2 ТИПОВЫЕ СТРУКТУРНО-АРХИТЕКТУРНЫЕ РЕШЕНИЯ ЦП И ПАМЯТИ МК ОБЩЕГО НАЗНАЧЕНИЯ

2.1 Основные компоненты структурно-архитектурных решений ЦП и памяти

Структурно-архитектурные решения ЦП и памяти рационально рассматривать совместно, поскольку они тесно взаимосвязаны. Их основными компонентами, очевидно, являются следующие [3]:

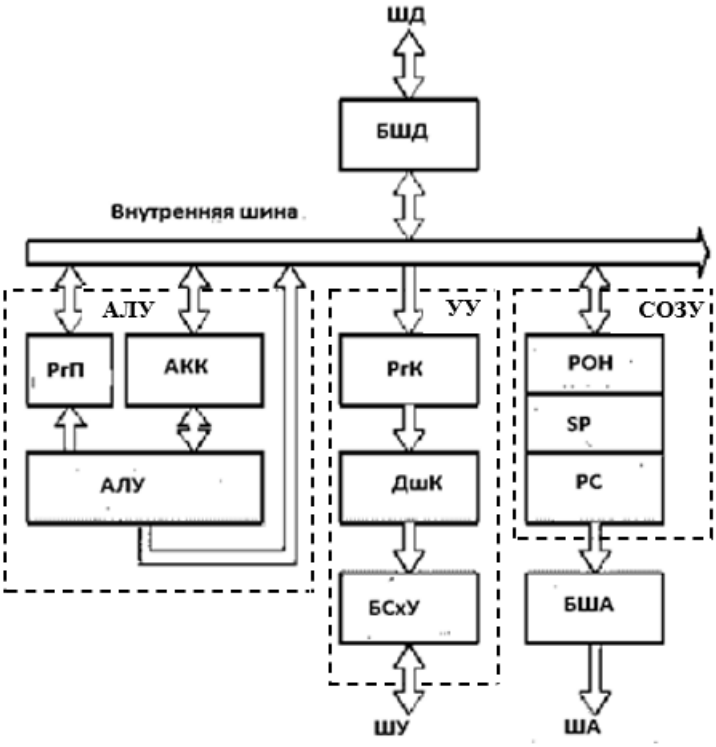
- структурная схема ЦП, состав и назначение его узлов и блоков,
- режимы работы ЦП;
- типы, форматы и разрядность обрабатываемых данных;
- состав, назначение и объем модулей и/или областей памяти, форматы адресов их содержимого и порядок доступа к нему;
- регистровая модель ЦП;
- состав системы команд машинного языка, их форматы и количество тактов синхрогенератора МК, необходимых для их выполнения.

2.2 Типовые структурные схемы и принципы работы ЦП

Как указано в табл. 1.1, ЦП является «ядром» МК, выполняющим функции обработки данных и управления другими функциональными блоками МК, в соответствии с последовательностью команд, хранимой в памяти программ. Она, в свою очередь, представляет собой совокупность запоминающих элементов, каждому из которых присвоен индивидуальный и неповторимый (в пределах соответствующего модуля памяти) **адрес**.

ЦП относительно простых МК (класса «*cost-sensitive*» и ряда семейств класса «*mainstream*») строятся по «классической» схеме ЦП электронно-вычислительных средств. В прошлом по ней реализовывались ЦП компьютеров практически всех классов, а в настоящее время – в основном, ЦП МК вышеназванных категорий.

Обобщенная структурная схема «классического» ЦП представлена на рис. 2.1 [3].



АЛУ – арифметико-логическое устройство
УУ – устройство управления
СОЗУ – сверхоперативное запоминающее устройство
АКК – регистр-аккумулятор
РгП – регистр признаков (статуса)
РгК – регистр команды
ДшК – дешифратор команд
РОН – регистры общего назначения
SP – указатель стека (*Stack Pointer*)
PC – программный счетчик (*Program Counter*)
БСхУ – блок синхронизации и управления
БША, БШД – буферы шины адреса и данных соответственно
ШУ, ШД, ША – шины управления, данных и адреса соответственно

Рис. 2.1. Обобщенная структурная схема «классического» ЦП

Основными блоками ЦП являются следующие:

- устройство управления (УУ), включающее в себя регистр команд, дешифратор команд и блок синхронизации и управления, которое реализует выборку из памяти двоичных кодов команд выполняемой программы и выработку сигналов управления узлами и блоками ЦП в соответствии с указанными кодами;

- арифметико-логическое устройство (АЛУ), выполняющее собственно обработку данных в двоичной системе счисления –

операции сложения, вычитания, сдвигов, поразрядного логического И, ИЛИ, исключающего ИЛИ и. т. п., к последовательности которых сводится любая задача, решаемая средствами цифровой вычислительной техники, будь то математические расчеты, управление техническим объектом и т. п.;

- сверхоперативное запоминающее устройство (СОЗУ), включающее в себя набор быстродействующих ячеек памяти (регистров), в том числе регистры общего назначения (РОН), используемые для хранения операндов и результатов выполнения операций, и регистры специальных функций (РСФ), состав и назначение которых рассмотрены далее.

Общий принцип работы ЦП следующий. Процесс обработки данных осуществляется под управлением **УУ**. Двоичные коды команд, в явном или неявном виде содержащие:

- код подлежащей выполнению операции;
- адреса или указатели адресов операндов и результата;
- адрес или указатель адреса следующей команды;

последовательно, команда за командой, загружаются из памяти в регистр команды. Адрес очередной команды при этом содержится в *РС*. По загрузке кода команды в регистр команды она поступает на дешифратор команд. Он представляет собой, по существу, блок преобразования двоичных кодов команд в управляющие сигналы, поступающие на узлы и блоки ЦП (на рис. 2.1, в избежание его загромождения, они не показаны). По поступлении кода очередной команды на выходы ее дешифратора выдаются соответствующие данному коду:

- управляющие сигналы для АЛУ, настраивающие его в режим работы, задаваемый кодом операции (сложения, вычитания, сдвига и т. п.);
- управляющие сигналы, открывающие для чтения регистр блока РОН или ячейку памяти, содержимое которого (которой), в соответствии с кодом команды, служит одним из операндов (вторым операндом в простейших ЦП всегда является содержимое регистра-аккумулятора);

- управляющие сигналы, открывающие для записи регистр блока РОН или ячейку памяти, в который (которую), в соответствии с кодом команды, должен быть записан результат ее выполнения;

- управляющие сигналы для *PC*, задающие режим вычисления адреса следующей команды.

По выполнении очередной команды из памяти в регистр команды поступает следующая команда, адрес которой равен содержимому *PC*, полученному по результатам выполнения предыдущей команды, и вышеописанный процесс повторяется.

Для читателей, желающих более глубоко понять принципы реализации УУ ЦП, в Приложении А представлен пример синтеза и реализации УУ простейшего ЦП, система команд которого содержит только 4 команды (сложения и вычитания содержимого аккумулятора с двоичным числом на входах ЦП, безусловного перехода и условного перехода по нулевому содержимому аккумулятора). Естественно, ЦП с таким набором команд не имеет практического смысла, однако он удобен для пояснения принципов синтеза и реализации УУ. Синтез осуществлялся на основе классического представления УУ как цифрового автомата (*state machine*) [3]. В Приложении А приведены:

- форматы и описания работы команд, реализуемых ЦП;

- структурные схемы операционного блока и УУ ЦП, составленные в соответствии с реализуемой им системой команд;

- диаграмма состояний и таблица функционирования УУ как цифрового автомата, реализующего систему команд синтезируемого ЦП;

- системы уравнений, реализуемых комбинационными устройствами синтезируемого цифрового автомата.

При этом схемная реализация данной системы уравнений очевидна, и, во избежание перегрузки изложенного в Приложении А материала, в нем не представлена.

Как для читателей, подробно ознакомившихся с представленными в Приложении А материалами, так и для ограничившихся их поверхностным рассмотрением, **важно отметить**, что общие принципы синтеза и построения реальных ЦП и их УУ аналогичны рассмотренным в Приложении А. Отличия

заключаются, в основном, в составе системы команд (на практике их не менее нескольких десятков) и функциональных узлов ЦП, количестве состояний цифрового автомата УУ и генерируемых им управляющих сигналов. В настоящее время синтез УУ и других блоков ЦП, естественно, не производится «вручную»; существует достаточно большой выбор средств автоматизации проектирования цифровых устройств.

Собственно процесс обработки данных осуществляется посредством **АЛУ**. АЛУ ЦП с относительно простой системой команд, в том числе АЛУ МК общего назначения реализуется в виде набора цифровых блоков, выполняющих элементарные арифметические и логические операции [3]. Настройка АЛУ на выполнение той или иной операции осуществляется подключением выходов реализующего ее блока к выходам АЛУ, посредством мультиплексора, управляемого дешифратором команды.

Для читателей, желающих более глубоко понять принципы реализации АЛУ ЦП, в Приложении Б представлен пример функциональной схемы простейшего однобитового АЛУ, реализующего одну арифметическую операцию (сложение) и три побитовые логические (И, ИЛИ и инверсию). Там же представлено краткое описание схемы АЛУ.

В состав АЛУ относительно простых ЦП, как правило, входит **регистр-аккумулятор** (см. рис. 2.1), содержимое которого, по умолчанию, служит одним из операндов.

Также структурно к АЛУ может быть отнесен регистр статуса (см. рис. 2.1), однако с точки зрения архитектуры ЦП данный регистр корректнее отнести к РСФ СОЗУ (см. далее).

СОЗУ фактически служит в качестве быстродействующей оперативной памяти ЦП и включает в себя, как указано ранее:

- регистры общего назначения (РОН, *GPR*), как правило, используемые для хранения операндов и результатов выполнения операций;

- регистры специального назначения, чаще называемые регистрами специальных функций (РСФ, *SFR*); в частности, в состав данной группы регистров СОЗУ входят регистр признаков (РгП), известный также под названием регистра статуса (*Status Register, SR,*

или *Flags Register, FLAGS*), программный счетчик (*Program Counter, PC*) и указатель стека (*Stack Pointer, SP*) (см. рис. 2.1).

Следует отметить, что в состав ЦП большинства МК входят, кроме вышеперечисленных, и другие РСФ, состав и назначение которых будут рассмотрены в подразделах, посвященных конкретным семействам (подсемействам) МК (см., например, пункт 2.5.5). Необходимо также указать, что понятие «РСФ» включает в себя не только регистры специального назначения ЦП, но и программно-доступные регистры периферийных (по отношению к ЦП) блоков МК, например портов ввода-вывода, контроллеров прерываний, таймеров, блоков интерфейса и т. п. Состав, назначение и типовые форматы данных регистров будут рассмотрены в разделах, посвященных типовым структурно-архитектурным решениям соответствующих классов периферийных устройств МК.

Регистр статуса служит для фиксации характерных признаков результата выполнения очередной команды, таких как нулевой или ненулевой результат, наличие или отсутствие переполнения и т. п. Данные признаки используются при условных переходах (по нулевому результату, по отрицательному результату, по переполнению и т. п.). Некоторые из битов данного регистра могут использоваться для мониторинга состояния ЦП и для программирования режимов его работы, в частности, для глобального разрешения или запрета прерываний (см., например, рис. 2.23).

Программный счетчик (*PC*) предназначен для хранения адреса очередной команды. Он работает в 2-х основных режимах:

- при выполнении команд вызова подпрограмм, безусловных переходов и условных переходов (при выполнении условия) в *PC* загружается адрес первой команды подпрограммы или, соответственно, адрес перехода; в коде команды при этом содержится собственно значение адреса или данные, необходимые для его вычисления (см. приведенные в п. 2.6.3 описания режимов адресации);

- при выполнении всех остальных разновидностей команд (в том числе условных переходов при несоблюдении условия ветвления) адрес следующей команды вычисляется как сумма текущего

содержимого *PC*, т. е. адреса выполненной команды, и ее длины (обычно в байтах); при этом *PC* фактически работает как суммирующий счетчик, откуда и его название – *Program Counter*.

Изменения содержимого программного счетчика также, в принципе, может осуществляться специальными командами пользовательской программы. Однако, некорректные манипуляции с содержимым программного счетчика могут привести к весьма серьезным нарушениям работы МК и системы на его основе. Поэтому в ряде семейств МК программный счетчик не доступен для записи / чтения. При разработке же ПО семейств МК, программный счетчик которых доступен для изменения, необходимо соблюдать все возможные меры предосторожности для предотвращения некорректных изменений его содержимого.

Указатель стека (*SP*) служит для адресации специальной области, выделяемой в памяти и организуемой по принципу *LIFO* (*Last In – First Out*, «последним пришел – первым вышел»). Основное назначение данной области – хранение адресов возврата при вызове подпрограмм или при обслуживании прерываний (см. раздел 7), в том числе при выполнении вложенных подпрограмм или при обслуживании вложенных прерываний. Также архитектура большинства семейств МК предполагает возможность использования стека как области ПД, организованной по принципу «последним пришел – первым вышел» (*LIFO*), для хранения операндов и промежуточных результатов обработки данных. Организация памяти по такому принципу является предпочтительной при реализации ряда алгоритмов обработки данных. Запись данных в стек и чтение из стека осуществляется командами *PUSH* и *POP* соответственно, присутствующими в системе команд большинства семейств МК (см., например, табл. 2.4 и 2.5).

Содержимое *SP* равно адресу «вершины» стека, под которой понимается максимальный или минимальный из не занятых адресов области памяти, выделенной под стек. При этом *SP* указывает на максимальный из не занятых адресов стека, если он заполняется, начиная с максимального из выделенных под него адресов. При заполнении стека, начиная с минимального из адресов выделенной

под него области памяти, *SP* указывает на минимальный из не занятых адресов данной области. На практике встречаются оба варианта заполнения стека, в зависимости от конкретного типа / семейства ЦП.

Типовой пример использования стека приведен на рис. 2.2. Он иллюстрирует процесс обращения к подпрограмме с некоторым начальным адресом *SR1*, которая, в свою очередь, содержит обращение к другой (вложенной) подпрограмме с начальным адресом *SR2*. На рис. 2.2: *PROGRAM MEMORY* – память программ; *STACK* – содержимое стека на соответствующем этапе вычислительного процесса. Сплошными стрелками показан ход выполнения программы, а пунктирными – порядок изменения содержимого стека. В данном примере предполагается, что стек заполняется, начиная с максимального из адресов области памяти, выделенной под него.

Обращение к подпрограмме инициируется командой ее вызова (*CALL*). По данной команде на «вершину» стека, т.е. по адресу, равному содержимому *SP*, записывается адрес возврата из подпрограммы, т. е. адрес команды, следующей за командой вызова подпрограммы (на рис. 2.2 – *A1* или *A2* соответственно). После этого содержимое *SP* уменьшается на разрядность адреса в байтах (обозначенную на рис. 2.2 как *NA*), в результате чего содержимое *SP* становится равным адресу «вершины» стека с учетом новой записи в него. Затем в программный счетчик загружается начальный адрес вызываемой подпрограммы (*SR1* или, соответственно, *SR2*), и осуществляется переход к ней.

Подпрограмма должна заканчиваться специальной командой возврата из нее, *RET* (*RETURN*, возврат). По этой команде содержимое *SP* уменьшается на разрядность адреса в байтах, в результате чего оно становится равным адресу последнего слова, записанного в стек. Этим словом, в свою очередь, является адрес возврата в программный модуль, вызвавший соответствующую подпрограмму (см. рис. 2.2). Данный адрес загружается в программный счетчик, и осуществляется переход к команде, следующей за командой вызова подпрограммы. Новое содержимое

SP при этом указывает на адрес «вершины» стека с учетом «выгрузки» из него очередного слова.

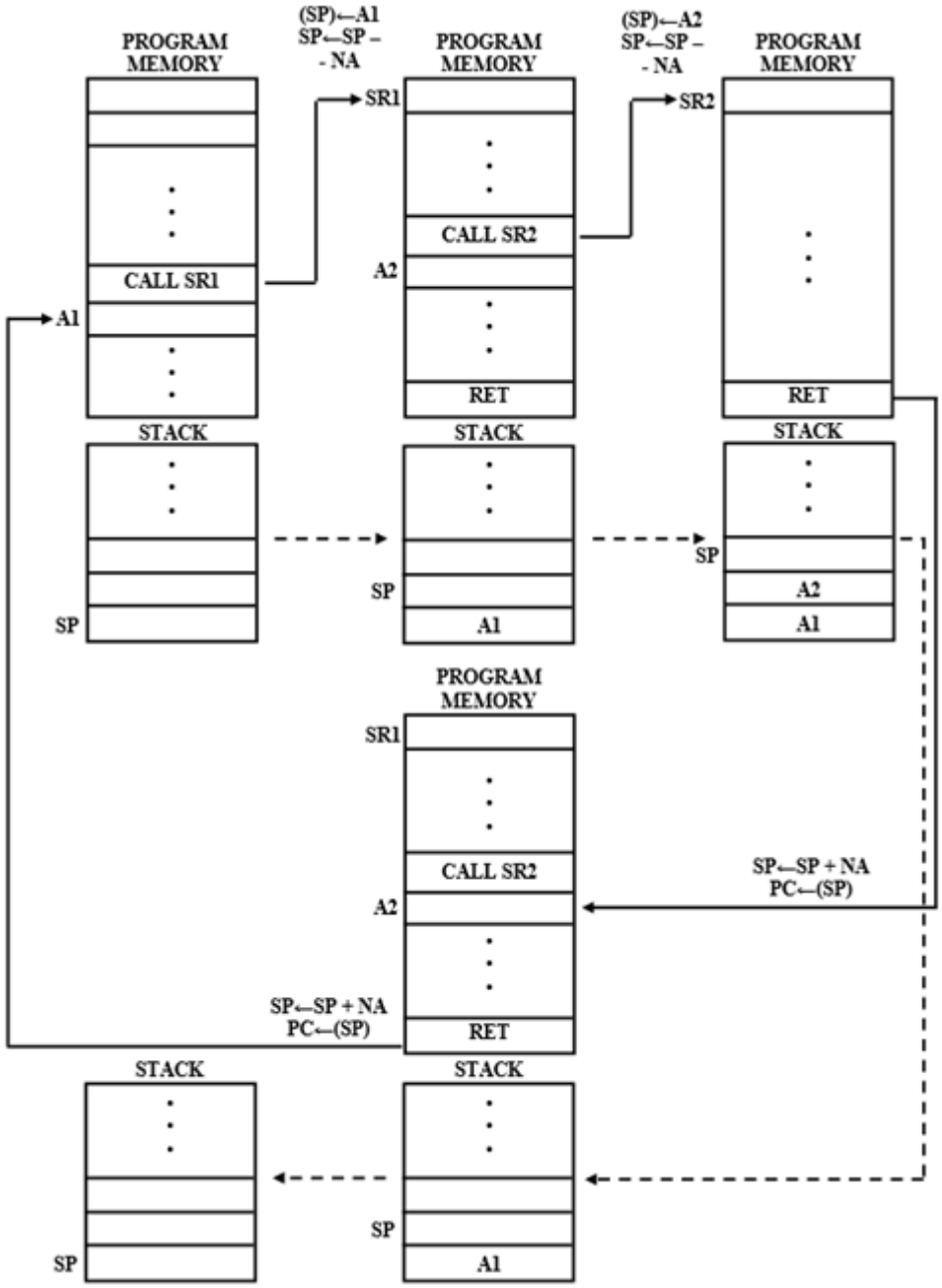
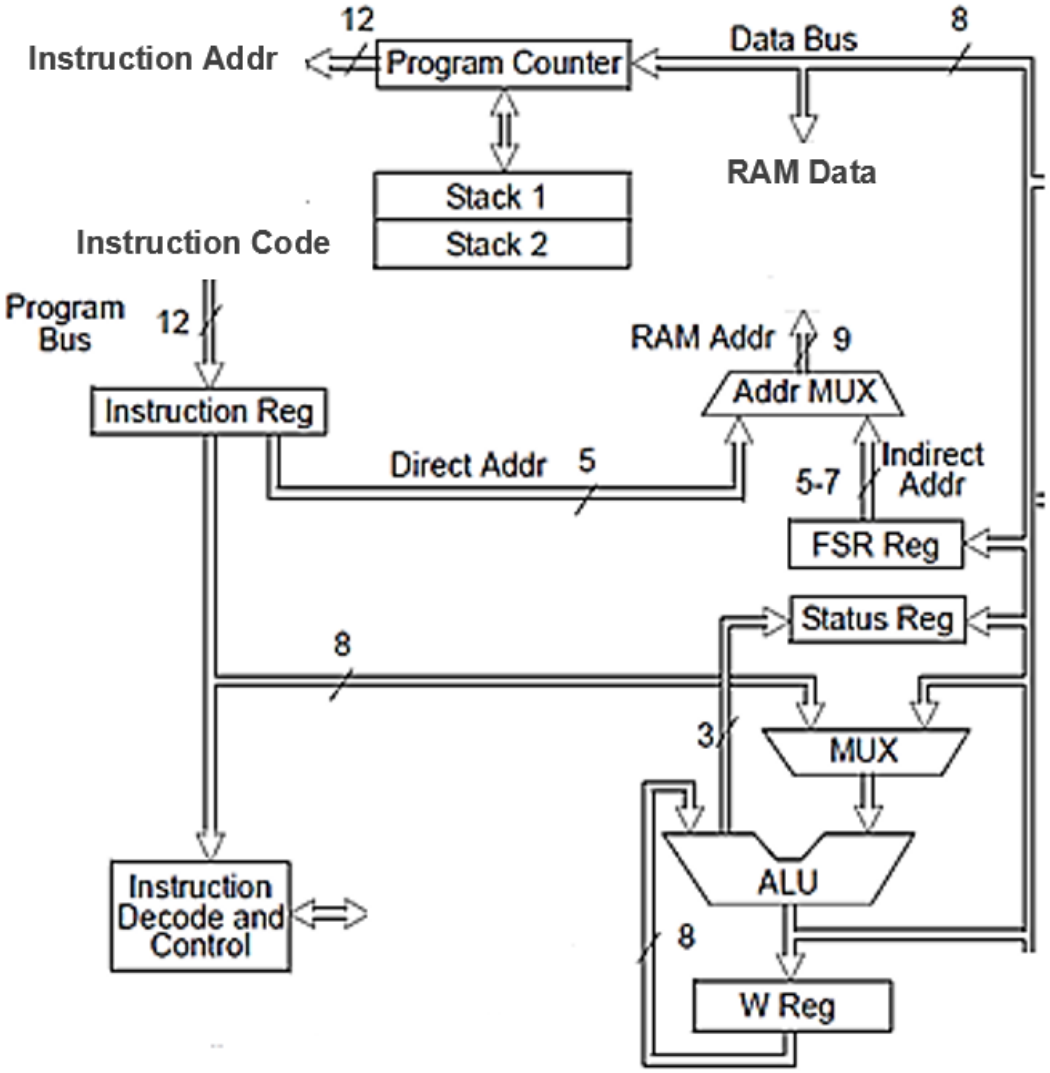


Рис. 2.2. Типовой пример применения стека (см. пояснения в тексте)

На рис. 2.3 – 2.5 представлены примеры структурных схем ЦП всех 3-х классов: на рис. 2.3 – МК *PIC16F505* (см. рис. 1.1), относящегося к младшему подсемейству *PIC*-МК (класс «*cost-sensitive*»); на рис. 2.4 – МК подсемейства *ATmega* (см. рис. 1.3) класса «*mainstream*»; на рис. 2.5 – упрощенная структурная схема

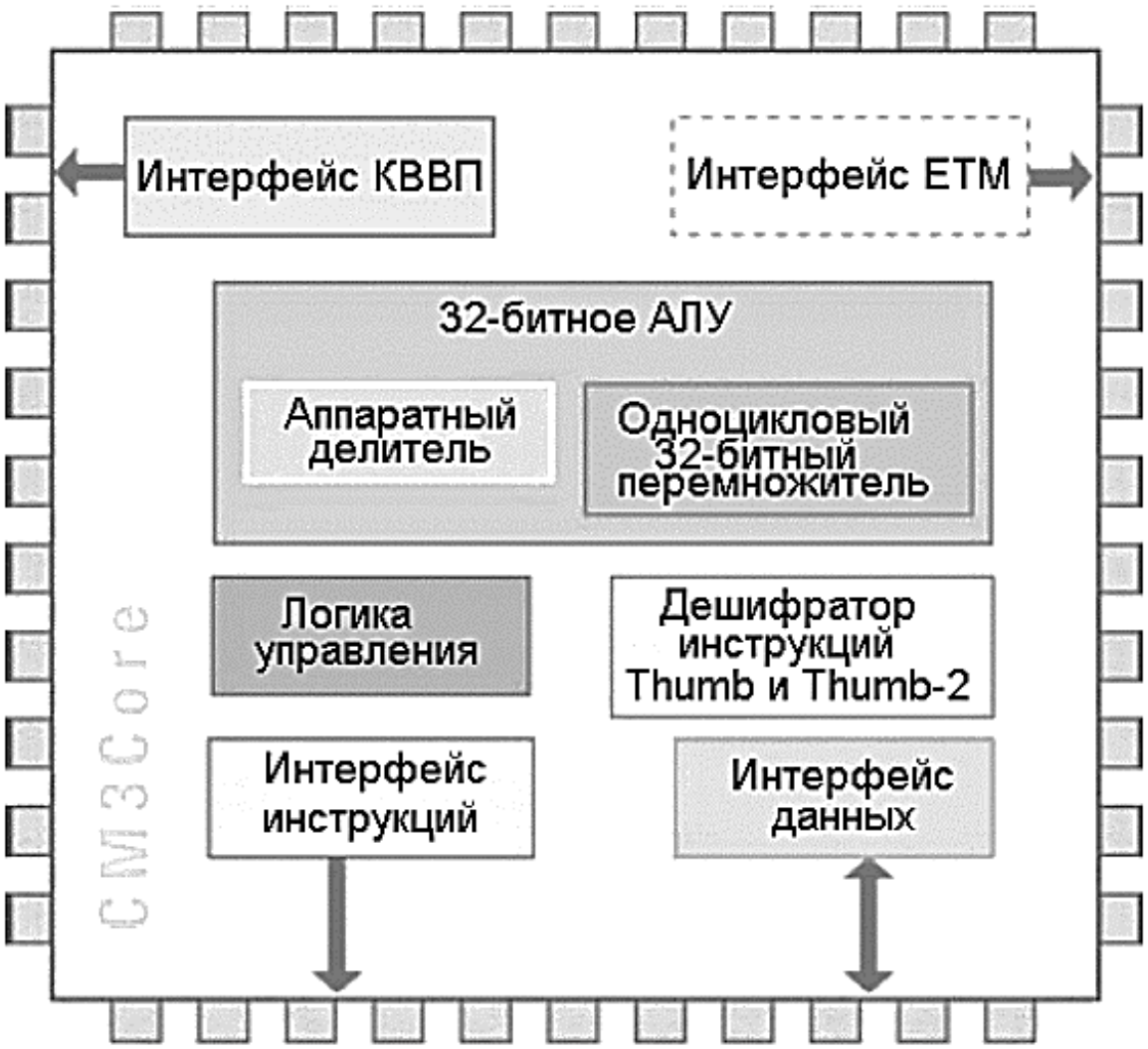
МК подсемейства *ARM Cortex-M3* (см. рис. 1.4 и 1.5) класса «*high performance*». Расшифровка названий узлов и блоков ЦП приведена в подрисуночных подписях (за исключением тех из них, названия которых совпадают с расшифрованными ранее в табл. 1.1 или в тексте).



Instruction Decode and Control – устройство управления (см. рис. 2.1);
FSR – регистр – указатель адреса (при косвенной адресации) и номера банка;
W Reg – регистр-аккумулятор (см. рис. 2.1);
MUX – мультиплексоры;
Direct Addr – прямой адрес;
Indirect Addr – косвенный адрес

Рис. 2.3. Структурная схема ЦП МК *PIC16F505* [4] (см. рис. 1.1)

Рис. 2.4. Структурная схема ЦП (ядра) МК подсемейства *ATmega* семейства *AVR* [8] (см. рис. 1.3)



КВВП – контроллер вложенных векторизированных прерываний
(векторный контроллер прерываний с возможностью вложения)
(см. пункт 7.3.2)
ETM – Embedded Trace Macrocell
(отладочный модуль)
Thumb и *Thumb-2* – системы команд МК семейства *ARM Cortex-Mx*
(см. пункт 2.6.5 и табл. 2.8)

Рис. 2.5. Упрощенная структурная схема ЦП (ядра) МК семейства *ARM Cortex-M3* [9] (см. рис. 1.4 и 1.5)

Сопоставляя структурные схемы, представленные на рис. 2.3 – 2.5 с обобщенной структурной схемой «классического» ЦП (см. рис. 2.1), можно сделать вывод, что, в целом, ЦП МК строятся в соответствии с ней, за исключением:

- отличий, обусловленных физическим разделением памяти программ и данных, характерных для Гарвардской архитектуры;

- наличия конвейера команд (в представленных на рис. 2.3 – 2.5 структурах присутствует неявно), благодаря которому, а также Гарвардской архитектуре, эффективное время выполнения большинства команд составляет 1 такт (подробнее – см. пункт 2.6.4).

В целом, приведенные на рис. 2.3 – 2.5 структурные схемы ЦП являются типовыми для большинства серийно выпускаемых в настоящее время МК общего назначения соответствующих классов.

2.3 Типовые режимы работы ЦП МК общего назначения

ЦП большинства МК общего назначения классов «*cost-sensitive*» и «*mainstream*» могут работать в следующих режимах:

- **основном режиме**;

- **энергосберегающих режимах**, в которые ЦП и МК в целом переводится специальной командой (у большинства семейств МК это команда *SLEEP*); как правило, существует несколько энергосберегающих режимов, которые различаются между собой составом «спящих» узлов и блоков МК, а также способами вывода МК из энергосберегающего режима.

Подробное рассмотрение энергосберегающих режимов работы МК представлено в разделе 3.

Некоторыми существенными особенностями отличаются режимы работы ЦП МК семейства *ARM Cortex-Mx*, относящихся, как указано ранее, к классу «*high performance*». Они, в отличие от ЦП более простых МК, имеет два не энергосберегающих режима работы [9]:

- **режим обработки исключительных ситуаций**, в том числе прерываний (*Handle*);

- **потоковый режим** (*Thread*), используемый при выполнении основной программы;

Переход в режим *Handle* осуществляется автоматически, по поступлении запроса на обработку исключительной ситуации (при условии, что данный переход не замаскирован, т. е. не запрещен). В свою очередь, переход в режим *Thread* также осуществляется автоматически, по окончании процедуры обработки исключительной ситуации.

Кроме того, для ЦП МК семейства *ARM Cortex-Mx* существуют понятия **привилегированного** и **непривилегированного** уровня выполнения программы [9]. Одно из основных их отличий – на привилегированном уровне программе доступны все регистры ЦП, на непривилегированном – только часть из них (см. приведенные в п. 2.5.8 описание регистровой модели МК данного семейства). В режиме *Thread* ЦП может работать как на привилегированном, так и на непривилегированном уровне (в зависимости от задаваемого программно состояния 0-го бита РСФ *CONTROL* ЦП, см. п. 2.5.8), в режиме *Handle* – только на привилегированном.

2.4 Типы и форматы данных МК общего назначения

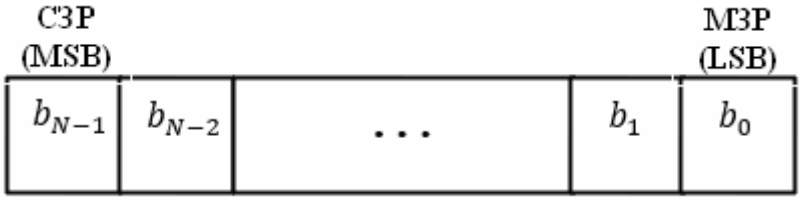
Процесс работы МК, в общем, представляет собой выполняемую под управлением ПО МК последовательность операций пересылки и обработки данных, т. е. хранящихся в различных блоках памяти (см. подраздел 2.5) двоичных чисел. Важно отметить, что под разрядностью МК подразумевается именно разрядность данных. Например, МК семейства AVR относятся к 8-битовым, поскольку разрядность их слов данных равна 8-и битам несмотря на то, что разрядность слов команд – 16 бит.

Обрабатываемые / пересылаемые данные относятся к одной из следующих категорий:

- двоичные (значительно реже – двоично-десятичные) числа, являющиеся операндами или результатами выполнения арифметических или логических операций;
- двоичные коды управления или состояния периферийных устройств.

2.4.1 Представление чисел в МК общего назначения

Системы команд большинства МК общего назначения классов «*cost-sensitive*» и «*mainstream*» предполагают, что операнды и результаты являются целыми *N*-битовыми двоичными числами, обобщенный формат которых представлен на рис. 2.6 [3]. По умолчанию полагается, что *N* – разрядность МК. Однако, системы команд большинства семейств МК с разрядностью большей 8-и бит (16- и 32-битовых) допускают разрядность операндов и результатов, меньшую разрядности МК, но, как правило, кратную 8-и битам, например, равную 8-и или 16-и битам для 32-битового МК.



СЗР (*MSB*) – старший значащий разряд (*Most Significant Bit*)
МЗР (*LSB*) – младший значащий разряд (*Least Significant Bit*)
$b_0 \ldots b_{N-1}$ – значения битов с соответствующими номерами

Рис. 2.6. Обобщенный формат *N*-битового целого двоичного числа

В зависимости от конкретной задачи обработки данных, целое двоичное число рассматривается как **беззнаковое** (*unsigned*) (если оно может быть только положительным или нулевым) или **знаковое** (*signed*) (если оно может быть как положительным, так и отрицательным). В табл. 2.1 представлены диапазоны беззнаковых и знаковых *N*-битовых целых двоичных чисел, а также выражения, описывающие зависимости значений этих чисел от состояний их битов. При этом предполагается, что отрицательные целые числа представляются в **дополнительном коде** (дополняют свой модуль до значения 2^N, откуда название). Такое представление практически без исключений применяется во всех аппаратных средствах ИТ, благодаря возможности выполнения операций вычитания посредством сложения уменьшаемого с представленным в дополнительном коде вычитаемым и, следовательно, отсутствию необходимости в специальных блоках вычитания.

Таблица 2.1

Представление целых двоичных чисел в МК

Тип числа	**Знак числа**	**Диапазон**	**Значение**	**Значение старшего бита,** b_{N-1}
Беззнаковое	Всегда считается положительным	От 0 до $2^N - 1$ (от 00...0 до 11…1)	$\sum_{i=0}^{N-1} b_i 2^i$	0 или 1, в зависимости от значения числа
Со знаком	Положительное	От 0 до $2^{N-1} - 1$ (от 00…0 до 01…1)	$\sum_{i=0}^{N-2} b_i 2^i$	0
	Отрицательное в дополнительном коде	От минус 2^{N-1} до минус 1 (от 10…0 до 11…1)	$-\left(2^N - \sum_{i=0}^{N-1} b_i 2^i\right)$	1

Большинство интегрированных сред разработки (*IDE*) ПО МК базируется на языке *C*, адаптированном к задачам программирования МК. Разрядность и форматы переменных при этом, как правило, не совпадают с разрядностью и форматами данных, используемыми системой команд МК. В *IDE* ПО МК применяются следующие основные типы переменных:

- знаковые и беззнаковые целочисленные переменные разрядностью 1 байт (*char, signed char, unsigned char*), 2 байта (*int, signed int, unsigned int*), 4 (*long int, unsigned long*);

- вещественные переменные с одинарной точностью (*float*) и с двойной точностью (*double*);

причем состав типов переменных и синтаксис их объявления могут различаться у различных *IDE*.

Целочисленные переменные после компиляции программного *C*-кода хранятся в памяти и представляются при обработке в форматах, аналогичных описываемым рис. 2.6 и табл. 2.1, за исключением того, что их разрядность может не совпадать с разрядностью МК. При этом вопросы размещения в памяти МК целых чисел различной разрядности освещены далее в пункте 2.5.2,

Вещественные переменные программ, разработанных в *IDE* ПО МК, представляются как двоичные числа с **плавающей точкой**, в соответствии со стандартом *IEEE-754*. При данном представлении число описывается следующим обобщенным выражением:

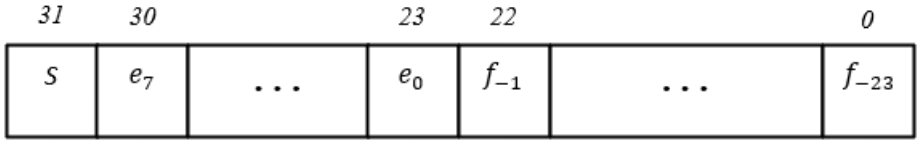
$$x = (-1)^S \times F \times 2^E; \qquad (2.1)$$

где s – знаковый бит, равный 0 для положительных чисел и 1 – для отрицательных;

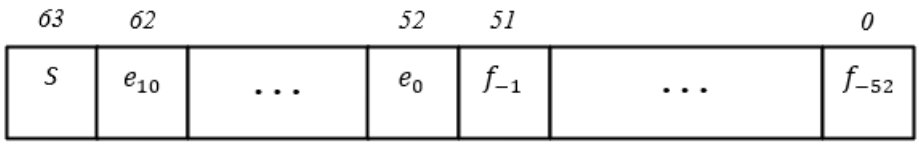
F – мантисса;

E – порядок.

Форматы представления вещественных чисел с одинарной и двойной точностью в соответствии со стандартом *IEEE-754* приведены на рис. 2.7а и 2.7б соответственно [3].



а)



б)

Рис. 2.7. Форматы представления вещественных чисел с одинарной (а) и с двойной (б) точностью в соответствии со стандартом *IEEE-754*

Разрядность вещественного числа с одинарной точностью – 32 бита, а с двойной точностью – 64 бита. Старший бит – знаковый (см. выражение (2.1)). Следующие за старшим 8 бит числа с одинарной точностью или, соответственно, 11 бит числа с двойной точностью содержат **смещенный порядок**, вычисляемый по выражению:

$$E_d = E + (2^{N_E - 1} - 1); \qquad (2.2)$$

где N_E – число битов, выделенных под смещенный порядок. Он всегда положителен, что упрощает аппаратную реализацию блоков обработки вещественных чисел.

Мантисса чисел, представляемых по стандарту *IEEE-754*, является нормированной, т. е. ее значение находится в пределах от 1,0…0 до 1,1…1 в двоичной системе счисления, благодаря чему все биты дробной части являются значащими, и обеспечивается наименьшая погрешность округления числа при заданной разрядности. Младшие 23 бита числа с одинарной точностью или, соответственно, младшие 52 бита числа с двойной точностью содержат модуль дробной части нормированной мантиссы. Целая часть по умолчанию равна 1, и под ее значение никаких битов не выделяется.

Модуль вещественного числа с ординарной точностью может находиться в диапазоне от 2^{-126} до 2×2^{127}, а с двойной точностью – в диапазоне от 2^{-1022} до 2×2^{1023}. Мантисса при этом представляется с точностью до 23-го и 52-го бита после двоичной точки соответственно, т. е. с абсолютной погрешностью округления, равной соответственно 2^{-23} и 2^{-52}.

Пример представления переменной, объявленной как *float*, в памяти МК семейства *AVR* приведен на рис. 2.8.

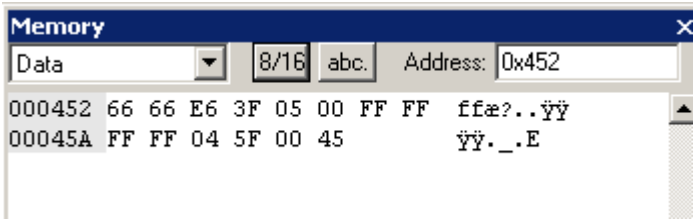


Рис. 2.8. Пример представления вещественной переменной типа *float* в памяти МК (см. пояснения в тексте)

Значение переменной равно 1,8 и расположено в памяти данных по адресам с *0x452* по *0x455*, начиная с младшего байта. В соответствии со стандартом *IEEE-754* данное значение представлено шестнадцатеричным числом *3FE66666*, которому соответствует двоичное число 00111111111001100110011001100110, то есть:

- знаковый бит равен 0;

- смещенный порядок равен 01111111 в двоичной системе счисления, т. е. 127 в десятичной, следовательно, в соответствии с выражением (2.2), учитывая, что в рассматриваемом случае N_E равно 8-и, несмещенный порядок числа равен 0;

- целая часть числа по умолчанию равна 1, при представлении по стандарту *IEEE-754* не отображается;

- дробная часть равна 0,11001100110011001100110 в двоичной системе счисления, т. е. приблизительно 0,79999995 в десятичной.

Следовательно, расположенное по адресам с *0x452* по *0x455* число равно $1{,}79999995 \times 2^0$, т. е. (с точностью до 23-го бита после двоичной точки (запятой)) 1,8.

Необходимо также отметить, что система команд ряда МК класса «*high performance*», в частности, МК подсемейства *ARM Cortex-M4* [16] включает в себя ряд математических операций над вещественными переменными с плавающей точкой, которые представляются как числа с одинарной точностью или с половинной точностью (*half-precision*). Первые из перечисленных представляются в формате, приведенном на рис. 2.7а, вторые – как 16-битовые двоичные числа в формате, представленном на рис. 2.9, также соответствующем стандарту *IEEE-754*.

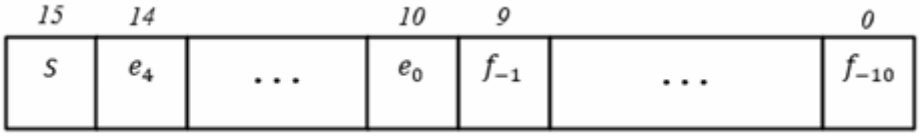


Рис. 2.9. Формат представления вещественных чисел с половинной точностью в соответствии со стандартом *IEEE-754*

Принцип представления аналогичен вышеописанному принципу представления вещественных чисел с одинарной и двойной точностью (см. выражения (2.1) и (2.2), рис. 2.7 и пояснения к ним), за исключением того, что под смещенный порядок выделяется 5 бит, а под дробную часть модуля нормированной мантиссы – 10 бит. При этом модуль вещественного числа с половинной точностью может находиться в диапазоне от 2^{-14} до 2×2^{15}. Модуль нормированной мантиссы при этом находится в пределах от 1,0000000000 до

1,1111111111 в двоичной системе счисления, и представляется с точностью до 10-го бита после двоичной точки соответственно, т. е. с абсолютной погрешностью округления, равной 2^{-10}.

2.4.2 Двоичные коды управления или состояния периферийных устройств МК

Коды управления периферийных устройств МК представляют собой формируемые ПО МК двоичные числа, загружаемые в регистры управления соответствующих устройств и задающие режимы их работы. В свою очередь, коды состояния представляют собой двоичные числа, считываемые из регистров состояния периферийных устройств и служащие для определения программным обеспечением МК наличия / отсутствия запросов на обслуживание; данных, подлежащих считыванию или передаче; ошибок приема / передачи и т. п.

Разрядность и форматы кодов управления и состояния совпадают с разрядностью и форматами вышеназванных регистров периферийных устройств, которые представлены в разделах, посвященных соответствующим классам данных устройств.

2.5 Типовые структурно-архитектурные решения памяти МК общего назначения

2.5.1 Состав памяти МК

В общем случае, резидентная (т. е. физически расположенная на кристалле) память МК включает в себя следующие типы ЗУ:

- регистровые ЗУ (СОЗУ);
- статическое ОЗУ;
- один или несколько модулей энергонезависимых ЗУ, обычно типа *FLASH*, в отечественной литературе также используется термин «Электрически стираемое перепрограммируемое ПЗУ» (ЭСППЗУ, см. рис. 1.5).

Следует отметить, что в структуре МК общего назначения практически не применяются динамические ОЗУ, использование

которых рационально при объемах памяти от сотен мегабайт и более [3], не характерных для МК.

На рис. 2.10 представлена иерархическая «пирамида» перечисленных выше типов ЗУ, ранжированных по объему и быстродействию [3]. По мере продвижения от основания «пирамиды» к ее вершине растет быстродействие ЗУ, а от вершины к основанию – объем ЗУ (см. направления стрелок на рис. 2.10). При тех же элементной базе и технологии и примерно при той же площади на кристалле регистровое ЗУ будет иметь максимальное быстродействие (откуда его второе распространенное название – сверхоперативное ЗУ, СОЗУ), но минимальный объем, а энергонезависимое ЗУ – максимальный объем, но минимальное быстродействие.

Каждый из типов ЗУ, применяемых в МК, используется для решения круга задач, определяемого свойствами ЗУ соответствующего типа.

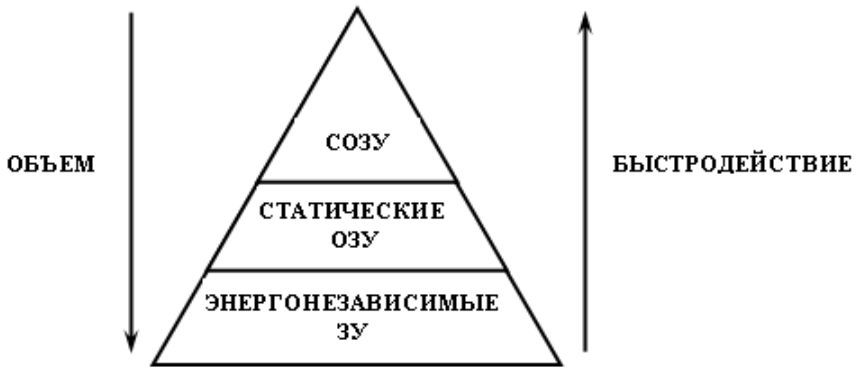


Рис. 2.10. Иерархия ЗУ МК по объему и быстродействию

Регистры СОЗУ, ввиду высокого быстродействия, но ограниченного их количества на кристалле, используются для хранения операндов и результатов непосредственно выполняемого фрагмента программы, для программирования ряда системных функций (например, для задания области стека), а также для связи с периферийными устройствами. Схемотехнически регистры СОЗУ, как правило, представляют собой совокупность триггеров с объединенными входами синхронизации и с параллельной или параллельно-последовательной записью данных.

В относительно простых МК класса *cost-sensitive* с объемом ПД порядка сотен байт СОЗУ может выполнять и ее функции в целом, что позволяет достичь потенциально наивысшего быстродействия выборки и записи данных при приемлемых аппаратурных затратах.

Регистры СОЗУ разделяются на две группы:

- программно-доступные, содержимое которых может считываться и записываться (или только считываться) пользовательским ПО;

- недоступные пользовательскому ПО ни для чтения, ни для записи (программно-недоступные), к ним относится, например, регистр команд.

Поскольку излагаемый материал посвящается моделям ЦП и памяти МК с точки зрения разработчика электронных средств на базе МК, в том числе ПО МК, ограничимся рассмотрением типового состава программно-доступных регистров (который, на взгляд автора, не совсем корректно, в литературе часто называют программной моделью МК).

Программно-доступные регистры, как указано ранее, разделяются на две основные группы:

- регистры общего назначения (РОН), в англоязычных источниках – General Purpose Registers (*GPR*);

- регистры специальных функций (РСФ), в англоязычной литературе – *Special Function Registers* (*SFR*), к которым относятся регистры периферийных устройств, а также некоторые программно-доступные регистры ЦП, в частности, регистр статуса, программный счетчик и указатель стека.

Каждому программно-доступному регистру присваивается определенный адрес и, как правило, некоторое имя-идентификатор, используемое при разработке ПО МК в определенной среде проектирования (*IDE*).

Статические ОЗУ используются в качестве ПД при ее объеме от нескольких килобайт и более. Следует, однако, отметить, что время обращения к ПД на основе статического ОЗУ, как правило, больше, чем к СОЗУ. Поэтому в фрагментах программ, критичных с точки зрения времени их выполнения, предпочтительно использовать регистры СОЗУ для хранения операндов и результатов.

Энергонезависимые ЗУ служат в качестве ПП, а также для хранения данных, которые не должны теряться при выключении питания (констант, уставок и т. п.).

Для большинства семейств МК общего назначения характерна **Гарвардская архитектура** [3], т. е. физическое разделение памяти программ (ПП) и памяти данных (ПД). При этом, в зависимости от архитектуры конкретного семейства, ПП и ПД могут как занимать различные области адресов в пределах общего адресного пространства, так и области с одинаковыми адресами в физически разделенных ЗУ.

Т**иповой состав** резидентных модулей памяти современного МК общего назначения следующий (см. рис. 1.1 – 1.5):

- энергонезависимая ПП, в которой, как правило, выделяются определенные области под векторы прерываний, под загрузочные сектора, а также под некоторые другие специальные цели (см. далее пункт 2.5.2);
- СОЗУ, включающее в себя РОН и РСФ, количество и состав которых зависит от конкретного семейства и / или модели МК;
- статическое ОЗУ данных (отсутствует у ряда моделей МК класса *cost-sensitive*);
- энергонезависимая ПД (у ряда семейств / моделей МК отсутствует как отдельный блок, в ее качестве используется часть ПП, не задействованная под прикладное ПО);
- ряд энергонезависимых ЗЭ системного назначения, предназначенных для хранения слов конфигурации МК, кодов идентификации и т. п.

Следует отметить, что архитектура большинства семейств современных МК, особенно классов «*mainstream*» и «*high performance*», позволяет, при необходимости, дополнить память МК внешними (по отношению к нему) модулями. На практике такой прием применяется достаточно редко, т. к. в современной электронике массогабаритные параметры устройства более важны, чем стоимость МК. Поэтому рациональнее использовать МК с б**о**льшим объемом резидентной памяти, чем дополнять недорогие МК внешними модулями памяти.

2.5.2. Размещение команд и данных в памяти МК

Как правило, минимальной адресуемой единицей информации в памяти МК является байт, т. е. каждому байту в ПП и в ПД присваивается уникальный (в пределах определенного модуля памяти) адрес. В зависимости от порядка расположения байтов многобайтного слова в памяти различают [3]:

- *Little Endian Format* («От младшего к старшему»), на профессиональном жаргоне – «Остроконечник», при котором по младшему из адресов, выделенных под слово, располагается его младший байт; последующие байты располагаются в порядке возрастания их номеров в слове; адресом слова служит адрес его младшего байта;

- *Big Endian Format* («От старшего к младшему»), на профессиональном жаргоне – «Тупоконечник», при котором по младшему из адресов, выделенных под слово, располагается его старший байт; последующие байты располагаются в порядке убывания их номеров в слове; адресом слова служит адрес его старшего байта.

Примеры вышеперечисленных форматов для 32-битового слова представлены на рис. 2.11.

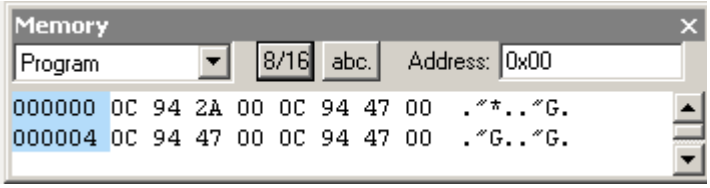
В целом, *Little Endian Format* более распространен на практике. В качестве типовых примеров данного формата на рис. 2.12 представлены фрагменты ПП и ПД МК семейства *AVR*. При этом в ПП по адресу 00000000 хранится команда из 2-х 16-битовых слов с шестнадцатеричным кодом *940C* и *002A* соответственно; в ПД по адресам 000456 и 000457 – переменная типа *int* с разрядностью 2 байта и значением 0005, причем слову переменной в целом присвоен адрес 000456. Подробнее особенности адресации ПП и ПД МК семейства *AVR* описаны в пункте 2.5.5.

Ряд семейств МК класса «*high performance*», в частности, семейства *ARM*, допускают использование как формата *Little Endian*, так и *Big Endian*. Выбор формата осуществляется программным путем, установкой / сбросом соответствующего бита в одном из управляющих регистров.

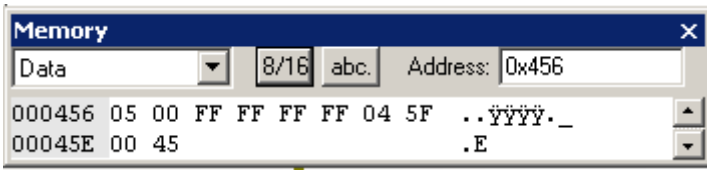
а)

б)

Рис. 2.11. Примеры расположения многобайтных слов в памяти в формате *Little Endian* (а) и *Big Endian* (б)



а)



б)

Рис. 2.12. Примеры расположения кодов команд (а) в ПП и данных в ПД (б) МК семейства *AVR* (см. пояснения в тексте)

2.5.3. Общие вопросы организации адресного пространства и адресации памяти МК

Адресация ПП осуществляется посредством программного счетчика, у ряда семейств / моделей МК – в сочетании с выделенными для данной цели РСФ (см. далее). Адреса ПД или исходные данные для их вычисления указываются в адресных полях кодов команд. Адресация ПД может также осуществляться посредством указателя стека.

Простейшим вариантом организации адресного пространства памяти является ее линейная («плоская») модель [3], при которой все адресное пространство ПП или ПД представляет собой единый массив ЗЭ, без разбиения на страницы, сегменты и т. п., адресуемый напрямую программным счетчиком или адресным полем кода команды соответственно. Например, если объем ПП некоторого МК равен 4096-ти словам команд, а ее адресация осуществляется программным счетчиком разрядностью $log_2 4096$, т. е. 12 бит, без использования каких-либо дополнительных РСФ или РОН, то модель ПП данного МК является линейной («плоской»).

«Плоская» модель памяти обеспечивает потенциально наивысшее быстродействие выполнения программ и простоту программирования, ввиду отсутствия необходимости в процедурах переключения между страницами или сегментами памяти. Однако, для организации ПП по такой модели необходимо, чтобы разрядность программного счетчика была не меньше, чем $log_2(V_{PM})$, где V_{PM} – объем ПП (в словах команд), а для реализации «плоской» модели ПД – чтобы разрядность адреса операнда в коде команды была не менее $log_2(V_{DM})$, где V_{DM} – объем ПД (в словах данных). Выполнение этих условий для ряда семейств МК, особенно класса *cost-sensitive*, сложно или невозможно. В таких случаях применяются другие способы организации адресного пространства, из которых наиболее распространены [3]:

- разбиение памяти на **банки**, адресуемые способом **переключения банков** (***bank switching***);

- **сегментная** организация памяти.

Организация памяти **первым** из перечисленных способов состоит в следующем. Память (ПП или ПД) разбивается на блоки фиксированного объема, называемые банками или страницами. Их размер и начальные адреса постоянны (фиксированы) и определяются архитектурой конкретной модели МК. Адрес ЗЭ состоит из 2-х частей: номера банка (старшая часть) и адреса ЗЭ в его пределах (младшая часть). Переключение банков / страниц осуществляется с помощью некоторого программно-доступного РСФ, содержащего старшую часть адреса ЗЭ, а адресация в пределах банка / страницы – программным счетчиком или, соответственно, полем адреса операнда в коде команды.

Типовые примеры организации адресного пространства ПП и ПД с использованием механизма переключения банков представлены в пункте 2.5.6 (см. рис. 2.13, 2.14 и 2.16, а также пояснения к ним).

Большей гибкостью характеризуется **сегментная** организация памяти МК. Принцип ее состоит в следующем. В пределах адресного пространства памяти выделяется определенное количество сегментов фиксированного или переменного размера; при этом число и размер сегментов зависят от архитектуры семейства или модели МК. Начальный адрес каждого из сегментов записывается в закрепленный за ним сегментный регистр, входящий в состав ЦП. При сегментной организации ПП программный счетчик содержит смещение адреса команды относительно начала содержащего ее сегмента. При сегментной организации ПД код команды, в общем случае, содержит номер сегмента, в котором находится операнд, и смещение адреса операнда относительно начала сегмента. Физический адрес команды или операнда вычисляется как сумма содержимого сегментного регистра и смещения адреса команды (операнда) относительно начала сегмента.

Сегментные регистры являются программно-доступными, поэтому границы сегментов могут оперативно изменяться в процессе работы. При этом каждый из сегментов может быть выделен для определенной цели, например, один – под глобальные переменные, один – под локальные и т. п.

В частности, по сегментному принципу организована ПД МК КР1878ВЕ1 (см. пункт 2.5.6).

Примечание. Несколько усложненный по сравнению с вышеописанным вариант сегментной организации памяти применяется в МП общего назначения при **многозадачном** режиме работы. Рассмотрение данного вопроса выходит за рамки настоящего пособия; интересующиеся лица могут ознакомиться с ней, например, по источнике [3].

Необходимо также остановиться на адресации по принципу ***wrap-around***, достаточно широко применяемой во многих семействах МК при обращениях к физически не существующим областям памяти (см., например, [4]). У ряда моделей МК разрядность программного счетчика больше необходимой для адресации физически существующей ПП или / и разрядность адресного поля кода команды больше необходимой для адресации физически существующей ПД. Такой «запас» по разрядности в ряде случаев обеспечивает совместимость с ПО, разработанным для моделей МК того же семейства с б**о**льшим объемом ПП или ПД. Например, при разрядности программного счетчика некоторого семейства МК, равной 12-ти битам, теоретически может осуществляться обращение по адресам от *000h* до *FFFh* в шестнадцатеричной системе счисления. Реальный объем резидентной ПП конкретной модели МК, принадлежащего данному семейству, может быть при этом равен 1024 12-битовых слов, поэтому максимальный физически существующий адрес равен *3FFh*. Область памяти с адресами, б**о**льшими, чем максимальный из реально существующих, называется *Unimplemented* (нереализованная, не существующая). Обращение по какому-либо из адресов данной области приведет к адресации ***wrap-around,*** т. е. по физическому адресу, равному остатку от деления адреса обращения на объем физически существующей области ПП. Например, при максимальном адресе физически существующей ПП, равном *3FFh* (1023), т. е. при объеме физически существующей ПП, равном 1024 (*400h*) слов, результатом обращения по адресу, равному *410h* (1040) будет обращение по физическому адресу, равному *010h* (16).

Адресация по принципу *wrap-around* применяется и при обращениях по физически не существующим адресам ПД (см. пункт 2.5.6, рис. 2.16).

Типовые примеры организации адресного пространства МК приведены в пунктах 2.5.6 – 2.5.8.

2.5.4. Общие вопросы организации памяти программ

Как указано ранее, память большинства семейств МК общего назначения организована по **Гарвардской архитектуре**. Диапазоны адресов, выделенных ПП и ПД, могут частично или полностью совпадать, но при этом ПП и ПД, включая аппаратные средства доступа к ним, физически разделены; различны и процедуры доступа к ним на программном уровне.

ПП, как правило, является энергонезависимой, как правило – типа *FLASH* (ЭСППЗУ). Структура подавляющего большинства современных МК включает в себя модуль **резидентной ПП** (*On-chip Program Memory*) объемом от сотен байт – единиц килобайт (МК класса «*cost-sensitive*») до сотен килобайт – единиц мегабайт (МК класса «*high performance*»). При необходимости, ПП большинства семейств МК может быть расширена за счет подключения внешних модулей.

В общем случае, резидентная ПП МК включает в себя следующие области:

- таблицу векторов прерываний;
- область размещения пользовательского ПО (*User Memory Space, Application Flash Section*);
- загрузочный сектор (*Boot Section*).

Некоторая область ПП МК всех классов и практически всех моделей выделяется под **таблицу векторов прерываний. Вектором прерываний** называют идентификационный номер, закрепленный за источником запросов на прерывания (см. табл. 1.1), а **таблицей векторов прерываний** – область памяти, содержащую начальные команды подпрограмм обслуживания прерываний от каждого из источников или начальные адреса данных подпрограмм [3]. Таблица векторов прерываний МК общего назначения, как

правило, строится следующим образом. За каждым из векторов прерываний закрепляется определенный адрес, по которому должна быть размещена начальная команда подпрограммы обслуживания соответствующего прерывания; обычно это команда безусловного перехода по начальному адресу размещения собственно подпрограммы обслуживания запроса на прерывание от соответствующего источника. См. также раздел 7.

Область размещения **пользовательского ПО** является основной. Архитектура ряда семейств МК, в составе которых не имеется специально выделенного модуля энергонезависимой ПД, предусматривает использование в его качестве части ПП, не задействованной под прикладное ПО.

Загрузочный сектор предназначен для размещения ПО, управляющего самопрограммированием, т. е. загрузкой кода в область размещения пользовательского ПО. Данная функция не реализуется в большинстве моделей МК класса «*cost-sensitive*».

Минимальной адресуемой единицей информации в ПП МК, как правило, является слово команды. Разрядность ЗЭ ПП, в зависимости от конкретного семейства МК, может быть равной или разрядности слова команды или одному байту. При однобайтной разрядности ЗЭ команды, как правило, размещаются в ПП в формате *Little Endian* (см., например, рис. 2.12а)

Типовые примеры организации ПП МК общего назначения различных классов рассмотрены далее, в пунктах 2.5.6 – 2.5.8.

2.5.5. Общие вопросы организации памяти данных

Как указано ранее, ПД, в общем случае, включает в себя следующие модули:

- СОЗУ, включающее в себя РОН и РСФ, количество и состав которых зависит от конкретного семейства и / или модели МК; при этом в состав РСФ практически всех семейств МК входят программный счетчик и регистр статуса, а также (кроме ряда подсемейств МК класса *cost-sensitive*) указатель стека;

- статическое ОЗУ данных (отсутствует у ряда моделей МК класса *cost-sensitive*);

- энергонезависимая ПД (у ряда семейств / моделей МК отсутствует как отдельный блок, в ее качестве используется часть ПП, не задействованная под прикладное ПО).

Диапазоны адресов, выделенные модулям ПД, у ряда семейств МК частично или полностью совпадают по значениям с адресами ПП (см. пункты 2.5.6 и 2.5.7), а например, у МК семейства *ARM Cortex-Mx* для ПП и ПД выделены различные адресные пространства (см. пункт 2.5.8). Как при совпадении, так и при несовпадении значений адресов, выделенных под ПП и ПД, у всех МК с Гарвардской архитектурой данные модули памяти физически разделены; различны и процедуры доступа к ним на программном уровне. Также могут частично или полностью совпадать диапазоны адресов, выделенных под энергозависимую и энергонезависимую ПД, но процедуры доступа к ним при этом тоже различаются.

Адреса РОН и РСФ, как правило, входят в общее адресное пространство ПД, в пределах которого под них выделены определенные диапазоны адресов (см. пункты 2.5.6 – 2.5.8). При этом у ряда семейств МК, например, *ARM Cortex-Mx*, понятие адреса РОН отсутствует; обращения к РОН возможны только по присвоенным им именам (идентификаторам).

Минимальной адресуемой единицей информации в ПД МК, как правило, является байт, т. е. каждому байту присваивается уникальный адрес в пределах модуля ПД, в котором он хранится. Многобайтные слова данных, как правило, хранятся в формате *Little Endian* (см. рис. 2.11а). У ряда семейств МК, например, *ARM Cortex-Mx*, определенные области ПД допускают адресацию к отдельным битам ЗЭ (см. пункт 2.5.8).

Типовые примеры организации ПД МК общего назначения различных классов рассмотрены далее, в пунктах 2.5.6 – 2.5.8.

2.5.6. Типовые примеры организации памяти и регистровых моделей МК класса *cost-sensitive*

В качестве типовых примеров МК класса *cost-sensitive* могут служить:

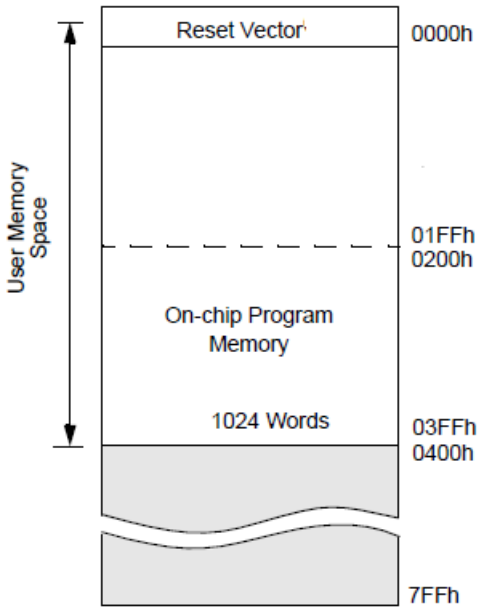
- МК *PIC16F505* [4] (см. рис. 1.1 и табл. 1.1), один из простейших МК данного класса (и МК в целом), являющийся 8-битовым *PIC*-МК младшего подсемейства с объемом резидентной ПП 1024 12-битовых слов команд, резидентной ПД – 72 8-битовых РОН; состав периферийных устройств – два 6-битовых порта ввода-вывода, один 8-битовый таймер / счетчик общего назначения, один сторожевой таймер;

- МК КР1878ВЕ1 (см. рис. 1.2 и табл. 1.1), по архитектуре близкий к среднему подсемейству *PIC*-МК [5].

Схема организации резидентной **ПП** (*On-chip Program Memory*) **МК *PIC16F505*** представлена на рис. 2.13. Она состоит из 2-х страниц (банков), адреса ЗЭ которых (в шестнадцатеричной системе счисления) находятся в диапазонах от *0000h* до *01FFh* и от *0200h* до *03FFh* соответственно. Разрядность ЗЭ равна разрядности команды (12-ти битам).

Адресация ПП осуществляется способом переключения банков (страниц). В процессе работы программы номер страницы задается 5-м битом регистра статуса (доступен для записи), а адрес команды на странице - младшим байтом программного счетчика (РСФ *PCL*). Переход со страницы на страницу осуществляется с помощью команд *GOTO* («Безусловный переход»), *CALL* («Вызов подпрограммы») или изменения содержимого программного счетчика (см. рис. 2.14). При этом переход в пределах страницы с одной ее половины на другую, т. е. изменение 8-го бита программного счетчика, может быть реализован только командой *GOTO*.

Разрядность программного счетчика МК *PIC16F505* равна 12-ти битам. Поэтому в процессе работы МК теоретически возможно обращение к физически не существующей области адресов, обозначенной на рис. 2.13 серым цветом, которое реально приведет к адресации ***wrap-around*** (см. пункт 2.5.3).



Reset Vector – адрес размещения начальной команды подпрограммы обслуживания прерывания по сбросу.
Серым цветом обозначено адресное пространство, физически не существующее, но к которому теоретически может осуществляться обращение (см. пояснения в тексте)

Рис. 2.13. Организация ПП МК *PIC16F505* [4]

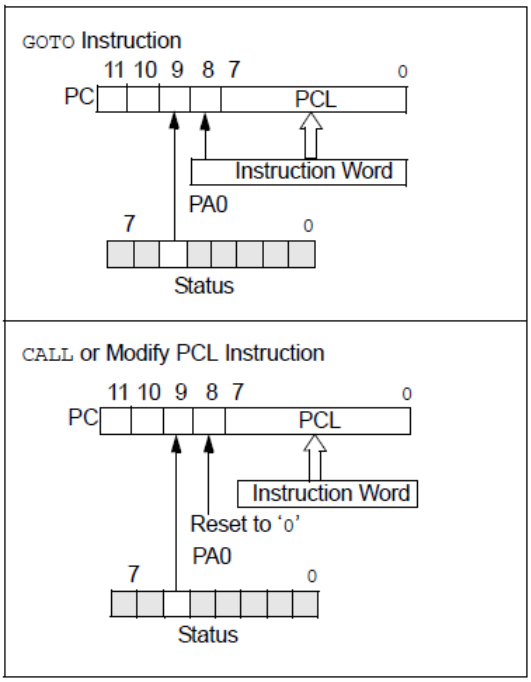


Рис. 2.14. Схемы загрузки программного счетчика МК *PIC16F505* при безусловном переходе и при вызове подпрограммы [4]

В свою очередь, модель **ПП МК КР1878ВЕ1** [5] является «плоской» и состоит из одной страницы объемом 1024 ЗЭ, адресуемой непосредственно программным счетчиком. Разрядность ЗЭ ПП равна разрядности кода команды (16-ти битам).

Начальная область ПП как МК *PIC16F505,* так и КР1878ВЕ1 выделена под таблицу векторов прерываний. У первого из них имеется только один источник прерываний – сброс (*Reset*), с номером вектора, равным 0. Поэтому таблица векторов прерываний МК *PIC16F505* занимает только один адрес ПП – нулевой, по которому должна быть расположена первая команда, выполняемая по сбросу МК.

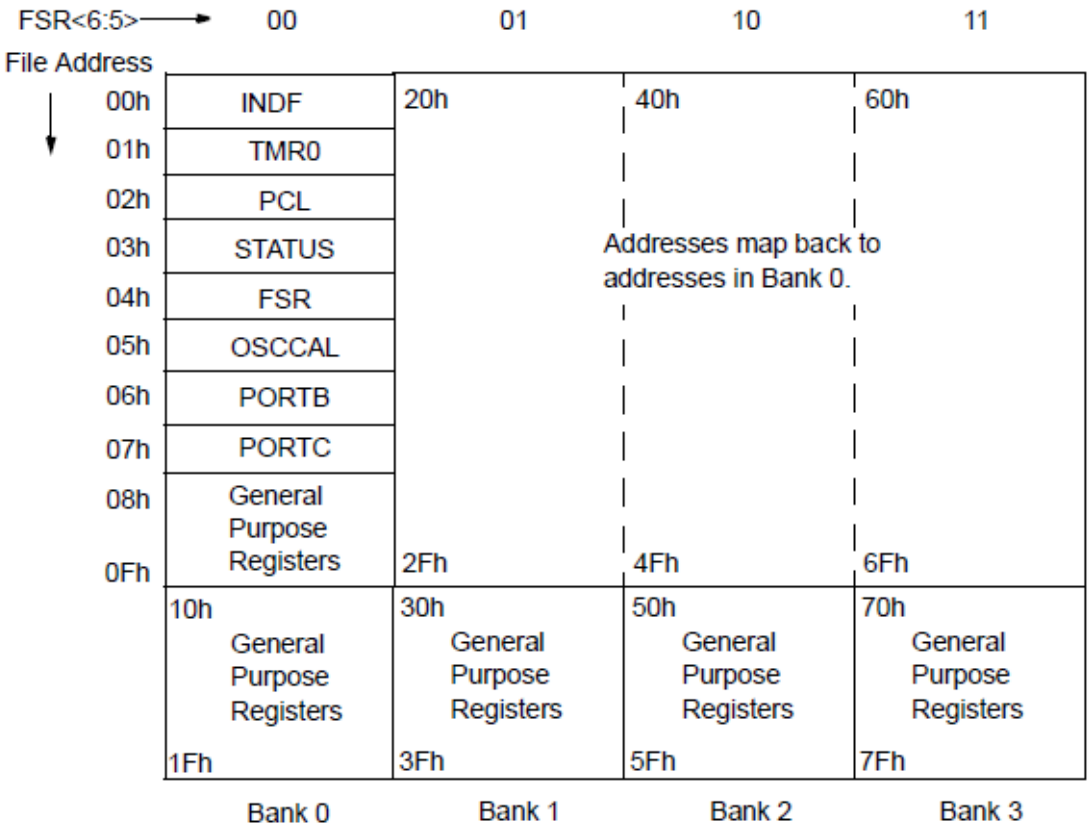
Таблица векторов прерываний МК КР1878ВЕ1 представлена на рис. 2.15. Номера векторов прерываний данного МК совпадают с адресами ПП, по которым должны быть расположены начальные команды подпрограмм обслуживания прерываний от соответствующих источников. Как указано в пункте 2.5.4, на практике это команды безусловного перехода по начальному адресу размещения собственно подпрограммы обслуживания соответствующего прерывания.

Вектор прерывания	Источник прерывания
0	Начальный пуск
1	Сторожевой таймер
2	Переполнение стека команд или данных
3	Интервальный таймер
4, 5	Резерв
6	Порт А
7	Порт В
$8 \div E_{16}$	Резерв
F_{16}	Завершение записи в блок ЭСППЗУ

Рис. 2.15. Таблица векторов прерываний МК КР1878ВЕ1 [5]

Схема организации **резидентной ПД МК *PIC16F505*** представлена на рис. 2.16 [4]. Она включает в себя только СОЗУ

(статическое ОЗУ отсутствует), разбитое на 4 банка. Адресация ПД осуществляется способом переключения банков (см. пункт 2.5.3); номер банка указывается в 5-м и 6-м битах регистра *FSR*. Адреса *00h – 07h* 0-го банка выделены под РСФ, остальные адреса – под РОН. Разрядность РОН и РСФ равна разрядности МК, т. е. 8-и битам. Области ПД с адресами *20h – 2Fh, 40h – 4Fh* и *60h – 6Fh* физически не существуют, при обращении к ним реально происходит переключение на область памяти с адресами *00h – 0Fh* способом *wrap-around* (см. рис. 2.16).



INDF – физически не существующий регистр, указание которого в адресном поле команды служит признаком косвенной адресации операнда;
TMR0 – регистр данных таймера-счетчика;
PCL – младший байт программного счетчика;
STATUS – регистр статуса;
FSR – регистр – указатель адреса (при косвенной адресации) и номера банка;
OSCCAL – калибровочное число внутреннего *RC*-генератора;
PORTB, PORTC – регистры данных порта *B* и *C*

Рис. 2.16. Схема организации ПД МК *PIC16F505* [4]

Структура **резидентной энергозависимой ПД МК КР1878ВЕ1** представлена на рис. 2.17 [5]. Она включает в себя ряд РСФ с адресами от *00h* до *3Fh* и ОЗУ данных, под которое выделены адреса от *40h* до *BFh*. Разрядность РСФ и ЗЭ ОЗУ равна 8-и битам (т. е. разрядности МК).

$Адрес_{16}$	Устройство	$Адрес_{16}$	Устройство
0	Регистр состояния процессора	1B, 1C	Свободный адрес
1	Рабочий регистр порта A	1D	Регистр управления сторожевого таймера
2	Рабочий регистр порта B	1E ÷ 37	Свободные адреса
3	Свободный адрес	38	Регистр управления блока ЭСППЗУ
4	Регистр управления таймера	39	Регистр адреса блока ЭСППЗУ
5	Рабочий регистр таймера	3A ÷ 3E	Свободные адреса
6 ÷ 18	Свободные адреса	3F	Регистр данных блока ЭСППЗУ
19	Регистр конфигурации порта A	40 ÷ **B** F	Оперативная память
1A	Регистр конфигурации порта B	C0 ÷ FF	Свободные адреса

Рис. 2.17. Структура энергозависимой ПД МК КР1878ВЕ1 [5]

Адресация ПД МК КР1878ВЕ1 осуществляется по **сегментному** принципу. В пределах адресного пространства ПД выделяются 4 сегмента размером по 8 байт каждый. Старшие 5 бит начального адреса каждого из них записываются в соответствующий сегментный регистр, входящий в состав ЦП; младшие 3 бита начального адреса сегмента всегда равны 0. Код команды при этом содержит 2-битовый номер сегмента, в котором находится операнд, и 3-битовое смещение адреса операнда относительно начала сегмента. Физический адрес операнда вычисляется как сумма содержимого сегментного регистра, умноженного на 8, и смещения адреса операнда относительно начала сегмента. Сегментные регистры являются программно-доступными, поэтому границы сегментов могут оперативно изменяться в процессе работы.

Пример формирования физического адреса ПД МК КР1878ВЕ1 приведен на рис. 2.18. В данном примере операнд находится в 1-м сегменте, начальный адрес сегмента (содержимое 1-го сегментного регистра, умноженное на 8) равен 10111000 в двоичной системе

счисления, т. е. *B8h*, смещение адреса операнда относительно начала сегмента равно 4-м, физический адрес операнда – *BCh*.

Энергонезависимая ПД у МК *PIC16F505* отсутствует, у МК КР1878ВЕ1 представляет собой ЭСППЗУ объемом 64 8-битовых слов. Запись и чтение ЭСППЗУ осуществляются с использованием 3-х РСФ: регистра адреса, регистра данных и регистра управления блока ЭСППЗУ. Процедуры записи и чтения описаны в [5].

Рис. 2.18. Пример формирования физического адреса ПД МК КР1878ВЕ1 (см. пояснения в тексте)

Формат **регистра статуса** МК *PIC16F505* представлен на рис. 2.19, а МК КР1878ВЕ1 – на рис. 2.20.

R/W-0	R/W-0	R/W-0	R-1	R-1	R/W-x	R/W-x	R/W-x
RBWUF	—	PA0	$\overline{TO}$	$\overline{PD}$	Z	DC	C
bit 7							bit 0

C и *DC* – признаки переноса из старшего разряда и переноса из младшей тетрады в старшую в результате выполнения текущей операции;
Z – признак нулевого результата текущей операции;
$\overline{PD}$ – признак перехода МК в режим пониженного энергопотребления (активное состояние – нулевое);
$\overline{TO}$ – признак переполнения сторожевого таймера (активное состояние – нулевое);
PA0 – номер страницы ПП (см. рис. 2.14);
RBWUF – бит сброса порта *B*;
R/W-0 – бит доступен для чтения и для записи, состояние бита после сброса МК – нулевое;
R-1 – бит доступен только для чтения, состояние бита после сброса МК – единичное;
R/W-x – бит доступен для чтения и для записи, состояние бита после сброса МК – неопределенное (может быть как нулевым, так и единичным)

Рис. 2.19. Формат регистра статуса МК *PIC16F505* [4]

Разряды	7	6	5	4	3	2	1	0
Мнемоника	Резерв		**DC**	**OF**	**IE**	**S**	**Z**	**C**

S – признак отрицательного результата выполнения команды;
IE – бит разрешения прерываний от периферийных устройств;
OF – признак арифметического переполнения в результате выполнения команды;
C, Z и *DC* – см. пояснения к рис. 2.19

Рис. 2.20. Формат регистра статуса МК КР1878ВЕ1 [5]

Программный счетчик МК ***PIC16F505*** – 12-битовый. Напрямую доступны его младшие 8 бит (РСФ *PCL*, см. рис. 2.13). Изменение 8-го и 9-го битов возможны только посредством команд безусловного перехода и вызова подпрограммы (см. рис. 2.14). Состояния 10-го и 11-го битов безразличны, поскольку, независимо от их значений, реально выборка команд осуществляется из физически существующей области ПП (адреса от *0000h* до *03FFh,* см. рис. 2.13), способом *wrap-around*.

Разрядность **программного счетчика** МК **КР1878ВЕ1** равна 10-и битам, что соответствует реальному объему ПП (1024 слова). Непосредственная запись адреса в программный счетчик МК КР1878ВЕ1 возможна только командами условных и безусловных переходов и вызова подпрограмм [5].

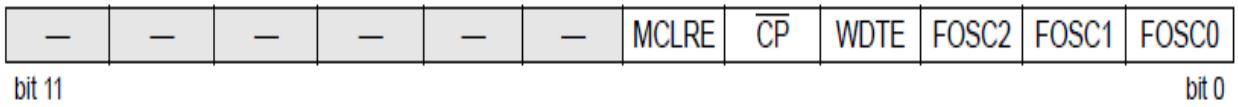
Стек как МК *PIC16F505,* так и КР1878ВЕ1 является **аппаратным** (что характерно для многих моделей МК класса *cost-sensitive*), т. е. представляет собой организованную по принципу *LIFO* группу из запоминающих элементов (регистров), структурно не входящую ни в ПП, ни в ПД.

Стек МК ***PIC16F505*** [4] используется только для хранения адресов возврата из подпрограмм; команды *PUSH* и *POP* в системе команд *PIC*-МК младшего подсемейства отсутствуют. Стек является двухуровневым, т. е. состоит из 2-х ЗЭ. Их разрядность равна разрядности программного счетчика, т. е. 12-ти битам.

В составе МК **КР1878ВЕ1** [5] имеется как стек команд (корректнее было бы назвать его стек адресов команд), так и стек данных (см. рис. 1.2). Первый из них используется для хранения

адресов возврата из подпрограмм, в том числе из подпрограмм обслуживания запросов на прерывания. Он состоит из 8-и ЗЭ с разрядностью, равной разрядности адреса ПП (10-и битам). Стек данных используется как модуль ПД с организацией по принципу *LIFO* и записью / чтением посредством команд *PUSH* и *POP*, имеющихся в системе команд МК КР1878ВЕ1. Он состоит из 16-и ЗЭ с разрядностью, равной разрядности слова данных (8 бит).

Как указано в пункте 2.5.1, в структуру практически всех МК входят ряд энергонезависимых ЗЭ системного назначения, предназначенных для хранения слов конфигурации МК, кодов идентификации и т. п. У МК класса *cost-sensitive* они, как правило, недоступны пользовательскому ПО; изменение их содержимого возможно только при программировании памяти МК. Например, к ЗЭ системного назначения относится слово конфигурации (*CONFIGURATION WORD*) МК *PIC16F505* [4], формат которого представлен на рис. 2.21.



FOSC0…FOSC2 – биты, задающие источник синхронизации МК (подробнее – см. [4]);
WDTE – бит разрешения работы сторожевого таймера;
$\overline{CP}$ – бит разрешения защиты ПП (активное состояние - нулевое);
MCLRE – бит разрешения использования 3-го вывода порта В в качестве входа сброса МК

Рис. 2.21. Формат слова конфигурации МК *PIC16F505* [4]

Более подробно организация памяти, состав и назначение регистров МК *PIC16F505* описаны в источнике [4], МК КР1878ВЕ1 – в [5], других типовых примеров МК класса *cost-sensitive* – например, в [17].

Необходимо также отметить, что к классу *cost-sensitive* относятся МК широко распространенного подсемейства *ATtiny* семейства *AVR* [7]. Организация памяти и регистровые модели МК семейства *AVR* рассмотрены далее, в пункте 2.5.7.

2.5.7. Организация памяти и регистровые модели МК семейства *AVR*

Организация памяти и регистровые модели всех МК семейства *AVR*, в целом, однотипны. Их память реализована по Гарвардскому варианту архитектуры (ПП и ПД физически разделены) и включает в себя [6]:

- 32 (у наиболее простых моделей подсемейства *ATtiny* – 16) 8-битовых РОН, структурно входящих в состав ЦП (см. рис. 2.4), из них три пары регистров в ряде команд могут служить в качестве 16-битовых регистров – указателей адреса с именами *X, Y* и *Z* (подробнее – см. пункт 2.6.3);

- несколько десятков 8-битовых и 16-битовых РСФ, количество, состав и форматы которых зависят от конкретной модели МК; часть РСФ входит в состав ЦП (например, программный счетчик, указатель стека и регистр статуса), большинство – в состав периферийных устройств;

- резидентное статическое ОЗУ данных;

- резидентное энергонезависимое ЗУ программ;

- в состав памяти ряда моделей МК семейства *AVR*, в частности, большинства моделей подсемейства *ATmega*, также входит энергонезависимое ЗУ данных (см. рис. 1.3).

Память конкретных моделей *AVR*-МК различается объемом вышеперечисленных модулей, количеством и составом РОН и РСФ, наличием или отсутствием энергонезависимого ЗУ данных, составом и форматами регистров системного назначения.

В качестве типового примера МК семейства *AVR* здесь и в дальнейшем будем использовать БИС 1887ВЕ7Т [8] класса «*mainstream*», являющуюся аналогом МК *ATmega128*. Структурная схема данной БИС представлена на рис. 1.3. При необходимости, в процессе изложения, будут отмечаться особенности структурно-архитектурных решений МК подсемейства *ATtiny* семейства *AVR* [7].

Резидентная память МК 1887ВЕ7Т включает в себя:

- энергонезависимую ПП объемом 65536 16-битовых слов (128 килобайт);

- 32 8-битовых РОН;

- 105 8-битовых РСФ;
- статическое ОЗУ данных объемом 4096 8-битовых слов;
- энергонезависимое ЗУ данных объемом 4096 8-битовых слова;
- энергонезависимые регистры системного назначения.

Структура **ПП** МК 1887ВЕ7Т / *ATmega128* представлена на рис. 2.22 [8]. Она, как и ПП большинства МК подсемейства *ATmega*, состоит из 2-х основных областей: области размещения прикладного ПО и загрузочного сектора, в котором размещается ПО загрузки памяти МК в режиме самопрограммирования (см. раздел 12). Данный сектор располагается в области адресов от *FFFFh – Boot Size* до *FFFFh*, где *Boot Size* – размер загрузочного сектора, задаваемый в процессе записи байтов конфигурации (см. далее). В структуре ПП большинства МК подсемейства *ATtiny* загрузочный сектор отсутствует.

Разрядность команд МК семейства *AVR* равна 16-ти или 32-м битам, поэтому ПП организована в виде массива из 16-битовых слов. Код команды может состоять из одного или двух слов разрядностью 16 бит. Адресом слова команды служит адрес ее младшего байта. Младший бит адреса всегда равен нулю. При адресации ПП физическое значение адреса слова команды (его младшего байта) вычисляется сдвигом содержимого программного счетчика на один бит влево с записью нуля в младший бит. Например, ассемблерная команда с адресом *$3FFF* (*3FFFh*), которой соответствует такое же содержимое программного счетчика, реально расположена в ПП по адресам *7FFEh* (младший байт) и *7FFFh* (старший байт).

Как у всех МК семейства *AVR,* в ПП МК 1887ВЕ7Т / *ATmega128* имеется область, выделенная под таблицу векторов прерываний (см. пункт 7.3.1). Число источников прерываний равно 35-ти [8], а начальной командой подпрограммы обслуживания прерывания обычно служит безусловный переход с абсолютной адресацией (см. табл. 2.7), разрядность кода которого равна 32-м битам. Поэтому под таблицу векторов прерываний выделено 70 16-битовых слов ПП. По умолчанию, они занимают младшие 70 адресов ПП, от *0000h* до *0045h,* однако под управлением прикладной программы данная таблица может быть перемещена в область ПП, занимающую первые 70 адресов загрузочного сектора (см. рис. 2.22).

Рис. 2.22. ПП МК 1887ВЕ7Т / *ATmega128* [8]
(в правом поле приведены начальный и конечный адрес в шестнадцатеричной системе счисления)

Структура **энергозависимой ПД** МК 1887ВЕ7Т / *ATmega128* представлена на рис. 2.23.

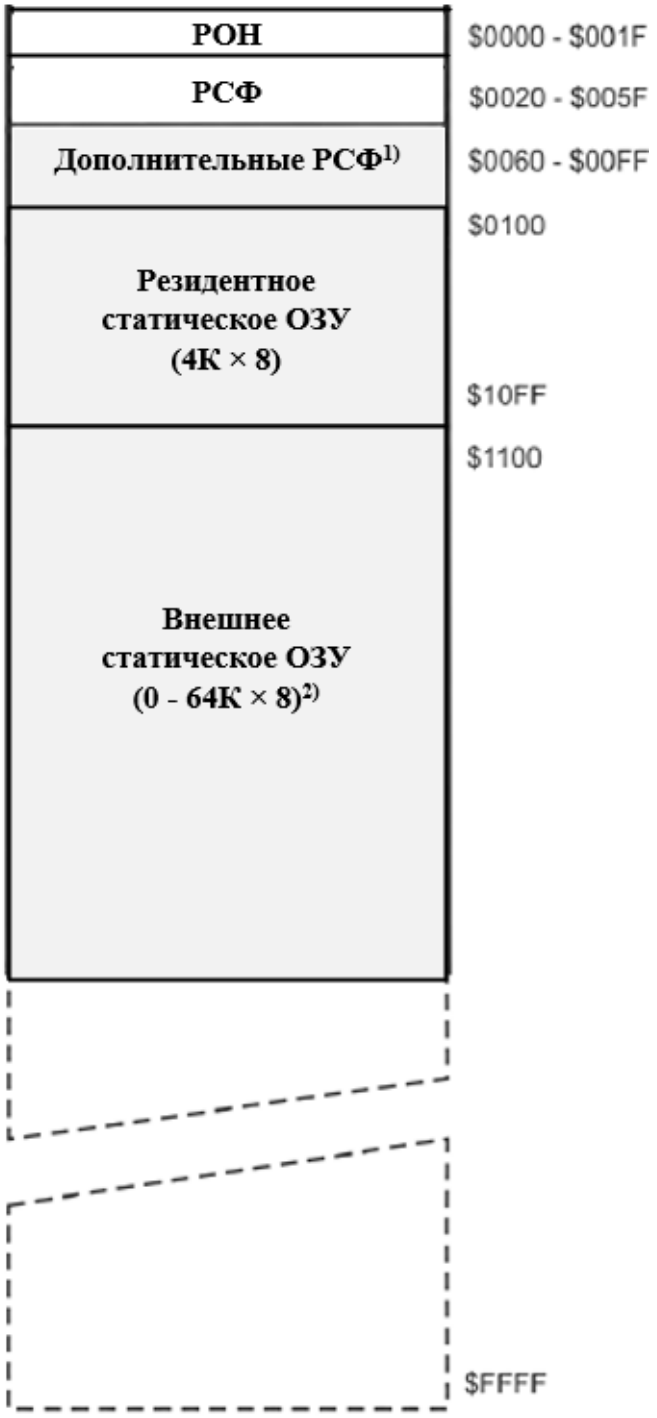
Энергозависимая ПД МК 1887ВЕ7Т / *ATmega128* включает в себя:

- 32 8-битовых РОН;

- 105 8-битовых РСФ, расположенных в адресном пространстве от *20h* до *FFh*, часть адресов которого не задействована (зарезервирована); в состав РСФ входят как регистры периферийных устройств, так ряд регистров системного назначения (в частности, регистр статуса и указатель стека);

- статическое резидентное ОЗУ данных объемом 4096 байта, занимающее адресное пространство от *0100h* до *10FFh*;

- при необходимости ПД может быть дополнена внешним ОЗУ объемом до 64-х килобайт, занимающим адресное пространство от *1100h* до *FFFFh*.



[1)] РСФ с данными адресами имеются только в составе МК подсемейства *ATmega*.

[2)] Необходимость во внешнем ОЗУ и его объем определяются конкретной задачей, решаемой МК. В большинстве случаев оно не требуется. 64 Кбайт – его максимально возможный объем.

Рис. 2.23. Энергозависимая ПД МК 1887ВЕ7Т / *ATmega128* [8] (в правом поле приведены начальный и конечный адрес в шестнадцатеричной системе счисления)

На практике внешнее ОЗУ используется редко, по причине, указанной в пункте 2.5.1. Поэтому вопросы организации интерфейса МК с ним здесь не рассматриваются; они подробно изложены в [8].

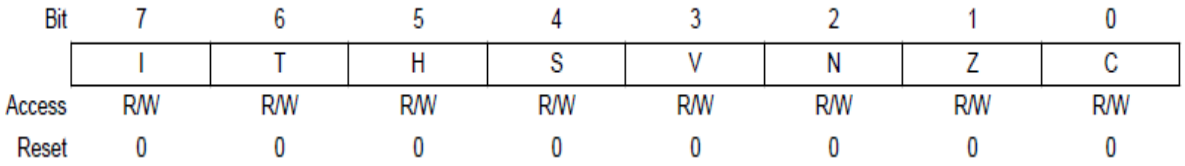
Разрядность ЗЭ ПД равны разрядности МК, т. е. 8-и битам. Все модули ПД, представленные на рис. 2.23, доступны прикладному ПО (в некоторых РСФ ряд битов доступны только для чтения). Обращения к РОН обычно осуществляются по присвоенным им именам (*R0, R1* и т. п., см. табл. 2.7).

Описания форматов основных РСФ периферийных устройств приведены в разделах, посвященных соответствующим классам периферийных устройств (портам, таймерам и т. п.).

В состав модулей памяти МК 1887ВЕ7Т / *ATmega128*, как и большинства МК подсемейства *ATmega*, входит также **энергонезависимая ПД**, реализованная по технологии ЭСППЗУ (*EEPROM*). Ее объем – 4096 8-битовых слов. Доступ к ней несколько сложнее, чем к энергозависимой ПД. Он осуществляется через РСФ *EEARH* (регистр старшей части адреса *EEPROM*), *EEARL* (регистр младшей части адреса *EEPROM*) и *EEDR* (регистр данных *EEPROM*). Описания процедур чтения и записи *EEPROM* и примеры их программной реализации представлены в [8].

Формат **регистра статуса** одинаков у всех МК семейства *AVR.* Он представлен на рис. 2.24. Все биты регистра статуса доступны как для чтения, так и для записи, их активное состояние – единичное. После сброса (*Reset*) МК все биты регистра статуса устанавливаются в пассивное (нулевое) состояние.

Программный счетчик МК 1887ВЕ7Т / *ATmega128* [8] содержит 16 старших битов абсолютного адреса команды, подлежащей выполнению, они же выступают в качестве адреса слова команды в ассемблерном коде программы. Как указано ранее, адресом слова команды МК семейства *AVR* служит адрес ее младшего байта. Младший бит адреса всегда равен нулю. При адресации ПП физическое значение адреса слова команды (его младшего байта) вычисляется сдвигом содержимого программного счетчика на один бит влево с записью нуля в младший бит.



Bit	7	6	5	4	3	2	1	0
	I	T	H	S	V	N	Z	C
Access	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Reset	0	0	0	0	0	0	0	0

C, H – признаки переноса из старшего разряда и из младшей тетрады в старшую в результате выполнения текущей операции;
Z – признак нулевого результата текущей операции;
N – признак отрицательного результата текущей операции;
$S = N \oplus V$ – признак, используемый при сравнении чисел со знаком (см. описания команд *BRGE* и *BRLT* в табл. 2.7);
V – признак переполнения в результате выполнения текущей операции;
T – разряд, используемый как «почтовый ящик» при пересылке бита из одного РОН в другой (см. описания команд BLD и BST в табл. 2.7);
I – бит глобального разрешения прерываний (см. раздел 7)

Рис. 2.24. Формат регистра статуса МК семейства *AVR* [8]

У всех МК семейства *AVR,* программный счетчик недоступен прикладному ПО для непосредственной записи или считывания. Его содержимое может перезагружаться только при выполнении команд условных и безусловных переходов, а также при вызовах подпрограмм, в том числе подпрограмм обслуживания прерываний (см. табл. 2.7 и раздел 7).

Стек МК семейства *AVR* представляет собой область, выделяемую разработчиком прикладного ПО в статическом ОЗУ данных [6], в отличие от аппаратного стека моделей МК, рассмотренных в пункте 2.5.6. Стек *AVR*-МК заполняется в направлении от запоминающих элементов (ЗЭ) с б**о**льшим адресом к ЗЭ с меньшим адресом. Содержимое указателя стека равно адресу его вершины, т. е. не заполненного ЗЭ стека с максимальным адресом. Стек используется как для хранения адресов возврата из подпрограмм, в том числе из подпрограмм обслуживания прерываний, так и для временного хранения операндов и результатов. В системе команд *AVR*-МК, в отличие от *PIC*-МК, имеются команды *PUSH* (загрузка содержимого РОН в стек) и *POP* (загрузка РОН из стека), см. табл. 2.7. При выполнении команд вызова подпрограмм или при обслуживании прерываний на вершину

стека загружается двухбайтный адрес возврата (см. рис. 2.2), занимающий два однобайтных ЗЭ стека, при этом, соответственно, содержимое указателя стека уменьшается на два. При выполнении команды *PUSH* в стек загружается однобайтное содержимое некоторого РОН, и указатель стека уменьшается на единицу. В свою очередь, при выполнении команды *POP* содержимое указателя стека увеличивается на единицу, а при выполнении команды возврата из подпрограммы (*RET*) или из прерывания (*RETI*) – увеличивается на два; см. также рис. 2.2 и табл. 2.7.

Во всех МК семейства *AVR* указатель стека, для обеспечения совместимости, имеет разрядность 16 бит. Под него выделены два 8-битовых программно-доступных РСФ: *SPL* и *SPH* (соответственно младший и старший байт указателя стека). При сбросе МК их содержимое устанавливается равным некоторому определенному значению, у ряда моделей – старшему адресу резидентного статического ОЗУ, у МК 1887ВЕ7Т – нулю [8]. Прикладным ПО указатель стека устанавливается на адрес, определяемый разработчиком для решения конкретной задачи. Его значение должно быть не меньше младшего адреса резидентного статического ОЗУ, у МК 1887ВЕ7Т / *ATmega128* равного *100h* (реально, для возможности практического использования стека – минимум на 2 больше данного адреса).

В состав **энергонезависимых регистров системного назначения** МК 1887ВЕ7Т / *ATmega128* (и большинства МК подсемейства *ATmega*) входят [8]:

- байт защиты ПП и энергонезависимой ПД (*Lock Bit Byte*), задающий режимы защиты энергонезависимых модулей памяти МК от записи и / или чтения;

- байты конфигурации (*Fuse Bytes*), в которых хранятся коды конфигурации МК на системном уровне, в частности, задающие размер загрузочного сектора, источник синхронизации МК и т. п.;

- сигнатурные байты (*Signature Bytes*), содержащие трехбайтный код идентификации МК; для МК 1887ВЕ7Т значения этих байтов равны *1Eh*, *97h* и *02h*;

- калибровочные байты (*Calibration Bytes*), в которые при производстве БИС МК загружаются 4 калибровочных числа

внутреннего *RC*-генератора синхроимпульсов МК (на рис. 1.3 обозначен как «Калиброванный генератор»), обеспечивающие настройку генератора на частоты 1, 2, 4 и 8 МГц (см. пункт 4.4.1).

Подробные сведения о формате байта защиты и порядке доступа к нему представлены в [8]. Форматы и назначение байтов конфигурации будут рассматриваться по мере описания архитектуры подсистем МК, к которым относятся те или иные их биты (подсистемы синхронизации, подсистемы сброса, подсистемы самопрограммирования и т. п.). Назначение и использование калибровочных байтов будут рассмотрены в разделе 4, посвященном подсистеме синхронизации МК.

2.5.8. Организация памяти и регистровые модели МК семейства *ARM Cortex-Mx*

2.5.8.1. Организация памяти и регистровые модели всех МК семейства *ARM Cortex-Mx*, в целом, идентичны [9]. Будем рассматривать их на примере архитектуры *ARM Cortex-M3* [18], характеризуемой средней сложностью в пределах семейства; в частности, к МК с данной архитектурой относятся МК модельного ряда *STM32F103x* (см. рис. 1.4) и МК К1986ВЕ92F1I (см. рис. 1.5). Основные отличия организации памяти и регистровых моделей МК других подсемейств *ARM Cortex-Mx* будут отмечаться по мере изложения материала настоящего пункта.

Память МК семейства *ARM Cortex-Mx*, в общем случае, включает в себя:

- резидентную (т. е. расположенную на кристалле БИС МК) ПП (*Code*), под которую выделена область адресов с *00000000h* по *1FFFFFFFh*;

- резидентное статическое ОЗУ данных (*SRAM*), под которое зарезервированы адреса с *20000000h* по *3FFFFFFFh*;

- регистры резидентных периферийных устройств (*Peripheral*), под которые выделены адреса с *40000000h* по *5FFFFFFFh*;

- внешнее (по отношению к БИС МК) ОЗУ (*External RAM*) и регистры внешних периферийных устройств (*External device*), под которые выделены адресные пространства с *60000000h* по *9FFFFFFFh* и с *A0000000h* по *DFFFFFFFh* соответственно; на

практике, как правило, необходимость в использовании внешних периферийных устройств и ОЗУ отсутствует;

- память системного назначения (*Private peripheral bus & Vendor-specific memory*), под которую выделены адреса с *E0000000h* по *FFFFFFFFh*;

- программно-доступное СОЗУ ЦП (см. далее рис. 2.30).

Необходимо отметить, что:

- несмотря на то, что ПП и ПД занимают определенные области в одном и том же адресном пространстве, они физически разделены, а доступ к ним осуществляется по различным шинам адреса и данных, т. е. с точки зрения организации памяти архитектура МК семейства *ARM Cortex-Mx* является Гарвардской;

- ни у одного из выпускаемых в настоящее время МК семейства *ARM Cortex-Mx* резидентные ПП и ПД, регистры периферийных устройств или память системного назначения не занимают **все** выделенное под них адресное пространство; например объемы ПП и статического ОЗУ данных МК модельного ряда *STM32F103x* равны 64 Кбайт и 20 Кбайт соответственно, см. рис. 1.4); указанные выше адресные пространства выделены с учетом дальнейшего развития семейства и совместимости ПО МК «снизу вверх».

Обобщенная карта памяти МК подсемейства *ARM Cortex-M3* представлена на рис. 2.25 [16]. Названия и назначение ее структурных компонентов соответствуют вышеописанной общей структуре памяти МК семейства *ARM Cortex-Mx*. Смысл понятий *Bit band region* и *Bit band alias* раскрыт далее; необходимо отметить, что данные области отсутствуют в памяти МК подсемейства *ARM Cortex-M0* (наиболее простых по структуре и архитектуре в пределах семейства).

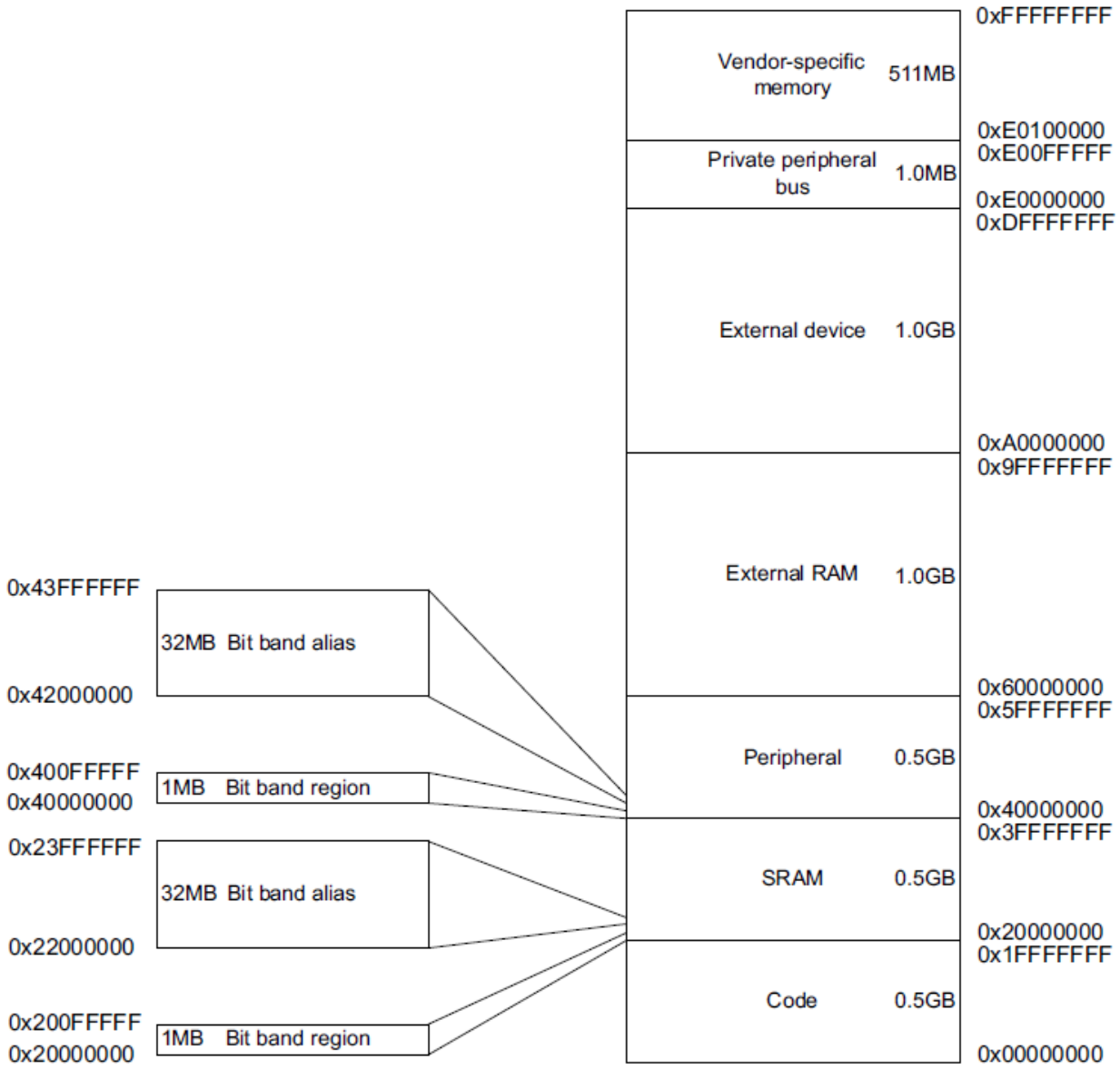


Рис. 2.25. Карта памяти МК семейства *ARM Cortex-M3* [16] (см. пояснения в тексте)

2.5.8.2. Необходимо отметить, что в пределах адресного пространства *Code* МК *ARM Cortex-Mx*, выделяемого под ПП (см. рис. 2.25) существуют области адресов определенного назначения, аналогично ПП МК семейства *AVR* (см. рис. 2.22). Типовым примером является распределение данного адресного пространства МК К1986ВЕ92F1I [11], представленное на рис. 2.26. Его младшие 2 Кбайт выделены под загрузочный сектор, в данной области адресов также находится таблица векторов прерываний. Код прикладного ПО располагается, начиная с адреса *0x08000000* (*08000000h*), что

характерно для большинства МК семейства *ARM Cortex-Mx*. Размер области расположения прикладного ПО равен объему ЭСППЗУ ПП.

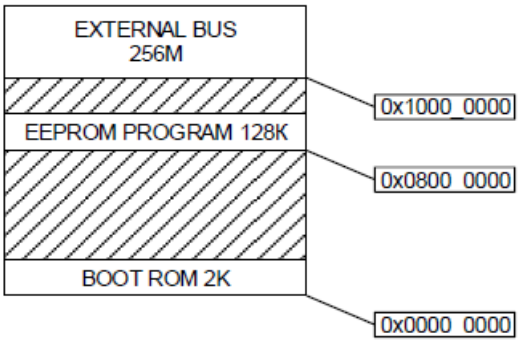


Рис. 2.26. Структура выделенного под ПП адресного пространства МК К1986ВЕ92F1I [11] (см. также рис. 2.25)

Также следует отметить, что у МК семейства *ARM Cortex-Mx,* в отличие от семейства *AVR*, отсутствует резидентная энергонезависимая ПД, реализованная как отдельный модуль. В ее качестве может использоваться область ЭСППЗУ ПП, не задействованная под загрузочный сектор и прикладное ПО.

У ряда моделей МК семейства *ARM Cortex-Mx,* в том числе К1986ВЕ92F1I, часть адресного пространства, выделенного под резидентную ПП, но не занятого физически реализованными модулями ПП, может использоваться для адресации внешней ПП. На рис. 2.26 это область, обозначенная как *EXTERNAL BUS* (т. е. доступная через внешнюю шину).

2.5.8.3. В выделенных под внутреннюю ПД и под РСФ ПУ адресных пространствах большинства моделей МК существуют не занятые резидентным ОЗУ и резидентными РСФ области адресов, которые могут использоваться для адресации внешних модулей ОЗУ и внешних периферийных устройств (см., например, рис. 2.27 и 2.28). Данные области также обозначены как *EXTERNAL BUS*. Резидентное статическое ОЗУ данных обозначено на рис. 2.27 как *SRAM*, а регистры резидентных периферийных устройств на рис. 2.28 – как *PERIPHERAL*. Штриховкой на рис. 2.26 – 2.28 обозначены физически не существующие и не задействованные области адресов.

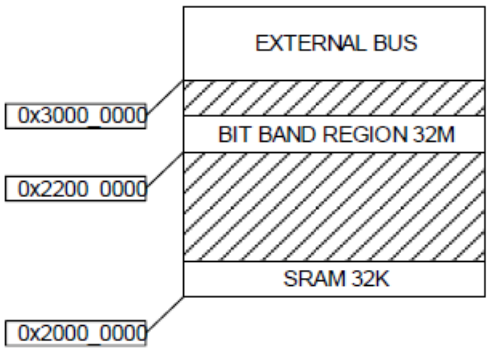


Рис. 2.27. Структура адресного пространства МК К1986ВЕ92F1I, выделенного под ОЗУ данных [11] (см. также рис. 2.25)

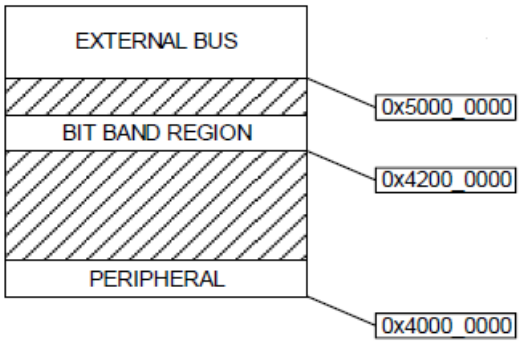


Рис. 2.28. Структура адресного пространства МК К1986ВЕ92F1I, выделенного под регистры ПУ [11] (см. также рис. 2.25)

2.5.8.4. Размещение команд и данных в памяти МК семейства *ARM Cortex-Mx*. В общем случае, минимальная адресуемая единица информации всех модулей памяти МК семейства *ARM Cortex-Mx* – байт. Разрядность команд равна 16-и или 32-м битам. Слова данных могут быть 8-и, 16-и или 32-битными (байт, полуслово и слово). Команды и данные размещаются в памяти в формате *Little Endian* (см. рис. 2.11). Адресом слова команды или данных, а также адресом РСФ служит адрес **младшего** байта. Многобайтные слова размещаются в памяти в порядке возрастания адресов от более младшего байта к более старшему (например, если адрес 0-го байта слова данных равен некоторому значению *A*, адрес 1-го байта равен *A+1*, и т. д.).

Особенностью размещения слов в ПД МК подсемейств *ARM Cortex-M3* и выше является их **фрагментация**, что позволяет

оптимизировать использование адресного пространства ПД при хранении слов различной разрядности (см. рис. 2.29).

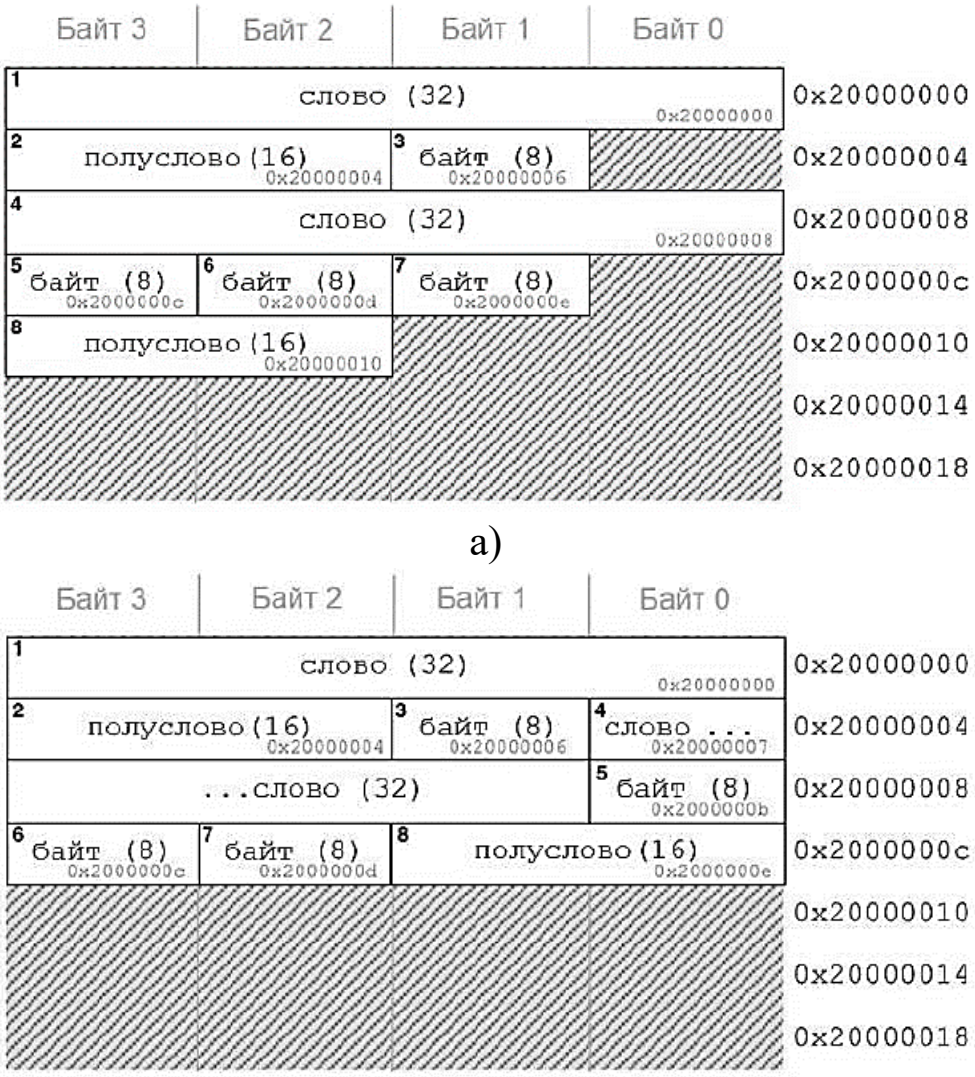


Рис. 2.29. Пример размещения не фрагментированных (а) и фрагментированных (б) переменных различной разрядности в ПД 32-разрядного МК [9]
(в скобках указана разрядность переменных в битах)

2.5.8.5. Области адресов с *20000000h* по *200FFFFFh* и с *40000000h* по *400FFFFFh* (*Bit band regions* на рис. 2.25) МК подсемейств *ARM Cortex-M3* и выше доступны для адресации **к отдельным битам**, реализуемой посредством процедуры *Bit band* [9]. Каждому биту ЗЭ / регистров данных областей присвоен определенный адрес (*Bit band alias*), вычисляемый по выражению:

$$BBA = BBA\_BA + (BYTE\_OFFS)*32 + (BIT\_NUMB)*4,$$

где *BBA_BA* – базовый адрес области *Bit band alias*, к которой принадлежит адресуемый бит (соответственно *22000000h* или *42000000h*);

BYTE_OFFS – смещение адреса байта, к которому принадлежит соответствующий бит, относительно базового адреса области *Bit band region,* к которой принадлежит данный байт (соответственно *20000000h* или *40000000h*);

BIT_NUMB – номер бита.

Например, 6-й и 7-й биты байта с адресом *200FFFFFh* в области *Bit band region* имеют статус виртуальных 32-битовых слов с адресами *22000000h + + FFFFFh*32 + 6*4 = 23FFFFF8h* и *22000000h + FFFFFh*32 + 7*4 = 23FFFFFCh* соответственно.

2.5.8.6. Программно-доступное СОЗУ МК семейства *ARM Cortex-Mx* включает в себя [9]:

- регистры программной модели ЦП прикладного уровня;
- регистры программной модели ЦП системного уровня;
- регистры периферийных устройств.

Состав регистров программной модели **прикладного уровня** ЦП МК семейства *ARM Cortex-M3* представлен на рис. 2.30.

В состав регистров программной модели прикладного уровня входят:

- 13 РОН (*R0 – R12*), причем РОН *R8 – R12* доступны только для 32-битовых версий команд;
- указатель стека (*R13*), через который осуществляется обращение к двум физически разделенным регистрам – указателю стека основной программы (*MSP, Main Stack Pointer*) и запускаемого из нее процесса (*PSP, Process Stack Pointer*); переключение между ними производится посредством 1-го бита регистра *CONTROL* (см. далее);
- регистр связи, *R14*, используемый для хранения адреса возврата в основную программу при выполнении подпрограмм (см. описания команд *BL* и *BLX* в табл. 2.8);
- программный счетчик (*R15*);
- регистр статуса выполняемой программы (*PSR*);

- регистры управления маскированием (т. е. запретом) обработки исключительных ситуаций, в том числе прерываний – *PRIMASK, FAULTMASK* и *BASEPRI*, а также управляющий регистр *CONTROL*. Регистры *FAULTMASK* и *BASEPRI* отсутствуют в МК подсемейства *ARM Cortex-M0* [19].

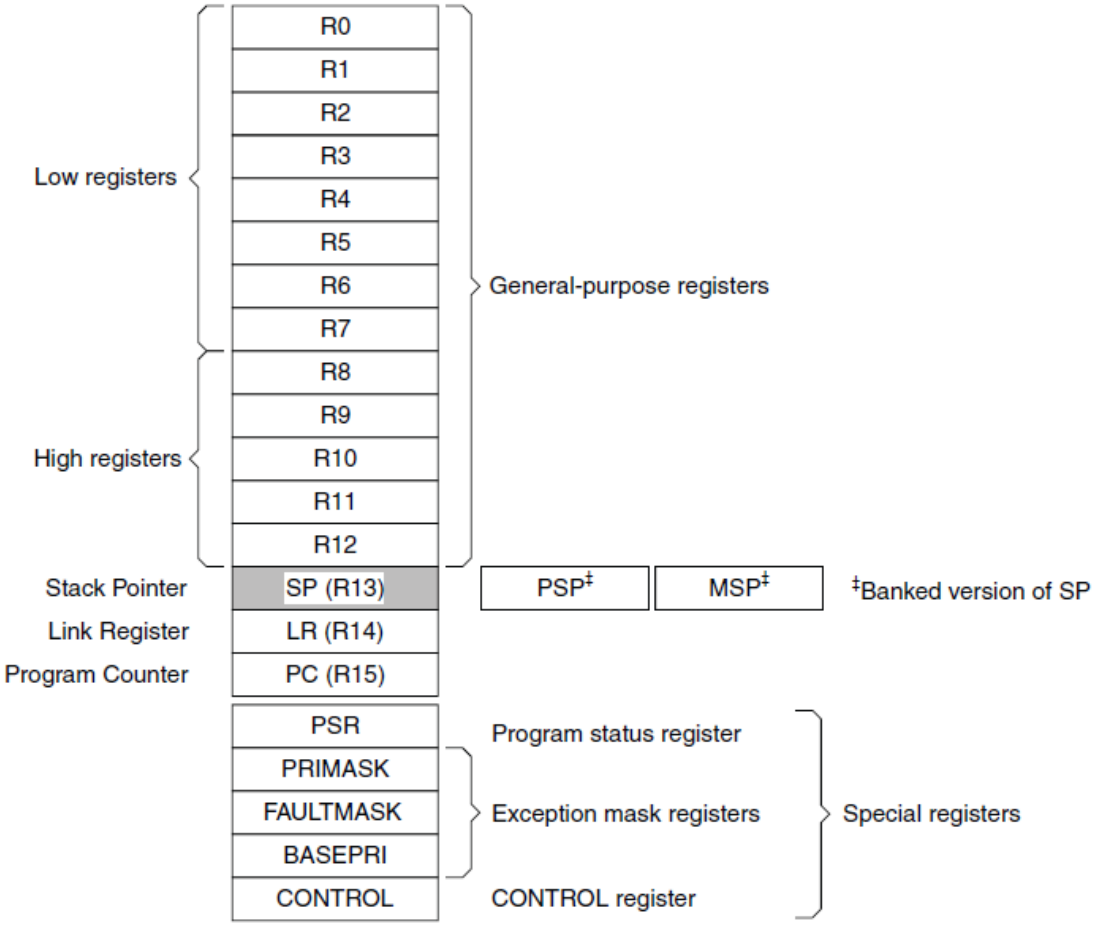


Рис. 2.30. Состав регистров программной модели прикладного уровня ЦП МК семейства *ARM Cortex-M3* [18] (пояснения – в тексте)

При этом:

- в регистре *PRIMASK* задействован только нулевой бит, записью единицы в который запрещается обработка исключительных ситуаций с конфигурируемым приоритетом;

- в регистре *FAULTMASK* задействован только нулевой бит, записью единицы в который запрещается обработка всех исключительных ситуаций, кроме немаскируемых прерываний (см. пункт 7.3.2);

- в регистре *BASEPRI* задействованы биты с 4-го по 7-й; в них указывается минимальное значение приоритета исключительных ситуаций, обработка которых не разрешена (следует отметить, что чем выше значение приоритета, тем ниже его уровень);

- в регистре *CONTROL* задействованы 0-й и 1-й биты; в 0-м бите указывается уровень выполнения программы (0 – привилегированный, 1 - непривилегированный), а состояние 1-го бита задает указатель стека, содержимое которого хранится в *R13* (0 – *MSP*, 1 – *PSP*, причем в режиме Handle может использоваться только *MSP*).

Из регистров, представленных на рис. 2.30, *MSP, PSR, PRIMASK, FAULTMASK*, *BASEPRI* и *CONTROL* доступны только на привилегированном уровне выполнения программы, остальные – также и на непривилегированном. Чтение и запись вышеперечисленных регистров осуществляется специальными командами *MRS* и *MSR* соответственно (см. табл. 2.8).

Программный счетчик (*PC*) МК семейства *ARM Cortex-Mx* доступен пользовательскому ПО как для чтения, так и для записи (с рядом ограничений) (см. табл. 2.8). Его содержимое равно адресу текущей команды, который совпадает с адресом ее младшего байта. Поскольку разрядность команд семейства МК *ARM Cortex-Mx* равна 2-м или 4-м байтам, младший бит программного счетчика равен 0.

Указатель стека (*SP*) МК семейства *ARM Cortex-Mx* содержит адрес слова, находящегося на вершине стека. При этом **стек**, как и у МК семейства *AVR*, представляет собой область памяти типа *LIFO*, организуемую в ПД под управлением прикладного ПО, и заполняемую в направлении от б**о**льших адресов к меньшим. При загрузке в стек содержимое его указателя уменьшается, а при извлечении из стека – увеличивается. Как указано ранее через *SP* осуществляется обращение к двум физически разделенным регистрам – *MSP* и *PSP* (см. пояснения к рис. 2.30).

2.5.8.7. Формат **регистра статуса** (*PSR*) МК семейства *ARM Cortex-Mx* представлен на рис. 2.31. Назначение его битов (битовых полей) указано в табл. 2.2.

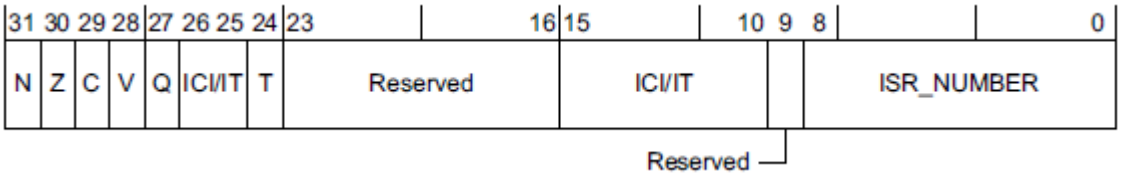


Рис. 2.31. Формат регистра статуса МК семейства *ARM Cortex-Mx* [16, 18, 19]

Таблица 2.2

Назначение битов (битовых полей) регистра статуса МК семейства ARM Cortex-Mx [16, 18, 19]

Бит (поле)	**Доступ**	**Назначение**
ISR_NUMBER	Только чтение	Идентификационный номер обрабатываемой исключительной ситуации (например, запросам на прерывания *IRQ0 – IRQ67* присвоены номера от 16 до 83 соответственно); нулевое значение данного поля указывает на отсутствие исключительной ситуации; см. также пункт 7.3.2
ICI/IT		При приостановке выполнения команды *LDM* или *STM* (см. табл. 2.8) из-за прерывания поле выполняет функцию *ICI* (*Interruptible-Continuable Instruction*); биты 12 – 15 содержат номер регистра – приемника / источника, с которого выполнение команды должно возобновиться. При выполнении блоков команд «*if-then*» (см. табл. 2.8) поле выполняет функцию *IT* и содержит информацию о текущем состоянии процесса выполнения блока.
T	Чтение и запись	Бит – указатель работы ЦП в системе команд *Thumb-2*, разрядность кода части из которых равна 16-и битам, части – 32-м; у ЦП МК семейства *ARM Cortex-Mx* всегда находится в единичном состоянии, т. к. ими поддерживается только данная система команд
Q		Признак достижения предельного значения результата при выполнении команд *SSAT* и *USAT* (см. табл. 2.8)
V		Признак переполнения в результате выполнения текущей операции
C		Признак переноса из старшего разряда в результате выполнения текущей операции
Z		Признак нулевого результата текущей операции
N		Признак отрицательного результата текущей операции

Особенностью регистра статуса данного МК семейства *ARM Cortex-Mx* является возможность программного обращения не только к регистру в целом, но и к группам его битов, имеющим статус отдельных регистров с именами:

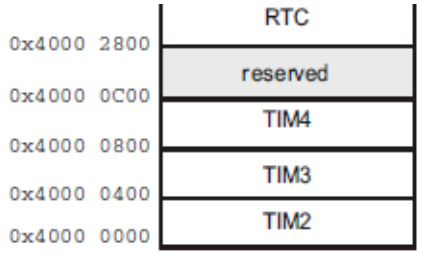
- *APSR* (биты 27 – 31);
- *EPSR* (биты 10 – 15, 25 и 26);
- *IPSR* (биты 0 – 8);

подробнее – см. [16, 18, 19].

2.5.8.8. Регистры программной модели ЦП **системного уровня** расположены в области памяти, обозначенной на рис. 2.30 как *Private peripheral bus*. В их состав входят, например, регистры контроллера прерываний, блока защиты памяти, системного таймера и т. п. Назначение и форматы ряда регистров данной группы будут рассмотрены в разделах, к которым они относятся с функциональной точки зрения (например, регистров контроллера прерываний – в пункте 7.3.2). Полное описание состава, назначения и форматов регистров программной модели ЦП системного уровня представлено, например, в документах [16, 18, 19].

2.5.8.9. Регистрам **периферийных устройств** МК семейства *ARM Cortex-Mx* выделены адреса ПД с *40000000h* по *5FFFFFFFh* (как и у ранее рассмотренных моделей МК, задействована только часть из них). Адреса данных регистров формируются как сумма базового адреса, выделенного под регистры соответствующего периферийного устройства конкретной модели МК, и смещения адреса определенного регистра относительно базового адреса, одинакового для всех моделей МК соответствующего подсемейства. Базовые адреса периферийных устройств указываются в техническом описании (*datasheet*) конкретной модели МК, а смещения адресов регистров периферийных устройств относительно базовых адресов – в руководстве пользователя (*User Guide*) соответствующего подсемейства МК. В качестве примеров на рис. 2.32а приведен фрагмент карты базовых адресов регистров ПУ МК *STM32F103x8* [10], а на рис. 2.32б - фрагмент таблицы регистров их таймеров общего назначения (*TIM2 – TIM5*) [13]. Например, адрес регистра состояния таймера *TIM2 (TIM2_SR)* равен *40000000h + 10h*

= *40000010h*, а регистра состояния таймера *TIM3 (TIM3_SR)* равен *40000400h + 10h = 40000410h.*



а)

Offset	Register	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x00	TIMx_CR1	Reserved																						CKD [1:0]		ARPE	CMS [1:0]		DIR	OPM	URS	UDIS	CEN
	Reset value																							0	0	0	0	0	0	0	0	0	0
0x04	TIMx_CR2	Reserved																								TI1S	MMS [2:0]			CCDS	Reserved		
	Reset value																									0	0	0	0	0			
0x08	TIMx_SMCR	Reserved																ETP	ECE	ETPS [1:0]		ETF[3:0]				MSM	TS[2:0]			Reserved	SMS[2:0]		
	Reset value																	0	0	0	0	0	0	0	0	0	0	0	0		0	0	0
0x0C	TIMx_DIER	Reserved																	TDE	COMDE	CC4DE	CC3DE	CC2DE	CC1DE	UDE	Reserved	TIE	Reserved		CC4IE	CC3IE	CC2IE	CC1IE	UIE
	Reset value																		0	0	0	0	0	0	0		0			0	0	0	0	0
0x10	TIMx_SR	Reserved																			CC4OF	CC3OF	CC2OF	CC1OF	Reserved		TIF	Reserved		CC4IF	CC3IF	CC2IF	CC1IF	UIF
	Reset value																				0	0	0	0			0			0	0	0	0	0

б)

Рис. 2.32. Фрагменты карты базовых адресов ПУ [10] (а) и таблицы регистров таймеров общего назначения (*TIM2 – TIM5*) [13] (б) МК *STM32F103xx*

2.5.8.10. В состав ряда моделей МК семейства *ARM Cortex-Mx* входит **модуль защиты памяти** (*Memory Protection Unit, MPU*) [9]. Его основное назначение – предохранение определенных, задаваемых разработчиком областей памяти от записи (в ряде вариантов конфигурации прав доступа – также и от чтения).

Возможны следующие варианты прав доступа к области памяти [9]:

- полный запрет доступа;

- доступ только для чтения на привилегированном уровне выполнения программы (см. подраздел 2.3); полный запрет доступа на непривилегированном уровне;

- на обоих уровнях – доступ только для чтения, с полным запретом доступа для записи;

- доступ для чтения и для записи на привилегированном уровне; полный запрет доступа на непривилегированном;

- доступ для чтения и для записи на привилегированном уровне; доступ только для чтения – на непривилегированном;

- полный доступ (и для чтения, и для записи) на обоих уровнях.

Нарушения правил доступа вызывают прерывание по нештатной ситуации *Memory Management Fault* (см. подпункт 7.3.2.5).

Границы областей памяти и права доступа к ним, а также ряд других параметров, относящихся к защите памяти [9, 16, 18] задаются посредством специальных программно-доступных (только на привилегированном уровне) регистров конфигурации *MPU*, расположенных в области *Private peripheral bus* памяти МК (см. рис. 2.25 и пояснения к нему).

Наличие или отсутствие *MPU* в составе той или иной модели МК определяется по ее *datasheet*. В частности, модуль *MPU* имеется в составе МК К1986ВЕ92FI, К1986ВЕ92F1I и К1986ВЕ94GI [11], отсутствует в МК модельного ряда *STM32F103x8* [10].

Следует отметить, что «рядовым» разработчикам средств контроля и управления на базе МК семейства *ARM Cortex-Mx* сравнительно редко приходится использовать модуль *MPU*. Поэтому углубленное рассмотрение его архитектуры выходит за рамки настоящего пособия. Ее подробное описание представлено в источниках [9, 11, 16, 18].

2.6 Система команд МК общего назначения

2.6.1. Общие вопросы разработки ПО МК

2.6.1.1. Неотъемлемым компонентом архитектуры ЦП МК является его система команд, под которой, как указано ранее,

понимается система двоичных кодов машинного языка, воспринимаемых ЦП без промежуточной трансляции (см. подраздел 1.1). Однако, прежде чем приступать к рассмотрению типовых систем команд МК, необходимо определить его целесообразность. Для этого, в свою очередь, следует остановиться на общих вопросах программирования МК.

Программирование непосредственно на машинном языке потенциально позволяет создать программный код с минимальным объемом и временем выполнения, за счет максимального использования особенностей структуры и архитектуры конкретного МК [3]. Однако программирование в двоичных кодах, даже в шестнадцатеричном их представлении, весьма трудоемко, при этом велика вероятность возникновения ошибок, а программы крайне неудобны для чтения и понимания. Поэтому программирование в машинных кодах в настоящее время практикуется весьма редко, хотя их знание может оказаться полезным при отладке устройств на МК, а также при решении задач анализа и прототипирования ПО МК в отсутствие исходных кодов на языках более высоких уровней.

2.6.1.2. При необходимости разработки программ или их фрагментов на уровне, равносильном по объему кода и быстродействию программам на машинном языке, программирование осуществляется на **языке ассемблера** МК. Его команды, по существу, представляют собой запись кодов машинного языка в удобно читаемой системе обозначений, базирующейся на естественном (обычно английском) языке. При этом каждой команде на языке ассемблера прямо соответствует некоторая команда на машинном языке. Например, двоичный код команды МК семейства *AVR*, выполняющей сложение содержимого РОН *R20* и *R21* с записью результата в регистр *R21*, имеет вид:

0000111101010100,

а та же команда на языке ассемблера указанного семейства выглядит следующим образом:

ADD R21,R20.

Нетрудно увидеть, что второй вариант записи намного удобнее для чтения и понимания, чем первый. В то же время после трансляции команда на языке ассемблера занимает такой же объем в памяти, как

и команда на машинном языке, и равносильна ей по времени выполнения. С другой стороны, программный код на языке ассемблера не воспринимается МК без трансляции в код на машинном языке специальной программой, также называемой Ассемблером.

2.6.1.3. В целом, в настоящее время разработка ПО МК осуществляется в интегрированных средах проектирования (*IDE*) на языках высокого уровня, обычно – на различных версиях языка *C*, адаптированных под задачи программирования МК. Это существенно снижает трудоемкость программирования и сроки разработки проектов, а также обеспечивает удобство чтения и понимания программных кодов и упрощает их переносимость с одной модели МК на другую в пределах семейства / подсемейства.

IDE создается, как правило, под определенное семейство МК или под номенклатуру МК, выпускаемую определенной компанией (обычно – разработчиком *IDE*). В свою очередь, адаптация языка *C* под задачи программирования МК состоит во включении в *IDE* библиотек интерфейса (взаимодействия) *C*-приложения с функциональными блоками МК, т. е. программных модулей, позволяющих приложению управлять ими и контролировать их состояние. Данное взаимодействие может осуществляться на различных уровнях. Например, пакеты *IDE* для разработки ПО МК семейства *ARM Cortex-Mx* могут включать в себя средства программного интерфейса следующих уровней [9]:

- *CMSIS* (*Cortex Microcontroller Software Interface Standard*);
- *SPL* (*Standard Peripherals Library*);
- *HAL* (*Hardware Abstraction Level*);

причем уровень *CMSIS* поддерживается практически всеми существующими *IDE*, уровень *SPL* – большинством, уровень *HAL* – наиболее «продвинутыми» *IDE*, например, *STM32Cube* [9].

2.6.1.4. Уровень *CMSIS* является наиболее «приближенным» к структуре и архитектуре конкретной модели МК. Взаимодействие на данном уровне осуществляется командами прямой записи управляющих кодов в регистры МК и непосредственного чтения их содержимого. Поэтому этот уровень можно назвать **регистровым**. При этом содержимое регистров и состояния их битов выступают в качестве переменных *C*-программы, а имена регистров и их битов, присвоенные им в соответствии с документацией на МК – в качестве идентификаторов переменных. Объявлять эти переменные в коде *C*-

программы не требуется, при условии, что в нее директивой *#include* включается файл описания программно-доступных регистров соответствующей модели / подсемейства МК из библиотеки *CMSIS*. Пример *C*-команды с использованием интерфейса *CMSIS*:

```
GPIOD->ODR |= 0x00000001;
```

Данная команда устанавливает в единичное состояние 0-й бит выходного регистра данных (*ODR*) порта ввода / вывода *D* (*GPIOD*) МК подсемейства *ARM Cortex-M4*. Команда не будет воспринята компилятором, если в заголовок *C*-файла не будет введена директива, включающая в него файл описания РСФ соответствующей модели / подсемейства МК, например:

```
#include "stm32f4xx.h"
```

Следует отметить, что интерфейс регистрового уровня, аналогичный *CMSIS*, широко используется и в *IDE* для разработки ПО других семейств МК. Например, в *C*-файле проекта ПО МК семейства *AVR* в *IDE Atmel Studio* команда установки в единичное состояние 0-го бита выходного регистра данных порта ввода / вывода *D* выглядит следующим образом:

```
DDRD = DDRD|0x01;
```

причем в заголовок файла должна быть введена директива:

```
#include <avr/io.h>
```

2.6.1.5. На **уровне *SPL*** управление функциональными блоками МК и контроль их состояния осуществляется посредством стандартных процедур, вызываемых из *C*-файла и входящих в состав библиотеки программных модулей *IDE*. Пример использования процедуры уровня *SPL* для установки в единичное состояние 0-го бита порта *D* МК подсемейства *ARM Cortex-M4* (см. выше реализацию той же операции на уровне *CMSIS*):

```
GPIO_SetBits(GPIOD, GPIO_Pin_0);
```

Для того, чтобы данная команда была воспринята компилятором, в заголовок *C*-файла должна быть введена не только директива, включающая в него файл описания РСФ соответствующей модели / подсемейства МК, например:

#include "stm32f4xx.h"

(см. выше), но и директива включения файла, содержащего процедуры управления портами ввода / вывода данной модели или подсемейства, например:

#include "stm32f4xx_gpio.h"

2.6.1.6. Наконец, на **уровне *HAL*** взаимодействие *C*-приложения с функциональными блоками МК осуществляется посредством библиотеки интерфейсных модулей (драйверов *HAL*), являющихся функционально- (а не аппаратно-) ориентированными. Поэтому данный уровень интерфейса можно назвать **функциональным**. При этом, с точки зрения разработчика проектного *C*-файла, вызов и выполнение процедур управления блоками МК и контроля их состояния может осуществляться без «привязки» к их структуре и регистровым моделям. Однако, для этого необходимо предварительно проинициализировать используемые в проекте функциональные блоки МК и настроить их на работу в заданном режиме. В *IDE STM32Cube* данная процедура осуществляется посредством графической утилиты *STM32CubeMX*. Например для настройки какого-либо вывода («пина») некоторого порта для формирования сигнала включения / выключения светодиода необходимо выполнение следующих действий.

1. Выбрать модель и тип корпуса МК.

2. Выбрать вывод МК, к которому будет подключен светодиод, и настроить его на выполнение функции «Выход общего назначения» (см. рис. 2.33).

3. Настроить режим работы выбранного вывода (см. рис. 2.34) и присвоить ему идентификатор, в рассматриваемом примере присвоен идентификатор *Led* (Светодиод).

4. После данных процедур настройки команда переключения светодиода (из одного состояния в противоположное) в *C*-файле может быть записана следующим образом:

```
HAL_GPIO_TogglePin(Led_GPIO_Port, Led_Pin);
```

При этом программный код, разработанный для какой-либо модели МК с применением интерфейса уровня *HAL*, с минимальными изменениями или без изменений может быть перенесен на любую другую модель МК того же семейства или подсемейства.

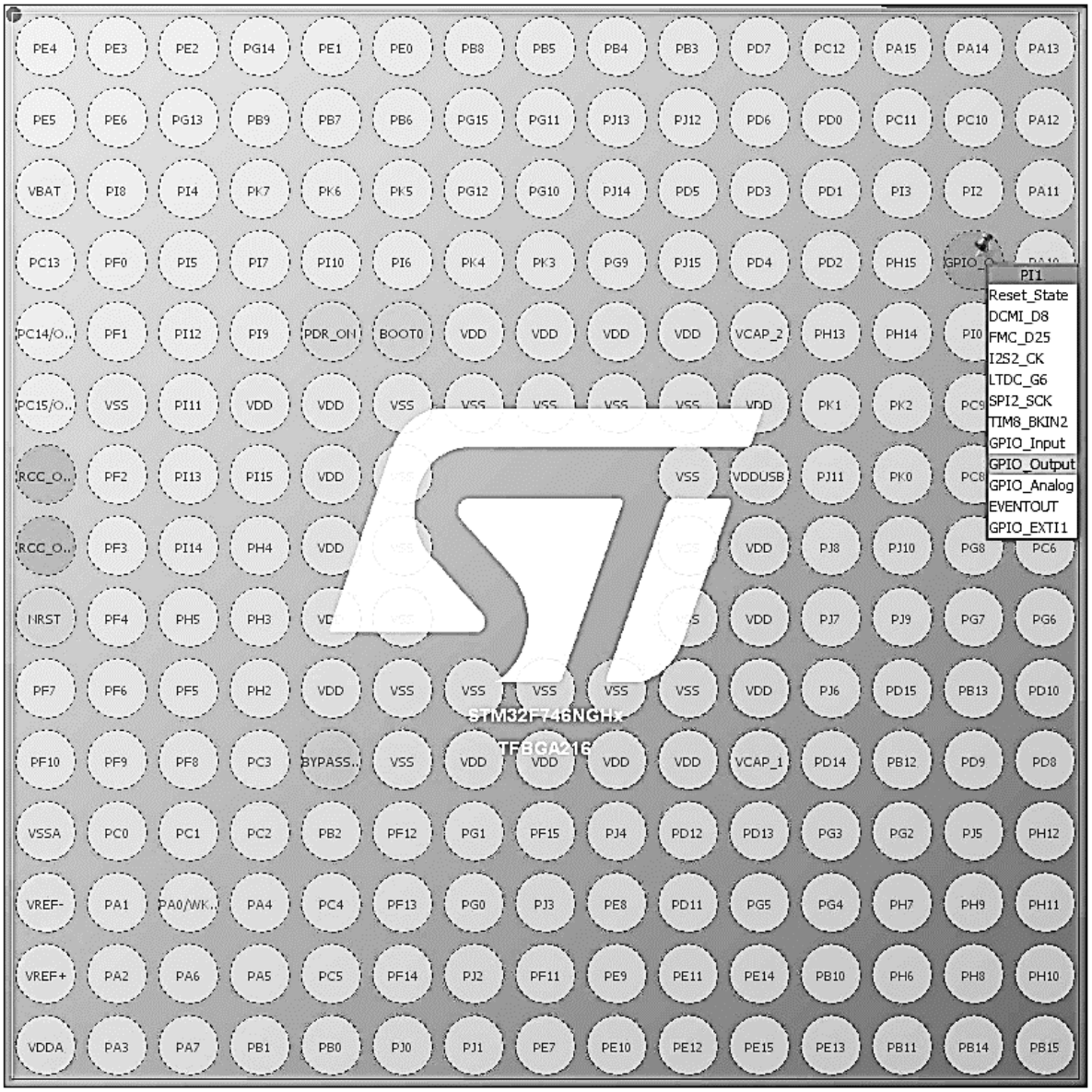


Рис. 2.33. Пример выбора вывода МК и настройки его функции посредством *STM32CubeMX*
(выбран 1-й вывод порта *I*, функция вывода – выход общего назначения (*GPIO Output*))

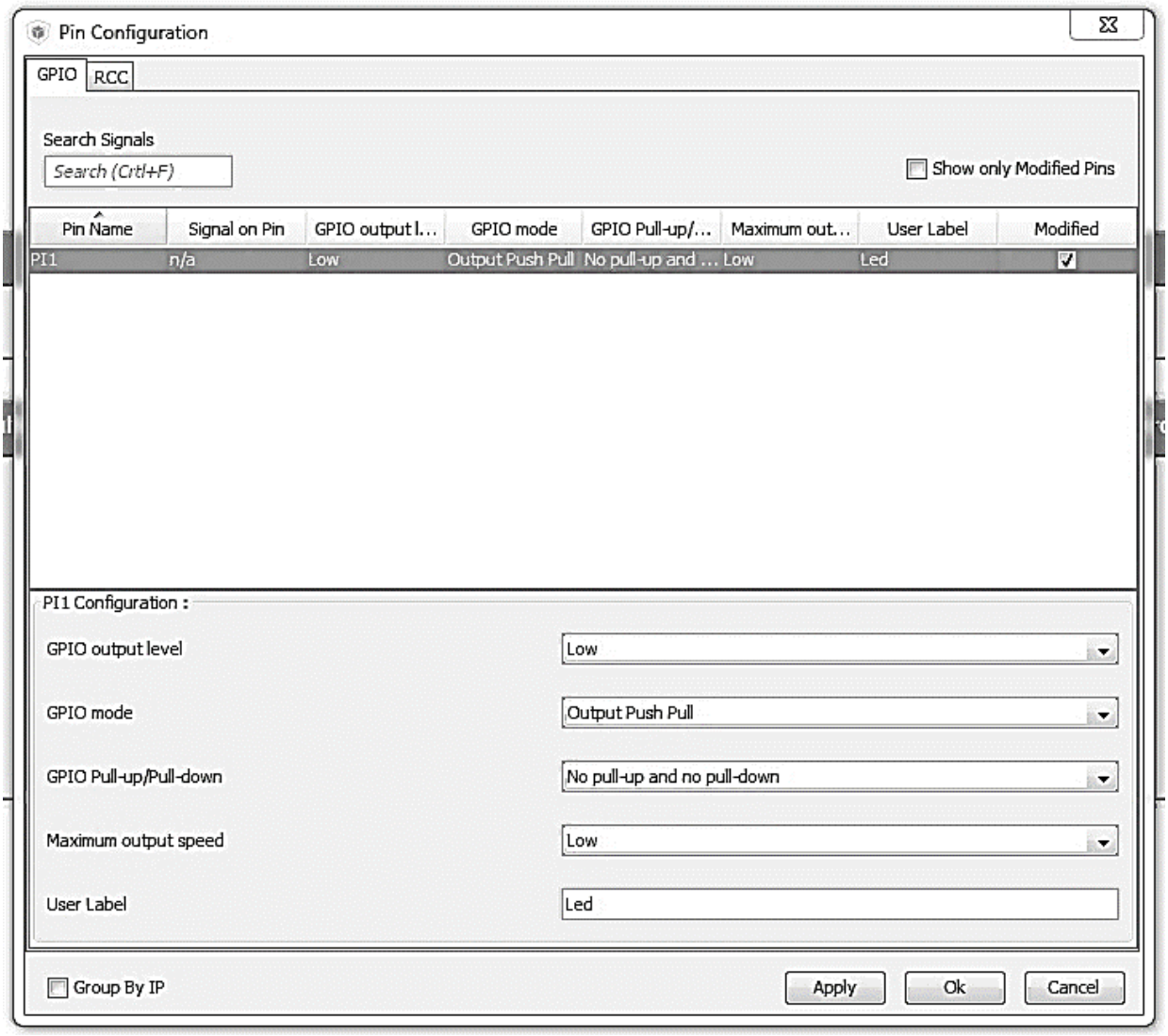


Рис. 2.34. Пример настройки режима работы вывода МК (вывод 1-й порта *I*, режим работы – двухтактный выход без подтягивающих резисторов, скорость переключения - низкая, идентификатор вывода - *Led*)

Функциональный уровень взаимодействия *C*-приложения с блоками МК, примерно аналогичный *HAL*, характерен также для среды разработки *Arduino IDE* [9].

2.6.1.7. Необходимо отметить, что процедуры настройки порта, функционально аналогичные вышеописанным, естественно, необходимы и при программировании на уровнях *CMSIS* и *SPL*. Например, настройка 0-го вывода порта *D* МК подсемейства *ARM Cortex-M4* на режим работы, аналогичный описываемому рис. 2.34, на уровне *CMSIS* осуществляется следующей последовательностью команд:

```
GPIOD->MODER = 0x00000001;
GPIOD->OSPEEDR = 0x00000000;
GPIOD->OTYPER = 0x00000000;
GPIOD->PUPDR = 0x00000000;
```

а на уровне *SPL*:

```
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;
GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_2MHz;
GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;
GPIO_Init(GPIOD, &GPIO_InitStructure);
```

2.6.1.8. Необходимо остановиться на двух важных вопросах **разработки ПО МК**:

- использование какого уровня интерфейса *C*-приложения с блоками МК (регистрового, функционального или библиотеки стандартных процедур взаимодействия с блоками МК) является предпочтительным;

- рационально ли применение языка ассемблера при разработке ПО МК или его отдельных модулей.

2.6.1.9. Корректный однозначный ответ на **первый** из вышеперечисленных вопросов дать сложно. Поскольку перечисленные уровни интерфейса различаются между собой, в первую очередь, степенью зависимости от структуры и регистровой модели МК, рационально рассмотреть основные достоинства и недостатки двух крайних, с точки зрения указанной зависимости, уровней. Ими являются функциональный и регистровый уровни интерфейса *C*-приложения с блоками МК (соответственно наименее и наиболее аппаратурно-зависимые). В дальнейшем, для краткости, будем использовать термины: **программирование МК на регистровом и на функциональном уровне.**

2.6.1.10. Основными **достоинствами** программирования на **функциональном** уровне являются:

- высокая степень автоматизации *IDE*, поддерживающих функциональный уровень программирования МК и, как следствие, простота и оперативность процесса создания проектов ПО;

- высокая «читабельность» программного кода;

- минимальные затраты времени на перенос программного кода с одной модели МК на другую в пределах подсемейства или

семейства, за счет малой зависимости или независимости кода от модели МК;

- относительная простота освоения программирования МК на функциональном уровне, за счет высокой автоматизации и «дружественного» пользовательского интерфейса *IDE*, поддерживающих данный уровень, а также отсутствия необходимости в знании архитектуры МК на регистровом уровне (однако, знания состава, функциональных возможностей и параметров блоков МК обязательны).

Основными **недостатками** программирования МК на **функциональном** уровне являются:

- существенно больший, чем при регистровом уровне программирования, объем исполняемого программного кода (автор на практике встречался со случаем, когда под разработанный на функциональном уровне и скомпилированный код относительно простого приложения не хватило объема ПП МК подсемейства *ARM Cortex-M0*);

- существенно большее время выполнения программ и их модулей, чем при программировании на уровнях, более «приближенных» к структуре и архитектуре МК (см. далее);

- ввиду высокой сложности *IDE* и, как следствие, повышенной вероятности «скрытых» ошибок при их разработке – возможность некорректной реализации некоторых процедур исполняемым кодом (в отсутствие ошибок разработчика прикладного ПО); такие случаи отмечались рядом пользователей *IDE STM32Cube* [9], встречались и в практике автора;

- относительная сложность синтаксиса команд управления блоками МК и проверки их состояния; необходимость некоторых затрат времени на освоение команд программирования МК на функциональном уровне;

- *IDE* ряда семейств / подсемейств / моделей МК, необходимость использования которых может возникнуть из-за организационно-экономических причин, не поддерживают функциональный уровень программирования;

- разработчику ПО МК, обучавшемуся и начинавшему программировать на функциональном уровне, значительно сложнее перейти, при возникновении необходимости, на регистровый уровень, так как он требует существенно более глубоких знаний структуры и архитектуры МК.

2.6.1.11. Основными **достоинствами** *C*-программирования МК на **регистровом** уровне по сравнению с другими уровнями разработки *C*-программ для МК являются:

- максимальная «близость» к структуре и архитектуре конкретной модели МК и, как следствие, возможность наиболее полного учета их функциональных возможностей и особенностей;
- минимальный объем ПП, занимаемый исполняемым программным кодом;
- минимальное время выполнения программ и их модулей (см. далее);
- поддержка программирования на регистровом уровне практически всеми существующими *IDE*;
- простота процедур обращения из *C*-программы к функциональным блокам МК (прямые чтение и запись их регистров) и, как следствие, минимальная вероятность некорректной работы приложения при отсутствии ошибок пользователя *IDE*;
- простота и однотипность синтаксиса команд чтения / записи регистров МК и проверки состояния их битов; минимальные затраты времени на освоение команд программирования МК на регистровом уровне;
- разработчики ПО МК, имеющие опыт программирования на регистровом уровне, обладают достаточно глубокими знаниями структуры и архитектуры МК, поэтому им проще перейти, при необходимости, на функциональный уровень программирования, чем обучавшимся и программирующим на функциональном уровне перейти на регистровый.

К основным **недостаткам** *C*-программирования МК на **регистровом** уровне (по сравнению с работой на более «высоких» уровнях) обычно относят:

- необходимость глубокого изучения структуры и архитектуры МК для программирования на данном уровне (однако, является ли это недостатком – вопрос дискуссионный);
- б**о**льшие затраты времени на разработку проекта;
- низкая «читабельность» программного кода;
- сложность переноса проекта на другую модель МК.

2.6.1.12. В качестве дополнительной информации для сопоставления различных уровней программирования МК, на рис. 2.35 и 2.36 представлены осциллограммы выходного сигнала одного из выводов порта МК подсемейства *ARM Cortex-M4*. Оба сигнала

сформированы по одному и тому же алгоритму, при одной и той же тактовой частоте МК, равной 8 МГц, в одной и той же *IDE*.

Формирование сигнала, осциллограмма которого представлена на рис. 2.35, осуществлялось следующим программным фрагментом на **функциональном** уровне:

```
while(1)
{
GPIO_SetBits(GPIOD, GPIO_Pin_0);
GPIO_ResetBits(GPIOD, GPIO_Pin_0);
}
```

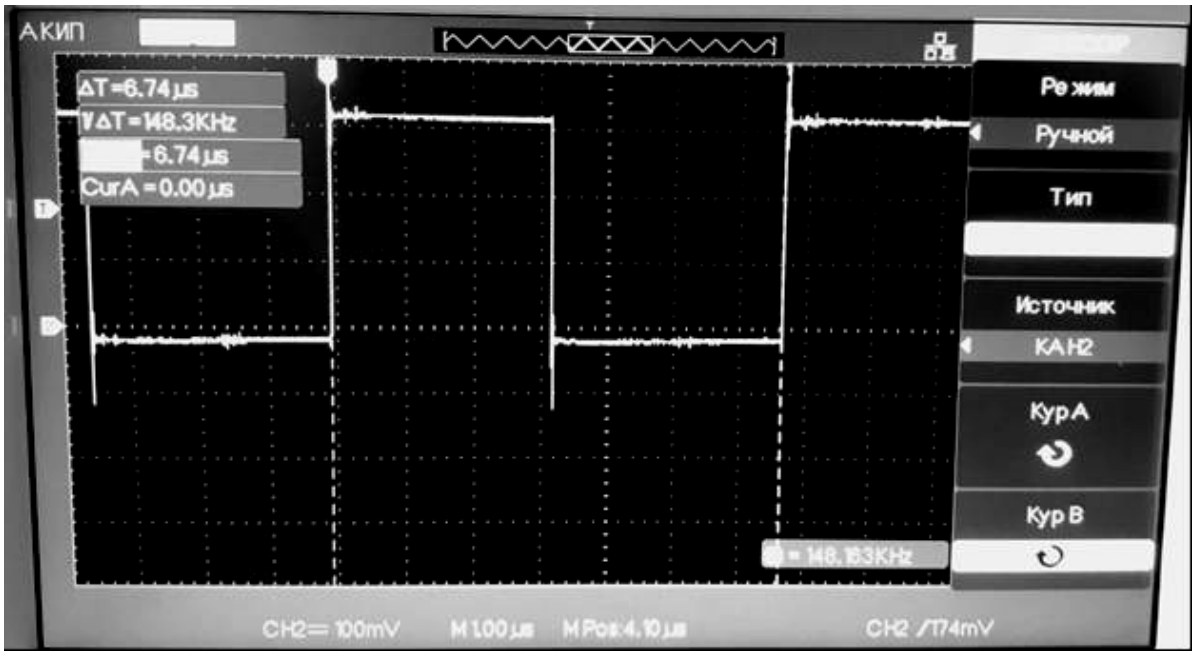


Рис. 2.35. Осциллограмма выходного сигнала порта МК подсемейства *ARM Cortex-M4,* сформированного программным фрагментом функционального уровня (см. выше)

В свою очередь, формирование сигнала, осциллограмма которого приведена на рис. 2.36, осуществлялось посредством следующего фрагмента программного кода на **регистровом** уровне:

```
while(1)
{
GPIOD->ODR=0x00000001;
GPIOD->ODR=0x00000000;
}
```

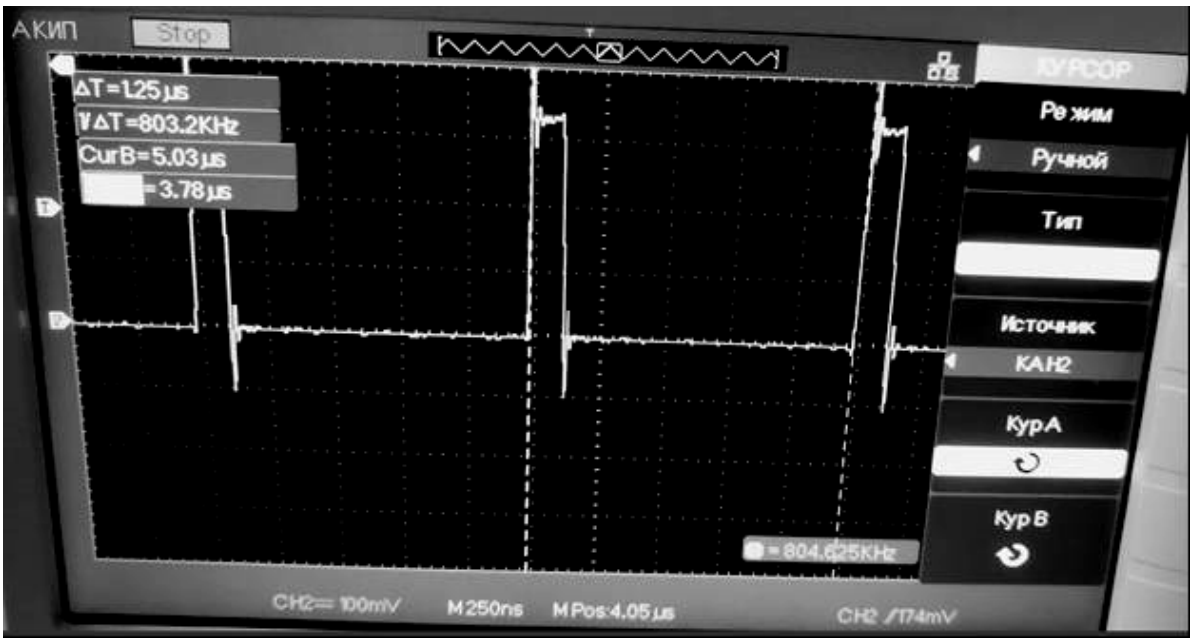


Рис. 2.36. Осциллограмма выходного сигнала порта МК подсемейства *ARM Cortex-M4,* сформированного программным фрагментом регистрового уровня (см. выше)

Сопоставляя осциллограммы, представленные на рис. 2.35 и 2.36, нетрудно заметить, что регистровый уровень программирования потенциально обеспечивает реальное время выполнения операций управления блоками МК, в несколько раз меньшее, чем функциональный уровень (в примерах, иллюстрируемых рис. 2.35 и 2.36 – приблизительно в 5 раз).

2.6.1.13. На основании вышеприведенного сопоставления достоинств и недостатков функционального и регистрового уровней программирования МК могут быть сделаны следующие выводы.

Программирование МК на **функциональном** уровне рационально при следующих условиях:

- программирование соответствующей модели / подсемейства МК на функциональном уровне поддерживается имеющейся в распоряжении или доступной *IDE*;

- быстродействие проектируемого устройства и объем памяти МК, занимаемый программой, не критичны;

- сроки на разработку ПО являются сжатыми;

- есть вероятность, что потребуется оперативный перенос разработанного ПО на другую модель / подсемейство МК;

- разработчик ПО обладает скорее «программистским», чем «схемотехническим» типом мышления, но не владеет глубокими знаниями в области архитектуры МК на регистровом уровне.

В свою очередь, программирование МК на **регистровом** уровне рационально при следующих условиях:

- программирование соответствующей модели / подсемейства МК на функциональном уровне не поддерживается имеющейся в распоряжении или доступной *IDE*;

- критичны быстродействие проектируемого устройства или / и объем памяти МК, занимаемый программой;

- разработчик ПО обладает глубокими знаниями в области архитектуры МК, при которых затраты времени на выполнение разработки и, при необходимости, перенос проекта на другую модель / подсемейство / семейство МК **сопоставимы** с затратами времени при программировании на функциональном уровне;

- в процессе обучения основам архитектуры и применения МК, с целью приобретения глубоких знаний в области структуры и архитектуры МК, а также навыков программирования на регистровом уровне, которые, в совокупности, позволят оперативно освоить программирование и применение практически любого семейства / подсемейства МК, а также, при необходимости, перейти и на функциональный уровень программирования.

С учетом вышесказанного, «право на применение», в зависимости от конкретных обстоятельств, имеет как функциональный, так и регистровый уровень программирования. В целом, по мнению автора (однако, не навязываемому никому), регистровый уровень программирования является более профессиональным и предпочтителен в большинстве практических случаев.

Что же касается освоения основ архитектуры и применения МК, то в его процессе, также по мнению автора, программирование рациональнее осуществлять на регистровом уровне. Это позволит максимально полно ознакомиться с типовыми структурно-архитектурными решениями МК, в том числе с назначением и форматами РОН и РСФ МК, а также, при необходимости, достаточно оперативно освоить программирование и на функциональном

уровне. Поэтому в примерах программных фрагментов, приведенных в настоящем пособии, в основном, используется программирование на регистровом уровне.

2.6.1.14. В связи с вышесказанным, необходимо остановиться еще на одном частном вопросе программирования на регистровом уровне. На данном уровне коды, присваиваемые содержимому регистров МК или их отдельных битовых полей, могут представляться:

- числами (константами) в двоичной, десятичной или шестнадцатеричной системе счисления;
- переменными с идентификаторами, присвоенными битам или битовым полям регистра.

Поясним вышесказанное примерами. В ранее приведенном фрагменте программного кода присваивание единичного значения младшему биту порта *D* осуществляется следующей командой:

GPIOD->ODR=0x00000001;

в которой используется первый из вышеперечисленных вариантов представления кодов. Независимая установка в единичное состояние младшего бита порта *D* при том же варианте представления реализуется командой:

GPIOD->ODR |= 0x00000001;

(побитовое логическое ИЛИ содержимого выходного регистра порта *D* с шестнадцатеричным числом 0x00000001).

Та же операция может быть реализована командой, использующей второй из вышеперечисленных вариантов представления:

GPIOD->ODR |= GPIO_ODR_ODR_0;

В данной команде операндом служит переменная с идентификатором *GPIO_ODR_ODR_0*, присвоенным 0-му биту выходного регистра порта (в данном случае – порта *D*).

Вопрос состоит в следующем: какой вариант представления содержимого регистров предпочтителен – числовой или в виде

переменных с идентификаторами, присвоенными соответствующим битам / битовым полям регистра?

Однозначного ответа на данный вопрос не существует. Основным достоинством числового представления является гарантированно корректное выполнение команд, естественно, при отсутствии ошибок в значениях операндов. Основным недостатком – необходимость некоторых затрат времени на вычисление данных значений.

В свою очередь, основным достоинством представления операндов в виде переменных с идентификаторами, присвоенными соответствующим битам или битовым полям, является отсутствие затрат времени на расчет значений операндов. С другой стороны, при этом необходимо корректное указание идентификаторов переменных, т. е. имен, присвоенных битам или битовым полям. Они, естественно, указываются в технической документации на соответствующую модель МК, а также в подсказках, «выплывающих» при наборе *C*-кода программы в большинстве *IDE*.

Примечание 1. Имена битов или битовых полей, указываемые в технической документации на МК и в подсказках *IDE*, могут не совпадать; в таких случаях в программном коде следует использовать имена, предлагаемые в подсказках *IDE*.

Примечание 2. Редко, но встречаются ситуации, в которых, при использовании предлагаемых *IDE* идентификаторов битов / битовых полей, отсутствии ошибок в математических выражениях команды присваивания и при компиляции кода, реально она не выполняется или выполняется некорректно; такие случаи несколько раз встречались автору, вероятно, они связаны с ошибками в модулях *IDE.*

Следует отметить, что при числовом представлении операндов некорректной работы команд присваивания значений битам или битовым полям регистров ни автором, ни его коллегами не выявлено.

Исходя из вышесказанного, автор отдает предпочтение числовому представлению операндов в вышеуказанных командах присваивания (не навязывая свою точку зрения). Поэтому в примерах, приводимых в последующих разделах, будет

использоваться именно данный вариант представления, с обоснованием значений, присваиваемых содержимому регистров или их битовых полей. Расчет данных значений не представляет серьезной проблемы. Краткое руководство по нему приведено в Приложении В.

Читатели, предпочитающие представление операндов в виде переменных с идентификаторами, присвоенными соответствующим битам или битовым полям, могут самостоятельно переработать приводимые примеры, используя техническую документацию на МК и подсказки *IDE*. Следует только заметить, что при этом необходимо проверить корректность работы программного кода (правильность работы представленных примеров проверена).

2.6.1.15. Далее, необходимо остановиться на втором из поставленных ранее фундаментальных вопросов: рационально ли применение **языка ассемблера** при разработке ПО МК или его отдельных модулей?

На данный вопрос большинство специалистов дает следующий ответ. Безусловно, разработка прикладного ПО МК **полностью** на языке ассемблера в подавляющем большинстве случаев не рациональна, из-за высокой трудоемкости программирования и переноса ПО на другие модели МК. Исключения, теоретически, составляют лишь следующие ситуации (весьма редко встречающиеся на практике):

- для соответствующего семейства / подсемейства МК не существует *IDE*, поддерживающей разработку ПО на языке *C* или каком-либо другом языке высокого уровня;
- имеются крайне жесткие ограничения на объем ПП, занимаемой программным кодом.

С другой стороны, в некоторых случаях может быть рациональной реализация на языке ассемблера программных фрагментов, критичных с точки зрения времени выполнения, так как, при прочих равных условиях, квалифицированно разработанный код на языке ассемблера характеризуется минимальным временем выполнения по сравнению с кодом на любом языке высокого уровня. Проиллюстрируем это несложным примером.

Пусть необходимо на МК подсемейства *ARM Cortex-M4* реализовать цифровой фильтр (ЦФ) 1-го порядка с бесконечной импульсной характеристикой (БИХ). В качестве отсчетов входного сигнала ЦФ служит последовательность отсчетов встроенного АЦП МК, а отсчеты выходного сигнала ЦФ поступают на ЦАП МК. Запуск АЦП осуществляется одним из таймеров МК, без участия ЦП. Цифровая фильтрация реализуется в соответствии с выражением:

$$y[i] = a_0 \times x[i] + a_1 \times x[i-1] + b_1 \times y[i-1], \qquad (2.3)$$

где $x[i]$ и $x[i-1]$ – соответственно текущий и предыдущий отсчеты входного сигнала; $y[i]$ и $y[i-1]$ – соответственно текущий и предыдущий отсчеты выходного сигнала; a_0, a_1 и b_1 – коэффициенты, являющиеся функциями от частоты дискретизации и от частоты среза ЦФ, причем $a_0 = a_1$. Разрядность отсчетов входного и выходного сигналов, а также коэффициентов ЦФ равны 8-и битам. Начальные значения $x[i-1]$ и $y[i-1]$ – нулевые.

В табл. 2.3 представлен программный фрагмент на языке ассемблера МК подсемейства *ARM Cortex-M4*, являющийся «телом» цикла работы ЦФ, наиболее критичным с точки зрения времени выполнения. В табл. 2.4 приведен функционально аналогичный фрагмент на языке *C* и дизассемблированный результат компиляции данного фрагмента при следующих ее параметрах: уровень оптимизации – максимальный (*O3*) + «*Optimize for Time*». Также в табл. 2.3 и 2.4 приведено время выполнения каждой из ассемблерных команд в периодах тактового генератора ЦП, в соответствии с *ARM Cortex-M4 Processor Technical Reference Manual*. В табл. 2.5 представлен дизассемблированный результат компиляции только команды, реализующей выражение (2.3), при нулевом уровне оптимизации.

Таблица 2.3

Программный код реализации БИХ-ЦФ на языке ассемблера МК подсемейства ARM Cortex-M4

Команда на языке ассемблера[1)]	**Время выполнения, T_{clk}**[2)]
Опрос (в цикле) флага готовности результата преобразования АЦП	
a1:LDR r4,[r0,#0x00]	2
TST r4,#0x0002	1
BEQ a1	2 – 4[3)]
При готовности – обнуление регистра состояния АЦП, считывание результата преобразования и обнуление его старших 3-х байт	
STR r2,[r0,#0x00]	2
LDR r4,[r0,#0x4c]	2
AND r4,r8	1
Фильтрация в соответствии с выражением (2.3)	
MLA r9, r3, r6, r5	1
MOV r5, r9	1
MLA r9, r4, r6, r5	1
MOV r5, r9	1
MUL r5, r7	1
LSR r5, r5, #0x08	1
Нормализация результата фильтрации	
LSR r9, r9, #0x08	1
Запись выходного отсчета ЦФ в регистр данных ЦАП	
STR r9, [r1, #0x1c]	2
Присвоение отсчету $x[i-1]$ значения отсчета $x[i]$	
MOV r3,r4	1
Переход к опросу флага готовности АЦП	
B a1	2 – 4[3)]
	Всего: 22 – 26

Примечания.

[1)] Описания команд – см. табл. 2.8. Функции, присвоенные РОН:
r0 – базовый адрес регистров АЦП (см. рис. 2.32); r1 – базовый адрес регистров ЦАП; r2 – нулевое значение; r3, r4 – значения отсчетов $x[i-1]$ и $x[i]$ соответственно; r5 – значение $b_0 \times y[i-1]$; r6 и r7 – коэффициенты a_0 и b_1 соответственно; r8 – значение маски (0x000000FF); r9 – значение $y[i]$.

[2)] T_{clk} – период тактового генератора ЦП.

[3)] В зависимости от числа тактов ЦП, требуемых для заполнения конвейера (см. пункт 2.6.4).

Таблица 2.4

Программный код реализации БИХ-ЦФ на языке С и результат его компиляции (профиль «О3 + Optimize for Time»)

Команда на языке *С*	**Последовательность команд на языке ассемблера**	**Время выполнения, T_{clk} [1)]**
Опрос (в цикле) флага готовности результата преобразования АЦП		
while(((ADC1->SR)&(0x00000002)) ==0x00000000);	LDR r2,[r12,#0x00]	2
	TST r2,#0x02	1
	BEQ 0x08000440	1 – 4[1)]
При готовности – считывание результата преобразования и присвоение его значения текущему отсчету входного сигнала ЦФ		
xin=ADC1->DR; x[1]=xin;	LDR r2,[r12,#0x4C]	2
	UXTB r2,r2	1
	STRB r2,[r5,#0x00]	2
Обнуление регистра состояния АЦП		
ADC1->SR = 0x00000000;	STR r4,[r12,#0x00]	2
Фильтрация в соответствии с выражением (2.3)		
y[1]=a[0]*x[0]+a[0]*x[1]+b[0]*y[0];	STR r2,[r0,#0x04]	2
	LDR r8,[r0,#0x00]	2
	LDR r9,[r1,#0x00]	2
	ADD r8,r8,r2	1
	MUL r8,r8,r3	1
	MLA r8,r6,r9,r8	1
Нормализация результата фильтрации		
y[1] = y[1]/256;	ASR r9,r8,#31	1
	ADD r8,r8,r9,LSR #24	1
	ASR r8,r8,#8	1
Запись выходного отсчета ЦФ в регистр данных ЦАП		
yout=y[1];	AND r9,r8,#0xFF	1
	STR r8,[r1,#0x04]	2
	STRB r9,[r5,#0x01]	2
DAC->DHR8R1=yout;	STR r9,[r7,#0x00]	2
Присвоение $x[i-1]$ значения $x[i]$, а $y[i-1]$ - значения $y[i]$		
x[0]=x[1];	STR r2,[r0,#0x00]	2
y[0]=y[1];	STR r8,[r1,#0x00]	2
Переход к опросу флага готовности АЦП		
while(1)	B 0x08000440	2 – 4[1)]
		Всего: 36 – 41
[1)] См. примечания к табл. 2.3		

Таблица 2.5

Дизассемблированный результат компиляции команды, реализующей выражение (2.3), при нулевом уровне оптимизации

Команда на языке ассемблера	**Время выполнения, T_{clk}[1)]**
LDR r0,[pc,#112]	2
LDR r0,[r0,#0x00]	2
LDR r1,[r1,#0x00]	2
MULS r0,r1,r0	1
LDR r1,[pc,#104]	2
LDR r1,[r1,#0x04]	2
LDR r2,[pc,#108]	2
LDR r2,[r2,#0x04]	2
MLA r0,r1,r2,r0	1
LDR r1,[pc,#96]	2
LDR r1,[r1,#0x00]	2
LDR r2,[pc,#100]	2
LDR r2,[r2,#0x00]	2
MLA r0,r1,r2,r0	1
LDR r1,[pc,#92]	2
STR r0,[r1,#0x04]	2
	Всего: 29[2)]

[1)] См. примечания к табл. 2.3

[2)] Как видно из табл. 2.3 и 2.4, время выполнения той же команды при максимальном уровне оптимизации равно 9-и периодам тактового генератора ЦП, а при реализации выражения (2.3) непосредственно на языке ассемблера – 6-и периодам.

Сопоставляя данные, приведенные в табл. 2.3 – 2.5, можно сделать следующие выводы:

- время выполнения программного фрагмента, реализующего БИХ-ЦФ на языке ассемблера, примерно в 1,5 раза меньше, чем функционально аналогичного ему фрагмента на языке *C* при профиле компиляции «*O3 + Optimize for Time*», максимально оптимизированном по времени выполнения;

- код дизассемблированного результата компиляции *C*-фрагмента при вышеуказанном профиле, в целом, сходен по принципам реализации каждой из процедур с кодом, изначально разработанным на языке ассемблера (см. табл. 2.3);

- код дизассемблированного результата компиляции *C*-команды, реализующей выражение (2.3) при нулевом уровне оптимизации,

существенно отличается от кода при максимальном уровне оптимизации, и характеризуется временем выполнения, в 4,8 раз б**о**льшим, чем аналогичный фрагмент на языке ассемблера, и в 3,2 раза б**о**льшим – чем результат компиляции той же команды при максимальном уровне оптимизации.

Таким образом, разработка на языке ассемблера программных фрагментов, критичных с точки зрения времени выполнения, может обеспечить некоторый выигрыш по быстродействию даже по сравнению с *C*-кодом при максимальном уровне оптимизации. В частности, в вышеприведенном примере этот выигрыш составляет примерно 1,5 раза. На первый взгляд, он не является значительным. Однако, с другой стороны, применительно к цифровой фильтрации, такой выигрыш позволит в 1,5 раза увеличить верхнюю граничную частоту входного сигнала ЦФ, что может быть немаловажно в ряде конкретных случаев.

2.6.1.16. Следует отметить, что, при реализации критичных по быстродействию программных фрагментов на языке ассемблера, МК могут служить рациональной заменой ПЛИС в некоторых задачах обработки сигналов и управления техническими объектами в реальном масштабе времени.

2.6.1.17. Также необходимо отметить, что при разработке ПО МК на языке *C* быстродействие, сопоставимое с программированием на языке ассемблера, может быть достигнуто корректным выбором профиля компиляции, как правило – максимальный уровень оптимизации по быстродействию, обычно также оптимальный по объему скомпилированного кода (см. табл. 2.4 и 2.5). Однако, на практике встречаются случаи, когда при выборе максимального уровня оптимизации компилятор «игнорирует» некоторые команды исходного *C*-кода, что приводит к некорректной работе программы. Поэтому при использовании повышенных уровней оптимизации необходим контроль дизассемблированного результата компиляции (см. далее).

2.6.1.18. Исходя из вышесказанного, наиболее рациональным подходом к разработке программ для МК представляется следующий:

- основная часть каждого из программных модулей разрабатывается на языке высокого уровня (в зависимости от требований по быстродействию и квалификации программиста – на регистровом или на функциональном уровне), с выбором профиля оптимизации компилятора, соответствующего требованиям к быстродействию или / и объему исполняемого кода и с контролем дизассемблированного результата компиляции;

- при необходимости, программные фрагменты, критичные с точки зрения времени выполнения или объема, разрабатываются на языке ассемблера соответствующего МК и оформляются как ассемблерные вставки в основной код (что допускается практически всеми используемыми в настоящее время *IDE*).

Изложение материалов данного пособия основывается на предположении, что программирование МК осуществляется на базе данного подхода. Большинство примеров программных фрагментов, представленных в пособии, написаны на языке *C*, адаптированном под задачи программирования МК. При этом в приведенных примерах используется регистровый, а не функциональный уровень программирования (см. приведенные ранее результаты сопоставления данных уровней).

С другой стороны, необходимо уделить внимание и языкам ассемблера, поскольку:

- в ряде практических случаев возникает необходимость реализации некоторых программных фрагментов на языке ассемблера для удовлетворения требований по быстродействию или /и по объему исполняемого кода (см. табл. 2.3 – 2.5);

- языки ассемблера широко используются при разработке **собственно *IDE***, поэтому их знание может потребоваться при изучении файлов библиотек *IDE* для понимания особенностей реализации тех или иных функций;

- знание языков ассемблера требуется для анализа дизассемблированного результата компиляции программы на языке высокого уровня, при проверке корректности выбранного уровня оптимизации (также см. табл. 2.4 и 2.5).

Следует отметить, что именно система команд ассемблера выступает в качестве системы команд МК, приводимой в его

техническом описании (*Data Sheet*) или руководстве по программированию (*Programming Manual*).

Типовым примерам языков ассемблера распространенных семейств МК посвящены последующие пункты. Команды машинного языка, т. е. двоичные коды ассемблерных команд, не приводятся и не рассматриваются, т. к. навыки программирования и чтения программ в машинных кодах редко требуются на практике. При необходимости, с системами данных кодов можно ознакомиться по документации на соответствующие семейства или модели МК.

2.6.2. Типовой состав системы команд языка ассемблера МК

Язык ассемблера практически каждого из семейств МК общего назначения включает в себя следующие группы команд [4, 8, 18]:

- команды пересылки данных;
- арифметические команды;
- побитовые логические операции;
- команды сдвигов;
- команды передачи управления;
- команды специального назначения (например, перехода в режим пониженного энергопотребления).

Типовой состав **команд пересылки данных** [4, 8, 18]:

- команды загрузки константы в РОН или в РСФ;
- команды пересылки из РОН в другой РОН;
- команды пересылки из РОН в РСФ или из РСФ в РОН;
- команды пересылки из РОН в ПД или из ПД в РОН;
- команды пересылки из ПП в РОН или из РОН в ПП (отсутствуют в системе команд ряда МК класса «*cost-sensitive*», а их выполнение возможно только в режимах, при которых ПП доступна для чтения или записи);
- команды записи в стек или чтения из стека.

В состав **арифметических команд** обычно входят [4, 8, 18]:

- сложение содержимого 2-х РОН или содержимого РОН с константой;

- вычитание из содержимого РОН содержимого другого РОН или константы;

- инкремент и декремент (соответственно увеличение и уменьшение на единицу) содержимого РОН;

- сравнение содержимого РОН с содержимым другого РОН или с константой, осуществляемое также посредством вычитания, но без записи результата в какой-либо из модулей памяти (РОН, РСФ или ПД), а только с изменением битов признаков в регистре статуса по результатам вычитания; команда сравнения обычно используется при реализации условных переходов;

- перемножение и деление содержимых РОН.

Типовой состав **побитовых логических операций** включает в себя [8, 18]:

- побитовое логическое И, И-НЕ, ИЛИ, ИЛИ-НЕ и исключающее ИЛИ 2-х операндов (обычно – содержимых 2-х РОН или содержимого РОН и константы);

- инверсия операнда (обычно - содержимого РОН, в системах команд ряда семейств МК – также РСФ);

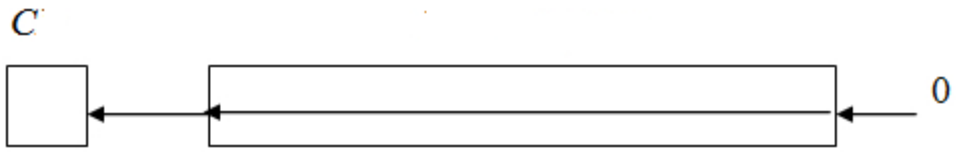
- выборочная установка, сброс или инверсия битов РОН или РСФ;

- в системах команд ряда семейств МК – команды проверки состояний битов РОН или РСФ посредством побитового логического И или исключающего ИЛИ с содержимым некоторого РОН или с константой, без записи результата, а только с изменением битов признаков в регистре статуса; данные команды, как и арифметические команды сравнения, используются при реализации условных переходов.

Типовой состав **команд сдвигов** языка ассемблера МК включает в себя логические, арифметические и циклические сдвиги содержимого РОН вправо или влево на один бит. Системы команд сдвига ряда семейств МК класса «*high performance*», в частности, семейств *ARM Cortex-Mx*, позволяют осуществлять сдвиг на число бит, большее 1, указываемое в коде команды.

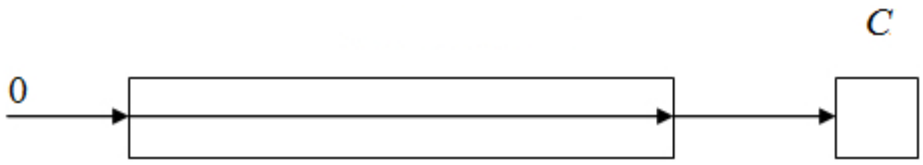
Правила выполнения вышеперечисленных разновидностей сдвигов иллюстрируются рисунками 2.37 – 2.43 [8, 18]. На всех

данных рисунках *C* – бит (флаг) признака переноса регистра статуса МК (см. рис. 2.19, 2.20, 2.24 и 2.31).



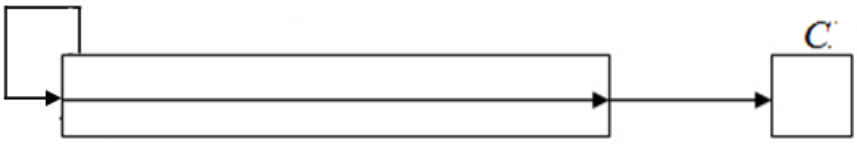
Пример.
Исходное значение операнда: $00110011_2 = 51_{10}$
После логического сдвига влево на 1 бит:
значение операнда – $01100110_2 = 102_{10}$, *C*=0

Рис. 2.37. Общая схема и пример выполнения логического сдвига влево на 1 бит



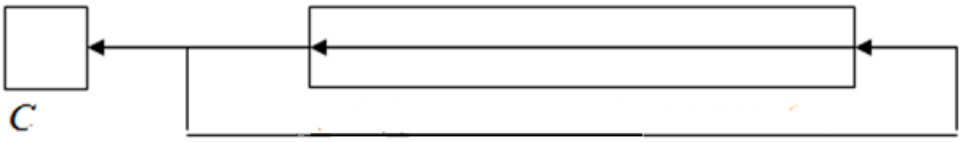
Пример.
Исходное значение операнда: $00110011_2 = 51_{10}$
После логического сдвига вправо на 1 бит:
значение операнда – $00011001_2 = 25_{10}$, *C*=1

Рис. 2.38. Общая схема и пример выполнения логического сдвига вправо на 1 бит



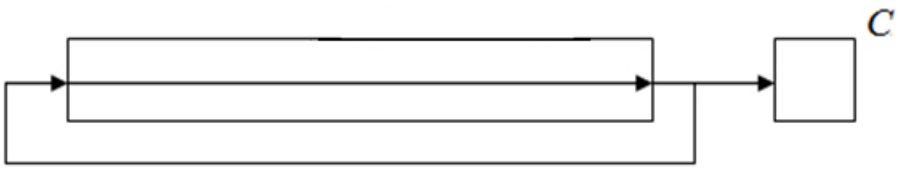
Пример.
Исходное значение операнда: $11110100_2 = -12_{10}$
После арифметического сдвига вправо на 1 бит:
значение операнда – $11111010_2 = -6_{10}$, *C*=0

Рис. 2.39. Общая схема и пример выполнения арифметического сдвига вправо на 1 бит



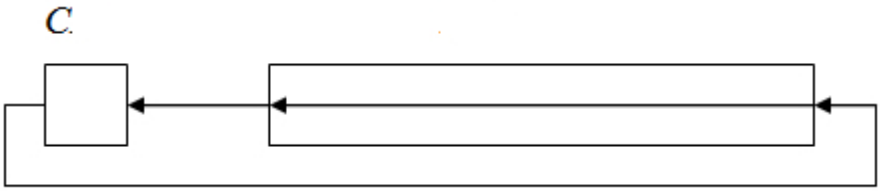
Пример.
Исходное значение операнда: 00110011_2
После циклического сдвига влево на 1 бит:
значение операнда – 01100110_2, C=0

Рис. 2.40. Общая схема и пример выполнения циклического сдвига влево на 1 бит



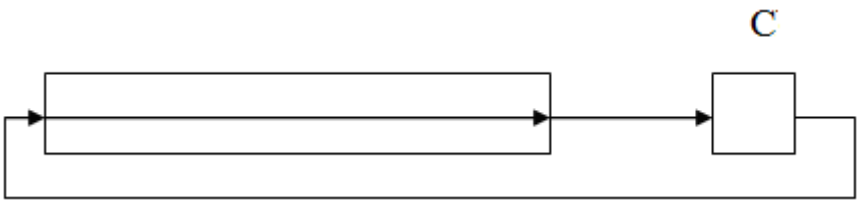
Пример.
Исходное значение операнда: 00110011_2
После циклического сдвига вправо на 1 бит:
значение операнда – 10011001_2, C=1

Рис. 2.41. Общая схема и пример выполнения циклического сдвига вправо на 1 бит



Пример.
Исходное значение операнда: 00110011_2
Исходное состояние флага переноса: C=1
После циклического сдвига влево на 1 бит через перенос:
значение операнда – 01100111_2, C=0

Рис. 2.42. Общая схема и пример выполнения циклического сдвига влево через перенос на 1 бит



Пример.
Исходное значение операнда: 00110011_2
Исходное состояние флага переноса: C=0
После циклического сдвига вправо на 1 бит через перенос:
значение операнда – 00011001_2, C=1

Рис. 2.43. Общая схема и пример выполнения циклического сдвига вправо через перенос на 1 бит

Логические сдвиги на 1 бит (см. рис. 2.37 и 2.38) состоят в перемещении каждого бита операнда на 1 позицию влево или, соответственно, вправо, с записью нуля в освобождающийся младший бит (при сдвиге влево) или старший бит (при сдвиге вправо). При этом старший (при сдвиге влево) или младший (при сдвиге вправо) бит операнда записывается в разряд признака переноса регистра статуса. Нетрудно заметить, что логический сдвиг влево или вправо на 1 бит равносилен умножению или, соответственно, делению беззнакового числа на 2.

По аналогичным правилам выполняется логический сдвиг влево или вправо на некоторое большее единицы число бит. Он эквивалентен умножению или, соответственно, делению беззнакового операнда на 2^k, где k – число бит, на которое сдвигается операнд.

Арифметический сдвиг (см. рис. 2.39) отличается от логического тем, что при нем сохраняется знак операнда, т. е. после сдвига положительное число остается положительным, а отрицательное – отрицательным. Практический смысл имеет только арифметический сдвиг вправо. В частности, в системах команд МК семейств *AVR* и *ARM Cortex-Mx* имеется только арифметический сдвиг вправо; арифметический сдвиг влево отсутствует. Общая схема выполнения арифметического сдвига вправо аналогична таковой логического сдвига, за исключением того, что состояние старшего значащего разряда при арифметическом сдвиге не изменяется, независимо от

числа бит, на которое осуществляется сдвиг. Арифметический сдвиг вправо эквивалентен делению на 2^k операнда со знаком, представленного в дополнительном коде, где k – число бит, на которое сдвигается операнд (см. рис. 2.39).

Необходимо отметить, что умножение на 2^k операнда со знаком осуществляется логическим сдвигом на k бит, при условии, что результат сдвига не выйдет за пределы диапазона чисел со знаком соответствующей разрядности (см. табл. 2.1).

Циклические сдвиги (см. рис. 2.40 и 2.41) отличаются от логических тем, что в освобождающийся бит записывается значение бита, «выталкиваемого» в результате сдвига. Циклические сдвиги через перенос (см. рис. 2.42 и 2.43) характеризуются тем, что бит (флаг) переноса служит в качестве одного из битов операнда, подвергаемого сдвигу.

Следует отметить, что логические, арифметические и описываемые рис. 2.40 и 2.41 циклические сдвиги могут быть реализованы посредством циклических сдвигов через перенос. Например, логический сдвиг может быть выполнен как последовательность операций, состоящая из установки в ноль бита переноса с последующим циклическим сдвигом через перенос. Поэтому команды сдвига ряда МК класса «*cost-sensitive*», в частности, некоторых МК семейства *PIC*, включают в себя только циклические сдвиги влево и вправо через перенос [4].

Команды передачи управления включают в себя [4, 8, 18]:

- команды безусловных переходов;
- команды условных переходов;
- команды вызова подпрограмм;
- команды возврата из подпрограмм, в том числе из подпрограмм обслуживания прерываний.

Команды безусловных переходов передают управления некоторой команде, абсолютный или относительный адрес которой (подробнее см. пункт 2.6.3) указан в коде команды безусловного перехода.

Примеры команд безусловных переходов:

- *RJMP label1* (система команд МК семейства *AVR* [8]) – безусловный переход с относительной адресацией (см. пункт 2.6.3) к команде с меткой *label1*;

- *JMP $3FF00* (система команд МК семейства *AVR* [8]) – безусловный переход с абсолютной адресацией (см. пункт 2.6.3) к команде с адресом *3FF00* в шестнадцатеричной системе счисления;

- *B label2* (система команд МК семейства *ARM Cortex-Mx* [18]) – безусловный переход с относительной адресацией (см. пункт 2.6.3) к команде с меткой *label2.*

Команды условных переходов передают управление команде, абсолютный или относительный адрес которой указан в их коде, если соблюдается условие перехода (например, результат выполнения предыдущей команды равен 0). Если же условие перехода не соблюдается – управление передается команде, следующей за командой условного перехода. Примеры команд условных переходов:

- *BREQ label3* (система команд МК семейства *AVR* [8]) – переход к команде с меткой *label3*, если результат выполнения предшествующей команды равен 0 (установлен в 1 флаг *Z* регистра статуса); в противном случае – переход к команде, следующей за *BREQ label3*;

- *BNE label4* (система команд МК семейства *ARM Cortex-Mx* [18]) – переход к команде с меткой *label4*, если результат выполнения предшествующей команды не равен 0 (флаг *Z* регистра статуса); в противном случае – переход к команде, следующей за *BNE label4*.

Команды вызова подпрограмм (т. е. некоторых блоков программного кода, многократно используемых при выполнении программы и доступных для вызова из различных ее точек) осуществляют передачу управления начальной команде соответствующей подпрограммы, с предварительной автоматической загрузкой в стек адреса возврата из подпрограммы (см. рис. 2.2 и пояснения к нему). Вызовы подпрограмм могут быть как безусловными, так и условными. Примеры команд вызова подпрограмм:

- *CALL $3F00* (система команд МК семейства *AVR* [8]) – вызов подпрограммы, расположенной в ПП, начиная с адреса *3F00* в шестнадцатеричной системе счисления;

- *BL label5* (система команд МК семейства *ARM Cortex-Mx* [18]) – переход к подпрограмме, начальная команда которой снабжена меткой *label5*; при этом адрес возврата в основную программу автоматически запоминается в регистре связи *LR* (*Link Register*) (см. рис. 2.34);

- *BLEQ label6* (система команд МК семейства *ARM Cortex-Mx* [18]) – условный (при единичном флаге *Z*) переход к подпрограмме, начальная команда которой снабжена меткой *label6*, с запоминанием адреса возврата в регистре *LR*.

После выполнения подпрограммы должен быть обеспечен возврат в вызвавшую ее программу (подпрограмму). Он реализуется специальными командами возврата, которыми в обязательном порядке должна заканчиваться каждая подпрограмма. В системах команд большинства семейств МК таковыми являются команды *RET* («Возврат из подпрограммы») и *RETI (*или *IRET)* (возврат из подпрограммы обслуживания прерывания). Команда *RET* осуществляет загрузку из стека в программный счетчик адреса возврата в вызвавшую программу (подпрограмму) (см. рис. 2.2). Команда *RETI (IRET)*, кроме этого, выполняет установку флага глобального разрешения прерываний (см., например, рис. 2.24).

В системах команд МК семейства *ARM Cortex-Mx* специальная команда возврата из подпрограмм отсутствует. Возврат осуществляется командой пересылки содержимого регистра связи (*LR*) в программный счетчик.

Наконец, в систему команд каждого семейства МК входят **команды специального назначения**, типовыми примерами которых являются [8, 18]:

- команда перевода МК в режим пониженного энергопотребления;

- команда сброса сторожевого таймера;

и т. п.

2.6.3. Структура кода команды. Способы адресации

Независимо от того, записана ли команда МК в двоичном коде или на языке ассемблера, она содержит, в явном или в неявном виде, следующую информацию [3]:

- код выполняемой операции;
- адреса операндов;
- адрес, по которому должен быть помещен результат выполнения команды;
- адрес следующей команды.

В системах команд практически всех семейств МК **код выполняемой операции** непосредственно указывается в двоичном или ассемблерном слове команды. В качестве примера приведем код команды младшего подсемейства *PIC*-МК [4], выполняющей сложение содержимого регистра-аккумулятора (см. рис. 2.1) и некоторого ЗЭ памяти данных:

- двоичный код команды – *000111dfffff*; здесь *000111* – код операции сложения, *fffff* – номер ЗЭ в пределах выбранного банка ПД (см. рис. 2.16), *d* – бит – указатель приемника результата (при $d = 0$ результат записывается в аккумулятор, а при $d = 1$ – по адресу второго операнда);
- код данной команды на языке ассемблера соответствующего подсемейства МК – *ADDWF f, d*; здесь кодом операции является аббревиатура *ADDWF*.

Значительно более разнообразны способы **адресации** (т. е. задания адресов) операндов, результата и следующей команды. В системах команд МК общего назначения применяются следующие основные способы адресации [4, 8, 18]:

- подразумеваемая, или неявная (*implied or implicit addressing*);
- прямая регистровая (*register direct addressing*);
- косвенная регистровая (*register indirect addressing*);
- непосредственная (*immediate addressing*);
- абсолютная (*absolute addressing*).

При **подразумеваемой адресации** адрес одного из операндов или / и результата или / и следующей команды не указывается в

явном виде, а определяется кодом команды. Например, в вышеописанной команде *ADDWF f, d* данным способом адресуются:

- первый операнд, которым, в соответствии с кодом операции, является содержимое аккумулятора (что характерно для двухоперандных команд многих МК класса «*cost-sensitive*»);

- следующая команда, в качестве которой, очевидно, служит расположенная в ПП непосредственно после команды *ADDWF f, d*, поскольку *ADDWF f, d* не является командой безусловного или условного перехода.

Еще одним примером использования подразумеваемой адресации является команда *ADD R0,R1* (система команд МК семейства *AVR*), выполняющая суммирование содержимых РОН *R0* и *R1* с записью результата в регистр *R0*. В данной команде неявно адресуются результат (в соответствии с кодом команды он записывается на место 1-го операнда) и следующая команда, которой, как и в предыдущем примере, является следующая по порядку расположения в ПП.

Необходимо отметить, что адрес следующей команды является подразумеваемым **во всех** командах, кроме команд передачи управления (см. пункт 2.6.2).

Прямая регистровая адресация используется, в основном, для указания адресов операндов и результатов. При данном способе адресации операндов в их качестве служит содержимое РОН (реже – РСФ), а при прямой регистровой адресации результатов они записываются также в РОН или РСФ. В коде команды при этом указываются номера регистров, содержащих операнды и / или выделенных для записи результата. Типичным примером применения прямой регистровой адресации операндов (при подразумеваемой адресации результата) является вышеприведенная команда *ADD R0,R1* [8]. Примером команды с прямой регистровой адресацией как операндов, так и результата является команда *ADD R0, R1, R2* (система команд МК семейства *ARM Cortex-Mx* [18]), выполняющая суммирование содержимого РОН *R1* и *R2* с записью результата в РОН *R0*.

Косвенная регистровая адресация. Используется для указания адресов как операндов и результатов, так и команд. В большинстве

источников под косвенной регистровой адресацией подразумевается только ее простейший вариант, при котором адрес операнда или команды (исполнительный адрес) содержится в некотором РОН или РСФ, а в коде команды указывается номер соответствующего РОН (РСФ). Типичными примерами применения простейшего варианта косвенной регистровой адресации являются следующие команды [18]:

- *LDR R8, [R10]* (система команд МК семейства *ARM Cortex-Mx*) – загрузка в РОН *R8* содержимого ЗЭ ПД, адрес которого хранится в РОН *R10*;

- *BLX R0* (система команд МК семейства *ARM Cortex-Mx*) – вызов подпрограммы с начальным адресом, хранящимся в *РОН R0*.

По мнению автора, к косвенной регистровой адресации могут также отнесены способы, при которых исполнительный адрес также хранится в некотором РОН или РСФ, но его содержимое некоторым образом модифицируется перед выполнением или после выполнения команды. Наиболее распространенными из них являются [8, 18]:

- автоинкрементная косвенная адресация с пре- и постинкрементом;

- автодекрементная косвенная адресация с пре- и постдекрементом;

- косвенная базовая (относительная) адресация.

В режимах **автоинкрементной** и **автодекрементной** косвенной адресации исполнительный адрес хранится в некотором РОН или (реже) РСФ, выполняющем функцию указателя. При каждом выполнении команды он автоматически увеличивается (автоинкремент) или уменьшается (автодекремент) на определенное значение. Увеличение (уменьшение) происходит перед выполнением команды (преинкремент, предекремент) или после ее выполнения (постинкремент, постдекремент). Данные режимы адресации удобны при обработке массивов данных (после считывания или записи очередного элемента массива регистр – указатель автоматически устанавливается на адрес следующего элемента). Примеры использования данных режимов адресации:

- *LD R16,X+* (система команд МК семейства *AVR* [8) – загрузка в РОН *R16* содержимого ЗЭ ПД, адрес которого хранится в регистровой паре *X* (см. рис. 2.4), с последующим увеличением указанного адреса на 1;

- *LD R18,-Y* (система команд МК семейства *AVR* [8]) – загрузка в РОН *R18* содержимого ЗЭ ПД, адрес которого хранится в регистровой паре *Y* (см. рис. 2.4), с уменьшением указанного адреса на 1 до выполнения загрузки;

- *STR R3, [R9], #4* (система команд МК семейства *ARM Cortex-Mx* [18]) – загрузка содержимого РОН *R3* в ЗЭ ПД, адрес которого хранится в РОН *R9*, с последующим увеличением указанного адреса на 4;

- *LDR R3, [R5], #-4* (система команд МК семейства *ARM Cortex-Mx* [18]) – загрузка в РОН *R3* содержимого ЗЭ ПД, адрес которого хранится в РОН *R5*, с последующим уменьшением указанного адреса на 4.

В режиме **базовой (относительной) косвенной адресации** исполнительный адрес вычисляется как сумма базового адреса (содержимого некоторого РОН или РСФ, называемого базовым) и смещения, которое может быть постоянным или переменным. Данный режим адресации в ряде случаев удобен при работе с массивами, а также при обращениях к РСФ семейств МК, у которых адрес РСФ состоит из базового адреса периферийного устройства и смещения относительно него (см. рис. 2.32). Примеры команд с использованием базовой адресации [18]:

- *STR R9, [R1, #0x34]* (система команд МК семейства *ARM Cortex-Mx*) – загрузка содержимого РОН *R9* в ЗЭ ПД, адрес которого равен сумме базового адреса, хранящегося в РОН *R1,* и смещения относительно базового адреса, равного 34 в шестнадцатеричной системе счисления;

- *STR R9, [R1, R2]* (система команд МК семейства *ARM Cortex-Mx*) – загрузка РОН *R9* в ЗЭ ПД, адрес которого равен сумме содержимого РОН *R1* и *R2*.

Важным частным случаем базовой (относительной) адресации, применяемым в командах передачи управления, является указание исполнительного адреса с использованием программного счетчика в

качестве базового регистра (***PC-relative addressing***). Адрес команды, которой передается управление, при этом вычисляется как сумма содержимого программного счетчика и смещения, приводимого в двоичном коде команды. В коде команды на языке ассемблера, как правило, указывается только метка команды, которой передается управление, а вычисление смещения происходит при трансляции ассемблерного кода в двоичный. Базовый регистр при таком способе адресации в коде команды не указывается; по умолчанию подразумевается, что в его качестве служит программный счетчик.

Примеры команд с использованием такого способа адресации [8, 18]:

- *RCALL label7* (система команд МК семейства *AVR* [8]) – вызов подпрограммы, начальная команда которой снабжена меткой *label7*;

- *B label5* (система команд МК семейства *ARM Cortex-Mx* [18]) – безусловный переход к команде с меткой *label5*.

Непосредственная адресация применяется только к операндам и характеризуется тем, что в коде команды приводится значение операнда, а не его адрес. Удобна при работе с константами. Примеры непосредственной адресации [8, 18]:

- *SUBI R16, $33* (система команд МК семейства *AVR* [8]) – вычитание числа 33 в шестнадцатеричной системе счисления из содержимого РОН *R16*;

- *MOV R1, #0xFA05* (система команд МК семейства *ARM Cortex-Mx* [18]) – запись в РОН *R1* числа *FA05* в шестнадцатеричной системе счисления.

Абсолютная адресация используется, в основном, в командах переходов (безусловных и условных) и вызова подпрограмм, и служит для указания адреса команды, которой передается управление. Абсолютное значение данного адреса указывается непосредственно в коде команды. Примеры использования данного способа адресации [8]:

- *JMP $3F00* (система команд МК семейства *AVR*) – безусловный переход к команде, расположенной в ПП по адресу *3F00* в шестнадцатеричной системе счисления;

- *CALL $3F00* (система команд МК семейства *AVR*) – вызов подпрограммы, расположенной в ПП, начиная с адреса *3F00* в шестнадцатеричной системе счисления.

2.6.4. Конвейер команд

Как указано ранее (см. выводы по подразделу 2.2), эффективное время выполнения большинства команд современных МК составляет 1 такт, ввиду наличия **конвейера команд**. Его реализация, в свою очередь, возможна благодаря Гарвардской *RISC* – архитектуре, являющейся базовым структурно-архитектурным решением подавляющего большинства современных МК общего назначения (см. подраздел 1.1).

Как указано в подразделе 1.1, система команд *RISC*-МК включает в себя только набор наиболее часто используемых простых команд, которые требуют для их выполнения минимальное количество тактов, как правило, два – один для выборки команды из ПП, другой – для ее выполнения (см. Приложение А). С другой стороны, благодаря физическому разделению ПП и ПД, характерному для Гарвардской архитектуры, в реализованных на ее основе МК возможно параллельное во времени выполнение текущей команды (как правило, требующее обращения к ПД) и извлечение из ПП следующей команды, т. е. конвейеризация извлечения и выполнения команд. Таким образом может быть реализован конвейер команд с **коэффициентом совмещения операций** (т. е. с количеством одновременно реализуемых этапов выполнения команды), равным 2-м. Временная диаграмма работы такого конвейера представлена на рис. 2.44.

Конвейер, временная диаграмма работы которого приведена на рис. 2.44, применяется в 8-битовых МК семейств *PIC* [17] и *AVR* [6], а также в ряде других семейств МК классов «*cost-sensitive*» и «*mainstream*».

В ряде МК класса «*high-performance*», в том числе в МК семейства *ARM Cortex-Mx*, ввиду относительной сложности системы команд по сравнению с МК классов «*cost-sensitive*» и «*mainstream*», выделяется дополнительный этап выполнения команды – ее декодирование. Коэффициент совмещения операций конвейера

команд при этом равен 3-м [9]. Временная диаграмма его работы приведена на рис. 2.45.

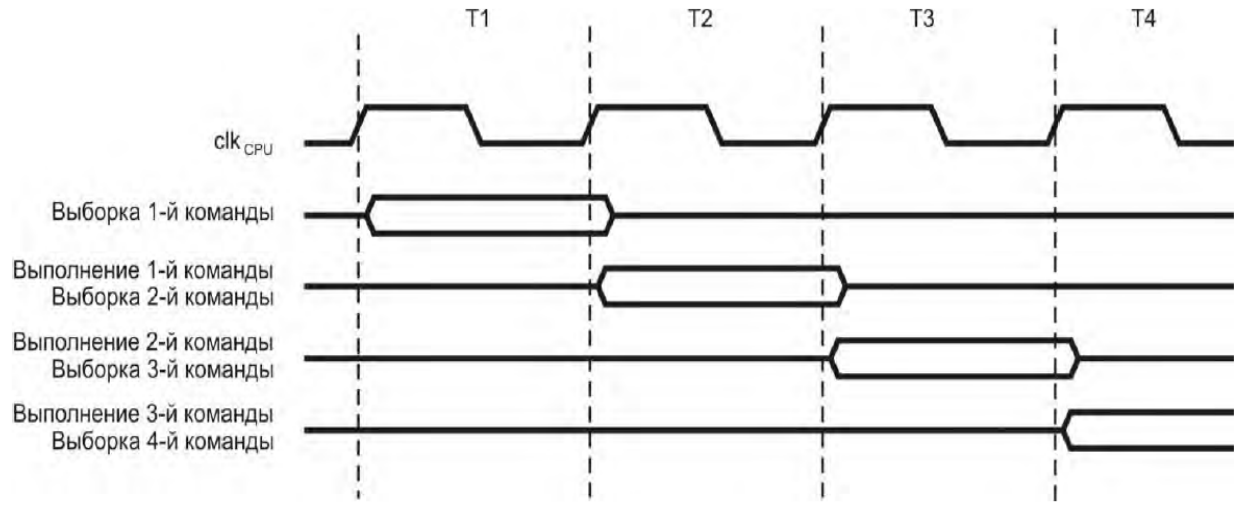


Рис. 2.44. Типовая временн**а**я диаграмма работы конвейера команд с коэффициентом совмещения операций, равным 2-м

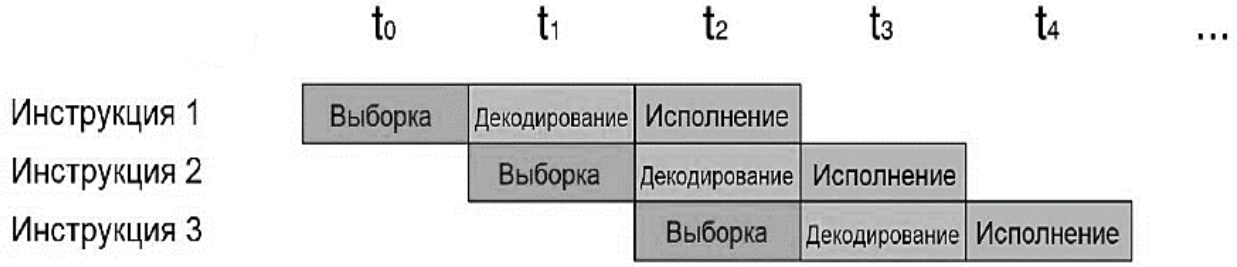


Рис. 2.45. Временн**а**я диаграмма работы конвейера команд ЦП МК семейства *ARM Cortex-Mx* с коэффициентом совмещения операций, равным 3-м (время указано в тактах ЦП) [9]

В результате применения конвейера команд эффективное время выполнения большинства из них в процессе работы программы равно длительности одного такта ЦП, за исключением:

- относительно сложных математических операций (умножения, деления и т. п.);

- команд, требующих манипуляций с программным счетчиком или / и перезагрузки конвейера (безусловные и условные переходы, вызов подпрограммы и возврат из нее и т. п.).

Вышесказанное иллюстрируют представленные далее в табл. 2.6 – 2.8 описания систем команд распространенных семейств МК общего назначения.

2.6.5. Типовые примеры систем команд МК общего назначения

В качестве типовых примеров систем команд МК общего назначения в табл. 2.6 – 2.8 представлены синтаксис и краткие описания основных ассемблерных команд:

- младшего подсемейства *PIC*-МК, относящихся к классу «*cost-sensitive*» (табл. 2.6);
- МК семейства *AVR* класса «*mainstream*» (табл. 2.7);
- МК семейства *ARM Cortex-Mx,* которые могут быть отнесены к классу «*high performance*» (табл. 2.8).

При этом система команд младшего подсемейства *PIC*-МК является показательным примером простейшей системы ассемблерных команд, а система команд МК семейства *ARM Cortex-Mx* – примером достаточно развитого языка ассемблера МК.

Примечание. Система команд МК семейства *ARM Cortex-Mx* представляет собой упрощенный вариант более развитой системы команд процессоров *ARM*, разрядностью кодов которых равна 32-м битам [9]. С учетом того, что МК семейства *ARM Cortex-Mx* не предназначены для решения сложных вычислительных задач, для них была разработана система команд *Thumb* [9], включающая в себя наиболее часто используемые *ARM*-команды, разрядность кодов которых уменьшена до 16-ти битов, с целью экономии объема памяти программ. При декодировании *Thumb*-команды автоматически преобразуются в соответствующие им 32-битовые *ARM*-команды. Подсемействами от *ARM Cortex-M3* и выше поддерживается также система команд *Thumb-2* [9], более развитая, чем *Thumb*, и включающая в себя как 16-, так и 32-битовые команды. См. также рис. 2.5 и примечания к табл. 2.8.

Таблица 2.6

Основные команды ассемблера младшего подсемейства PIC-МК [4]

Синтаксис	**Описание**	**Биты регистра статуса, изменяемые командой (см. рис. 2.19)**	T_{clk}
1	2	3	4
Команды пересылки данных			
MOVF f, d	Пересылка «регистр – аккумулятор» W ← f при d = 0; f ← f [1)] при d = 1	Z	1
MOVWF f	Пересылка «аккумулятор – регистр» f ← W	-	
MOVLW k	Непосредственная загрузка аккумулятора W ← k		
Арифметические операции			
ADDWF f, d	Сложение W ← W + f при d = 0; f ← W + f при d = 1	Z, DC, C	1
SUBWF f, d	Вычитание W ← f – W при d = 0; f ← f – W при d = 1		
INCF f, d	Инкремент W ← f + 1 при d = 0; f ← f + 1 при d = 1	Z	
DECF f, d	Декремент W ← f – 1 при d = 0; f ← f – 1 при d = 1		
Логические операции			
ANDWF f, d	Побитовое логическое И W ← f ∧W при d = 0; f ← f ∧W при d = 1	Z	1
ANDLW k	Побитовое логическое И с константой W ← W ∧ k		

Продолжение таблицы 2.6

1	2	3	4
IORWF f, d	Побитовое логическое ИЛИ W ← f ∨W при d = 0; f ← f ∨W при d = 1	Z	1
IORLW k	Побитовое логическое ИЛИ с константой W ←W∨ k		
XORWF f, d	Побитовое исключающее ИЛИ W ← f ⊕W при d = 0; f ← f ⊕W при d = 1		
XORLW k	Побитовое исключающее ИЛИ с константой W ← W⊕ k		
COMF f, d	Побитовая инверсия W ← FFh – f при d = 0; f ← FFh – f при d = 1		
CLRF f	Сброс регистра f ← 00h		
CLRW	Сброс аккумулятора W ← 00h		
Операции сдвигов			
RLF f, d	Циклический сдвиг влево на 1 бит содержимого регистра f через бит C регистра статуса См. рис. 2.42	C	1
RRF f, d	Циклический сдвиг вправо на 1 бит содержимого регистра f через бит C регистра статуса См. рис. 2.47		
SWAPF f, d	Перестановка тетрад регистра f f(0…3) ↔ f(4…7)	-	
Примечание. Во всех операциях сдвигов при d = 0 результат сохраняется в регистре W, а при d = 1 – в регистре f			
Битовые операции			
BCF f, b	Сброс бита в регистре f f(b) ← 0	-	1
BSF f, b	Установка бита в регистре f f(b) ← 1		

Продолжение таблицы 2.6

1	2	3	4
Команды передачи управления			
INCFSZ f, d	Инкремент и пропуск следующей команды при нулевом результате W ← f + 1 при d = 0; f ← f + 1 при d = 1; пропуск следующей команды при нулевом значении f + 1	-	1(2)[2)]
DECFSZ f, d	Декремент и пропуск следующей команды при нулевом результате W ← f - 1 при d = 0; f ← f - 1 при d = 1; пропуск следующей команды при нулевом значении f - 1	-	1(2)[2)]
BTFSC f, b	Пропуск следующей команды при f(b) = 0		
BTFSS f, b	Пропуск следующей команды при f(b) = 1		
GOTO k	Безусловный переход с абсолютной адресацией[3)] PC[8…0] ← k, 0 ≤ k ≤ 511, PC[9]← STATUS[5]		2
CALL k	Вызов подпрограммы с абсолютной адресацией[4)] STACK ← PC + 1, PC[7…0] ← k, 0 ≤ k ≤ 255, PC[8]← 0 PC[9]← STATUS[5]		
RETLW k	Возврат из подпрограммы с непосредственной загрузкой аккумулятора[5)] PC ← STACK, W ← k		
Специальные операции			
NOP	Пустая операция	-	1
SLEEP	Перевод МК в режим пониженного энергопотребления	TO, PD	
CLRWDT	Сброс сторожевого таймера		
OPTION[6)]	OPTION ← W	-	

Окончание таблицы 2.6

1	2	3	4
TRIS f	Загрузка регистра управления портом ввода / вывода TRIS f ← W	-	1

Примечания.

T_{clk} – время выполнения команды в тактах ЦП;

f – регистр с номером f в пределах текущего банка и/или его содержимое;

W – аккумулятор и/или его содержимое;

k – константа, если не оговорено иное – 8-битовая в дополнительном коде;

b – номер бита;

STACK – вершина стека;

PC – содержимое программного счетчика;

f(b), W(b), PC(b) – состояние b-го бита соответствующего регистра;

Z, DC, C, TO, PD – соответствующие биты регистра статуса (см. рис. 2.19).

[1] Данный вариант может использоваться для проверки содержимого регистра на ноль.

[2] При выполнении условия пропуска следующей команды время выполнения равно 2-м тактам, в противном случае – одному такту.

[3] См. рис. 2.14.

[4] См. рис. 2.2 и 2.14.

[5] См. рис. 2.2.

[6] OPTION – доступный только для записи РСФ младшего подсемейства PIC-МК, содержащий биты управления системой сброса, портами ввода / вывода и таймерами. У данного РСФ отсутствует адрес в пространстве ПД; обращение к нему возможно только посредством команды OPTION.

Таблица 2.7

Основные команды ассемблера МК семейства AVR [8]

Синтаксис	**Описание**	**Биты регистра статуса, изменяемые командой (см. рис. 2.24)**	T_{clk} [1)]
1	2	3	4
Команды пересылки данных			
MOV Rd, Rs	Пересылка «РОН – РОН» Rd←Rs	-	1
LDI Rd, k	Загрузка константы в РОН[2)] Rd←k		
	Загрузка РОН косвенно адресуемым операндом		
LD Rd, X	Rd←(X)		2
LD Rd, Y	Rd←(Y)		
LD Rd, Z	Rd←(Z)		
	Загрузка РОН операндом, адресуемым косвенно с пост- инкрементом		
LD Rd, X+	Rd←(X), X←X+1		2
LD Rd, Y+	Rd←(Y), Y←Y+1		
LD Rd, Z+	Rd←(Z), Z←Z+1		
	Загрузка РОН операндом, адресуемым косвенно с пре- декрементом		
LD Rd, -X	X←X-1, Rd←(X)		2
LD Rd, -Y	Y←Y-1, Rd←(Y)		
LD Rd, -Z	Z←Z-1, Rd←(Z)		
	Загрузка РОН операндом, адресуемым косвенной базовой адресацией		
LDD Rd, X+q	Rd←(X + q)		2
LDD Rd, Y+q	Rd←(Y + q)		
LDD Rd, Z+q	Rd←(Z + q)		
LDS Rd, ADM	Загрузка РОН абсолютно адресуемым байтом ПД Rd←(ADM)		2
LPM	Загрузка РОН косвенно адресуемым байтом ПП R0←PM(Z)		3

Продолжение таблицы 2.7

1	2	3	4
	Запоминание содержимого РОН в косвенно адресуемой ячейке ПД	-	
ST X, Rs	(X)←Rs		2
ST Y, Rs	(Y)←Rs		
ST Z, Rs	(Z)←Rs		
	Запоминание содержимого РОН в ячейке ПД, адресуемой косвенно с пост- инкрементом		
ST X+, Rs	(X)←Rs, X←X+1		2
ST Y+, Rs	(Y)←Rs, Y←Y+1		
ST Z+, Rs	(Z)←Rs, Z←Z+1		
	Запоминание содержимого РОН в ячейке ПД, адресуемой косвенно с пре- декрементом		
ST -X, Rs	X←X-1, (X)←Rs		2
ST -Y, Rs	Y←Y-1, (Y)←Rs		
ST -Z, Rs	Z←Z-1, (Z)←Rs		
	Запоминание содержимого РОН в ячейке ПД, адресуемой косвенной базовой адресацией		
STD Y+q, Rs	(Y + q)←Rs		2
STD Z+q, Rs	(Z + q)←Rs		
STS ADM,Rs	Запоминание содержимого РОН в абсолютно адресуемой ячейке ПД (ADM)← Rs		
SPM[3)]	Запоминание содержимого регистровой пары R1:R0 в косвенно адресуемом ЗЭ ПП PM(Z)←R1:R0		-[4)]
IN Rd, ADS	Загрузка РОН содержимым абсолютно адресуемого РСФ Rd←(ADS)		1
OUT ADS, Rs	Запоминание содержимого РОН в абсолютно адресуемом РСФ (ADS)← Rs		

Продолжение таблицы 2.7

1	2	3	4
PUSH Rs	Запоминание содержимого РОН в стеке (SP)←Rs, SP←SP-1	-	2
POP Rd	Загрузка РОН из стека SP←SP+1, Rd←(SP)		
Арифметические операции			
ADD Rd, Rs	Сложение Rd←Rd + Rs	Z, C, N, V, S, H	1
ADC Rd, Rs	Сложение с учетом состояния бита C[5)] Rd←Rd + Rs + C		
SUB Rd, Rs	Вычитание Rd←Rd - Rs		
SBC Rd, Rs	Вычитание с учетом состояния бита C[5)] Rd←Rd – Rs - C		
SUB Rd, k	Вычитание константы[2)] Rd←Rd - k		
SBCI Rd, k	Вычитание константы[2)] с учетом состояния бита C[5)] Rd←Rd - k - C		
CP Rd, Rs	Сравнение[6)] Rd - Rs		
CPC Rd, Rs	Сравнение[6)] с учетом состояния бита C[5)] Rd - Rs - C		
CPI Rd, k	Сравнение[6)] с константой[2)] Rd - k		
NEG Rd	Дополнение содержимого РОН Rd←00h - Rd		
INC Rd	Инкремент Rd←Rd + 1	Z, N, V	
DEC Rd	Декремент Rd←Rd - 1		
MUL Rd, Rs	Перемножение содержимых 2-х РОН как беззнаковых чисел[3)] R1:R0←Rd×Rs	Z, C	2

Продолжение таблицы 2.7

1	2	3	4
MULS Rd, Rs	Перемножение содержимых 2-х РОН как чисел со знаком в дополнительном коде[3)] R1:R0←Rd×Rs	Z, C	2
Логические операции			
AND Rd, Rs	Логическое И Rd←Rd∧Rs	Z, N, V, S	1
ANDI Rd, k	Логическое И с константой[2)] Rd←Rd∧ k		
OR Rd, Rs	Логическое ИЛИ Rd←Rd∨Rs		
ORI Rd, k	Логическое ИЛИ с константой[2)] Rd←Rd∨ k		
EOR Rd, Rs	Исключающее ИЛИ Rd←Rd⊕Rs		
SBR Rd, k	Установка битов в РОН[2)] Rd←Rd∨ k		
CBR Rd, k	Сброс битов в РОН[2)] Rd← Rd∧ (FFh - k)		
TST Rd	Проверка содержимого РОН[6)] Rd←Rd∧Rd		
CLR Rd	Обнуление содержимого РОН Rd←00h		
COM Rd	Инверсия содержимого РОН Rd←FFh - Rd	Z, C, N, V, S	
SER Rd	Установка всех битов РОН в единичное состояние Rd←FFh	-	
Операции сдвигов			
ASR Rd	Арифметический сдвиг содержимого РОН Rd вправо на 1 бит См. рис. 2.43	Z, C, N, V	1
LSL Rd	Логический сдвиг содержимого РОН Rd влево на 1 бит См. рис. 2.41	Z, C, N, V, H	
LSR Rd	Логический сдвиг содержимого РОН Rd вправо на 1 бит См. рис. 2.42	Z, C, N, V	

Продолжение таблицы 2.7

1	2	3	4
ROL Rd	Циклический сдвиг содержимого РОН Rd на 1 бит влево через бит С См. рис. 2.46	Z, C, N, V, H	1
ROR Rd	Циклический сдвиг содержимого РОН Rd на 1 бит вправо через бит С См. рис. 2.47	Z, C, N, V	
SWAP Rd	Перестановка тетрад РОН Rd(0…3)↔Rd(4…7)	-	
Битовые операции			
BST Rs, b	Запоминание b-го бита РОН Rs в бите Т регистра статуса (см. рис. 2.23) T← Rs(b)	T	1
BLD Rd, b	Загрузка b-го бита РОН Rd битом Т регистра статуса (см. рис. 2.23) Rd (b) ← T	-	
BSET b	Установка в «1» b-го бита регистра статуса		
BCLR b	Сброс b-го бита регистра статуса		
SBI ADS, b	Установка в «1» b-го бита РСФ с адресом ADS	-	2
CBI ADS, b	Сброс в «0» b-го бита РСФ с адресом ADS		
Команды передачи управления			
RJMP k[7)]	Безусловный переход с относительной адресацией PC←PC + k + 1	-	2
RCALL k[7)]	Вызов подпрограммы с относительной адресацией (SP)←PC + 1, SP←SP - (2 или 3)[8)], PC←PC + k + 1		3/4[9)]
IJMP	Безусловный переход с косвенной адресацией PC(0…15)←Z, PC(16…21) ←0		2
ICALL	Вызов подпрограммы с косвенной адресацией (SP)←PC+1, SP←SP-(2 или 3)[8)] PC(0…15)←Z, PC(16…21) ←0		3/4[9)]

Продолжение таблицы 2.7

1	2	3	4
JMP АРМ[10)]	Безусловный переход с абсолютной адресацией[3)] PC←АРМ	-	3
CALL АРМ[10)]	Вызов подпрограммы с абсолютной адресацией[3)] (SP)←PC + 2, SP←SP - (2 или 3)[8)], PC← АРМ		4/5[11)]
RET	Возврат из подпрограммы (см. рис.2.2) PC←(SP), SP←SP+(2 или 3)[8)]		4/5[11)]
RETI	Возврат из подпрограммы обслуживания прерывания I[12)]←1, PC←(SP), SP←SP+(2 или 3)[9)]	I	4/5[11)]
	Пропуск следующей команды при:		
CPSE Rd, Rs	Равенстве содержимых РОН Rd и Rs	-	1/2/ 3[13)]
SBRC Rd, b	Нулевом b-ом бите РОН Rd		
SBRS Rd, b	Единичном b-ом бите РОН Rd		
SBIC ADS, b	Нулевом b-ом бите РСФ с адресом ADS		
SBIS ADS, b	Единичном b-ом бите РСФ с адресом ADS		
При выполнении всех команд пропуска: PC←PC + (2 или 3)[14)], если условие пропуска выполняется; иначе PC←PC + 1			
	Условные переходы с относительной адресацией: если условие перехода выполняется, PC←PC + k[7)] + 1; иначе PC←PC + 1		
BREQ k[7)]	Переход, если равно 0 (при Z=1)	-	1/2[15)]
BRNE k[7)]	Переход, если не равно 0 (при Z=0)		
BRCS k[7)]	Переход при наличии переноса (C = 1)		
BRCC k[7)]	Переход при отсутствии переноса (C = 0)		
BRSH k[7)]	Переход, если больше или равно (применительно к числам без знака), условие перехода – C = 0		

Продолжение таблицы 2.7

1	2	3	4
BRLO k[7)]	Переход, если меньше (применительно к числам без знака), условие перехода – C = 1	-	1/2[15)]
BRGE k[7)]	Переход, если больше или равно (применительно к числам со знаком), условие перехода – S = N⊕V = 0		
BRLT k[7)]	Переход, если меньше (применительно к числам со знаком), условие перехода – S = N⊕V = 1		
BRMI k[7)]	Переход, если результат отрицателен (при N=1)		
BRPL k[7)]	Переход, если результат положителен (при N=0)		
BRVS k[7)]	Переход при переполнении (V = 1)		
BRVC k[7)]	Переход при отсутствии переполнения (V = 0)		
BRTS k[7)]	Переход при T = 1		
BRTC k[7)]	Переход при T = 0		
BRHS k[7)]	Переход при наличии переноса из младшей тетрады в старшую (H = 1)		
BRHC k[7)]	Переход при отсутствии переноса из младшей тетрады в старшую (H = 0)		
BRIE k[7)]	Переход при глобальном разрешении прерываний (I = 1)		
BRID k[7)]	Переход при глобальном запрете прерываний (I = 0)		
Специальные операции			
NOP	Пустая операция	-	1
SLEEP	Перевод МК в режим пониженного энергопотребления		
WDR	Сброс сторожевого таймера		
Примечания. T_{clk} – время выполнения команды в тактах ЦП; Rd, Rs – содержимые РОН с номерами d и s соответственно; X, Y, Z – содержимые регистровых пар X, Y и Z соответственно; SP – содержимое указателя стека; PC – содержимое программного счетчика;			

Продолжение таблицы 2.7

1	2	3	4
ADS – адрес РСФ, 00h ≤ d ≤ 3Fh (см. рис. 2.22 и 2.25); ADM – адрес ПД; APM – адрес ПП; (X), (Y), (Z), (ADS), (ADM) – содержимое ЗЭ ПД с адресом X, Y, Z ADS и ADM соответственно; (SP) – содержимое ЗЭ аппаратного стека с адресом SP; PM(Z) – содержимое ЗЭ ПП с адресом Z; Rd(b), ADS(b), SR(b) – состояние b-го бита соответственно регистра Rd, РСФ с адресом ADS и регистра статуса; Z, C, N, V, S, H, T, I – соответствующие биты регистра статуса (см. рис. 2.24); k – 8-битовая константа (если не оговорено иное); q – смещение, 0 ≤ q ≤ 63. [1)] У ряда моделей МК время выполнения команд может отличаться от указанного в таблице. См. *datasheet* конкретной модели МК. [2)] 16 ≤ d ≤ 31. [3)] Данные команды отсутствуют в системе команд ряда МК подсемейства *ATtiny*. [4)] Время выполнения данной команды зависит от конкретной модели МК. [5)] Данные команды удобны при сложении / вычитании / сравнении операндов с разрядностью, большей, чем разрядность РОН. [6)] При выполнении данных команд результат не записывается в какой-либо из модулей памяти, а только изменяются биты признаков в регистре статуса; данные команды обычно используются при реализации условных переходов. [7)] -64 ≤ k ≤ 63. Большинство версий ассемблера AVR-МК допускает использование только меток в качестве смещения k в ассемблерном коде команд с относительной адресацией. При этом необходимо, чтобы смещение адреса команды, снабженной меткой, относительно адреса команды перехода не выходило за пределы диапазона от минус 64 до плюс 63. [8)] Содержимое указателя стека уменьшается на 2 при вызове подпрограммы и увеличивается на 2 при возврате из подпрограммы или прерывания у моделей МК с 16-битовым программным счетчиком. У моделей с 22-битовым программным счетчиком содержимое указателя стека при выполнении указанных команд уменьшается / увеличивается на 3.			

Окончание таблицы 2.7

1	2	3	4
[9] Время выполнения команды равно 3-м тактам при разрядности программного счетчика 16 бит и 4-м тактам – при 22-битовом программном счетчике. [10] В данных командах допустимо также использование метки вместо абсолютного значения адреса. [11] Время выполнения команды равно 4-м тактам при разрядности программного счетчика 16 бит и 5-и тактам – при 22-битовом программном счетчике. [12] I – бит глобального разрешения прерываний в регистре статуса (см. рис. 2.24). Данный бит автоматически сбрасывается при переходе к подпрограмме обслуживания прерывания. Если данная подпрограмма завершается командой RETI – при выходе из нее глобальное разрешение прерываний восстанавливается. [13] При выполнении условия пропуска следующей команды содержимое программного счетчика увеличивается на 2, если разрядность указанной команды равна 16-и битам, и на 3 – если ее разрядность равна 32-м битам. [14] Время выполнения команды равно 1 такту при невыполнении условия пропуска следующей команды; 2-м тактам – при выполнении данного условия и разрядности следующей команды, равной 16-и битам; 3-м тактам – при выполнении условия пропуска следующей команды и ее разрядности, равной 32-м битам. [15] Время выполнения команды равно 1 такту при невыполнении условия перехода и 2-м тактам – при его выполнении.			

Таблица 2.8

Основные команды ассемблера МК семейства ARM Cortex-Mx [18]

Синтаксис	**Описание**	**Биты регистра статуса, изменяемые командой (см. рис. 2.31)**	T_{clk}
1	2	3	4
Команды пересылки данных			
MOV{cond} Rd, <op2>	Загрузка 32-битового регистра ЦП Rd ← <op2>	N, Z, C	1
MOVW{cond} Rd, #imm16	Загрузка 16-битовой константы в регистр ЦП Rd ← imm16		
MOVT{cond} Rd, #imm16	Загрузка константы в старшие 16 бит 32-битового регистра ЦП Rd[31:16] ← imm16		
MOV{cond} PC, Rm	Загрузка программного счетчика содержимым Rm[1)] PC ← Rm		1 + P
	Загрузка регистра ЦУ косвенно адресуемым содержимым памяти:		
LDR{cond} Rd, [Rn, <op2>]	Rd ← (Rn + <op2>)	-	2
LDR{cond} Rd, [Rn], #imm[2)]	Rd ← (Rn), Rn ← Rn + imm[2)]		
LDR{cond} PC, [Rn, <op2>]	Загрузка программного счетчика косвенно адресуемым содержимым памяти PC ← (Rn + <op2>)		2 + P
LDM{cond} Rn, {<reglist>}	Загрузка регистров ЦП, указанных в списке <reglist>, содержимым области памяти, базовый адрес которой равен содержимому Rn		1 + k

Продолжение таблицы 2.8

1	2	3	4
	Rd1 ← (Rn); … Rdk ← (Rn + 4*(k-1))		
	Загрузка содержимого регистра ЦП в память по косвенному адресу:		
STR{cond} Rd, [Rn, <op2>]	(Rn + <op2>) ← Rd	-	2
STR{cond} Rd, [Rn], #imm[2)]	(Rn) ← Rd Rn ← Rn + imm[2)]		
STM{cond} Rn, {<reglist>}	Загрузка регистров ЦПУ, указанных в списке <reglist>, в область памяти, базовый адрес которой равен содержимому Rn (Rn) ← Rs1; … (Rn + 4*(k-1)) ← Rsk		1 + k
PUSH{cond} {<reglist>}	Загрузка в стек содержимого регистров ЦП, указанных в списке <reglist> (SP - 4) ← Rsk; … (SP - 4k) ← Rs1; SP ← SP - 4k		
POP{cond} {<reglist>}	Выгрузка из стека в регистры ЦП, указанные в списке <reglist> Rd1 ← (SP); … Rdk ← (SP + 4*(k-1)); SP ← SP + 4k		
	Чтение / запись *Special Registers* ЦП (см. рис. 2.30)[4)]		
MRS{cond} Rd, spec_reg	Rd ← spec_reg		
MSR{cond} spec_reg, Rn	spec_reg ← Rn		

Продолжение таблицы 2.8

1	2	3	4
Арифметические операции			
ADD{cond} Rd, Rn, <op2>	Сложение Rd ← Rn + <op2>	N, Z, C, V	1
ADD{cond} PC, PC, Rm	Модификация содержимого программного счетчика на величину смещения, равную содержимому регистра ЦП Rm PC ← PC + Rm	N, Z, C, V	1 + P
ADC{cond} Rd, Rn, <op2>	Сложение с переносом Rd ← Rn + <op2> + C		1
SUB{cond} Rd, Rn, <op2>	Вычитание Rd ← Rn - <op2>		
SBC{cond} Rd, Rn, <op2>	Вычитание с заемом переноса Rd ← Rn - <op2> - $\overline{C}$		
	Сравнение (без изменения операндов):		
CMP{cond} Rn, <op2>	Rn - <op2>		
CMN{cond} Rn, <op2>	Rn + <op2>		
MUL{cond} Rd, Rn, Rm	Умножение Rd ← Rn*Rm	N, Z	
MLA{cond} Rd, Rn, Rm	Умножение с накоплением Rd ← Rd*Rn + Rm	-	2
MLS{cond} Rd, Rn, Rm	Умножение с антинакоплением Rd ← Rm - Rd*Rn		
SDIV{cond} Rd, Rn, Rm	Деление чисел со знаком Rd ← Rn / Rm		2… 12[3)]
UDIV{cond} Rd, Rn, Rm	Деление чисел без знака Rd ← Rn / Rm		
SSAT Rd, #n, Rm	Знаковое насыщение: Rd ← Rm при $-2^{n-1} \le Rm \le 2^{n-1} - 1$; Rd ← -2^{n-1} при Rm < -2^{n-1}; Rd ← $2^{n-1} - 1$ при Rm > $2^{n-1} - 1$	Q	1

Продолжение таблицы 2.8

1	2	3	4
USAT Rd, #n, Rm	Беззнаковое насыщение: Rd ← Rm при $0 \le Rm \le 2^n - 1$; Rd ← 0 при Rm < 0; Rd ← $2^n - 1$ при Rm > 2^n - 1	Q	1
Логические операции			
AND{cond} Rd, Rn, <op2>	Логическое И Rd ← Rn ∧ <op2>	N, Z, C	1
ORR{cond} Rd, Rn, <op2>	Логическое ИЛИ Rd ← Rn ∨ <op2>		
ORN{cond} Rd, Rn, <op2>	Логическое ИЛИ-НЕ $Rd \leftarrow \overline{Rn \vee <op2>}$		
EOR{cond} Rd, Rn, <op2>	Исключающее ИЛИ Rd ← Rn ⊕ <op2>		
BIC{cond} Rd, Rn, <op2>	Выборочное обнуление битов регистра ЦПУ $Rd \leftarrow Rn \wedge \overline{<op2>}$	N, Z, C	1
MVN{cond} Rd, <op2>	Инверсия операнда с записью в регистр ЦПУ $Rd \leftarrow \overline{<op2>}$		
	Проверка содержимого регистра ЦПУ (без изменения операндов):		
TST{cond} Rn, <op2>	Rn ∧ <op2>		
TEQ{cond} Rn, <op2>	Rn ⊕ <op2>		
Операции над битовыми полями			
BFC{cond} Rd, #imm1, #imm2	Сброс в лог. 0 битов регистра Rd с номерами от imm1 до imm1 + imm2 - 1	-	1
BFI{cond} Rd, Rn, #imm1, #imm2	Замена битов регистра Rd с номерами от imm1 до imm1 + imm2 – 1 битами регистра Rn с теми же номерами		

Продолжение таблицы 2.8

1	2	3	4
Операции сдвига содержимого регистра ЦП Rn на число битов, равное <op2>, с записью результата сдвига в регистр ЦП Rd			
ASR{cond} Rd, Rn, <op2>	Арифметический сдвиг вправо (см. рис. 2.39)	N, Z, C	1
LSR{cond} Rd, Rn, <op2>	Логический сдвиг влево (см. рис. 2.37)		
LSL{cond} Rd, Rn, <op2>	Логический сдвиг вправо (см. рис. 2.38)		
ROR{cond} Rd, Rn, <op2>	Циклический сдвиг вправо (см. рис. 2.41)		
RRX{cond} Rd, Rn	Циклический сдвиг вправо на 1 бит через бит C (см. рис. 2.43)		
Команды передачи управления			
B{cond} label	Переход к команде с меткой label при выполнении условия cond, в противном случае – переход к следующей команде	-	1 + P
BL{cond} label[5)]	При выполнении условия cond – переход к команде с меткой label, с сохранением адреса команды, следующей за BL, в регистре R14 ЦП. В противном случае – переход к следующей команде		
BX{cond} Rm	При выполнении условия cond - переход к команде с адресом, равным содержимому Rm. В противном случае – переход к следующей команде		

Продолжение таблицы 2.8

1	2	3	4
BLX{cond} Rm[5)]	При выполнении условия cond – переход к команде с адресом, равным содержимому Rm, с сохранением адреса команды, следующей за BL, в регистре R14 ЦП. В противном случае – переход к следующей команде	-	1 + P
CBZ Rn, label	Переход к команде с меткой label при нулевом или, соответственно, ненулевом содержимом Rn. В противном случае – переход к следующей команде		
CBNZ Rn, label			
IT{x{y{z}}} cond	Блок команд «if-then». IT{x{y{z}}} cond – заголовок блока. Блок может содержать от 1-й до 4-х команд. Условием выполнения 1-ой является cond; 2-й, 3-й и 4-й (при их наличии) – x, y и z соответственно. Подробности и примеры – см. [18]		1
Команды перевода МК в энергосберегающий режим			
WFE	См. табл. 3.2 и пояснения к ней	-	1 + W
WFI			

Примечания.
В данной таблице указаны команды, входящие в состав ассемблера большинства МК семейства *ARM Cortex-Mx*. Конкретный состав команд каждого подсемейства МК, их синтаксис и описания представлен в руководстве по программированию (*Programming manual*) соответствующего подсемейства МК.

Продолжение таблицы 2.8

1	2	3	4

T_{clk} – время выполнения команды в тактах ЦП. У ряда моделей МК время выполнения команд может отличаться от указанного в таблице. См. *datasheet* конкретной модели МК.

{cond} – опциональный суффикс условия выполнения команды, в соответствии с **таблицей 2.9**; например, синтаксис MOVEQ R0, R1 означает, что команда MOV будет выполнена только при установленном (т. е. равном лог. 1) флаге Z регистра статуса (*x*PSR) ЦП (см. рис. 2.31). При отсутствии суффикса {cond} команда выполняется безусловно.

Rd, Rm, Rn – содержимое регистров R0 – R15 ЦПУ (см. рис. 2.30); в ряде команд – кроме программного счетчика (подробности см. в [18]).

PC – содержимое программного счетчика.

SP – содержимое указателя стека.

spec_reg – регистр ЦП из группы *Special Registers* (см. рис. 2.30).

<op2> - второй операнд команды, в качестве которого может служить содержимое регистра ЦПУ (с опциональным предварительным сдвигом) или непосредственно указываемый операнд; например, возможные варианты синтаксиса команды MOV:

MOV{cond} Rd, Rm;

MOV{cond} Rd, Rm, LSR #4 (запись в регистр Rd содержимого регистра Rm, предварительно сдвинутого логически на 4 бита влево);

MOV{cond} Rd, #imm32.

imm32, imm16 – непосредственно указываемый 32-битовый или, соответственно, 16-битовый операнд, например, 0x35FF0012 (число 35FF0012 в шестнадцатеричной системе счисления).

Rd[31:16] – биты с 31-го по 16-й регистра Rd.

(Rn), (Rn + <op2>) – содержимое ЗЭ с адресом, равным Rn и Rn + <op2> соответственно.

{<reglist>} – список регистров; пример синтаксиса – LDMNE R0, {R1-R5,R9,R12}.

Rdi, Rsi – регистры-приемники, или, соответственно, регистры-источники из списка {<reglist>}.

k – количество регистров в списке.

P – число циклов ЦП, требуемое для перезагрузки конвейера (см. выше) и равное, в зависимости от конкретного состояния конвейера, от 1-го до 3-х.

W – число циклов ЦП, необходимое для перехода в энергосберегающий режим

[1)] Специальное назначение имеет команда загрузки программного счетчика содержимым регистра связи, R14 (MOV{cond} PC, R14), служащая командой безусловного или условного возврата из подпрограммы (см. также сноску [5)]).

[2)] Число imm может быть как положительным, так и отрицательным.

Окончание таблицы 2.8

1	2	3	4
[3)] В зависимости от числа ненулевых младших значащих битов в результате деления. [4)] Данные команды доступны для выполнения только на привилегированном уровне выполнения программ. [5)] Команды BL{cond} и BLX{cond} служат, по существу, командами перехода к подпрограмме (условного или безусловного), с сохранением адреса возврата в регистре связи. При этом подпрограмма должна завершаться командой MOV{cond} PC, R14 (см. сноску [1)]).			

Таблица 2.9

Суффиксы условий (cond) выполнения команд (см. табл. 2.8) [18]

Суффикс	**Состояние флагов**	**Семантика**
EQ	Z установлен	Равно нулю
NE	Z сброшен	Не равно нулю
CS	C установлен	Больше или равно (для чисел без знака)
CC	C сброшен	Меньше (для чисел без знака)
HI	C установлен и Z сброшен	Больше (для чисел без знака)
LS	C сброшен или Z установлен	Меньше или равно (для чисел без знака)
MI	N установлен	Отрицательный результат
PL	N сброшен	Положительный результат либо ноль
VS	V установлен	Переполнение
VC	V сброшен	Нет переполнения
GE	N равно V	Больше или равно (для чисел со знаком)
LT	N не равно V	Меньше (для чисел со знаком)
GT	Z сброшен и N равно V	Больше (для чисел со знаком)
LE	Z установлен или N не равно V	Меньше или равно (для чисел со знаком)
AL или суффикс отсутствует	Состояние флагов игнорируется	Безусловно

2.7 Выводы по разделу 2

2.7.1. Описанные в разделе 2 примеры структурно-архитектурных решений ЦП и памяти МК являются типовыми.

2.7.2. Практически для всех распространенных семейств МК общего назначения характерно построение ЦП по типовым структурным схемам, варианты которых (в зависимости от класса МК) представлены на рис. 2.3 – 2.5; нетрудно при этом заметить, что данные варианты мало различаются по общему принципу реализации и, в целом, соответствуют «классической» структуре ЦП (см. рис. 2.1);

2.7.3. Основными блоками ЦП МК являются УУ, АЛУ и СОЗУ. Их назначение и общие принципы их работы, а также функционирования ЦП в целом описаны в подразделе 2.2. Принципы технической реализации УУ и АЛУ изложены в Приложениях А и Б.

2.7.4. ЦП большинства МК общего назначения классов «*cost-sensitive*» и «*mainstream*» могут работать в следующих режимах: основном или в одном из энергосберегающих, различающихся между собой составом «спящих» узлов и блоков МК, а также способами вывода МК из энергосберегающего режима (см. далее подраздел 3). Для ряда подсемейств МК класса «*high performance*» характерно наличие двух или нескольких не энергосберегающих режимов работы (например, режим обработки исключительных ситуаций и потоковый режим в МК семейства *ARM Cortex-Mx*, см. подраздел 2.3 и п. 2.5.8).

2.7.5. Данные в МК относятся к одной из следующих категорий:

- двоичные (значительно реже – двоично-десятичные) операнды и результаты выполнения операций, которые могут быть беззнаковыми целыми числами, целыми числами со знаком, в дополнительном коде (см. рис. 2.6 и табл. 2.1) или вещественными числами с плавающей точкой (см. рис. 2.7 и 2.9);
- двоичные коды управления или состояния периферийных устройств.

В памяти МК данные, как правило, хранятся побайтно, обычно в формате «От младшего к старшему» (*Little Endian*), реже – в формате «От старшего к младшему» (*Big Endian*), см. рис. 2.11 и 2.12 и пояснения к ним.

2.7.6. Организация памяти современных МК общего назначения отличается следующими основными особенностями.

Память большинства семейств МК строится по Гарвардской архитектуре, характеризуемой физическим разделением ПП и ПД, что позволяет совместить во времени выполнение текущей команды (как правило, требующее обращения к ПД) и извлечение из ПП следующей команды (см. п. 2.6.4).

Типовой состав модулей памяти современного МК общего назначения следующий:

- энергонезависимая память программ, в которой, как правило, выделяются определенные области под векторы прерываний, под загрузочные сектора, а также под некоторые другие специальные цели;
- СОЗУ, включающее в себя РОН и РСФ, количество и состав которых зависит от конкретного семейства и / или модели МК;
- статическое ОЗУ данных (отсутствует у ряда моделей класса *cost-sensitive,* его функции при этом выполняют РОН);
- энергонезависимая память данных (у многих семейств / моделей МК отсутствует как отдельный блок, в ее качестве используется часть памяти программ, не задействованная под прикладное ПО);
- набор энергонезависимых ЗЭ системного назначения, предназначенных для хранения слов конфигурации МК, кодов идентификации и т. п.

Типовые примеры организации памяти различных классов МК общего назначения представлены в подразделе 2.5.

2.7.7. Под **системой команд МК** понимается система двоичных кодов машинного языка, воспринимаемых ЦП без промежуточной трансляции. Программирование на машинном языке потенциально позволяет создать программный код с минимальным объемом и временем выполнения, однако практикуется редко из-за большой трудоемкости. Разработка программ или их фрагментов на уровне, равносильном по объему кода и быстродействию программам на машинном языке, осуществляется на **языке ассемблера** МК. Его команды представляют собой запись кодов машинного языка на естественном (английском) языке и транслируются в команды машинного языка посредством специального ПО.

Именно система команд ассемблера выступает в качестве системы команд МК, приводимой в его руководстве пользователя (*Reference Manual*) или руководстве по программированию (*Programming Manual*).

2.7.8. С точки зрения системы команд для большинства семейств МК общего назначения характерна RISC-архитектура (*Reduced Instruction Set Computer*, «Компьютер с сокращенным набором команд»). Их система команд включает в себя только набор наиболее часто используемых простых инструкций. Более сложные процедуры реализуются в виде подпрограмм, написанных с использованием указанного простого набора команд. Сочетание Гарвардской и RISC-архитектуры, а также конвейеризации (см. п. 2.6.4) обеспечивает выполнение большинства ассемблерных команд за 1 такт ЦП.

2.7.9. Язык ассемблера практически каждого из семейств МК общего назначения включает в себя следующие группы команд:

- команды пересылки данных;
- арифметические команды;
- побитовые логические операции;
- команды сдвигов;
- команды передачи управления;
- команды специального назначения (например, перехода в режим пониженного энергопотребления).

Код каждой команды содержит (в явном или в неявном виде): код подлежащей выполнению операции, адреса или указатели адресов операндов и результата, адрес или указатель адреса следующей команды.

Типовые системы команд МК различных классов представлены в табл. 2.6 – 2.8.

2.7.10. В настоящее время разработка ПО МК осуществляется в интегрированных средах проектирования (*IDE*). Рациональным подходом к разработке программ для МК представляется следующий:

- основная часть каждого из программных модулей разрабатывается на языке высокого уровня, обычно – на версиях языка *C*, адаптированных под задачи программирования МК (что

снижает трудоемкость и сроки разработки проектов, а также обеспечивает удобство понимания программных кодов и упрощает их переносимость с одной модели МК на другую);

- программные фрагменты, критичные с точки зрения времени выполнения или объема, разрабатываются на языке ассемблера соответствующего МК и оформляются как ассемблерные вставки в основной код.