержит, следовательно, одну *внешнюю пересылку*, к которой могут добавляться *внутренние операции* процессора по расшифровке команды и обработке данных. В зависимости от типа и формата команды, способов адресации и числа операндов, число машинных циклов в одном командном цикле может быть различным – от одного до нескольких. Так же и в одном машинном цикле может выполняться различное число операций внутри процессора над байтами/словами (чаще одна-две).

Процессор – динамическое устройство. Для того чтобы он функционировал, на него должны поступать побуждающие толчки – импульсы электрического напряжения от специального устройства – тактового генератора (п.2.1). С помощью импульсов, проходящих через все блоки процессора, генератор подталкивает их на выполнение очередных операций над данными и синхронизирует ход операций по отдельным блокам. Период следования тактовых импульсов (промежуток между началами двух соседних импульсов) называют *машинным тактом*. Длительность машинных тактов всегда одинакова. Выполнение различных операций – внешних пересылок и внутренних операций процессора может занимать от долей такта до многих тактов в зависимости от команды и процессора. Общая тенденция заключается в уменьшении количества тактов, затрачиваемых на выполнение операций.

Таким образом, просматривается иерархия процедур при работе микропроцессора:

Машинная программа ® *Командный цикл* ® *Машинный цикл* ® *Машинная операция* ® *Машинный такт*.

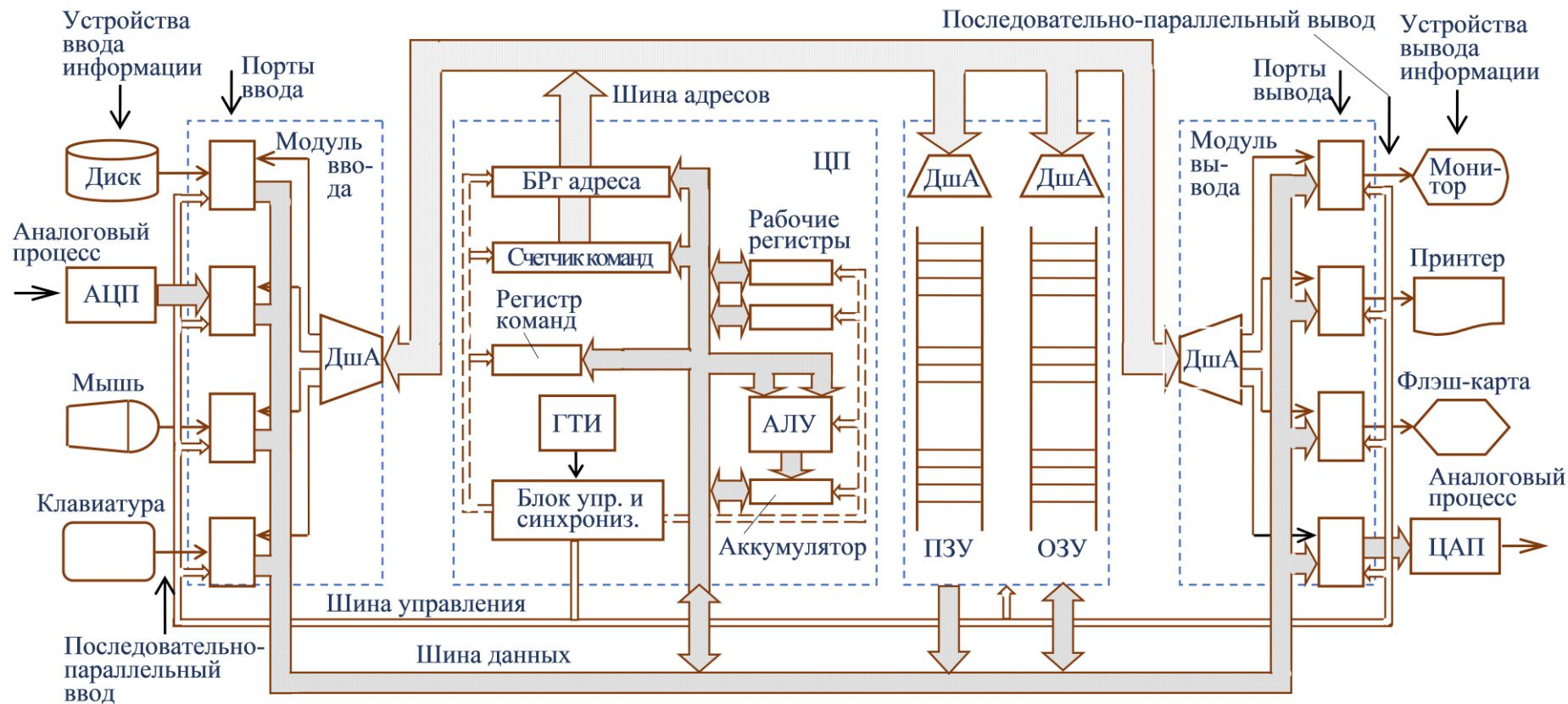


Рисунок Б.1 – Поток информации в вычислительной системе.

ЦП – центральный процессор; АЛУ – арифметико-логическое устройство;
 ДшА – дешифратор (селектор) адреса; БРг – буферный регистр;
 АЦП и ЦАП – аналого-цифровой и цифро-аналоговый преобразователь

Семь типов машинных операций. ЦП работает в составе компьютерной системы и постоянно обменивается информацией с памятью и периферийными устройствами. Для обмена он использует системную магистраль. Знание принципов обмена информацией по системной магистрали необходимо для успешного решения задач, возникающих при организации интерфейса разрабатываемой вычислительной системы.

Основными шинами системной магистрали, используемыми при описании передачи цифровых сигналов в ЭВМ, являются шины данных, адресов и управления (см. рис.2.2, 2.9). Модель системы с тремя шинами позволяет описать все *семь типов базовых машинных операций вычислительной системы* [18]:

1) **Чтение** центральным процессором одной порции данных из памяти. В этой операции процессор извлекает порцию данных из ячейки основной памяти. Порция данных пересылается под управлением процессора по шине данных и заносится в один из его регистров:

Регистр процессора (Рг) $\leftarrow$ ШД $\leftarrow$ ячейка памяти (ЯП).

2) **Запись** процессором одной порции данных («данного») в память. В этой операции данное передается из процессора в основную память:

Рг $\rightarrow$ ШД $\rightarrow$ ЯП.

3) **Ввод** процессором данного из периферийного устройства (устройства ввода) – из его регистра («порта»), которое пересылается из порта по шине данных и заносится в регистр процессора:

Рг $\leftarrow$ ШД $\leftarrow$ порт ввода.

4) **Вывод** данного из регистра процессора в ПУ (устройство вывода) – в его порт вывода:

Рг $\rightarrow$ ШД $\rightarrow$ порт вывода.

5) **Подтверждение прерывания** (или просто «прерывание») – ввод процессором номера прерывания от периферийного устройства.

Эта операция осуществляется не по указке машинной программы, а под влиянием внешнего сигнала прерывания ЗПр (запроса прерывания, INTR – Interruption). Сигнал ЗПр поступает на процессор от периферийного устройства (обычно – через контроллер прерываний) и вызывает ответные автоматические действия процессора, состоящие из операций «подтверждение прерывания», «запись в стек» и «чтение из памяти».

В операции «подтверждение прерывания» процессор вводит из контроллера прерываний специальный двоичный код (номер, тип) прерывания, идентифицирующий устройство – источник запроса:

Пр $\leftarrow$ ШД $\leftarrow$ номер прерывания n

и затем вычисляет по этому номеру n адрес ячейки памяти, в которой хранится так называемый вектор прерывания (ВП) – адрес перехода на подпрограмму, обслуживающую данное устройство, – подпрограмму прерывания (ПП) (см. п.8.2, 8.3). На этом подтверждение прерывания (операция 5) заканчивается.

Вслед за циклом «прерывание» (операцией 5) процессор (также автоматически) выполняет операции типа 2 – запись и типа 1 – чтение:

запоминает в памяти (в стеке) адрес возврата из подпрограммы прерывания (ПП):

Пр: адрес возврата из ПП $\rightarrow$ ШД $\rightarrow$ стек

и считывает из памяти, из ячейки, хранящей вектор прерывания, (адрес которой он определил в цикле «прерывание») сам вектор – адрес перехода на ПП:

Пр $\leftarrow$ ШД $\leftarrow$ адрес перехода на ПП (ВП).

По окончании указанных действий (прерывания, записи и чтения) процессор переходит на ПП по вектору прерывания. Режим прерываний описывается в главе 8.

6) **Прямой доступ** периферийного устройства (ПУ) к основной памяти компьютера (ПДП, DMA – Direct Memory Access). Прямой доступ к памяти заключается в пересылке данных между ПУ и памятью напрямую, минуя центральный процессор. Этот режим осуществляется под управлением контроллера ПДП (DMA Controller). Процессор на время ПДП вообще отключается от системной магистрали и не принимает никакого участия в пересылке данных,

для чего контроллер ПДП подает на вход процессора сигнал захвата HLD (Зх), отсоединяющий внутренние блоки процессора от захваченных внешних шин.

В микроконтроллерах прямой доступ к памяти отсутствует.

7) Все **внутренние операции** процессора по обработке и преобразованию данных, происходящие на его собственных, внутренних узлах:

- а) дешифрация команд, поступающих на процессор из памяти;
- б) передача операндов из регистра на регистр процессора (внутренние, межрегистровые пересылки);
- в) арифметические и логические действия над операндами;
- г) другие внутренние операции.

Пересылка – это копирование данных. При выполнении любых операций по пересылке данных, как по системной магистрали, так и внутри процессора, приемное устройство копирует в своем регистре (регистре-приемнике) данное, содержащееся в выдающем устройстве (регистре-источнике). Содержимое регистра-источника при этом не уничтожается, остается без изменения.

Состав машинных операций отдельных команд. Каждая команда сопровождается всегда проведением не одной, а нескольких базовых машинных операций. Кроме *основной, «целевой»* операции, для осуществления которой команда предназначена (например, сложить два числа), проводится ряд *вспомогательных* операций, поддерживающих основную и не отраженных в названии команды. К таким, вспомогательным, операциям относятся всегда *чтение кода* команды из памяти и его *дешифрация*, являющиеся обязательными для всех команд. Для большинства команд выполняются также *вспомогательные пересылки* операндов из регистра на регистр или на ячейку памяти и др. Полный состав машинных операций любой команды может быть различен. Однако какова бы ни была машинная команда, она всегда выполняется процессором при помощи операций, относящихся к названным выше базовым типам 1-4 и 7. Состав операций каждой команды зависит от ее назначения, размера в байтах, формата, режима адресации операнда и др. характеристик. Но любая команда генерирует всегда один и только один состав машинных операций из числа 1-4 и 7.

Функциональная полнота машинных операций. Мы перечислили семь типов машинных операций. Пять из них (1-4 и 7) генерируются программой (машинными командами), а две (5 и 6) – аппаратурой: по сигналу прерывания (INTR) (операция 5) или при участии сигнала захвата (HLD) (операция 6).

Любую выполняемую системой операцию можно отнести к одному из семи названных типов. Однако вовсе не обязательно, чтобы система выполняла все эти операции; существуют весьма эффективные специализированные системы, выполняющие только три или четыре операции. Обязательными для любой компьютерной системы являются операции 1, 2 (чтение, запись) и 7 (дешифрация команд и обработка данных внутри процессора). Эти операции выполняются любым процессором.

Б.3. Операции внешних пересылок

В процессе работы компьютера пересылка информации по одной и той же магистрали в один и тот же момент времени происходит только между двумя устройствами **по принципу «точка-точка», «активный-пассивный»**. Одно из этих устройств является активным (*ведущим, задатчиком, управляющим*), другое – пассивным (*исполнителем, ведомым, управляемым*). Остальные устройства в это время должны быть отключены от магистрали.

Активное устройство начинает процедуру обмена и управляет ею. Пассивное устройство выполняет предписания активного. Обычно активным является процессор (за исключением операции перехода на подпрограмму прерывания и режима прямого доступа к памяти, в которых активно периферийное устройство, а процессор пассивен). Оперативная память – всегда пассивное устройство.

Передача данных происходит между процессором, с одной стороны, и памятью или ПУ, с другой; или, в режиме ПДП, – между ПУ и памятью. Возможна (но не у микроконтроллеров) передача между двумя ПУ, тогда одно – задатчик, другое – исполнитель.

Физическим средством проведения внешних пересылок между органами ядра ЭВМ – процессором, памятью и контроллерами ПУ служит **системная магистраль** ЭВМ. Она состоит из трех шин (см. рис.2.2, 2.9):

- шины данных (ШД),
- шины адресов (ША) и
- шины управления (ШУ).

Системные шины формируются на основе внешних (*собственных, локальных, «первичных»*) шин микрочипа процессора при помощи устройства шинного интерфейса, или системного контроллера. Контроллер системной магистрали вырабатывает полный набор необходимых, унифицированных сигналов этой магистрали, используя собственные, порой уникальные сигналы микрочипа процессора, выставляемые тем на своих внешних выводах («ножках»). В микроконтроллерах все указанные компоненты находятся внутри микрочипа МК.

Физически *шина* представляет собой несколько проводников (электрических линий, цепей), объединенных в единую магистраль. По каждому проводнику осуществляется передача одного двоичного разряда (бита) информации. Высокий уровень электрического потенциала соответствует обычно логической «1», низкий уровень – логическому нулю «0».

Проводники *шины данных* и передаваемые по ним сигналы обозначаются буквами $D_{n-1} \dots D_0$, где n – число проводников, разрядов. Для разных процессоров величина n может принимать значения 8, 16, 32, 64. У МК AVR $n=16$ для шины данных программной памяти и $n=8$ – памяти данных.

Сигналы и проводники *шины адреса* обозначаются $A_{m-1} \dots A_0$, m – число разрядов шины адреса: $m = 16, 20, 24, 32, 64$.

У процессоров IA-32 разрядность шины данных $n = 64$, адреса $m = 32$. У современных типов процессоров Xeon и Pentium 4 (так называемой архитектуры IA-64), процессоров семейства K8 (архитектуры AMD64, от названия фирмы-разработчика Advanced Micro Devices) и ряда других $m = 64$.

У МК AVR шина данных программной памяти имеет разрядность $n=16$, а шина данных памяти данных – $n=8$. Разрядность адресной шины AVR меняется в пределах $m=10\dots17$ – для программной памяти и $m=14$ – для памяти данных (у МК ATmega16).

Сигналы (линии) *шины управления* разнотипны и носят индивидуальные имена, отражающие функции сигналов и зависящие от типа магистрали (AGP, PCI, VESA, IESA). Типичными управляющими сигналами являются (см. рис.2.3):

1. Чт, или MEMR – сигнал управления чтением памяти (Memory Read);
2. Зп, или MEMW – сигнал записи в память (Memory Write);
3. Вв, или IOR – сигнал ввода из порта (Input-Output Read);
4. Выв, или IOW – сигнал вывода в порт (Input-Output Write);
5. ЗПр, или INTR – сигнал прерывания, запрос прерывания (Interrupt);
6. ППр, или INTA – сигнал подтверждения прерываний (Interruption Acknowledge).

Существует и множество других сигналов, передаваемых по шине управления, – сигналы тактирования от кварцевого генератора, синхронизации, арбитража (захвата шин) и т.д.

Шина данных служит физическим средством для передачи данных между выбранным устройством и центральным процессором. Под *данными* понимаются *команды* и *данные*, подлежащие обработке, т.е. *операнды*. (В узком смысле *данными, собственно данными* называются только операнды.) Сигналы, указывающие на источник или приемник информации (адреса ячеек памяти и портов в/в), в системе с отдельными ША и ШД по шине данных не передаются. Единственная задача шины данных – обеспечить прохождение данных (ко-

манд и операндов) между ЦП и устройством, адрес которого задан шиной адреса (или, вообще, между активным и пассивным устройством).

Порция данных передается по ШД в параллельном формате, одновременно по всем задействованным для этого проводникам шины данных, например по всей ширине шины $D_{n-1} \dots D_0$, если данное имеет n разрядов. Таким образом, число проводников в шине данных равно максимальной разрядности двоичного числа или кода, который может быть передан по ШД.

В зависимости от выполняемой операции информация по ШД может передаваться в двух направлениях: от внешних устройств к ЦП (чтение, ввод, подтверждение прерывания) и от ЦП к внешним устройствам (запись, вывод). Следовательно, шина данных процессора является двунаправленной. Этим она отличается от шины адреса, по которой адрес передается всегда в одном направлении: от ЦП к дешифратору адреса внешних по отношению к ЦП устройств.

По **шине адреса** центральный процессор передает адрес той ячейки памяти или того порта периферийного устройства, с которым он будет обмениваться порцией данных – принимать или выдавать данное. Каждой ячейке и каждому порту присвоен определенный адрес, известный программисту. Этот адрес записывается в выполняемой ЦП программе и поступает в ЦП при выборке программы из памяти. По шине адреса информация передается всегда в виде кодовой комбинации двоичных сигналов (нулей и единиц). Она передается в параллельном формате (одновременно по всем проводникам ША). Таким образом, число проводников в ША равно разрядности двоичного адреса, передаваемого по ША. Передаваемый из ЦП адрес поступает на входы дешифраторов памяти и периферийных устройств, которые выбирают нужную ячейку или порт. В системе с отдельными шинами адреса и данных данные, которые будут пересылаться между адресованной ячейкой (или портом) и ЦП, по шине адреса не передаются. По шине адреса не передаются также электрические сигналы, определяющие тип внешней пересылки (чтение или запись, в память или в ПУ), начало и окончание пересылки и другие управляющие сигналы. Передача адреса памяти или ПУ и обеспечение тем самым правильного места пересылки данных является основной функцией системной шины адреса.

По **шине управления** ЦП передает электрические сигналы, определяющие тип и моменты начала и окончания нужной операции обмена информацией с внешними устройствами. Эти сигналы включают устройства на выполнение нужных операций. Например: сигнал Чт (чтение), выдаваемый процессором, включает память на выполнение операции чтения ячейки – открывает ячейку по чтению; сигнал Зп (запись) открывает целевую ячейку по записи в нее информации; сигнал Вв (ввод) служит для управления вводом данного из порта в процессор; сигнал Выв (вывод) – выводом данного из процессора в порт; ППр – активизирует контроллер прерываний на выдачу им номера прерывания в процессор; и т.д. По ШУ процессор принимает также сигналы от внешних устройств, несущие запросы на обслуживание этих устройств процессором (запросы прерываний) или на другие действия (например, на переключение процессора в режим ПДП). По шине управления передаются, кроме того, общие (стартовые и синхронизирующие) сигналы.

Управляющие сигналы передаются по шине управления, как правило, одиночными битами – электрическими потенциалами по отдельным линиям (цепям) шины управления (см. рис.2.3). Этим ШУ отличается от шин данных и адреса, информация по которым передается байтами и словами.

Набор управляющих сигналов, передаваемых по ШУ, их временные диаграммы, электрические характеристики и наименования могут отличаться у разных компьютеров. Однако перечисленные функции указанных выше сигналов одинаковы и не зависят от вида процессора и компьютера.

Способы назначения устройства обмена. Для проведения пересылки одной порции данных между ЦП и внешними по отношению к нему устройствами (памятью и ПУ) процес-

сор должен назначить это устройство. Возможны два способа назначения устройства: *адресный* и *безадресный*. Адресное назначение осуществляется путем подачи процессором адреса устройства в системную шину адреса; оно используется при выполнении операций *чтения*, *записи*, *ввода* и *вывода* (операции 1 – 4). *Безадресное* назначение устройства происходит при *подтверждении прерывания процессором* (операция 5).

Потоки данных, проходящие по системной магистрали во время проведения *адресных внешних пересылок* (чтения, записи, ввода и вывода), показаны на рис.Б.2, а во время *безадресной пересылки* (подтверждения прерывания) – в нижней части рис.Б.2 [19]. **Временные диаграммы сигналов** в этих операциях изображены на рис.Б.3-Б.5. [18]

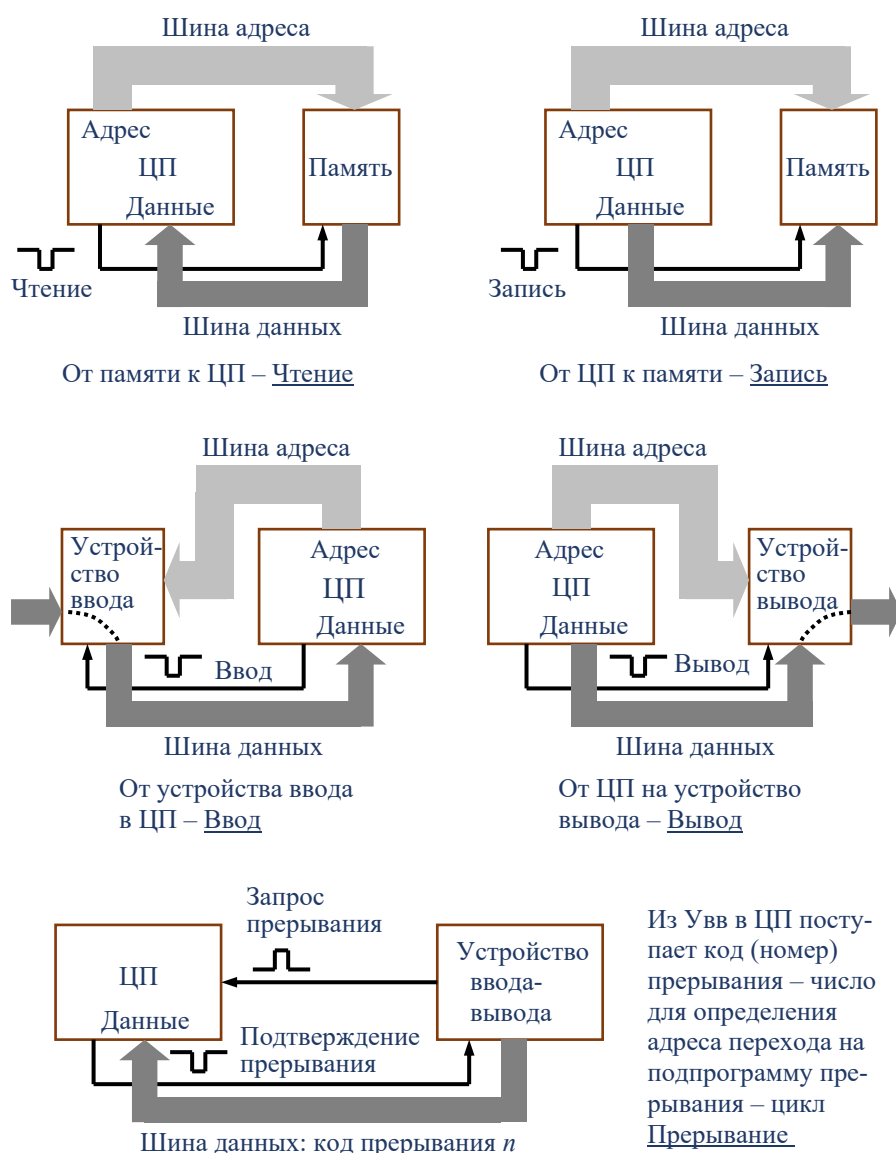


Рисунок Б.2 – Пять операций обмена данными в компьютере

Чтение данного из памяти (операции 1 и 5 – чтение). На рис.Б.2 показаны направления потоков информации при выполнении операции чтения из памяти системы, а на рис.Б.3 – временная диаграмма цикла чтения. Эта диаграмма должна соблюдаться при выполнении операции чтения данных из полупроводникового запоминающего устройства (ЗУ). Формировать временную диаграмму должен ЦП (или другое активное устройство, например контроллер ПДП – в режиме ПДП). На рис.Б.3 показаны электрические сигналы, которые долж-

но получать ЗУ при считывании данных независимо от того, микропроцессор какого типа используется в составе ЦП.

Шина адреса обеспечивает передачу правильного адреса той ячейки памяти, из которой должны считываться данные. Затем на шину управления подается сигнал Чт (Чтение). Пока он имеет активный уровень (низкий), данные поступают из адресованной ячейки на шину данных и остаются стабильными, а микропроцессор считывает двоичную информацию с шины данных и записывает ее во внутренний регистр. Через определенное время сигнал Чт сбрасывается и передача данных прекращается.

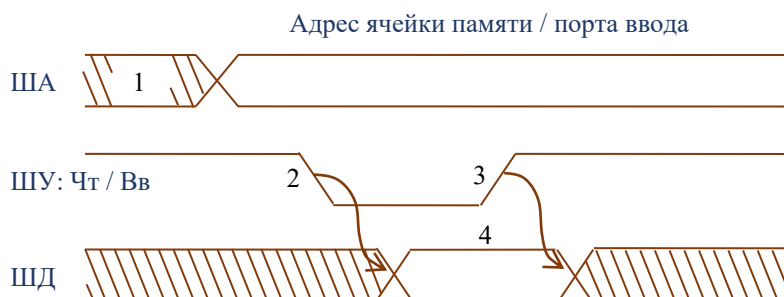


Рисунок Б.3 – Временная диаграмма чтения данных из памяти и ввода из ПУ.

Для цикла чтения из памяти: 1 – адрес поступает в память; 2 – сигнал управления установлен (разрешена выдача данных из памяти на шину данных); 3 – сигнал управления сброшен; 4 – данные на выходе памяти стабильны.

Для цикла ввода из ПУ: 1 – на шине адреса установлен адрес порта ввода данных; 2 – на шине управления установлен сигнал разрешения ввода данных на системную шину данных (начата передача данных); 3 – передача данных завершена; 4 – стабильная входная информация

Запись данных в память. На рис.Б.2 показаны потоки данных, а на рис.Б.4 – временная диаграмма цикла записи данных в полупроводниковое ЗУ (операции 2 и 5 – запись). При записи должна строго выдерживаться временная последовательность подаваемых сигналов, определяемая типом используемого ЗУ. Формировать временную диаграмму должен ЦП (активное устройство). На рис.Б.4 показаны электрические сигналы, которые должно получать ЗУ при записи данных независимо от того, микропроцессор какого типа использован для построения ЦП.

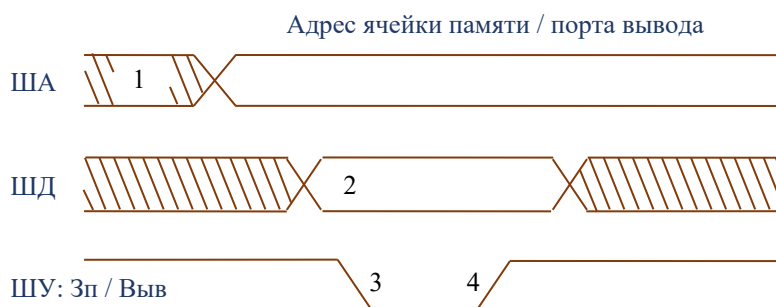


Рисунок Б.4 – Временная диаграмма записи данных в память и вывода в ПУ.

Для цикла записи в память: 1 – адрес поступает на схемы памяти; 2 – на входы данных памяти подана стабильная информация; 3 – выдача сигнала Зп (данные на шине данных стабильны); 4 – сброс сигнала Зп (передача данных завершена).

Для цикла вывода в ПУ: 1 – на адресной шине установлен адрес порта вывода; 2 – на шине данных установлена передаваемая в ПУ информация; 3 – передача данных завершена; 4 – стабильная входная информация; 3 и 4 – Выв – сигнал управления передачей данных в выбранный порт вывода

При выполнении операции записи по ША передается адрес той ячейки памяти, в которую ЦП запишет данные. Затем ЦП выдает в ШД данные. Во время выполнения операции записи сигнал управления записью Зп не подается на ШУ до тех пор, пока ЦП не подаст ста-

бильные данные на ШД. Через фиксированный промежуток времени сигнал Зп сбрасывается. Запоминающее устройство использует сигнал Зп для инициирования операции приема данных из ШД в адресованную ячейку. Сигнал на шине управления остается стабильным лишь ограниченное время, меньшее, чем время выполнения передачи данных по шине данных.

Ввод данных из ПУ (из порта ввода) (операция 3). При вводе данных из порта ввода периферийного устройства ЦП принимает данные из любого устройства, кроме ЗУ, например с оптического диска, клавиатуры, флэш-карты, аналого-цифрового преобразователя или из другого устройства, которое может пересылать данные в центральный процессор (см. рис.Б.2). Различие между операциями ввода данных из ПУ и чтения из памяти заключается в том, что для этих операций используются разные сигналы системной шины управления.

Временная диаграмма цикла ввода данных из ПУ аналогична диаграмме цикла чтения из памяти (рис.Б.3). Функцией адресной шины является обеспечение правильного электрического соединения процессора с внешним устройством. В цикле ввода шина адреса обеспечивает подачу адреса порта ввода на адресные входы дешифратора периферийных устройств. Затем на системную шину управления подается сигнал ввода Вв. Пока его уровень остается активным, данные поступают из ПУ (из назначенного порта) на системную шину данных. ЦП принимает эту двоичную информацию с шины данных и записывает ее в свой внутренний регистр. Через определенное время сигнал Вв сбрасывается, и передача данных заканчивается.

Вывод данных в ПУ (в порт вывода) (операция 4). Устройством вывода данных может быть любое устройство кроме ЗУ, воспринимающее передаваемую микропроцессором информацию, например экран монитора, оптический диск, флэш-карта, цифро-аналоговый преобразователь, др. устройства.

Показанная на рис.Б.4 временная диаграмма описывает взаимодействие ЦП с устройством вывода данных (см. рис.Б.2). Оно аналогично операции записи в память. ША обеспечивает подачу правильного адреса порта. ШД используется для передачи данных с выхода ЦП на информационный вход устройства вывода. Затем на ШУ подается сигнал управления выводом (активный уровень) и начинается передача данных.

Подтверждение прерывания. В этой операции (типа 5) периферийное устройство сообщает центральному процессору (чаще – с помощью контроллера прерываний) свой номер прерывания n , на основании которого процессор определяет адрес вектора прерывания АВП этого устройства. Потоки информации в этой операции показаны на рис.Б.2, а временная диаграмма сигналов – на рис.Б.5. Формирует эту диаграмму центральный процессор.

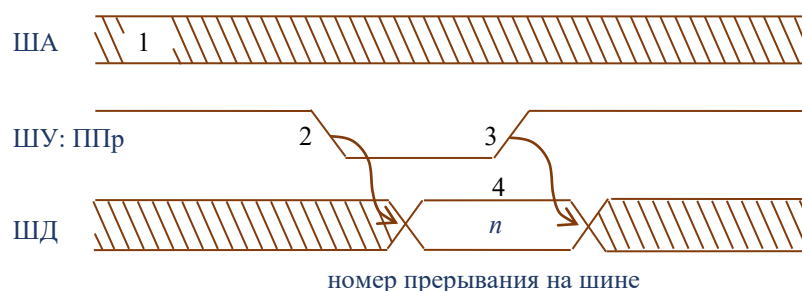


Рисунок Б.5 – Временная диаграмма цикла обработки прерывания.

1 – никакой адрес в шину адреса не поступает (безадресное назначение устройства процессором);
2 – сигнал подтверждения прерывания установлен; 3 – сигнал ППр сброшен; 4 – код адреса перехода (номер прерывания n) на выходе периферийного устройства (контроллера прерываний) стабилен

Данная операция – "цикл подтверждения прерывания" (или просто цикл "Прерывание") генерируется процессором по сигналу запроса прерывания ЗПр (INTR – Interruption), поступающему на процессор от периферийного устройства (через контроллер прерываний). После поступления ЗПр процессор закачивает текущую команду, и далее, если прерывания

разрешены, информирует периферийное устройство (контроллер прерываний) о переключении в режим прерывания, выставляя на линию подтверждения прерывания ППР (INTA – Interrupt Acknowledge) активный уровень напряжения. По сигналу ППР периферийное устройство (контроллер прерываний) сообщает процессору по шине данных номер прерывания n , после чего процессор снимает сигнал ППР, а контроллер прекращает передачу по шине данных. Далее на основании номера n процессор вычисляет адрес вектора прерывания ПУ. На этом цикл прерывания заканчивается.

Последующие действия процессора состоят в сохранении контекста программы в стеке, переходе на подпрограмму прерывания и ее исполнении (глава 8).

Обратите внимание, что процессор назначает устройство обмена в цикле прерывания *безадресным способом* (с помощью сигнала ППР), т.е. не выставляет никакого истинного адреса в шину адреса.

Б.4. Машинные циклы

Следующим по иерархии компонентом рабочего процесса, после машинной операции, является **машинный цикл**. Каждый МЦ соответствует одному обращению процессора к памяти или периферийным устройствам и некоторым внутренним операциям процессора над данными, выполняемым после этого обращения к шине и до начала следующего. К внутренним операциям относятся: расшифровка считанного из памяти кода команды, сложение операндов, сдвиг числа на определенное количество бит и т.п. Если внутренние операции в машинном цикле отсутствуют, то он сводится к одной пересылке по системной магистрали. Вместо термина «машинный цикл» используются также термины «цикл обращения к магистрали», «цикл обращения к каналу ЭВМ», «цикл магистрали», «цикл шины», «магистральный цикл», «шинный цикл». Длительность машинного цикла может варьироваться в определенных пределах в зависимости от модели процессора и выполняемой операции, но она всегда равна целому числу периодов (или долей периода) следования синхроимпульсов тактового генератора – тактов T .

Из машинных циклов складываются, в свою очередь, командные циклы, затрачиваемые на выполнение отдельных команд, а из командных циклов составляется законченный программный процесс (п.Б.2).

Итак, *машинный цикл* – это действия процессора по выполнению

- а) *одной внешней пересылки* по шине данных и
- б) некоторых *внутренних операций* по обработке данных, выполняемых на его собственных узлах после завершения данной пересылки и до начала следующей, которые (внутренние операции), однако, могут и отсутствовать.

Машинным циклом называют также дискретный промежуток времени выполнения этих действий.

В соответствии со сказанным весь интервал времени функционирования ЭВМ от пуска до останова подразделяется на целое число дискретных промежутков времени – машинных циклов, как показано на рис.Б.6.

Поскольку внутренние операции процессора протекают скрыто от пользователя и не отражаются на состоянии внешних выводов микросхемы, то для пользователя машинный цикл выглядит как одна внешняя пересылка, т.е. как один цикл шины, шинный цикл. Соответственно весь процесс функционирования процессора от пуска до останова представляется на системной магистрали в виде последовательности шинных циклов – внешних пересылок данных между процессором и памятью или периферийными устройствами.

Внутренние операции требуют для своего проведения определенного времени. Для этого процессор может вводить в машинный цикл дополнительные временные задержки («холостые» такты). В холостых тактах никакие внешние пересылки по ШД процессором не производятся, и он вообще отключается от внешних шин. Холостые такты обычно добавляются в конце машинных циклов. У современных высокопроизводительных процессоров,

располагающих параллельно работающими операционным, управляющим и интерфейсным устройствами, внутренние операции по обработке ранее введенных команд и данных осуществляются одновременно с выборкой последующих команд или пересылкой данных текущих команд (так называемая «конвейерная обработка команд»). Поэтому необходимость во введении холостых тактов у таких процессоров возникает крайне редко и фактически не влияет на производительность процессора.

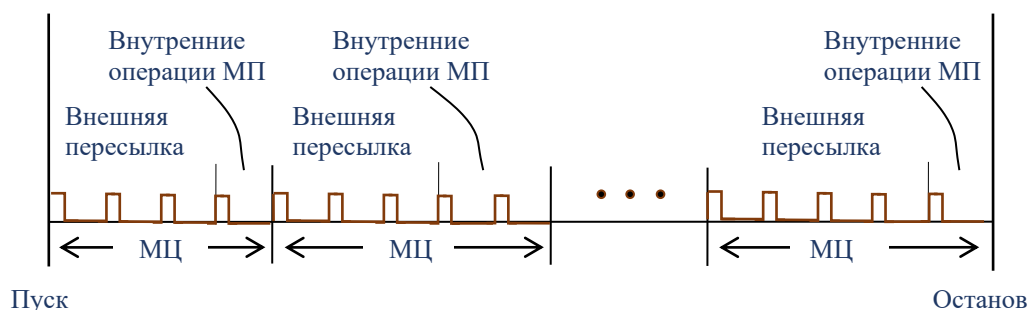


Рисунок Б.6 – Последовательность машинных циклов при работе ЭВМ

По существу, термин «цикл канала», «цикл шины» эквивалентен понятию машинного цикла процессора. Однако в многопроцессорных системах этот термин несколько шире в том смысле, что цикл канала может инициировать не только центральный процессор, но и, например, процессор ввода-вывода или арифметический сопроцессор.

На основе семи типов машинных операций, рассмотренных в п.Б.2, процессор формирует *пять типов машинных циклов*, отличающихся производимыми в них *внешними пересылками*:

- 1) машинные циклы чтения одной порции данных из памяти – циклы типа "Чтение";
- 2) машинные циклы записи данного в память – циклы "Запись";
- 3) циклы ввода данного в процессор из порта ПУ – циклы "Ввод";
- 4) циклы вывода данного из процессора в порт ПУ – "Вывод";
- 5) машинные циклы подтверждения прерывания, предназначенные для ввода в процессор из ПУ (из контроллера прерываний) кода пускового адреса подпрограммы прерывания или номера прерывания n – циклы "Прерывание", или "Подтверждение прерывания".

Все внутренние операции процессора по обработке данных (операции типа 7) выполняются в течение названных типов машинных циклов: в холостых тактах этих циклов или одновременно с выполнением внешних пересылок следующих циклов.

В режиме ПДП (операция 6) данные передаются между основной памятью и периферийным устройством, минуя ЦП; ЦП на это время отключается от системной шины и не участвует ни в каких пересылках. Следовательно, можно считать, что ПДП проводится в промежутках *между* машинными циклами процессора.

Машинные циклы можно различать дополнительно по смыслу проводимых в них *внутренних операций* над данными. Например: цикл "Чтение команды", в котором команда выбирается из памяти и расшифровывается процессором; цикл "Чтение операнда", используемый для чтения операнда из памяти и, например, сложения его с содержимым регистра процессора; цикл "Запись операнда в стек", вовлекающий в работу специальный операционный регистр процессора – указатель стека для адресации стековой памяти с последующей записью в нее операнда; и т.д.

Циклы можно различать также по *размеру передаваемой в них порции данных*, например: "Ввод байта", "Запись слова" и т.д.

Временные диаграммы машинных циклов выглядят так же, как и диаграммы соответствующих внешних пересылок (см. рис.Б.3-Б.5), но с разбиением на такты. Внутренние

операции не проявляются на состоянии внешних выводов микросхемы и на временных диаграммах не отображаются. К диаграммам могут добавляться лишь холостые такты, в которых процессор выполняет внутренние операции.

Наряду с временными диаграммами сигналов действия процессора в машинных циклах удобно описывать при помощи условных символьных выражений – **логических диаграмм**. Они представлены на рис.Б.7-Б.9.

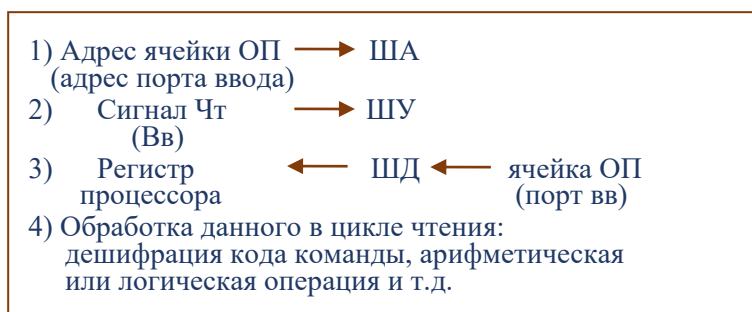


Рисунок Б.7 – Действия ЦП в машинных циклах «Чтение» и «Ввод»

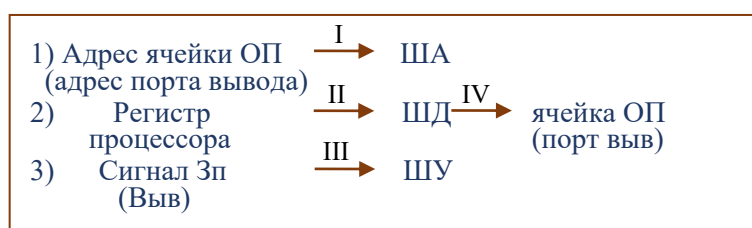


Рисунок Б.8 – Действия ЦП в машинных циклах «Запись» и «Вывод».
 I – первая операция, II – вторая, III – третья, IV – четвертая



Рисунок Б.9 – Действия ЦП в машинном цикле «Прерывание»

Согласно диаграммам на рис.Б.3 и Б.4 и сказанному ранее каждое обращение процессора к ячейке памяти (по чтению или записи) или к порту в/в (по вводу или выводу), т.е. *каждый машинный цикл адресного типа* (1 - 4) всегда начинается с того, что:

- 1) процессор выставляет адрес ячейки (или порта), к которой он будет производить обращение в этом цикле, в шину адреса и
- 2) обращается к памяти (или периферийному устройству) управляющим сигналом, формируемым по одной из линий шины управления для инициирования нужной операции чтения или записи (ввода или вывода);
- 3) после этого происходит передача данного по шине данных между процессором и адресованной ячейкой (портом) в нужном направлении: при чтении и вводе – из устройства в процессор, при записи и выводе – из процессора в устройство, причем при записи и выводе процессор сначала выдает данное, и только потом – сигнал записи или вывода

(это делается с целью стабилизации данного на шине к моменту занесения его во внешнее устройство);

- 4) при необходимости в машинном цикле процессор производит внутренние операции над данными. Для этого он может вводить в цикл дополнительные, «холостые» такты.

Подобные действия производятся в *адресных* машинных циклах (типов 1-4). При выполнении же *безадресного* цикла (типа 5) шина адреса не используется, а передача данного происходит под управлением одного только сигнала подтверждения прерывания ППр (см. рис.Б.5).

Важное заключение. В заключение необходимо сказать, что основное внимание было уделено нами последовательности событий; временные диаграммы при этом можно описывать в общем виде, поскольку важнее понять состав и назначение вырабатываемых сигналов и общую последовательность их формирования, чем углубляться в рассмотрение конкретных временных характеристик и их точных значений. Понимание принципа действия позволит сравнивать различные системные сигналы; при необходимости можно использовать детальный анализ временных соотношений.

При разработке большинства схем интерфейса вычислительной системы не обязательно проводить детальный анализ временных диаграмм. Если придерживаться некоторых простых правил, интерфейсы можно разрабатывать и отлаживать, зная общую последовательность действий, выполняемых при работе интерфейса. Эти правила были рассмотрены в приведенных положениях по организации интерфейса вычислительных систем.

Возвращаясь к приведенным временным диаграммам, можно отметить, что выполнением всех временных соотношений управляет центральный процессор, т.е. при выполнении каждой операции процессор будет вырабатывать в соответствующие моменты времени сигналы, подаваемые на каждую из трех шин. Поэтому, при разработке внутримашинного интерфейса можно просто использовать шины, полагая, что правильность выполнения временной диаграммы работы системы обеспечивается процессором. [17]

В. Порядок работы компьютера

В.1. Программный процесс

При функционировании компьютера по отработке машинной программы базовые *машинные операции* и формируемые из них *машинные циклы* складываются в *командные циклы* процессора, затрачиваемые на отработку отдельных машинных команд, а те, в свою очередь, следуя один за другим, выстраиваются в целостную процедуру выполнения *машинной программы* целиком (п.Б.2-Б4). Это означает, что машинная программа выполняется в компьютере в виде последовательности простых событий – машинных операций, т.е. процесса. Соответственно, *процессом* в информатике называют программу в стадии выполнения.

В.2. Размещение программы в основной памяти

Существенное условие размещения программы в основной памяти. Программист, работающий на языке машинных команд или ассемблера, должен следить за правильным размещением программы в основной памяти компьютера:

1. Все команды программы должны быть размещены подряд в последовательных ячейках памяти в том порядке, в каком они записаны в программе. Такое размещение программы называется *естественным*. И только те команды, на которые передается управление с помощью команд перехода, *принудительно* размещаются по адресам, указанным в этих командах перехода – адресам перехода. Поэтому команды выполняются, как правило, в порядке возрастания адресов. Этому правилу отвечает программа для МК AVR, взятая из примера, рассмотренного в п.7.3, и представленная в табл.В.1. Все команды в ней располагаются последовательно, друг за другом (естественно) за исключением команд `eor` и `ld`, находящихся в ячейках по адресам перехода \$0500 и \$0506 (принудительно).
2. Входная команда программы помещается в некоторую отправную ячейку памяти, начиная с которой располагается вся программа. Эта ячейка должна находиться за пределом области векторов прерываний (если они используются; в случае МК AVR эта область занимает адреса \$0002...\$0029, глава 8). Переход на входную команду программы осуществляется командой передачи управления из других мест программной памяти; или программа сразу начинается со стартовой команды компьютера, если вектора не используются.
3. Стартовая команда компьютера (запускаемая самой первой по сигналу системного сброса/пуска) должна находиться во вполне определенной – «пусковой» ячейке памяти – ячейке, с которой запускается процессор по этому сигналу. Для большинства процессоров адрес пусковой ячейки строго фиксирован. Для процессоров IA-32 он равен \$FFFFFFF0, для МК AVR – \$0000. Поэтому они запускаются на счет с ячеек \$FFFFFFF0 и \$0000 соответственно. В пусковую ячейку помещают, как правило, команду безусловного перехода на начало стартовой, инициализационной программы компьютера. Такая команда `jmp` расположена в ячейке \$0000 для программы из табл.В.1, и она передает управление на ячейку \$0030, где начинается некоторая программа ввода массива чисел.

Применение директив ассемблера резко облегчает соблюдение перечисленных правил 1-3.

Пусковая ячейка должна входить в состав ПЗУ – энергонезависимого ЗУ. В ПЗУ находится и вся стартовая программа компьютера. Поэтому после включения питания компьютер всегда готов к запуску стартовой программы. Остальные программы загружаются в ОЗУ с носителя внешней памяти (например, с жесткого диска). В микроконтроллерах все

программы, включая стартовую команду, находятся в ПЗУ. Загрузка программ в ПЗУ микроконтроллера выполняется редко, иногда всего один раз при изготовлении устройства.

Таблица В.1 – Программа суммирования элементов массива из примера в п.7.3

Память данных (ОЗУ)		
Адреса	Данные	Комментарии
\$0060		
...		
	<i>Массив 32 байта</i>	
\$0100	\$xx	Массив чисел,
\$0101	\$xx	вводимый
\$0102	\$xx	в ячейки
...	...	\$0100...\$011F
\$011E	\$xx	программой ввода
\$011F	\$xx	ниже
	<i>Сумма 2 байта</i>	
\$0120		Сюда помещается
\$0121		сумма чисел
Память программ (ПЗУ)		
Адреса	Команды	Комментарии
\$0000	jmp \$0030	Вектор сброса. Перепрыгнуть
\$0001		через векторы прерываний
\$0002	Векторы прерываний	
...	(по две ячейки	
\$0029	на каждый вектор)	
<i>Некоторая программа ввода массива чисел</i>		
\$0030	...	
...	...	
	jmp \$0500	Переход на нашу программу
...	...	суммирования
<i>Программа суммирования</i>		
<i>Инициализация</i>		
\$0500	eor R0, R0	Сбросить
\$0501	eor R1, R1	аккумулятор R1:R0
\$0502	eor R3, R3	Обнулить старший байт элемента
\$0503	ldi XL, 0x00	Адрес массива \$0100
\$0504	ldi XH, 0x01	в регистре X
\$0505	ldi R16, 0x20	Число элементов в R16
<i>Суммирование</i>		
\$0506	ld R2, X+	Очередной элемент в R2
\$0507	add R0, R2	Прибавить элемент к
\$0508	adc R1, R3	аккумулятору с учетом переноса
\$0509	dec R16	Это последний элемент?
\$050A	brne \$0506	Нет, повторить суммирование
\$050B	sts \$0120, R0	Да, запомнить сумму
\$050C	sts \$0121, R1	в ячейках \$120, \$121
...		

В.3. Продвижение процессора по программе

Порядок выполнения команд. Команды программы выполняются, как правило, в порядке возрастания адресов и начиная с некоторого начального, отправного адреса: S, S+1, S+2, ... Исключение составляют лишь команды, принимающие на себя управление по указке команд перехода и выполняемые с нарушением естественного порядка. Например, в случае из табл.В.1 отправной адрес программы ввода массива чисел есть \$0030, а программы суммирования – \$0500.

Назначение программного счетчика РС. Для поддержания указанного порядка выполнения команд служит специальный собственный регистр процессора, называемый *программным счетчиком РС* (Program Counter) или *указателем инструкций IP* (Instruction Pointer) или *счетчиком команд СчК*. Этот регистр всегда содержит адрес следующей по порядку, очередной, еще не исполненной команды, которая будет выбираться из памяти и исполняться вслед за текущей командой; или адрес очередного слова выбираемой команды (если команда длинная и выбирается за несколько циклов шины). Благодаря регистру РС процессор всегда "знает", откуда получить следующую команду или ее очередное слово.

Правила модификации программного счетчика были описаны в п.5.3.

В.4. Пуск компьютера и его функционирование после пуска.

Командные циклы

Сброс, инициализация и пуск компьютера. После того как компьютер подготовлен к работе и в основную память занесена, тем или иным способом, программа в кодах и исходные данные, можно запускать компьютер на счет.

Обобщенный порядок функционирования компьютера представлен на рис.В.2. Работа компьютера начинается с загрузки программного счетчика, инициализации (установки в начальное состояние) и пуска процессора по сигналу сброса (блоки 1 и 2).

Пуск процессора (или микроконтроллера) производится при включенном питании и тактовых импульсах активизацией сигнала сброса. Он приводит к инициализации – установке внутренних блоков и внешних сигналов процессора/МК в определенные начальные состояния (в зависимости от модели процессора/МК). Например, порты ввода-вывода МК AVR немедленно сбрасываются в исходное состояние, когда активируется источник сброса. В процессе сброса во все регистры ввода-вывода записываются их начальные значения. После снятия сигнала сброса (по его заднему фронту) процессор/МК запускается с некоторой фиксированной ячейки памяти программ (процессор IA-32 – с ячейки \$FFFFFFF0, МК AVR – с ячейки \$0000), выполняя цикл чтения из нее машинной команды.

После пуска процессор/МК начинает работать при запрещенных прерываниях (I=0).

После пуска процессор/МК приступает к выполнению программы начиная с пусковой ячейки программной памяти, скажем с 0-й ячейки. В этой ячейке размещается пусковая команда компьютера/МК (называемая зачастую «вектором сброса») – команда безусловного перехода на начало стартовой программы (например, команда `jmp $0030` в 0-й ячейке программной памяти в табл.В.1). Любая программа выполняется последовательно, по командам, путем обращения процессора к очередным ячейкам памяти, в которых она размещена.

Командный, или рабочий, цикл процессора. Выполнение операций по извлечению из памяти и выполнению одной команды, а также интервал времени, требующийся для этого, носит название *командного или рабочего цикла процессора* (п.Б.2). Цикл команды состоит в свою очередь из одного или нескольких *машинных (шинных) циклов*, каждый из которых соответствует одному обращению процессора к памяти или периферийным устройствам (п.Б.4). Это означает, что общее число машинных циклов в командном цикле равно числу обращений процессора к внешним устройствам (ОП и ПУ) за все время отработки команды.

Итак, командный цикл включает в себя (см. рис.В.2):

- выборку и дешифрацию команды (блок 7) и

- выполнение команды (блок 8).

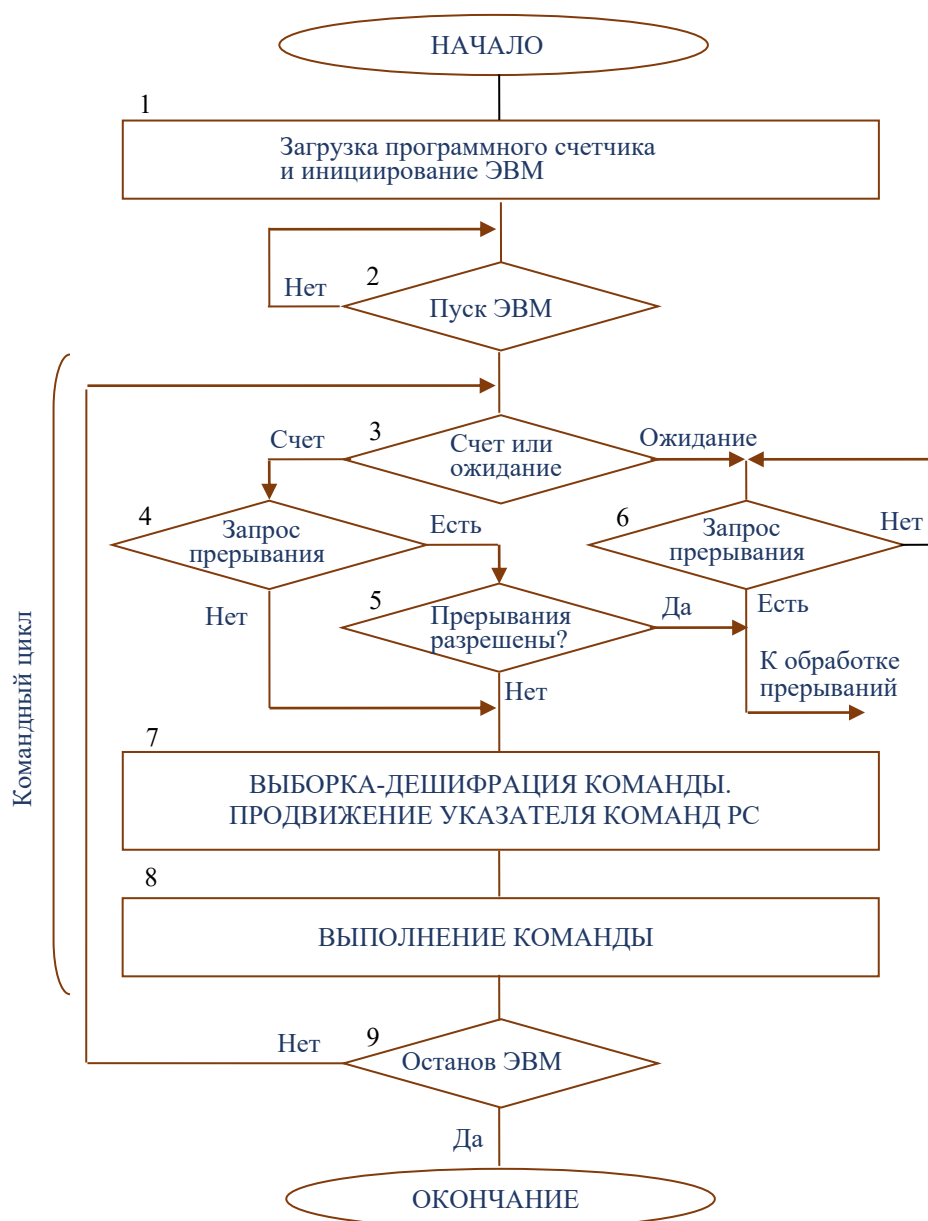


Рисунок В.2 – Обобщенный порядок работы ЭВМ

Каждое из этих действий может потребовать исполнения одного или нескольких машинных циклов. Выборка команды занимает ровно столько машинных циклов, сколько машинных слов она содержит (сколько ячеек памяти размером в ширину шины данных занимает). В каждом машинном цикле выбирается одно слово команды. Все машинные циклы выборки команды относятся к циклам типа "Чтение" ("Чтение команды") (п.Б.4).

При выполнении команды могут потребоваться дополнительные внешние пересылки – пересылки операндов или результатов операции, занимающие соответствующее число машинных циклов. Они реализуются процессором в форме циклов "Чтение" ("Чтение операнда"), "Запись (операнда)", "Ввод" и "Вывод" (п.3.3, 3.4). Если таких пересылок не требуется, то указанные машинные циклы отсутствуют, и тогда общее число машинных циклов в команде равно числу машинных циклов ее выборки.

Закономерен вопрос, каким образом процессор различает код операции команды (КОП) и операнды (или их адреса), если и те и другие поступают в процессор по шине дан-

ных и представлены двоичными кодами. Дело в том, что в начальной ячейке любой команды всегда записан КОП, и каждый КОП содержит информацию, указывающую общее число слов (байт) данной команды. Таким образом, процессор точно знает, что он собирается получить в определенный момент времени. Если программист сделает ошибку, то МП может принять данные за команду или наоборот.

Другие действия процессора при выполнении программы. Во время выполнения программы процессор следит за внешними управляющими сигналами, а также идентифицирует свое внутреннее состояние.

В начале каждого машинного цикла процессор распознает, какое из двух альтернативных состояний – *Счет* или *Останов (Ожидание прерывания)* имеет место (блок 3). В состоянии *Останов (Ожидание прерывания)* никакие программы не выполняются, а процессор ждет прихода сигнала прерывания INTR (ЗПр), после чего управление переходит к соответствующей прерывающей подпрограмме, переводящей процессор в состояние *Счет*.

Процессор IA-32/AMD64 переходит в состояние *Останов* под действием команды останова `hlt` или ожидания `wait`, а выходит из него по сигналам пуска RESET и прерывания INTR (а также NMI – No Masked Interruption, немаскируемого прерывания). МК AVR – под действием команды `sleep`, переводящей МК в режим энергосбережения SLEEP и останавливающей его работу, а выходит по сбросу или прерыванию (п.4.3, 6.4).

В состояние *Счет* процессор переходит:

- сразу после пуска (по сигналу сброса, воспринимаемому безусловно, в любом состоянии, в т.ч. в состоянии *Останов*);
- в ответ на запрос прерывания (по сигналу прерывания; причем запросы обслуживаются и в момент *Останов*);
- и после завершения любой команды, не являющейся командой останова `hlt` или ожидания `wait` или (для МК AVR) сна `sleep`.

В состоянии *Счет* при отсутствии запросов прерываний или при условии, что они запрещены, (блоки 4 и 5) процессор выбирает и выполняет очередную команду. Если же имел место запрос прерывания, и при этом прерывания были разрешены, то происходит выход из нормального командного цикла и переход к подпрограмме обработки прерывания (блоки 4, 5 и 6).

В.5. Разложение команд по машинным циклам

Анализ отработки команд путем описания командных циклов и составляющих их машинных циклов помогает лучше понять работу процессора и системного интерфейса при выполнении программы. Проведем этот анализ на примере команд микроконтроллеров AVR.

Примеры команд микроконтроллеров AVR. В табл.В.2 приведены некоторые примечательные команды микроконтроллеров семейства AVR, их форматы и действие с точки зрения программиста. Далее действие этих команд рассматривается в части операций процессора на шине и внутренних узлах. Описываются машинные циклы, генерируемые процессором при выборке и исполнении команд, и производимые в них базовые операции по пересылке и преобразованию данных. Описание циклов представлено в форме условных символьных выражений – логических диаграмм. При этом используются следующие обозначения:

← (или : =) – операция пересылки (копирования, присвоения);

ША, ШУ, ШД – шины адресов, управления, данных для памяти программ и памяти данных отдельно;

Чт, Зп – сигналы управления чтением, записью;

РгК – регистр команд;

$m(PC)$ – содержимое ячейки памяти программ (memory), адрес которой находится в регистре PC;

$m(X)$ и $m(SP)$ – содержимое ячейки памяти данных (memory), адресуемой регистрами X и SP соответственно.

Таблица В.2 – Описание команд для программиста

<Формат> (мнемокод)	<Действие> (псевдокод)
Запись в память: <code>st Y, R1</code>	$m(Y) \leftarrow R1$
Чтение памяти: <code>ld R0, X</code>	$R0 \leftarrow m(X)$
Вталкивание в стек: <code>push R7</code>	$m(SP) \leftarrow R7, SP \leftarrow SP - 1$
Выталкивание из стека: <code>pop R8</code>	$SP \leftarrow SP + 1, R8 \leftarrow m(SP)$

Выполнение команд по машинным циклам. Логические диаграммы командных циклов. Сначала любая команда выбирается из памяти и ее код дешифруется. На выборку команды затрачивается ровно столько машинных циклов, сколько ячеек памяти она занимает. Все циклы выборки команды являются циклами чтения.

Затем команда выполняется в смысле реализации ею заданных машинных операций по обработке данных и переходам. Для этого процессору могут потребоваться дополнительные внешние пересылки, реализуемые в форме машинных циклов типов "Чтение", "Запись", "Ввод" или "Вывод", используемых для пересылки и преобразования операндов.

Командный цикл команды `st Y, R1`

<u>Маш. цикл 1 "Чтение команды (КОП)"</u>	<u>Маш. цикл 3 "Запись операнда "</u>
1) $PC \rightarrow ША,$ 2) $Чт \rightarrow ШУ$ 3) $РгК \leftarrow ШД \leftarrow m(PC), PC \leftarrow PC + 1$ 4) Дешифрация команды	1) $X \rightarrow ША$ 2) $Зп \rightarrow ШУ$ 3) $R1 \rightarrow ШД \rightarrow m(X)$

Командный цикл команды `ld R0, X`

<u>Маш. цикл 1 "Чтение команды (КОП)"</u>	<u>Маш. цикл 3 "Чтение операнда"</u>
1) $PC \rightarrow ША,$ 2) $Чт \rightarrow ШУ$ 3) $РгК \leftarrow ШД \leftarrow m(PC), PC \leftarrow PC + 1$ 4) Дешифрация команды	1) $Y \rightarrow ША$ 2) $Чт \rightarrow ШУ$ 3) $R0 \leftarrow ШД \leftarrow m(Y)$

Командный цикл команды push R7

<u>Маш. цикл 1 "Чтение команды (КОП)"</u>	<u>Маш. цикл 3 "Запись в стек "</u>
1) PC $\rightarrow$ ША,	1) SP $\rightarrow$ ША
2) Чт $\rightarrow$ ШУ	2) Зп $\rightarrow$ ШУ
3) РгК $\leftarrow$ ШД $\leftarrow m(PC)$, PC $\leftarrow$ PC + 1	3) R7 $\rightarrow$ ШД $\rightarrow m(SP)$, SP $\leftarrow$ SP – 1
4) Дешифрация команды	

Командный цикл команды pop R8

<u>Маш. цикл 1 "Чтение команды (КОП)"</u>	<u>Маш. цикл 3 "Чтение из стека"</u>
1) PC $\rightarrow$ ША,	1) SP $\leftarrow$ SP + 1, SP $\rightarrow$ ША
2) Чт $\rightarrow$ ШУ	2) Чт $\rightarrow$ ШУ
3) РгК $\leftarrow$ ШД $\leftarrow m(PC)$, PC $\leftarrow$ PC + 1	3) R8 $\leftarrow$ ШД $\leftarrow m(SP)$
4) Дешифрация команды	

Учебное издание

Козырев Владимир Георгиевич

**ПРОГРАММИРОВАНИЕ
МИКРОКОНТРОЛЛЕРОВ***Учебное пособие*

В авторской редакции

Изд. № 53/2023. Объем 15,25 п.л. Усл. печ. л. 14,18. Уч.-изд. л. 14,945.
РИИЦМ ФГАОУ ВО «Севастопольский государственный университет»