Оглавление

1 Базовая информация

1.1 Системы счисления

16	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
10	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111

У систем счисления (СС) с основаниями степеней двойки (то есть 4-ичная, 8-ичная итд) есть особенность, позволяющая быстро переводить их в 2-ичную СС. Чтобы произвести такой перевод, достаточно перевести отдельно каждую цифру в 2-ичную СС, выделив для каждой из них количество бит, равное показателю степени двойки основания СС.

Пример 1. Перевод из 8-ичной в 2-ичную СС. (метод триад)

Основание изначальной СС - 8, что равно 2^3. Так как показатель степени равен 3, то для каждой цифры 8-ичной СС будет выделяться 3 бита. Переведём 1267_8 в 2-ичную СС. Для этого по очереди переведём каждую цифру: $1_8=001_2$, $2_8=010_2$, $6_8=110_2$, $7_8=111_2$. Запишем получившиеся числа слитно в том же порядке: 001.010.110.111. Избавимся от незначащих нулей и перевод окончен: $1267_8=1010110111_2$. Такой перевод из 8-ичной в 2-ичную СС называется методом триад, где триадой называют каждую тройку двоичных цифр.

Пример 2. Перевод из 16-ичной в 2-ичную СС. (метод тетрад)

Аналогично предыдущему примеру вычисляем показатель степени - 4 ($16=2^4$). Раскладываем каждую цифру по четырём битам (тетрадам). $2F3D_h=0010.1111.0011.1101_2$

Существует двоично-десятичный код. Это 16-ичное число, в котором нет цифр больше 9. Алгоритм перевода из двоично-десятичного кода в двоичный описан в пункте 7.5.

1.2 Отрицательные числа и операция вычитания

Для различия положительных и отрицательных чисел в математике используют знак «-». В двоичном коде этот знак зашифрован в старшем (левом) бите числа. Отрицание числа реализовано его переводом в дополнительный код (ДК). ДК - это обратный код, к которому нужно добавить 1.

1 добавляется, чтобы избавиться от двух представлений числа 0. Второе представление нуля возникает при преобразовании его в обратный код, пытаясь сделать из него отрицательный 0. Пусть у нас есть 8-разрядное число. Положительный 0 будет записываться как 00000000, а отрицательный как 11111111 (если бы отрицательное число делалось обратным кодом). Но так как -0=+0, то было принято решение добавлять 1. Таким образом мы получаем 1.00000000. Единица вышла за пределы разрядности числа, значит её можно не учитывать.

Операция вычитания реализована как операция сложения с отрицательным числом. A-B=A+(-B).

Пример 1. Найти разность 12-9 для 8-разрядных чисел.

12 = 00001100, -9 = -00001001 (прямой код) = 11110110 (обратный код) + 1 = 11110111 (дополнительный код)

```
  00001100
+ 11110111
-----------
1.00000011 (старший бит 0 ⇒ разность положительная) = 3
```

Пример 2. Найти разность 9-12 для 8-разрядных чисел.

9 = 00001001, -12 = -00000100 = 11110011+1 = 11110100

```
  00001001
+ 11110100
----------
 11111101
```

11111101 (разность отрицательная ⇒ нужно перевести из ДК) = 11111100+1 = -00000011 = -3

1.3 Дробные числа, фиксированная точка, плавающая точка

Разряды дробной части двоичного числа нумеруются по убыванию, начиная со старшего, номер которого -1. Номер разряда является показателем степени двойки.

Пример. Перевод $100,1101_2$ в 10-ичную СС.

$100,1101_2 = 1*2^2 + 0*2^1 + 0*2^0 + 1*2^{-1} + 1*2^{-2} + 0*2^{-3} + 1*2^{-4} = 4 + 1/2 + 1/4 + 1/16 = 4,8125$

Фиксированная точка (ФТ)

В форматах чисел с ФТ всегда отведено определённое постоянное количество разрядов для целой и для дробной части. Такой формат позволяет оптимизировать место, занимаемое числом, и скорость вычислений с ним, что хорошо для специализированных ЭВМ, необходимая точность вычисления которых постоянна и заранее известна.

Плавающая точка (ПТ)

Является более универсальным форматом чисел, так как позволяет, в зависимости от решаемой на данной момент задачи, уделять большее количество места для хранения очень больших или очень маленьких чисел. Числа с ПТ состоят из знака, порядка и мантиссы. Мантисса хранит в себе двоичное число (без старшего бита, равного единице), точка раздела которого на дробную и целую часть не выделена. За указание разряда, с которого начинается целая часть мантиссы, отвечает порядок. Отрицательные числа в ПТ хранятся в прямом коде, от положительных они отличаются только значением бита знака. Наиболее популярны 2 вида форматов с ПТ: float (single), double (двойная точность). Float занимает 32 бита: 1 - знак, 8 - порядок, 23 - мантисса. Double занимает 64 бита: 1 - знак, 11 - порядок, 52 - мантисса. Для 16-разрядных процессоров удобно использовать формат половинной точности, занимающий 16 бит: 1 - знак, 5 - порядок, 10 - мантисса.

Пример. Представить 5,125 в float.

Представим число как сумму степеней двойки.

$5,125 = 4+1+0,125 = 2^2+2^0+2^{-3} = 101,001_2$ (мантисса M=01001, старшая единица в мантиссах не записывается)

Для представления числа в ПТ его нужно поделить/умножить на такую степень двойки, которая оставит в целой части числа только одну двоичную цифру.

$101,001_2 = 10,1001_2*2^1 = 1,01001_2*2^{\mathbf{2}}$ (смещение 2)

К показателю степени двойки добавляется 127 (2^{n-1}-1, где n - количество разрядов, выделенное на порядок, которое в float равно 8 бит).

2+127 = 128+1 = 10000001 (порядок E=10000001)

Знак числа положительный (знак S=0)

5,125 = 0.10000001.01001000000000000000000

В общем виде представление числа в float выглядит так: $(-1)^S * 1,M * 2^{(E - 127)}$

В общем виде представление числа в double выглядит так: $(-1)^S * 1,M * 2^{(E - 1023)}$

В общем виде представление числа в любом виде ПТ: $(-1)^S * 1,M * 2^{(E - c)}$, где $c = 2^{n-1}-1$

Операции над числами в ПТ описаны в пункте 7.3.

1.4 Виды сдвигов, применение сдвигов

Операции сдвигов сдвигают двоичные числа на один разряд. Например, есть число 11001011, которое после логического сдвига вправо будет равно 01100101. Каждая цифра переместилась на более младший бит, младшая единица вылетела из числа, а на пустое место (старший бит) записался 0. Более наглядно этот сдвиг можно показать так:

```
  1  1  0  0  1  0  1  1 →
   ↘  ↘  ↘  ↘  ↘  ↘  ↘  ↘
→ 0  1  1  0  0  1  0  1
```

Есть несколько видов сдвигов: логический (влево, вправо), арифметический (вправо), циклический (влево, вправо), циклический через перенос (влево, вправо).

Логический сдвиг - сдвиг всего регистра на 1 бит влево или вправо. При сдвиге вправо младший бит исходного числа пропадает, а в старший бит результата записывается 0. При сдвиге влево старший бит исходного числа пропадает, а в младший бит результата записывается 0. Пример логического сдвига вправо показан выше. Пример логического сдвига влево: 11001011 ⇒ 10010110.

Арифметический сдвиг - сдвиг регистра с сохранением знака (старшего бита). Арифметический сдвиг бывает только вправо (о причинах этого написано в конце пункта). Работает как логический сдвиг вправо, только старший бит остаётся неизменным. Но при этом старший бит исходного числа дублируется в результат на 1 бит младше. Примеры арифметического сдвига вправо: **1**0011011 ⇒ **1**1001101, **0**1010100 ⇒ **0**0101010.

Циклический сдвиг - сдвиг, при котором цифра, выходящая из регистра, записывается в него с противоположной стороны. При логическом сдвиге один из битов пропадает с одной стороны, а с другой записывается 0. При циклическом вместо 0 записывается значение вышедшего бита. Примеры циклического сдвига вправо: 1100101**1** ⇒ **1**1100101, 1010101**0** ⇒ **0**1010101. Пример циклического сдвига влево: **1**1001011 ⇒ 1001011**1**.

Циклический сдвиг через перенос - циклический сдвиг с дополнительным битом за пределом регистра. Когда из регистра выходит значение, оно сохраняется во флажке переполнения C (carry). При циклическом сдвиге через перенос в вышедший из регистра бит записывается в C, а с другой стороны в регистр входит бит, который был в C до сдвига. Примеры циклического сдвига вправо: C=0 1100101**1** ⇒ C=**1** 01100101 ⇒ C=1 **1**0110010, C=0 1010101**0** ⇒ C=**0** 01010101. Пример циклического сдвига влево: C=0 **1**1001011 ⇒ C=**1** 10010110 ⇒ C=1 00101101. В Micro флажок C не связан со сдвигателями, поэтому такой сдвиг не реализован.

У сдвигов есть много применений. Одно из них - умножение или целочисленное (без учёта остатка) деление на 2. Рассмотрим аналог логических сдвигов чисел в 10-ичной системе счисления (СС). Рассмотрим число 789. При десятичном сдвиге влево мы получим число 7890, то есть в 10 раз большее. Если 789 сдвинуть вправо, то мы получим 78, то есть результат целочисленного деления на 10. Так как в 10-ичной СС 10 цифр, то приписывание нуля справа равносильно повтору числа 10 раз, то есть умножению на 10. В двоичной СС две цифры, а значит операции сдвигов умножают или целочисленно делят его на 2. Например, число 5 в

двоичной СС равно 101. При сдвиге влево получается число 1010, что равно 10 в десятичной СС. Рассмотрим умножение чисел на основание СС в общем виде. Пусть дано число abc с основанием СС s. При раскладывании его на сумму произведений получим:

$abc = a*s^2 + b*s^1 + c*s^0$

Умножим это число на s и обратно запишем результат:

$abc * s = a*s^2*s^1 + b*s^1*s^1 + c*s^0*s^1 = a*s^3 + b*s^2 + c*s^1 + 0*s^0 = abc0$

Очевидно, что abc0 является логическим сдвигом влево числа abc в основании СС s. Таким образом, операции с умножением, либо целочисленным делением на степень двойки можно заменить на соответствующие сдвиги, если итоговое число не выйдет за пределы максимального или минимального числа в регистре. С помощью сдвигов и сложений можно достаточно оптимально реализовать умножение и на другие числа. Например, умножение на 10 можно представить следующей формулой: (x*4+x)*2, которую можно заменить её аналогом с логическими сдвигами: (x>>2+x)>>1.

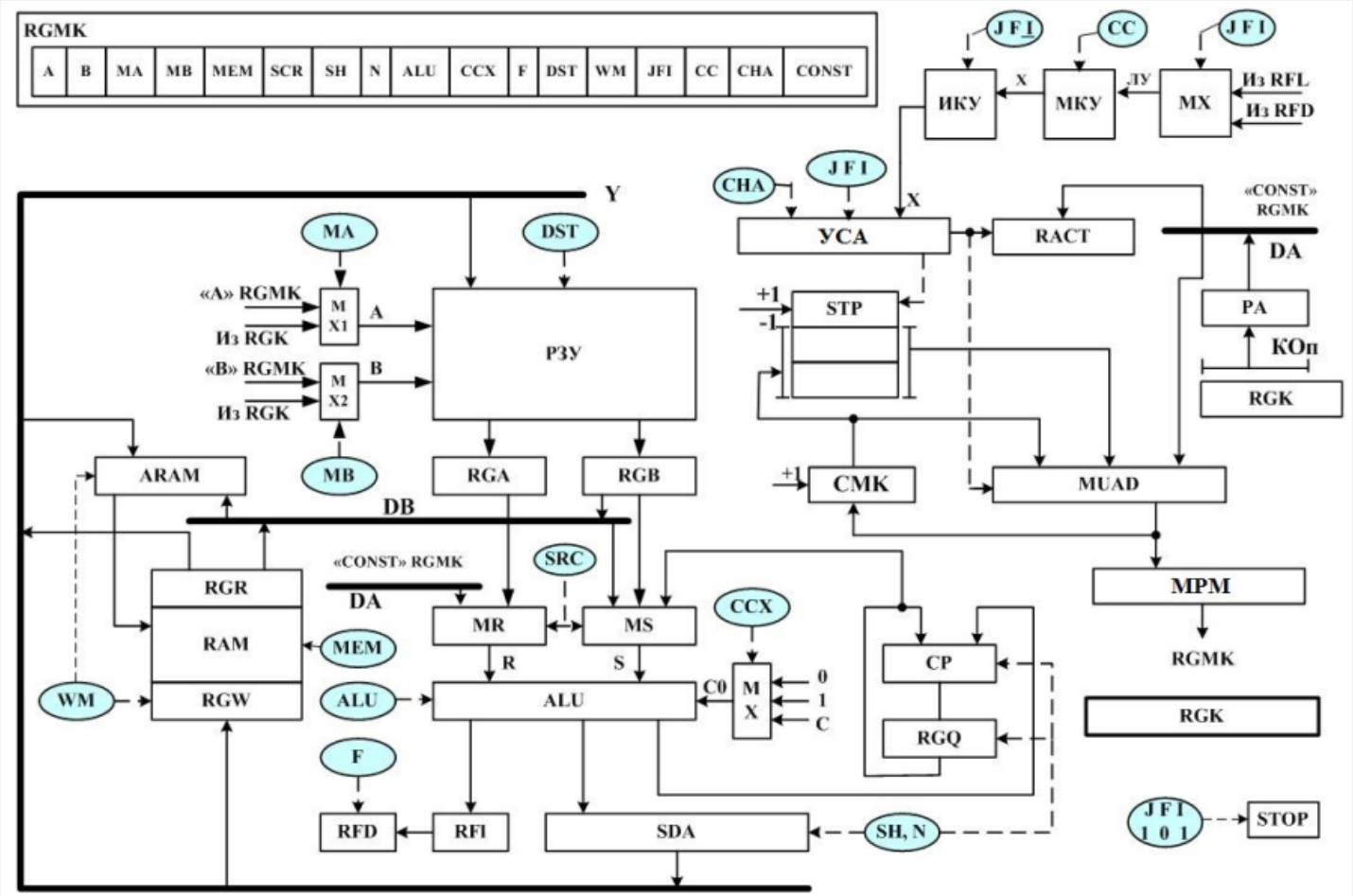
Логические сдвиги позволяют выполнять эти операции, когда в регистре хранится только неотрицательное число, то есть бит знака (старший бит) учитывать не нужно. Для выполнения операции деления на степень двойки над целыми числами со знаком существует арифметический сдвиг. Пусть есть число -128, которое в двоичной СС в восьмибитном регистре будет выглядеть как **1**0000000 (подробнее о представлении отрицательных чисел в двоичном коде в пункте 1.2). При делении его на 2 должно получиться число -64. Так как старший бит отвечает за знак, то он должен сохранится, а значит применяется арифметический сдвиг вправо **1**1000000. Для умножения на 2 в любых целых числах используется логический сдвиг влево. Например, чтобы получить из -64 число -128, его нужно логически сдвинуть влево **1**1000000⇒**1**0000000. Знак сохранился, операция выполнена корректно. Но тут может возникнуть вопрос, как производить умножение на 2 числа **1**0100000, ведь при логическом сдвиге влево знак изменится, что невозможно при умножении на 2. Для разрешения этого вопроса нужно определить диапазон отрицательных чисел, старший бит перед знаком которых равен 0. Получается, мы рассматриваем числа вида 10xxxxxx, где x - это любое значение бита. Так как число отрицательное, то осуществим его перевод в прямой код (то есть найдём модуль числа). После вычитания 1 (перевода из дополнительного кода в обратный) есть 2 сценария изменения вида числа: 01111111 (все x были равны 0, что привело к изменению старших бит), 10xxxxxx (старшие биты не изменились, так как изначально хотя бы один x был равен 1, а значит сейчас хотя бы один x равен 0, так как все единицы могут получиться только в случае, описанном выше). После инверсии (перевода в прямой код) в двух сценариях получаются следующие числа: 10000000, 01xxxxxx. Так как модуль числа всегда неотрицательный, то старший бит в полученных числах не нужно выделять под знак. В первом случае мы получили число 10000000=128, во втором $01xxxxxx = 0*2^7 + 1*2^6 + x*2^5 + x*2^4 + x*2^3 + x*2^2 + x*2^1 + x*2^0 = 64 + x*2^5 + x*2^4 + x*2^3 + x*2^2 + x*2^1 + x*2^0$. Так как хотя бы один x в обратном коде был равен 0, то после инверсии эти x равны 1, а значит в любом случае выражение $x*2^5 + x*2^4 + x*2^3 + x*2^2 + x*2^1 + x*2^0$ не равно 0, то есть всё число в любом случае больше 64. Восьмибитное целое число с учётом знака имеет ограничение на значения -128..+127. При попытке умножить отрицательное число, большее по модулю, чем 64, получается число, большее по модулю, чем 128, что невозможно при учёте ограничения на значение, которое можно уместить в восьми

двоичных разрядах. Таким образом, операция логического сдвига на число со знаком вида 10xxxxxx не имеет смысла, так как результат не уместится в восьмибитном регистре. Ещё нагляднее бессмысленность арифметического сдвига влево можно продемонстрировать на положительных числах. Рассмотрим число, которое при логическом сдвиге влево меняет знак, из-за чего может показаться необходимым применение арифметического сдвига влево. 01xxxxxx. А теперь вычислим его возможные значения. Минимальным это число будет, если все x равны 0. Число 01000000 равно 64 в десятичной СС. При умножении на 2 получится число 128, которое выходит за пределы допустимых значений 8-битного числа. При других значениях x число будет больше 64, а значит при умножении на 2 оно тем более будет выходить за пределы допустимых значений. Если бы арифметический сдвиг влево существовал, то он выдавал бы другие значения при операциях, приводящих к переполнению допустимых значений, но они не имели бы смысла так же как не имеют смысла результаты переполнения при выполнении логического сдвига влево. Выше было доказано, что при всех возможных случаях, не приводящих к переполнению, результаты арифметического и логического сдвигов влево идентичны, поэтому для оптимизации арифметический сдвиг влево не был реализован.

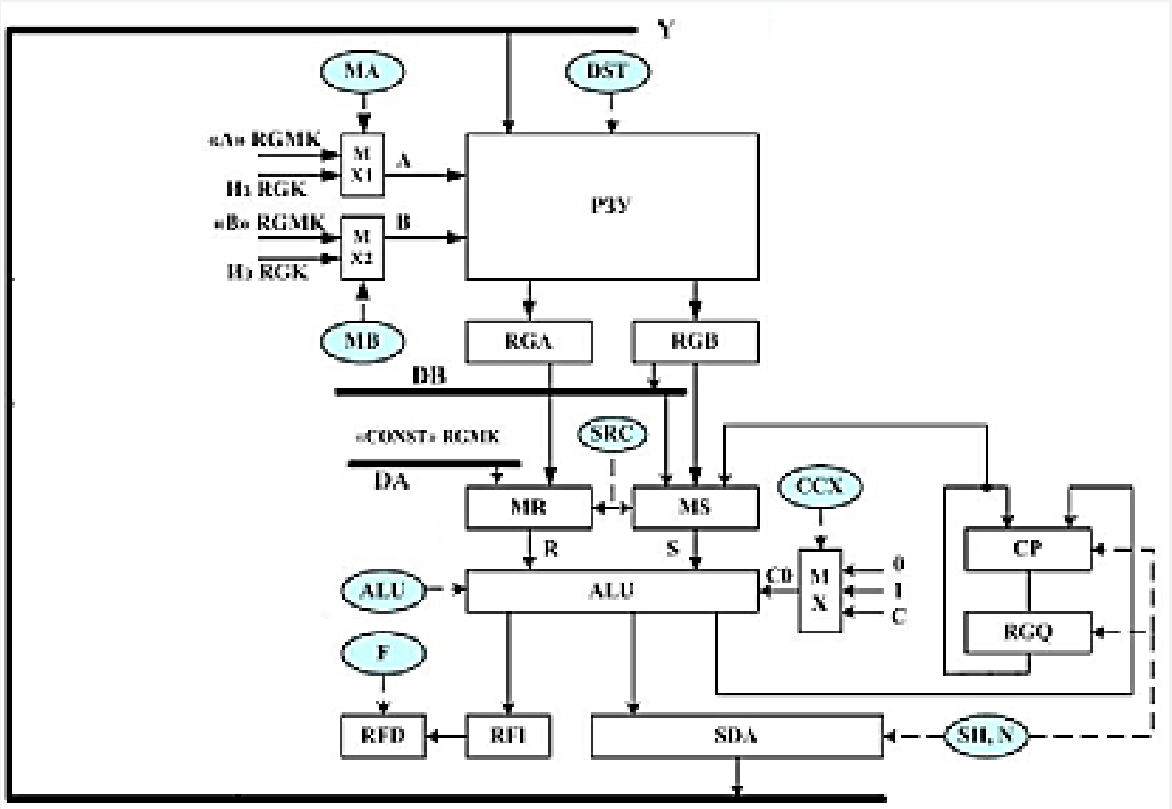
2 Общие сведения о процессоре

Процессор 16-разрядный, то есть работает с регистрами, хранящими 16-разрядные (16-битные) двоичные числа. Для удобства в интерфейсе Micro эти двоичные числа представлены в 16-ичной системе счисления (но это не значит, что процессор работает с 16-ичной системой счисления, все процессоры работают только с двоичным кодом). 16-разрядность процессора никак не связана с 16-ичной системой счисления, а лишь указывает количество разрядов в большинстве регистров. Все регистры в процессоре 16-разрядные, кроме регистра микрокоманд (RGMK), он 64-разрядный.

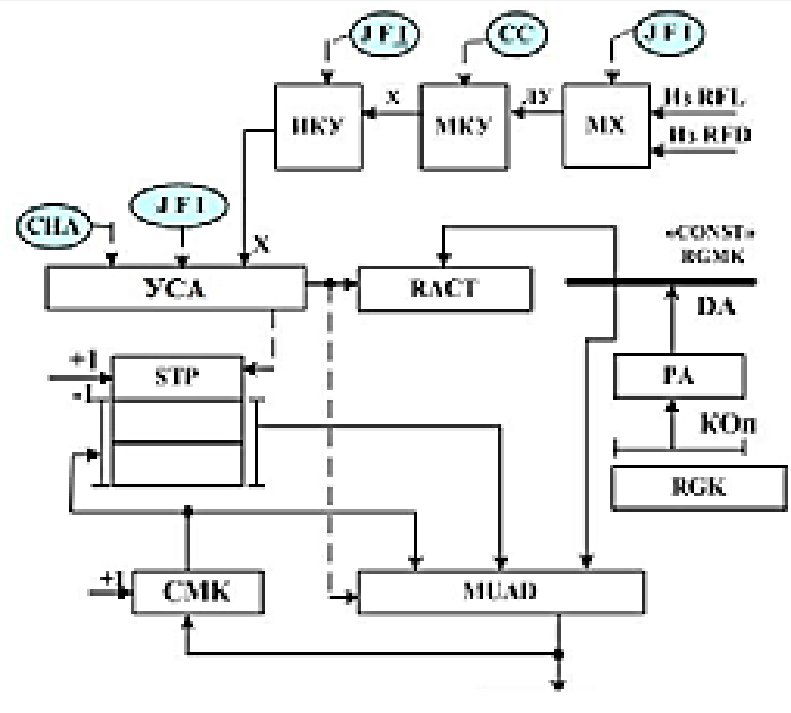
Общая схема процессора:



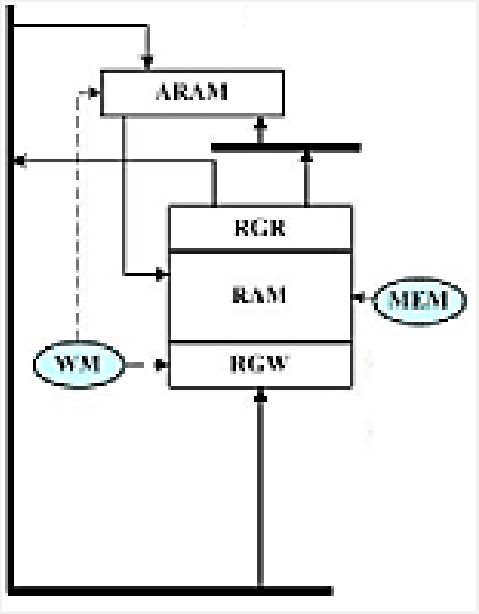
Состав процессора:
1 Операционный блок (ядро процессора)



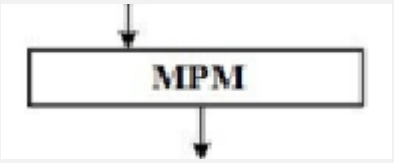
2 Блок микропрограммного управления



3 Оперативная память (RAM)



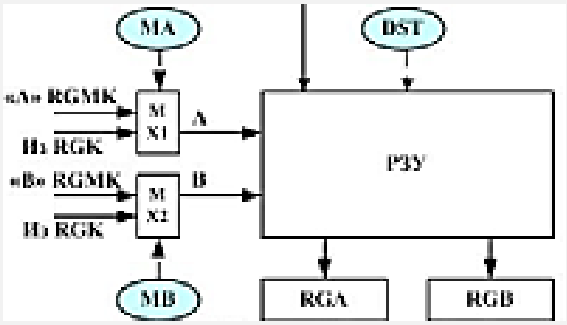
4 Микропрограммная память (MPM)



Процессор работает по инструкциям, прописанным в микропрограммной памяти. Микропрограммная память хранит в себе микрокоманды, которые делятся на поля, указывающие, какие микрооперации должен выполнить процессор за 1 такт. Микрокоманды выполняются по порядку, если в них не прописан переход на другие микрокоманды.

3 Работа с ядром процессора

3.1 РЗУ - регистровое запоминающее устройство

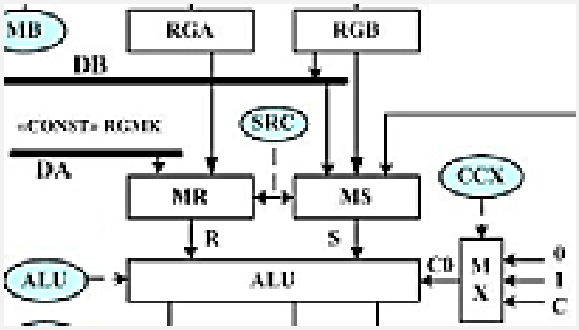


РЗУ хранит в себе 16 регистров общего назначения (РОН). Эти регистры могут иметь разный смысл в зависимости от их использования в процессе выполнения микропрограммы. Есть рекомендуемое назначение каждого из РОНов, описанное в пункте 6.2. РЗУ - самая быстрая память процессора, так как она находится в его ядре и уже готова к выводу значений регистров для выполнения операций над ними. За 1 такт из РЗУ выходят 2 значения, записываемые в регистры RGA и RGB, которые могут использоваться для операций в этом такте. Чтобы указать, значения каких именно РОНов пойдут в RGA и RGB, в полях A и B соответственно нужно указать номера этих РОНов.

Поля A/B	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
РОН	AX	CX	DX	BX	SP	BP	SI	DI	CS	SS	DS	ES	IP	PSW	RGK	RW

В полях MA и MB можно выбрать способ выбора адреса РОНа: из микрокоманды, то есть из полей A и B (по умолчанию MA и MB равны 0, что этому способу и соответствует), либо из регистра команд RGK (подробнее в разделе 6).

3.2 АЛУ - арифметико-логическое устройство



Далее после RGA и RGB идут мультиплексоры (устройства, которые из нескольких входов выбирают только 1 и ведут без изменений его значение к единственному выходу) MR и MS, выбирающие операнды, которые пойдут в АЛУ (арифметико-логическое устройство). Управление выбором этих мультиплексоров осуществляется полем микрокоманды SRC. При выборе операнда RGR берётся то значение, которое было там на начало такта.

Поле SRC	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Операнд R	0000	RGA	RGA	RGA	RGA*2	CONST	CONST	CONST
Операнд S	0000	RGB	RGQ	RGR	RGB	RGB	RGR	RGQ

По умолчанию SRC=1, что отвечает за выбор RGA и RGB как операндов для АЛУ. Ещё в качестве операндов могут выступать: нули; регистр RGR чтения оперативной памяти; константа CONST, указанная в микрокоманде; рабочий регистр RGQ для одного из сдвигателей.

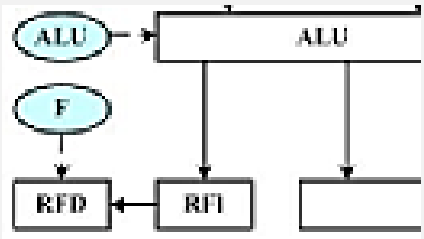
АЛУ выполняет некоторые арифметические и логические операции над операндами R и S,

выводя значение результата только в 1 регистр. Если операция включает в себя +C0, то это означает, что есть возможность к результату добавить 1 (сделать инкремент). Это осуществляется указанием 1 в поле микрокоманд CCX. По умолчанию CCX=0 ⇒ C0=0.

Поле ALU	Операция АЛУ	Флажки				Поле ALU	Операция АЛУ	Флажки			
		N	Z	V	C			N	Z	V	C
0	На всех выходах «0»	0	1	0	0	8	Умножение на 2 бита	+	+	+	+
1	S - R - 1 + C0	+	+	+	+	9	R & S	+	+	0	0
2	R - S - 1 + C0	+	+	+	+	A	$R \& \overline{S}$	+	+	0	0
3	R + S + C0	+	+	+	+	B	$\overline{R \& S}$	+	+	0	0
4	S + C0	+	+	+	+	C	R V S	+	+	0	0
5	$\overline{S}$ + C0	+	+	+	+	D	$\overline{R \vee S}$	+	+	0	0
6	R + C0	+	+	+	+	E	$R \oplus S$	+	+	0	0
7	$\overline{R}$ + C0	+	+	+	+	F	$\overline{R \oplus S}$	+	+	0	0

3.3 Флажки

По результату выполнения операции в RFI формируются флажки, показывающие характеристики результата и выполнения операции.



В Micro есть 6 флажков.

N (negative) - флажок отрицательного числа. Если флаг C=0, то N равен старшему биту регистра. Если C=1, то всегда N=1.

Z (zero) - флажок нуля. Если весь регистр и C равны нулю, то Z=1.

V (overflow) - флажок внутреннего переноса. Если в результате выполнения операции какой-то из битов наехал на старший бит, либо переполнение, то V=1.

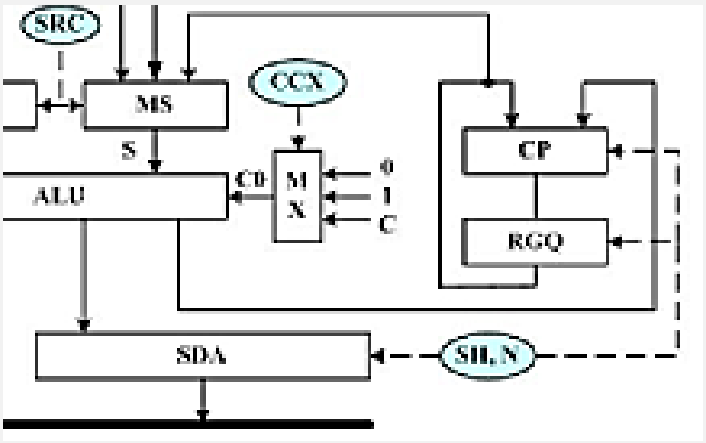
C (carry) - флажок переполнения. Если в результате выполнения операции какой-то из битов вышел из старшего разряда регистра, то C=1.

P (parity) - флажок паритета. Если во всём регистре и C нечётное количество единиц, то P=1.

M - признак переноса при выполнении операции умножения на два разряда.

На каждом новом такте в RFI формируются новые флажки на основании числа, полученного в АЛУ. При необходимости сохранить значения флажков из регистра RFI в RFD нужно указать 1 в поле микрокоманды F.

3.4 Сдвигатели

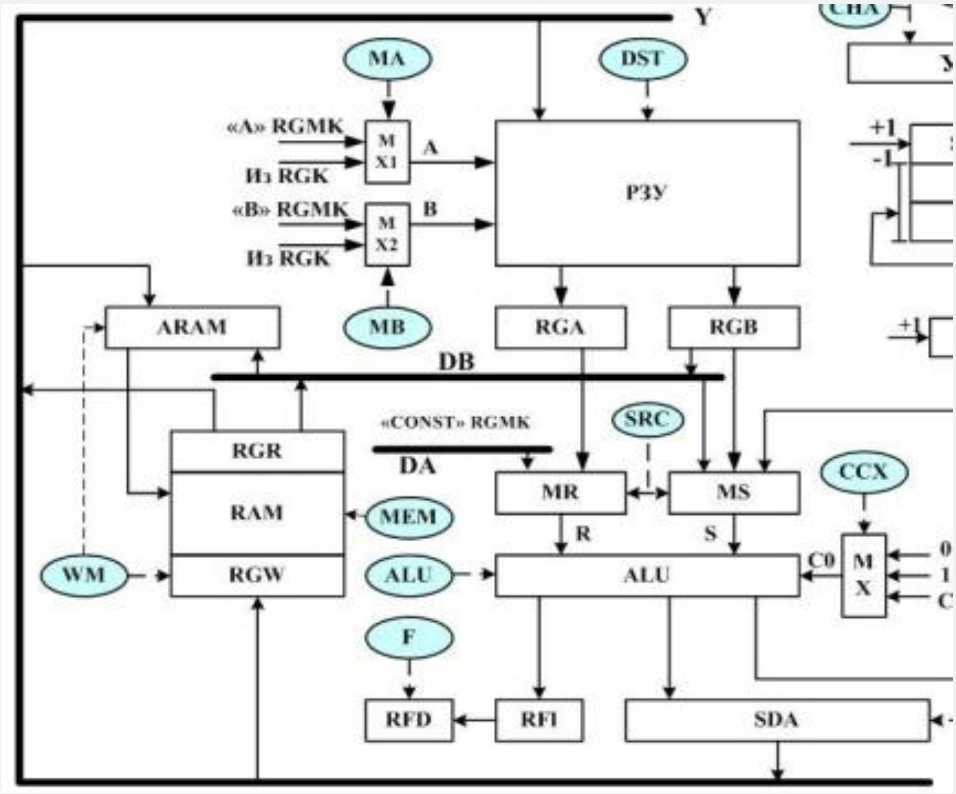


Для выполнения операций сдвигов есть 2 сдвигателя: SDA, CP. Управляются они полями микрокоманды SH и N, где SH указывает вид сдвига, а N - количество сдвигов.

Поле SH	**Операция**
0	Без сдвига
1	АС АЛУ вправо
2	ЛС АЛУ вправо
3	АС АЛУ, RGQ вправо
4	ЛС АЛУ, RGQ вправо
6	RGQ⇐ALU (загрузка из ALU в RGQ без изменения значения АЛУ)
8	ЛС АЛУ влево
A	ЛС АЛУ, RGQ влево
E	Расширение знака (дублирование старшего бита младшего байта АЛУ на каждый бит старшего байта, что позволяет расширить однобайтовое число со знаком на всё слово)

Для RGQ используется сдвигатель CP, для АЛУ сдвигатель SDA. Если разрядов АЛУ не хватает для выполнения какой-то операции, то можно использовать рабочий регистр RGQ. При выполнении сдвигов двух регистров АЛУ и RGQ образуют 32-битное число, в котором АЛУ - старшие 16 бит, RGQ - младшие.

3.5 Запись в РЗУ



Для записи значения регистра в РЗУ по шине Y используется поле микрокоманд DST. Поле DST выполняется после действия в АЛУ и действия над оперативной памятью.

Поле DST	**0**	**1**	**2**	**3**	**4**
Источник	Без записи	RGR	RGRL	RGRH	SDA
Приемник	Без записи	РЗУ	РЗУН	РЗУL	РЗУ

Выбор РОНа, в который осуществится запись, определяется полем микрокоманд B (если поле MB=0). Запись может осуществляться: из сдвигателя SDA, в которое всегда выходит значение из АЛУ; из регистра чтения оперативной памяти RGR (подробнее в разделе 4). Есть возможность записи в РОН не всего слова из RGR, а только одного из байтов, старшего (H - high) или младшего (L - low). Например, при указании DST=2 и B=0 (MB=0) в старший байт регистра AX запишется значение младшего байта регистра RGR.

4 Работа с оперативной памятью (RAM)

RAM - random access memory - память с произвольным доступом - в большинстве случаев энергозависимая часть системы компьютерной памяти, в которой во время работы компьютера хранится выполняемый машинный код (программы), а также входные, выходные и промежуточные данные, обрабатываемые процессором.

Данные RAM находятся в разделе данные/память. Есть представление данных как по словам, так и по байтам. Адрес RAM указывает на соответствующий байт. Слово состоит из двух байтов, записанных в порядке убывания. То есть слово с адресом 000 будет иметь в младшей половине байт с адресом 000, а в старшей байт с адресом 001.

(старшие тетрады адреса расположены по вертикали, младшая тетрада по горизонтали)

Байты:	**Адр**	**0**	**1**	**2**	**3**
	00	0F	13	28	6B

Слова:	**Адр**	**0**	**2**
	00	130F	6B28

У RAM есть 3 регистра, позволяющие работать с ней. ARAM - адрес RAM, RGR - регистр чтения из RAM, RGW - регистр записи в RAM.

Для чтения/записи данных из RAM, нужно указать адрес байта или слова. Этот адрес должен быть записан в регистре ARAM. Это можно сделать как через регистр RGB, так и через SDA, указав соответствующее значение в поле микрокоманд WM. Также WM позволяет записать какое-то значение из SDA в RGW для последующей записи какого-то значения в RAM. Поле WM выполняется после действия в АЛУ, но перед действием над ОП.

Поле WM	**0**	**1**	**2**	**3**
Источник	Без записи	SDA	SDA	RGB
Приемник	Без записи	RGW	ARAM	ARAM

Для чтения/записи слова/байта нужно указать соответствующее значение в поле микрокоманды MEM.

Поле MEM	**4**	**5**	**6**	**7**
	Чтение байта	Чтение слова	Запись байта	Запись слова

Прочитанный байт или слово присваивается в регистр RGR. Запись байта или слова осуществляется из регистра RGW.

Поле MEM выполняется после действия в АЛУ, но перед записью в РЗУ.

Если требуется прочитать слово из RAM и произвести с ним какую-то операцию в АЛУ, то за 1 такт это сделать не получится, так как по очереди микроопераций чтение из RAM произойдёт после операции в АЛУ. Для этого микрооперацию занесения адреса читаемого слова в ARAM и само чтение нужно сделать в одной микрокоманде, а операцию в АЛУ в следующей микрокоманде.

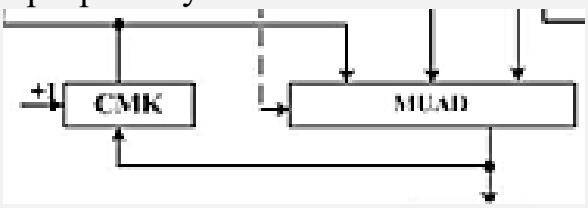
Если требуется произвести операцию в АЛУ над словом и потом записать его в RAM, то тут не возникает проблемы с тем, чтобы разместить это всё в одной микрокоманде, так как сначала выполнится операция в АЛУ, а потом в последнюю очередь произведётся запись слова в RAM.

Если требуется прочитать слово из RAM и записать его в РЗУ, то эти микрооперации тоже можно разместить в одной микрокоманде.

5 Блок микропрограммного управления (БМУ)

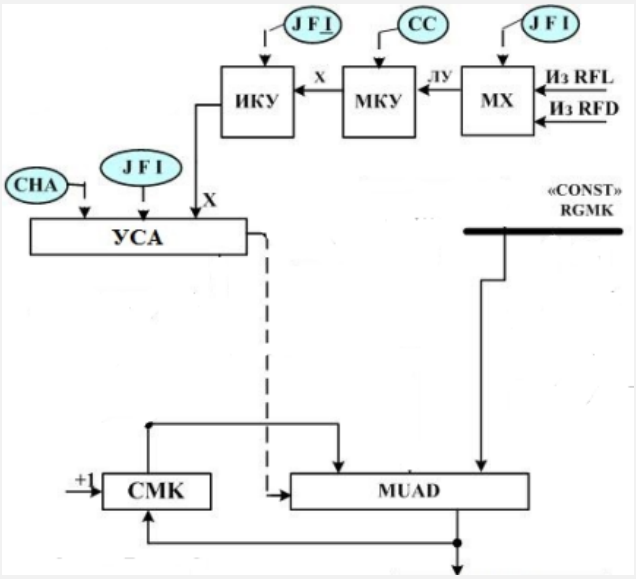
5.1 Общие сведения о БМУ

Основная задача БМУ - формирование адреса следующей выполняемой микрокоманды. По умолчанию, адрес следующей микрокоманды определяется инкрементом к адресу текущей микрокоманды. Адрес микрокоманды, которая будет выполняться следующей, выбирается мультиплексором адреса MUAD, после чего переносится в счётчик микрокоманд СМК и в микропрограммную память МРМ.



Существуют следующие способы формирования следующего адреса: инкремент к текущему адресу; возврат адреса из стека при выходе из какой-то подпрограммы; условный переход в зависимости от значения флажков; безусловный переход; переход по счётчику цикла с постусловием RACT; переход к началу команды.

5.2 Условный переход по флажкам и безусловный переход. Остановка процессора



За формирование условия перехода отвечает поле микрокоманды CC. Условие перехода формируется на основании значения флажков.

Поле CC	Вид перехода	Условие перехода
0	JP, JNP*	P=1
1	JZ, JNZ*	Z=1
2	JS, JNS*	N=1
3	JO, JNO*	V=1
4	JC, JNC*	C=1
5	JL, JNL*	$N \oplus V=1$
6	JLE, JNLE*	$Z \vee (N \oplus V)=1$
7	JBE, JNBE*	$C \vee Z=1$

За интерпретацию условия отвечает поле микрокоманды JFI. Оно состоит из трёх бит, расположенных в той же последовательности: J (jump) - безусловный переход при J=1, условный переход при J=0; F (flag) - бит, ответственный за регистр, из которого брать флаги.

RFI (значения флажков на текущем такте) при F=0, RFD (регистр длительного хранения флажков) при F=1; I (inversion) - инверсия условия при I=1, сохранение условия неизменным при I=0.

Например, если необходимо осуществить переход по ненулевому значению в АЛУ, то в поле CC указывается 1 (переход по нулю Z=1), а в поле JFI указывается 1 (условный переход J=0, переход по флажкам RFI текущего значения АЛУ F=0, инверсия условия I=1).

Поле микрокоманды CHA отвечает за способ формирования следующего адреса (на схеме CHA определяет работу УСА - управления следующем адресом). Рассмотрим некоторые из значений CHA. X - условие, определённое полями CC и JFI. X=1 - условие выполнено, X=0 - не выполнено. При JFI=4 (J=1) условие всегда выполнено. Y - адрес перехода.

Поле CHA	**Мнемоника**	**X=0**	**X=1**
		Y	**Y**
0	JZ	0	0
3	CJP	CMK	CONST
7	CONT	CMK	CMK

CHA=0 - безусловный переход к нулевой микрокоманде.

CHA=3 - переход по константе при выполнении условия, иначе к следующей по порядку микрокоманде.

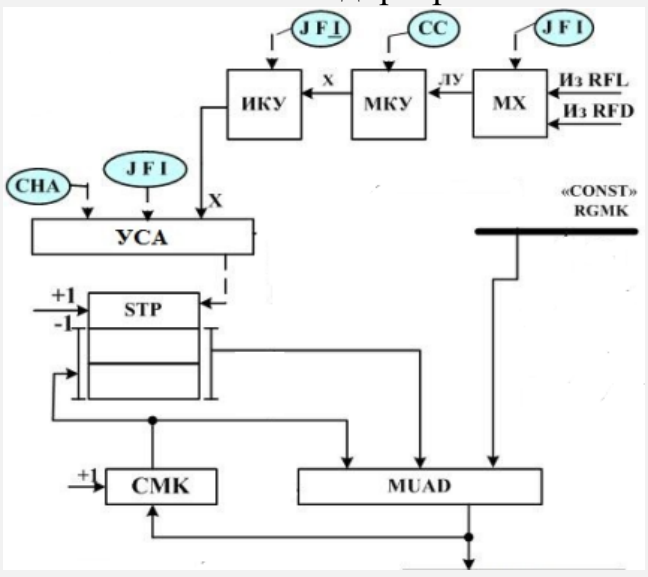
CHA=7 (по умолчанию) - безусловный переход к следующей по порядку микрокоманде.

Остановка процессора осуществляется указанием JFI=5 (J=1, I=1). Так как при J=1 условие считается всегда выполненным, а при I=1 истинность условия инвертируется, то возникает противоречие, останавливающее процессор.

Пример: Микрокоманда 00 перехода к микрокоманде с адресом 09, если результат суммы AX и DX не вызвал переполнение (C=0), иначе переход к следующей по порядку микрокоманде, вызывающей остановку процессора.

Адрес	A	B	MA	MB	MEM	SRC	SH	N	ALU	CCX	F	DST	WM	JFI	CC	CHA	CONST
00	0	2	0	0	0	1	0	0	3	0	0	0	0	1	4	3	0009
01	0	0	0	0	0	1	0	0	6	0	0	0	0	5	0	7	0000

5.3 Стек. Подпрограммы



Программы часто содержат в себе подпрограммы (ПП). ПП обычно удалены от основной программы. Для вызова ПП в CONST указывается адрес первой микрокоманды ПП. При этом, чтобы можно было вернуться к основной части программы после выполнения ПП, нужно указать адрес возврата. Для этого в конце ПП можно так же указать в CONST адрес перехода к микрокоманде основной части программы, идущей следующей за вызовом ПП. Но этот способ не всегда удобен, поэтому существует стек, автоматически запоминающий адрес возврата. При вызове ПП в стек заносится следующая микрокоманда после этого вызова и по CONST осуществляется переход к ПП. При выходе из ПП адрес перехода автоматически выгружается из стека и основная часть программы выполняется дальше. Стек может хранить в себе несколько адресов. Это нужно для возможности входа из одной ПП в другую. При этом из стека всегда будет выгружаться последний загруженный в него адрес. Рассмотрим пример использования стека.

↓ - вход в ПП, ↑ - выход из ПП.

Адреса микрокоманд основной программы: 00, 01, 02↓, **03**, 04.

Адреса микрокоманд ПП1: 10, 11, 12, 13↓, **14**↑.

Адреса микрокоманд ПП2: 20, 21↑.

Стек после выполнения микрокоманды 02: **03**, где 03 - вершина стека.

Стек после выполнения микрокоманды 13: **03, 14**, где 14 - вершина стека.

Стек после выполнения микрокоманды 21: **03**, где 03 - вершина стека.

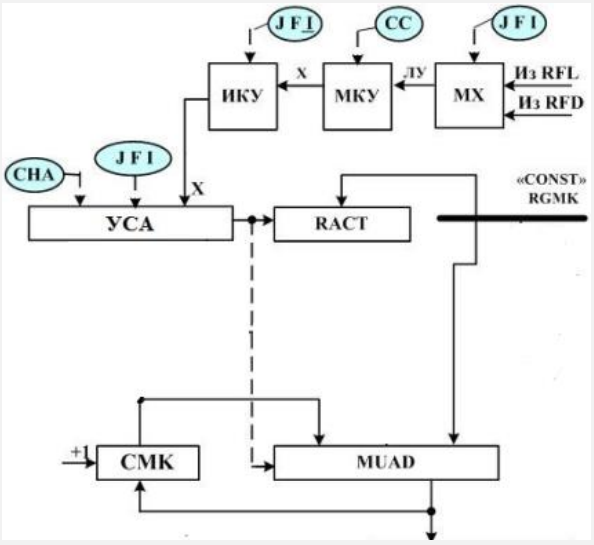
Стек после выполнения микрокоманды 14: пустой.

Рассмотрим некоторые из значений поля CHA, позволяющие управлять стеком.

Поле CHA	**Мнемоника**	**X=0**		**X=1**	
		Y	**Стек**	**Y**	**Стек**
0	JZ	0	Очистка	0	Очистка
1	CJS	CMK	Хранение	CONST	Загрузка
5	CRTN	CMK	Хранение	Стек	Выгрузка

Для входа в ПП должно быть выполнено условие, либо указан безусловный переход, в CONST записан адрес первой микрокоманды ПП, CHA=1. Для выхода из ПП должно быть выполнено условие, либо указан безусловный переход, CHA=5.

5.4 RACT. Циклы с параметром



Один из видов циклов - цикл с параметром (счётчиком). Особенность этих циклов заключается в том, что они выполняются определённое количество раз. Если количество итераций (прохождений циклов) неизвестно, но известно, в каком регистре оно должно хранится, то в конце тела цикла нужно расположить операцию декремента счётчика и проверки результата на 0. Если результат нулю не равен, то переход к началу цикла, иначе переход на следующую по порядку микрокоманду. Но если количество итераций заранее известно, то такой цикл можно реализовать через счётчик RACT. Перед началом тела в цикла из CONST нужно загрузить в RACT число на единицу меньшее, чем количество итераций. В конце тела цикла нужно расположить операцию декремента RACT. После осуществления декремента, процессор автоматически проверяет неравенство RACT нулю. Если RACT ненулевой, то условие считается выполненным и по CONST происходит переход на начало тела цикла, иначе переход к следующей по порядку микрокоманде. Рассмотрим некоторые из значений поля CHA, позволяющие управлять счётчиком RACT. X - условие неравенства RACT нулю.

Поле CHA	**Мнемоника**	**X=0**	**X=1**	**RACT**
		Y	**Y**	
4	RPCT	CMK	CONST	Декремент
6	LDCT	CMK	CMK	Загрузка

Пример микропрограммы, 10 раз прибавляющей к AX значение регистра DX с сохранением промежуточного результата в AX и остановкой процессора после выполнения цикла.

Адрес	A	B	MA	MB	MEM	SRC	SH	N	ALU	CCX	F	DST	WM	JFI	CC	CHA	CONST
00	0	0	0	0	0	1	0	0	6	0	0	0	0	0	0	6	0009
01	2	0	0	0	0	1	0	0	3	0	0	4	0	0	0	4	0001
02	0	0	0	0	0	1	0	0	6	0	0	0	0	5	0	7	0000

Микрокоманда 00: загрузка 9 в RACT.

Микрокоманда 01: сложение AX и DX, декремент RACT и возврат при ненулевом RACT на начало цикла (на эту же микрокоманду, так как цикл состоит из одной микрокоманды).

Микрокоманда 02: остановка процессора.

6 Выполнение команд ассемблера в процессоре

6.1 Общие сведения о выполнении команд

Программист пишет программы либо на ассемблере, либо на языках более высокого уровня, конвертируемых в ассемблерный код. Программы ассемблера представляют из себя некоторый набор команд, выполняемых процессором. У каждой команды есть свой двоичный код. В принстонской архитектуре ЭВМ, команды и данные хранятся в одной памяти. Пользователь или программист может взаимодействовать только с оперативной памятью, микропрограммная память всегда остаётся неизменной. Микропрограммная память обычно выполняет роль хранилища большого количества подпрограмм (ПП), каждая из которых соответствует определённой команде. При считывании кодов команд из оперативной памяти процессор определяет, где находится начальный адрес ПП, реализующей выполнение этой команды, и выполняет её. Так как программирование основано на наборах команд, а не микрокоманд, то некоторые привычные функции процессора будут работать по-другому.

На уровне команд:

Выполнение происходит не по микрокомандам в микропрограммной памяти, а по командам в оперативной памяти, каждая из которых представляет собой набор микрокоманд;

Роль СМК (адреса следующей выполняемой микрокоманды) выполняет регистр IP (адрес следующей выполняемой команды);

Стек реализован занесением адреса следующей по порядку команды (из IP) в сегмент оперативной памяти, ответственной за стек. Заполнение стека происходит с конца сегмента. За указание адреса вершины стека отвечает регистр SP;

Счётчиком цикла с параметром является регистр CX, а не RACT;

Флажки проверяются не в данный момент по RFI, а отдельной командой по флажкам, сохранённым с прошлой команды в RFD;

Переходы между командами и циклы выполняются изменением адреса оперативной памяти, хранящегося в IP;

Не микропрограммист, а программист выбирает операнды, для чего микропрограммы должны быть более универсальными…

6.2 Сегменты оперативной памяти. Назначения РОНов

Оперативная память обычно делится на сегменты, у каждого из которых своё предназначение. Существуют сегменты программ, данных, устройств ввода/вывода и некоторые другие. Сегменты также могут иметь свою структуру. Например, сегмент данных может хранить в себе несколько массивов, на каждый из которых отведён определённый отрезок сегмента оперативной памяти.

У регистров общего назначения, не смотря на своё название, рекомендуемое назначение всё-таки имеется.

РОНы:

AX (accumulator) - аккумулятор. Хранит в себе промежуточные вычисления.

CX (count) - счётчик. Хранит в себе счётчики для некоторых вычислений.

DX (data) - данные. Хранит в себе дополнительные данные.

BX (base) - база. Указывает на базовый (начальный) адрес памяти.

Указатели и индексы:

SP (stack pointer) - указатель стека. Указывает на адрес вершины стека.

BP (base pointer) - указатель базы. Используется для доступа к параметрам функции и локальным переменным в стеке.

SI (source (src) index) - индекс источника. Указывает индекс массива при чтении.

DI (destination (dst) index) - индекс получателя. Указывает индекс массива при записи.

Сегментные регистры:

CS (code segment) - сегмент кода. Указывает начальный адрес сегмента кода.

SS (stack segment) - сегмент стека. Указывает начальный адрес сегмента стека.

DS (data segment) - сегмент данных. Указывает начальный адрес сегмента данных.

ES (extra segment) - сегмент дополнительных данных. Указывает начальный адрес сегмента дополнительных данных.

Специальные регистры:

IP (instruction pointer). Указывает адрес следующей выполняемой команды.

RGK (command register) - регистр команд. Хранит в себе код выполняемой команды.

6.3 Структура команд

Команды могут содержать: код операции (КОп); режим адресации (mod); номера регистров РЗУ, участвующих в операции (reg, r/m); операнд (данные, участвующие в операции) (data); смещение адреса команды (disp); адрес операнда в оперативной памяти (addr). В зависимости от типа команды, какие-то из перечисленных выше элементов могут быть обязательными, необязательными или полностью отсутствовать в команде. Код операции - обязательный элемент любой команды.

Типы команд:

R – операция над регистром reg;

AR – операция над регистром reg и аккумулятором AX;

RM – операция над регистром r/m или словом памяти;

RRM – операция над регистром reg и регистром r/m или словом памяти;

ARM – операция над AX и регистром r/m или словом памяти;

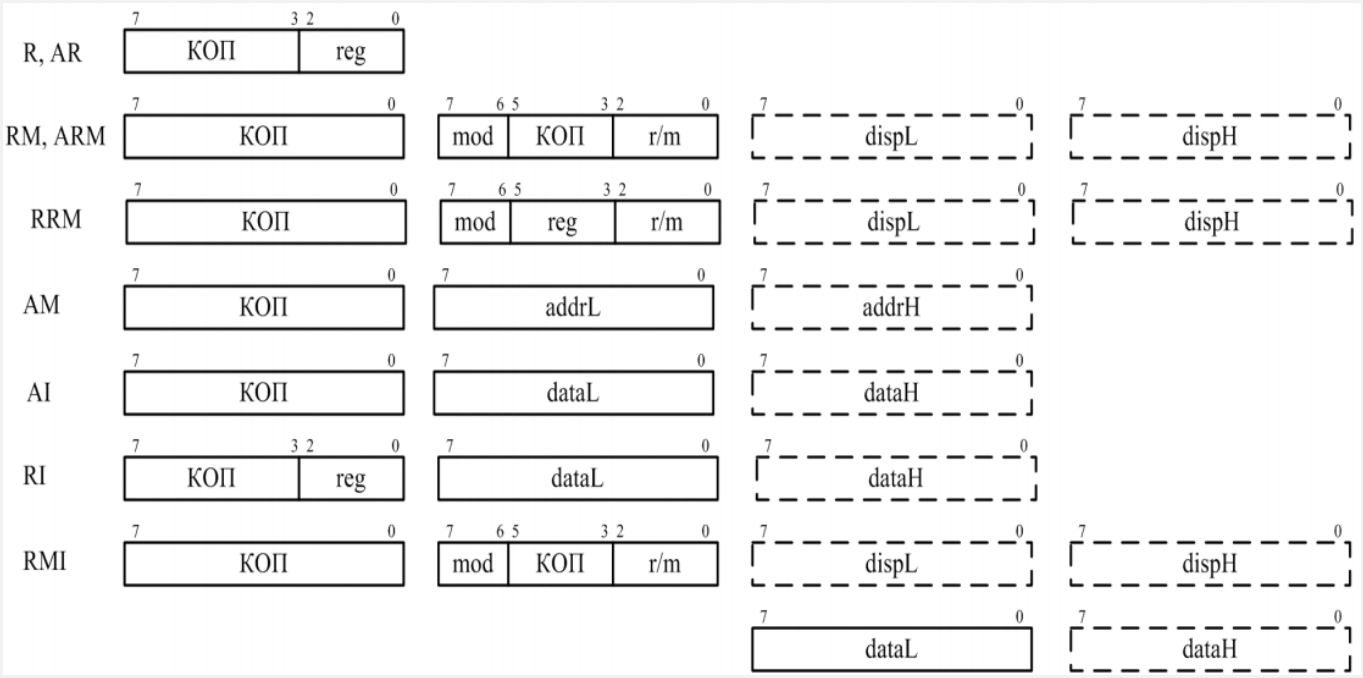
AM – операция над AX и словом памяти [addr];

AI – операция над AX и непосредственным одно- или двухбайтовым операндом;

RI – операция над регистром reg и непосредственным операндом;

RMI – операция над регистром r/m или словом памяти и непосредственным операндом.

Схема, показывающая структуру разных типов команд (пунктиром выделены необязательные элементы):



В трёхбитном поле команды reg записывается номер регистра РЗУ, участвующего в операции. Нумерация у регистров такая же, как и при указывании их в полях микрокоманды A и B. Так как поле команд reg трёхбитное, то в нём можно указать только первые 4 РОНа, указательные и индексные регистры.

При выборе режима адресации mod=11 поле команд r/m работает идентично полю reg. Другие режимы адресации рассмотрены в пункте 6.6.

В полях addr указывается адрес операнда (данного, участвующего в операции), хранящегося в оперативной памяти. Если адрес умещается в одном байте, то нет необходимости задействовать поле addrH.

В полях data хранится сам операнд, а не ссылка на него.

В полях disp хранится смещение IP. IP отвечает за указание следующей для выполнения команды. По умолчанию команды идут по порядку, но при необходимости в поле disp можно указать, на сколько ячеек (байтов) оперативной памяти нужно сдвинуться относительно текущего положения IP, чтобы перейти к нужной команде. Смещение может быть как положительным, так и отрицательным, поэтому старший бит смещения отведён под знак.

6.4 Виды команд

Обозначения в таблицах:

src – источник; dst – приемник;

бит s задает длину непосредственного операнда - слово (s=0) или байт (s=1);

бит d задает функцию регистра reg - источник (d=0) или приемник (d=1);

бит v управляет сдвигами - на 1 бит (v=0) или на число битов, заданное в регистре CX (v=1);

бит w - операнды байты (w=0) слова (w=1).

Состояния флажков обозначены следующим образом:

0 (1) – значения флажков равны этому значению при любом исходе операции, флажки результата необходимо сохранять в RFD;

X – флажок устанавливается в соответствии с результатом операции, флажки результата

необходимо сохранять в RFD;

“ – “ – флажок не имеет смысла, нет необходимости сохранения его в RFD.

Пересылочные операции. Отвечают за перемещение данных из одного места в другое.

	Операция	**Тип**	**Байт 1** **7654 3210**	**Байт 2** **76 543 210**	**Флажки** **S Z N C**
1	MOV – передать MOV dst, src; dst ← (src)	RRM RMI RI AM	1000 10d1 1100 0111 1011 1reg 1010 00d1	mod reg r/m mod reg r/m	– – – –
2	XCHG – обменять XCHG dst, src; (dst) ←→ (src)	RRM AR	1000 0111 1001 0reg	mod reg r/m	– – – –
3	PUSH – включить в стек PUSH src; 1) SP ← SP – 2; 2) (SP) ← (src).	RM R	1111 1111 0101 1reg	mod 110 r/m	– – – –
4	POP – извлечь из стека POP dst; 1) (SP) → (dst); 2) SP ← SP + 2.	RM R	1000 1111 0101 1reg	mod 000 r/m	– – – –

Арифметические и логические операции.

	Операция	**Тип**	**Байт 1** **7654 3210**	**Байт 2** **76 543 210**	**Флажки** **S Z N C**
1	ADD – сложить ADD dst, src; dst:=(dst) + (src)	RRM RMI AI	0000 00d1 1000 00s1 0000 0101	mod reg r/m mod 000 r/m	X X X X
2	SUB – вычесть SUB dst, src; dst:=(dst) - (src)	RRM RMI AI	0010 10d1 1010 00s1 0010 1101	mod reg r/m mod 000 r/m	X X X X
3	INC – инкремент INC src; (src):=src+1	RM R	1111 1110 0100 0reg	mod 000 r/m	X X X X
4	DEC – декремент DEC src; (src):=src-1	RM R	1111 1111 0100 1reg	mod 000 r/m	X X X X
5	CMP – сравнить CMP dst, src; (dst) - (src)	RRM RMI AI	0011 10d1 1011 00s1 0011 1101	mod reg r/m mod 000 r/m	X X X X
6	AND – объединить по И AND dst, src; dst:=(dst) & (src)	RRM RMI AI	0010 00d1 1000 0001 0010 0101	mod reg r/m mod 100 r/m	X X 0 0
7	TEST – проверить TEST dst, src; (dst) &(src)	RRM RMI AI	1000 0101 1111 0111 1010 1001	mod reg r/m mod 100 r/m	X X 0 0
8	OR – объединить по ИЛИ OR dst, src; dst:=(dst) v (src)	RRM RMI AI	0000 10d1 1000 0001 0000 1101	mod reg r/m mod 001 r/m	X X 0 0
9	XOR – сложение по mod2 XOR dst, src; dst:=(dst) ⊕ (src)	RRM RMI AI	0011 00d1 1000 0001 0011 0101	mod reg r/m mod 001 r/m	X X 0 0

Операции сдвигов.

	Операция	**Тип**	**Байт 1 7654 3210**	**Байт 2 76 543 210**	**Флажки S Z N C**
1	SHL – логический сдвиг влево	RM	1101 00v1	mod 100 r/m	X X X X
2	SHR – логический сдвиг вправо	RM	1101 00v1	mod 101 r/m	X X X X
3	SAR – арифметический сдвиг вправо	RM	1101 00v1	mod 111 r/m	X X X X
4	ROL – циклический сдвиг влево	RM	1101 00v1	mod 000 r/m	X X X X
5	ROR – циклический сдвиг вправо	RM	1101 00v1	mod 001 r/m	X X X X

Операции передачи управления. Отвечают за условный или безусловный переход между командами.

	Мнемоника	**Код**	**Операция**	**Коды условия (по флажкам RFD)**
1	JMP	1110 1011	Безусловный переход	-
2	JNZ	0111 0011	Переход по неравенству нулю	Z=0
3	JZ	0111 0100	Переход по равенству нулю	Z=1
4	JNS	0111 1001	Переход по плюсу	S=0
5	JS	0111 1000	Переход по минусу	S=1
6	JNO	0111 0001	Переход по непереполнению	V=0
7	JO	0111 0000	Переход по переполнению	V=1
8	JNC	0111 1011	Переход по отсутствию переноса	C=0
9	JC	0111 1010	Переход по переносу	C=1
10	JNL	0111 1101	Переход, если больше или равно	S⊕V=0
11	JL	0111 1100	Переход, если меньше	S⊕V=1
12	JNLE	0111 1111	Переход, если больше	Z∨(S⊕V)=0
13	JLE	0111 1110	Переход, если меньше или равно	Z∨(S⊕V)=1
14	JNBE	0111 0111	Переход, если больше (без знака)	C∨Z=0
15	JBE	0111 0110	Переход, если меньше или равно (без знака)	C∨Z=1
16	LOOP	1110 0010	Зациклить	CX≠0
17	LOOPZ	1100 0001	Зациклить, пока нуль или равно	Z=1 и CX≠0
18	LOOPNZ	1110 0000	Зациклить, пока нуль или не равно	Z=0 и CX≠0
19	JCXZ	1110 0011	Перейти, если CX=0	CX=0
20	HALT	1111 1111	Остановка	

Операции умножения, деления, преобразования.

	Операция	Тип	Байт 1 7654 3210	Байт 2 76 543 210	Флажки S Z N C
1	MUL src – умножение чисел без знака AX ← AL * (src) DX.AX ← AX * (src)	RM	1111 011w 1111 011w	mod 100 r/m 11 100 reg	X X X X
2	IMUL src – умножение чисел со знаком AX ← AL * (src) DX.AX ← AX * (src)	RM	1111 011w 1111 011w	mod 101 r/m 11 101 reg	X X X X
3	DIV src – беззнаковое деление AL ← AX / (src) AX ← DX.AX / (src)	RM	1111 011w 1111 011w	mod 110 r/m 11 110 reg	X X X X
4	IDIV src –деление чисел со знаком AL ← AX / (src) AX ← DX.AX / (src)	RM	1111 011w 1111 011w	mod 111 r/m 11 111 reg	X X X X
5	BLD src – преобразовать BCD (двоично-десятичное) число в двоичное	RM	1111 1111	mod 011 r/m	– – – –
6	XLAT – команда преобразования AL:=[BX+AL]		1101 0111		– – – –

6.5 Алгоритм микропрограммирования команд и программирования

Для микропрограммирования команд в микропрограммной памяти (МПП) должны иметься выборка команд и реализации самих команд. Выборка эта одинакова для всех команд и располагается в начале МПП. Выглядит она так:

Адрес МК	Операция	Поле	Значение	Функция
00	ARAM := IP IP := IP + 2	B WM SRC ALU DST CONST	C 3 5 3 4 2	IP ARAM:= RGB CONST, RGB Сложение Запись в РЗУ
01	Чтение ОП RGK := RGR Дешифрация	MEM B DST CHA	5 E 1 2	Чтение слова RGK РЗУ[B]:= RGR JMAP

В регистр адреса оперативной памяти (ОП) заносится значение IP (указателя адреса команд). К IP прибавляется 2, так как большинство команд имеют код, занимающий 2 байта. Все команды, даже однобайтовые, выровнены по слову, то есть несколько команд не могут располагаться в одном слове. Обычно, лишний байт заполняется нулями и не учитывается при распознавании (дешифрации) команды. Если есть необходимость изменить положение IP (следующая выполняемая команда идёт не следующим словом после текущего или текущая команда занимает больше двух байт), то значение IP меняется в подпрограмме (ПП), ответственной за реализацию команды. После указания адреса IP и добавления 2 к IP, в выборке команд читается слово ОП по указанному ранее в ARAM адресу. Прочитанное слово содержит в себе весь код операции команды. Прочитанное слово заносится в регистр команд RGK и указанием CHA=2 осуществляется дешифрация занесённого в RGK кода команды. Дешифрация заключается в определении адреса микрокоманды в МПП, с которого начинается ПП, реализующая выполнение команды. Для возможности дешифрации, в таблице

преобразования адресов должен быть указан шаблон дешифруемой команды, по которому можно её однозначно идентифицировать, и начальный адрес соответствующей ПП в МПП. Для универсальности команды, например, для возможности указывать разные регистры, над которыми необходимо производить действия, некоторые части одной и той же по смыслу команды могут отличаться (обычно, это поля reg, mod, r/m, addr, data, disp). Поэтому на местах, где эти универсальные поля расположены, в таблице преобразования адресов должны стоять X. Для примера рассмотрим команду XCHG типа RRM. Её код будет выглядеть так: 1000 0111 11 reg r/m (mod=11, другой режим адресации будет соответствовать другой команде). Первый (старший) байт команды неизменяемый, а второй может иметь разные значения в зависимости от выбранных регистров. Поэтому в таблице преобразования адресов шаблон команды будет записан следующим образом (02 в этом примере - начальный адрес ПП выполнения этой команды):

Начальный адрес:	Код операции:
02	1000011111XXXXXX

Команда из этого примера обменивает значения двух регистров. При режиме адресации mod=11 поля reg и r/m указывают номера регистров РЗУ, над которыми осуществляется действие. Так как эти номера регистров могут быть разными в зависимости от потребностей программиста, микропрограммист не может предвидеть их и не может в полях микрокоманды A и B указать эти регистры. Для этого есть поля MA и MB, позволяющие брать эти номера из RGK. В таблице представлены значения полей MA и MB (reg1 - поле reg первого байта, reg2 - поле reg второго байта). В примере имеются только поля reg2 и r/m.

Поле MA/MB	**0**	**1**	**2**	**3**
Источник адресов регистров РЗУ	Поле RGMK A или B	Поле RGK reg1	Поле RGK reg2	Поле RGK r/m

При указании в MA или MB значения 1-3 поля A или B соответственно будут игнорироваться, а номер регистра будет браться из соответствующих бит команды. Рассмотрим пример выполнения следующей программы:

MOV BX, SI (переместить в BX значение SI)

XCHG BX, BP (обменять значения BX и BP)

ADD AX, DX (записать в AX сумму AX и DX)

JNC -4 (переход предыдущей команде, если нет переноса)

HALT (остановить процессор)

В подобных командах на первом месте всегда указывается приёмник (регистр, в который заносится результат), а на втором источник (регистр из которого только берётся значение). Рассмотрим эти команды как команды типа RRM. Таким образом у команд могут быть следующие шаблоны:

Операция	**Тип**	**Байт 1 7654 3210**	**Байт 2 76 543 210**	**Флажки S Z N C**
MOV – передать MOV dst, src; dst ← (src)	RRM	1000 10d1	mod reg r/m	– – – –
XCHG – обменять XCHG dst, src; (dst) ←→ (src)	RRM	1000 0111	mod reg r/m	– – – –

ADD – сложить ADD dst, src; dst:=(dst) + (src)	RRM	0000 00d1	mod reg r/m	X X X X
JNC		0111 1011		
HALT		1111 1111		

Имеются 2 поля команды, в которых можно указать пару операндов, в которых одно из них будет указывать на источник, другое на приёмник. Для выбора назначения поля reg есть бит d в командах MOV и ADD (команд, в которых источник и приёмник имеют разный функционал). Укажем поле reg как ответственное за указание приёмника, d=1 (подробнее по обозначениям в пункте 6.4).

Для начала микропрограммирования этих команд, заполним МПП нужными ПП.

В начале МПП расположим выборку команд.

Адрес	A	B	MA	MB	MEM	SRC	SH	N	ALU	CCX	F	DST	WM	JFI	CC	CHA	CONST
00	0	C	0	0	0	5	0	0	3	0	0	4	3	0	0	7	2
01	0	E	0	0	5	1	0	0	6	0	0	1	0	0	0	2	0000

Далее расположим ПП команды MOV.

Адрес	A	B	MA	MB	MEM	SRC	SH	N	ALU	CCX	F	DST	WM	JFI	CC	CHA	CONST
02	0	0	3	2	0	1	0	0	6	0	0	4	0	0	0	0	0000

В этой ПП из РЗУ в RGA выходит регистр под номером, указанным в поле r/m (MA=3), а в RGB выходит регистр под номером, указанным в поле reg2 (MB=2). Значение RGA проходит без изменений через ALU, SDA и записывается обратно в РЗУ (DST=4) в регистр под номером, указанном в поле reg2 (MB=2). После выполнения присвоения происходит возврат на нулевой адрес (CHA=0), то есть к выборке команд. CHA=0 должно присутствовать в конце каждой ПП.

ПП команды XCHG может выглядеть следующим образом:

Адрес	A	B	MA	MB	MEM	SRC	SH	N	ALU	CCX	F	DST	WM	JFI	CC	CHA	CONST
03	0	0	3	0	0	1	6	0	6	0	0	0	0	0	0	7	0000
04	0	0	2	3	0	1	0	0	6	0	0	4	0	0	0	7	0000
05	0	0	0	2	0	2	0	0	4	0	0	4	0	0	0	0	0000

Микрокоманда 03: занесение регистра РЗУ под номером r/m (РЗУ[r/m]) в RGQ.

Микрокоманда 04: занесение в РЗУ[r/m] значения РЗУ[reg2].

Микрокоманда 05: занесение значения из RGQ в ALU и запись его в РЗУ[reg2], переход к выборке команд.

Далее запишем ПП команды ADD.

Адрес	A	B	MA	MB	MEM	SRC	SH	N	ALU	CCX	F	DST	WM	JFI	CC	CHA	CONST
06	0	0	3	2	0	1	0	0	3	0	1	4	0	0	0	0	0000

Главное отличие этой ПП от предыдущих - это необходимость сохранять флажки (F=1) для дальнейшего их анализа в следующей команде.

ПП команды JNC -4:

Адрес	A	B	MA	MB	MEM	SRC	SH	N	ALU	CCX	F	DST	WM	JFI	CC	CHA	CONST
07	0	E	0	0	0	1	E	0	4	0	0	4	0	2	4	3	0009
08	E	C	0	0	0	1	0	0	3	0	0	4	0	0	0	7	0000
09	0	0	0	0	0	1	0	0	6	0	0	0	0	0	0	0	0000

Так как код операции команд передачи управления однобайтовый, то в следующем байте того же слова можно указать смещение адреса, если его можно уместить в 1 байт. Так как нужно переместиться на предыдущую команду, то есть на команду ADD, при выполнении отсутствия

переполнения, то в смещение (dispL) записывается -4. Так как после дешифрации команды JNC значение IP уже увеличилось на 2, то есть переключилось на ожидаемый адрес следующей команды, то нам из IP нужно вычесть 2 при выполнении условия, так как в таком случае нам не нужен переход на следующую по порядку команду. При вычитании 2 IP вернётся на команду JNC, а по заданию необходим переход на ещё одну команду назад. Так как ни одна команда этой программы не превысила размеров двух байт, то для перехода к предыдущей по порядку команде нужно снова вычесть 2 из IP. Так и получается, что в данном случае переходом к предыдущей команде является смещение -4.

Микрокоманда 07: расширение знака (SH=E) RGK (расширение значения младшего байта на всё слово), анализ флажков RFD (JFI=2) на переполнение (CC=4). Если переполнение имеется, то не изменять IP и перейти (CHA=3) к микрокоманде 09 (CONST=0009) выхода из ПП. Если переполнения нет, то переход к следующей по порядку микрокоманде.

Микрокоманда 08: занесение в IP суммы IP и RGK.

Микрокоманда 09: выход из ПП.

ПП команды HALT реализовано установкой значения поля JFI=5 микрокоманды по адресу 0A.

МПП заполнена нужными ПП. Теперь нам известны адреса первых микрокоманд каждой из ПП, ответственной за каждую из команд. Далее заполним таблицу преобразования адресов шаблонами этих команд, указав в каждой из них режим адресации mod=11.

Начальный адрес:	Код операции:
02	1000101111XXXXXX
03	1000011111XXXXXX
06	0000001111XXXXXX
07	01111011XXXXXXXX
0A	11111111XXXXXXXX

Считывание команд из ОП готово, команды замикропрограммированы, распознавание команд и пересылки на нужные ПП реализованы, осталось лишь ввести эти команды.

Коды команд будут выглядеть следующим образом:

Команда	Двоичный код					16-ичная система счисления
	1 байт		2 байт			
MOV BX, SI	КОп		mod	reg2 (BX)	r/m (SI)	8BDE
	10001	011	11	011	110	
XCHG BX, BP	КОп		mod	reg2 (BX)	r/m (BP)	87DD
	10000	111	11	011	101	
ADD AX, DX	КОп		mod	reg2 (AX)	r/m (DX)	03C2
	00000	011	11	000	010	
JNC -4	КОп		dispL (-4)			7BFC
	01111	011	11	111	100	
HALT	КОп		-			FF00
	11111	111	00	000	000	

Пусть сегмент кода в ОП начинается с нулевого адреса. IP изначально равен 0, поэтому первая команда считается из нулевого слова ОП. Впишем команды программы по порядку с учётом того, что каждая команда начинается с нового слова.

Адр	0	2	4	6	8
00	8BDE	87DD	03C2	7BFC	FF00

Выполнение программы можно запустить в любом из четырёх режимов (выбирается в данные/регистры/режим). МК - выполнение по одной микрокоманде, Авто МК - бесконечное выполнение по микрокомандам (останавливается после встречи микрокоманды, в поле JFI которой указано 5), К - выполнение по командам, Авто К - бесконечное выполнение по командам (останавливается после встречи команды, содержащей микрокоманду, в поле JFI которой указано 5). При работе в любых режимах кроме МК убедитесь, что программа работает правильно и, на всякий случай, сохраните всё содержимое ОП, МПП, регистров и таблицы преобразования адресов перед запуском!

6.6 Способы адресации

В пункте 6.5 были рассмотрены команды с регистровым способом адресации (он использовался в командах MOV, XCHG, ADD). Это означает, что операндами являются регистры из РЗУ, номера которых указываются прямо в команде.

Пример регистровой адресации MOV AX, DX (загрузить значение из регистра РЗУ DX в регистр AX). Существует ещё несколько способов указать адрес операнда.

Непосредственная адресация. При таком способе адресации операнд хранится непосредственно в команде. За это отвечают поля команды data. Вся команда вместе с этим операндом может уместиться как в одном слове, так и в двух. В двухбайтовой команде с непосредственным операндом (это возможно в типах команд AI и RI, когда операнд однобайтовый), его можно считать накладыванием маски 00FF на слово, хранящееся в RGK. Когда операнд (байт или слово в командах типа RMI; слово в командах типа AI и RI) полностью хранится во втором слове команды, он считывается из оперативной памяти (ОП) по адресу IP (после считывания кода операции в RGK, IP автоматически настраивается на следующее слово), после чего к IP прибавляется 2, сместив его указание на следующее слово, в котором ожидается увидеть следующую команду, если отсутствует смещение disp адреса команды.

Пример непосредственной адресации MOV AX, $9A87_h$ (загрузить значение $9A87_h$ в AX).

Абсолютная адресация. При таком способе адресации операнд хранится в сегменте данных ОП. В команде указывается адрес этого операнда (поле команды addr). Этот адрес через какой-то регистр РЗУ (желательно не IP, чтобы не потерять адрес следующей команды) заносится в ARAM и происходит считывание операнда.

Пример абсолютной адресации MOV AX, [$10B_h$] (загрузить значение слова оперативной памяти по адресу $10B_h$ в регистр AX).

Косвенно-регистровая адресация. При таком способе адресации операнд хранится в сегменте данных ОП. Но, в отличие от абсолютной адресации, адрес этого операнда находится не в команде, а в каком-то из регистров РЗУ. В команде указывается этот регистр.

Пример косвенно-регистровой адресации MOV AX, [BX] (загрузить в регистр AX значение

слова оперативной памяти по адресу, хранящемуся в регистре BX).

Базовая адресация. ОП поделена на сегменты. Сегменты в свою очередь могут иметь некоторые структуры внутри себя. Например, сегмент данных может содержать в себе несколько массивов. Есть несколько способов обратиться к какому-то из элементов какой-то структуры. Можно как в абсолютной или косвенно-регистровой адресациях напрямую указать адрес этого элемента в памяти. В базовой и индексной адресациях этот адрес формируется из двух частей: базы (начального адреса структуры), индекса (порядкового номера элемента внутри структуры). При базовой адресации в одном из базовых регистров (BX или BP) хранится база структуры, а в смещении (disp), то есть в одном из полей команды, хранится индекс этого элемента. Для определения итогового адреса этого элемента в ОП база и индекс складываются.

Пример базовой адресации MOV AX, [BP]12_h (загрузить в регистр AX значение слова оперативной памяти по адресу, получаемому сложением значения в BP и 12_h).

Индексная адресация. Работает почти как базовая адресация, только база указывается в смещении (disp), а индекс хранится в одном из индексных регистров (SI или DI).

Пример базовой адресации MOV AX, 12_h[SI] (загрузить в регистр AX значение слова оперативной памяти по адресу, получаемому сложением 12_h и значения в SI).

Базово-индексная адресация. Иногда ОП содержит в себе структуры, у которых есть ещё внутренние структуры. Например, матрица. Матрица - это двумерный массив, то есть массив массивов с какими-то элементами. Для нахождения адреса элемента такой структуры, необходимо знать базовый (начальный) адрес матрицы, внутри этой структуры надо знать, с какого места начинается нужный одномерный массив, после чего указывается индекс элемента этого массива. В таком способе адресации смещение указывает на начальный адрес сложной структуры, базовый регистр (BX/BP) хранит в себе базовый адрес внутренней структуры, индексный регистр (SI/DI) хранит в себе индекс элемента.

Пример базово-индексной адресации MOV AX, $A0_h$[BX][DI] (загрузить в регистр AX значение слова оперативной памяти по адресу, получаемому сложением $A0_h$, значения в BX и значения в DI).

Помимо разных способов адресации операнда, существуют способы адресации команды.

Относительная адресация. В пункте 6.5 относительная адресация была продемонстрирована на примере команды JNC. В этом способе адресации в смещении (disp) указывается значение, которое нужно добавить к IP для перехода на нужную команду.

В коде команды способ адресации определяется значением полей команды mod и r/m. Значение поля mod зависит от необходимости использовать смещение в адресации и от его длины. Значение поля r/m зависит от используемых регистров, используемых как слагаемые адреса.

mod r/m	**00**	**01**	**10**
000	BX+SI	BX+SI+dispL	BX+SI+dispH,dispL
001	BX+DI	BX+DI+dispL	BX+DI+ dispH,dispL
010	BP+SI	BP+SI+dispL	BP+SI+ dispH,dispL
011	BP+DI	BP+DI+dispL	BP+DI+ dispH,dispL
100	SI	SI+dispL	SI+ dispH,dispL
101	DI	DI+dispL	DI+ dispH,dispL
110	dispH,dispL	BP+dispL	BP+ dispH,dispL
111	BX	BX+dispL	BX+ dispH,dispL

Пример реализации команд с базово-индексной адресацией. Так как в одной микропрограммной памяти хранится много команд с одной и той же адресацией, то часть реализации команды, формирующей адрес операнда, можно оформить в виде подпрограммы (ПП), к которой будут обращаться все команды с этим типом адресации. Реализуем команды:

MOV CX, 10_h[BX][SI] (dst:=src перемещение значения)

TEST DX, 20_h[BX][SI] (dst&src конъюнкция без присвоения для формирования флажков)

Обе команды обращаются к одному из операндов с помощью базово-индексной адресации вида: BX+SI+dispL. Для выбора этого способа адресации в команде нужно указать значения mod=01, r/m=000 (определяются по таблице выше). Помимо этого в команде должно присутствовать поле dispL для однобайтового смещения и поле reg для регистровой адресации второго операнда. Всеми этими полями обладает тип команд RRM.

Операция	**Тип**	**Байт 1** **7654 3210**	**Байт 2** **76 543 210**	**Флажки** **S Z N C**
MOV – передать MOV dst, src; dst ← (src)	RRM	1000 10d1	mod reg r/m	– – – –
TEST – проверить TEST dst, src; (dst) &(src)	RRM	1000 0101	mod reg r/m	X X 0 0

Команда	Двоичный код						16-ичная система счисления
	1 байт		2 байт			3 байт	
MOV CX, 10_h[BX][SI]	КОп		mod (dispL)	reg2 (CX)	r/m (BX+SI)	dispL	8B4810
	10001	011	01	001	000	0001 0000	
TEST DX, 20_h[BX][SI]	КОп		mod (dispL)	reg2 (DX)	r/m (BX+SI)	dispL	855020
	10000	101	01	010	000	0010 0000	

Запишем коды этих команд и исходные данные в ОП.

Адр	0	2	4	6
00	8B48	1000	8550	2000
01	0000	0000	0000	1111
02	0000	0000	0000	2222

Напишем выборку команд.

Адрес	A	B	MA	MB	MEM	SRC	SH	N	ALU	CCX	F	DST	WM	JFI	CC	CHA	CONST
00	0	C	0	0	0	5	0	0	3	0	0	4	3	0	0	7	2
01	0	E	0	0	5	1	0	0	6	0	0	1	0	0	0	2	0000

Напишем ПП формирования адреса операнда.

МК 02: увеличение IP на 2 и выход из ПП.

МК 03: обнуление регистра RW, в котором будет формироваться адрес операнда.

МК 04: перенос байта смещения из старшей части прочитанного слова в младшую.

МК 05-06: суммирование RW с BX и SI, чтение операнда по адресу RW.

Адрес	A	B	MA	MB	MEM	SRC	SH	N	ALU	CCX	F	DST	WM	JFI	CC	CHA	CONST
02	0	C	0	0	0	5	0	0	3	0	0	4	3	0	0	7	0002
03	0	F	0	0	0	0	0	0	6	0	0	4	0	0	0	7	0000
04	0	F	0	0	5	1	0	0	6	0	0	3	0	0	0	7	0000
05	3	F	0	0	0	1	0	0	3	0	0	4	0	0	0	7	0000
06	6	F	0	0	5	1	0	0	3	0	0	4	2	4	0	5	0000

Реализуем команду MOV.

МК 07: вызов ПП формирования адреса и чтения операнда.

МК 08: запись RGR в регистр reg2 и выход из команды.

Адрес	A	B	MA	MB	MEM	SRC	SH	N	ALU	CCX	F	DST	WM	JFI	CC	CHA	CONST
07	0	0	0	0	0	1	0	0	6	0	0	0	0	4	0	1	0002
08	0	0	0	2	0	3	0	0	4	0	0	4	0	0	0	0	0000

Реализуем команду TEST.

МК 09: вызов ПП формирования адреса и чтения операнда.

МК 0A: конъюнкция регистра reg2 и RGR, сохранение флажков и выход из команды.

Адрес	A	B	MA	MB	MEM	SRC	SH	N	ALU	CCX	F	DST	WM	JFI	CC	CHA	CONST
09	0	0	0	0	0	1	0	0	6	0	0	0	0	4	0	1	0002
0A	0	0	2	0	0	3	0	0	9	0	1	0	0	0	0	0	0000

Теперь, когда известны начальные адреса команд, запишем их коды в таблицу преобразования адресов. Так как команды с разными способами адресации являются разными командами и реализуются отдельно, а поля mod и r/m в данном случае отвечают именно за способ адресации, то они помечаются как часть кода операции, которую невозможно выбрать произвольно, используя одну и ту же команду. В поле reg2 наоборот операнд может выбираться свободно регистровым способом адресации.

Начальный адрес:	Код операции:
07	1000101101XXX000
09	1000010101XXX000

7 Описание алгоритмов некоторых сложных арифметических операций

7.1 Алгоритмы целочисленного умножения

В Micro в АЛУ не реализована операция умножения, поэтому есть несколько способов реализовать его на сложении и сдвигах.

Самый простой и самый неоптимальный способ реализовать умножение, это сделать цикл добавления к промежуточному результату одного из множителей. На каждой итерации цикла из другого множителя будет вычитаться 1, пока этот множитель не обнулится, что и станет окончанием цикла. Математически этот способ выглядит так: x*y = x+x+x+…+x (y раз).

Для умножения на константу можно использовать комбинацию сдвигов и сложений. Это очень оптимальный способ, но он не является универсальным, так как в качестве обоих множителей нельзя использовать переменную. В пункте 1.4 показана такая реализация умножения на 10: (x>>2+x)>>1. Число было представлено в виде суммы и умножений степеней двоек (x*10 = (x*4+x)*2), после чего умножения на степени двойки были заменены сдвигами. Для каждой константы можно подобрать свою оптимальную комбинацию сложений и сдвигов. Например для умножения на 12 эта операция может выглядеть так (x+x+x)*4 ⇒ (x+x+x)>>2.

Для быстрого и универсального умножения двух целых чисел без знака существуют 4 основных алгоритма умножения (умножение дробных чисел со знаком представлено в пункте 7.3).

Эти алгоритмы имеют некоторую схожесть с математическим умножением в столбик. В математике это реализовано следующим образом: анализируется младший разряд множителя и к сумме частичного произведения (СЧП) добавляется множимое, просуммированное столько раз с самим собой, сколько указано в младшем разряде множителя, и сдвинутое на столько разрядов, сколько разрядов множителя уже было проанализировано, далее анализируется следующий разряд и к СЧП добавляется новое частичное произведение (ЧП) уже со сдвигом на один разряд влево.

В двоичной системе умножение в столбик ещё проще, чем в десятичной, так как частичное произведение образуется либо умножением на 1 (то есть просто добавлением сдвинутого множимого), либо умножением на 0 (то есть просто добавлением нулей). Для примера рассмотрим умножение 11 на 5. Важно обратить внимание, что при умножении двух операндов одинакового размера, произведение имеет размер в 2 раза больший.

				<<	1	0	1	1								
	ЧП:			>>	0	1	0	1				СЧП:				
1)	+				1	0	1	1		0	0	0	0	1	0	1	1
2)	+			0	0	0	0			0	0	0	0	1	0	1	1
3)	+		1	0	1	1				0	0	1	1	0	1	1	1
4)	+	0	0	0	0					0	0	1	1	0	1	1	1
	0	0	1	1	0	1	1	1	↙								

Представим этот алгоритм в виде комбинации имеющихся в распоряжении операций. В примере имеется 4 ЧП, после добавления каждого из которых менялась СЧП, окончательное значение которой и является итоговым произведением. Для реализации этого уже нужно задействовать минимум 3 регистра: множимое, множитель, СЧП. Умножение будет представлять из себя цикл добавления ЧП к СЧП в зависимости от состояния анализируемого

на данный момент бита множителя. Каждое ЧП представляет из себя либо 0, либо сдвинутое множимое. Таким образом на каждой итерации цикла будет происходить сдвиг множимого на 1 бит влево. Так как при единичном значении младшего бита множителя ЧП представляет из себя изначальное значение множимого, то сдвиг множимого нужно разместить в конце цикла, чтобы не получить по итогу сдвинутое произведение. В начале цикла уместно расположить анализ младшего бита множителя, так как это определяет дальнейший план действий (добавление множимого или пропуск операции сложения). Один из основных способов проверки значения младшего бита - накладывание маски 0001 на регистр с множителем (выполнение побитовой операции конъюнкции соответствующих битов множителя), что сохранит значение младшего бита, но обнулит остальные. Таким образом после выполнения конъюнкции, в случае нулевого младшего бита, все биты будут равны 0, что приведёт к установке единицы в флаг Z. Иначе Z=0. Результат конъюнкции не нужно сохранять в регистре, свою роль эта операция уже выполнила. Далее в зависимости от флага Z нужно либо добавлять ЧП к СЧП, либо пропустить это сложение. Чтобы на следующей итерации можно было использовать эту же маску, но уже для анализа следующего бита множителя, множитель нужно сдвинуть на 1 бит вправо. Максимальное количество итераций для полного выполнения операции умножения равно количеству разрядов множителя. Но для некоторых входных чисел эта операция может выполняться быстрее. Когда анализируется бит множителя, равный нулю, СЧП не меняется. Если были проанализированы все разряды множителя, равные единице, то дальше цикл можно не повторять, так как итоговый результат от этого не изменятся. Поэтому условием выхода из цикла может быть равенство нулю (Z=1) множителя на какой-то из итераций. По итогу этот алгоритм умножения выглядит следующим образом (данный алгоритм называется алгоритмом умножения с анализом младшего бита множителя и сдвигом множимого):

1) Накладывание маски 0001 на множитель. Если результат равен нулю, то переход на действие 3. Иначе выполнение по порядку.

2) Добавление к СЧП множимого.

3) Сдвиг множителя на 1 бит вправо. Если сформирован признак равенства нулю, то выход из цикла. Иначе выполнение по порядку.

4) Сдвиг множимого на 1 бит влево. Безусловный переход к действию 1.

Теперь рассмотрим похожий алгоритм. Алгоритм умножения с анализом старшего бита множителя и сдвигом множимого. Рассмотрим его на том же примере умножения 11 на 6.

			1	0	1	1	>>									
ЧП:			0	1	0	1	<<				СЧП:					
1)	+	0	0	0	0				0	0	0	0	0	0	0	0
2)	+		1	0	1	1			0	0	1	0	1	1	0	0
3)	+			0	0	0	0		0	0	1	0	1	1	0	0
4)	+				1	0	1	1	0	0	1	1	0	1	1	1
		0	0	1	1	0	1	1	1	↙						

Так как от перестановки мест слагаемых сумма не меняется, то можно спокойно складывать ЧП в обратную сторону. Если при анализе одних и тех же бит (пусть анализ и происходит в обратную сторону) ЧП получаются те же, то и СЧП по итогу будет тем же. В прошлом

алгоритме чем левее располагался анализируемый бит множителя, тем левее сдвигалось множимое. Так как в этом алгоритме анализ начинается с самого левого бита, то и множимое должно быть изначально сдвинуто влево на количество разрядов минус 1 (именно столько разрядов располагается до старшего разряда множителя). Далее, при анализе всё более правого бита множителя, всё больше будет сдвигаться вправо множимое. Предположим, что данный алгоритм применяется для умножения однобайтовых операндов в двухбайтовом регистре. Тогда изначальный сдвиг множимого до входа в цикл будет осуществлён на 7 бит влево. Для анализа старшего бита однобайтового множимого используется маска 0080_h. Итоговый алгоритм выглядит следующим образом:

1) Сдвиг множимого на 7 бит влево.

2) Накладывание маски 0080_h на множитель. Если результат равен нулю, то переход на действие 4. Иначе выполнение по порядку.

3) Добавление к СЧП множимого.

4) Сдвиг множителя на 1 бит влево. Если сформирован признак равенства нулю, то выход из цикла. Иначе выполнение по порядку.

5) Сдвиг множимого на 1 бит вправо. Безусловный переход к действию 2.

В предыдущих алгоритмах для соответствия ЧП анализируемым разрядам множителя сдвигалось множимое. Но вместо сдвига множимого на каждой итерации можно использовать сдвиг СЧП. Рассмотрим алгоритм умножения с анализом младшего бита множителя и сдвигом СЧП.

				1	0	1	1		
				0	1	0	1	>>	
1)	ЧП:			1	0	1	1		
СЧП:	+	0	1	0	1	1	0	0	0
2)	>>	0	0	1	0	1	1	0	0
	ЧП:			0	0	0	0		
СЧП:	+	0	0	1	0	1	1	0	0
3)	>>	0	0	0	1	0	1	1	0
	ЧП:			1	0	1	1		
СЧП:	+	0	1	1	0	1	1	1	0
4)	>>	0	0	1	1	0	1	1	1
	ЧП:			0	0	0	0		
СЧП:	+	0	0	1	1	0	1	1	1
		0	0	1	1	0	1	1	1

Здесь множимое изначально сдвинуто на количество разрядов множителя минус 1 влево. При анализе младшего бита множителя в старшую часть СЧП записывается ЧП, после чего на следующей итерации СЧП сдвинется вправо, передвинув влияние этого ЧП на более младший разряд. Таким образом, что раньше входит в СЧП, то оказывает большее влияние на младшие биты итогового произведения, так как чем дальше происходит анализ, тем в более младшие разряды сдвигается влияние более ранних ЧП. Так как влияние самых ранних ЧП должно оказаться в младших разрядах СЧП, то нельзя делать досрочного выхода из цикла после обнуления множителя, потому что не будет произведено достаточное количество сдвигов СЧП влево. Количество итераций равно количеству разрядов множителя. Итоговый алгоритм для

однобайтовых операндов и двухбайтового произведения выглядит следующим образом:

1) Сдвиг множимого на 7 бит влево. Настройка счётчика цикла на 8 итераций.

2) Сдвиг СЧП на 1 бит влево.

3) Накладывание маски 0001 на множитель. Если результат равен нулю, то переход на действие 5. Иначе выполнение по порядку.

4) Добавление к СЧП множимого.

5) Сдвиг множителя на 1 бит влево. Если счётчик цикла не обнулился, переход на действие 2. Иначе выполнение по порядку.

Похожим образом работает алгоритм умножения с анализом старшего бита множителя и сдвигом СЧП.

						1	0	1	1
					<<	0	1	0	1
1)	ЧП:					0	0	0	0
СЧП:	+	0	0	0	0	0	0	0	0
2)	<<	0	0	0	0	0	0	0	0
	ЧП:					1	0	1	1
СЧП:	+	0	0	0	0	1	0	1	1
3)	<<	0	0	0	1	0	1	1	0
	ЧП:					0	0	0	0
СЧП:	+	0	0	0	1	0	1	1	0
4)	<<	0	0	1	0	1	1	0	0
	ЧП:					1	0	1	1
СЧП:	+	0	0	1	1	0	1	1	1
		0	0	1	1	0	1	1	1

Чем более старший бит множителя анализируется, тем большее влияние полученное ЧП должно оказывать на старшие биты итогового результата. Так как СЧП сдвигается влево, то более ранние ЧП будут оказывать большее влияние на старшие биты итогового результата, поэтому изначальный сдвиг множимого не требуется. Итоговый алгоритм для однобайтовых операндов и двухбайтового произведения выглядит следующим образом:

1) Настройка счётчика цикла на 8 итераций.

2) Сдвиг СЧП на 1 бит вправо.

3) Накладывание маски 0080_h на множитель. Если результат равен нулю, то переход на действие 5. Иначе выполнение по порядку.

4) Добавление к СЧП множимого.

5) Сдвиг множителя на 1 бит вправо. Если счётчик цикла не обнулился, переход на действие 2. Иначе выполнение по порядку.

7.2 Алгоритмы делений целочисленного и в формате фиксированной точки

Рассмотрим 2 алгоритма деления целых беззнаковых чисел (деление дробных чисел со знаком представлено в пункте 7.3). Так как и входные, и выходные числа целые, то частным является только целая часть от деления. Для выполнения таких операций делимое может быть в 2 раза длиннее делителя и в 2 раза длиннее частного. Рассмотрим деление двоичных чисел в столбик. Для примера возьмём числа: 29 - делимое, 7 - делитель. Подбираем старшую часть

делимого, из которой можно вычесть делитель, не получив отрицательного числа. Вычитаем, записываем в частное 1. Больше нет частей делимого, из которых можно вычесть делитель. В частное справа приписываем столько нулей, сколько осталось свободных разрядов. Если среди них есть единицы, то они являются остатком.

0	0	0	1	1	1	0	1	|	0	1	1	1	
	-	0	1	1	1			|		1	0	0	(ост 1)
		0	0	0	0	0	1						

Для реализации деления в процессоре нужно написать похожий алгоритм. Для начала нужно придумать способ нахождения старшей части делимого, из которой можно вычесть делитель, не получив отрицательного числа. Так как делимое в 2 раза длиннее делителя, то делитель нужно сдвинуть на свою длину, чтобы начать анализировать делимое с самой его старшей части. Если после вычитания получилась отрицательное делимое, то к нему обратно добавляется делитель. Так как эта часть делимого не является той, из которой можно вычесть делитель, то частное не меняется. Если бы разность получилось неотрицательной, то к делимому не добавляется делитель, а к частному прибавляется 1. Чтобы дальше анализировать делимое, его надо сдвинуть влево на 1 бит. Так как значение делимого стало в 2 раза больше (результат сдвига влево на 1 бит), то и значения частного должно стать в 2 раза больше. То есть частное тоже надо сдвинуть на 1 бит влево. Тело цикла разработано, осталось определить количество его итераций. Делитель в результате всего деления должен быть вычтен из каждой версии делимого, получаемого при сдвиге влево, пока не будет вычтена самая младшая часть изначального делимого. Так как в первой итерации вычитание происходит из старшей половины делимого и в каждой итерации оно сдвигается на бит влево, то количество итераций для перебора всех битов равно половине количества разрядов делимого плюс 1, либо количество разрядов делителя плюс 1.

	Делимое:									/					Делитель
1)		-	0	1	1	1	0	0	0	0		0	0	0	0	<<
	< 0		**1**	0	1	0	1	1	0	1		0	0	0	0	
		+	0	1	1	1	0	0	0	0						
	<<		0	0	0	1	1	1	0	1		↓	↓	↓	↓	
			0	0	1	1	1	0	1	0						
2)		-	0	1	1	1	0	0	0	0		0	0	0	0	<<
	< 0		**1**	1	0	0	1	0	1	0		0	0	0	0	
		+	0	1	1	1	0	0	0	0						
	<<		0	0	1	1	1	0	1	0		↓	↓	↓	↓	
			0	1	1	1	0	1	0	0						
3)		-	0	1	1	1	0	0	0	0		0	0	0	0	<<
	<<		**0**	0	0	0	0	1	0	0		0	0	0	1	
			0	0	0	0	1	0	0	0		↓	↓	↓	↓	
4)		-	0	1	1	1	0	0	0	0		0	0	0	1	<<
	< 0		**1**	0	0	1	1	0	0	0		0	0	1	0	
		+	0	1	1	1	0	0	0	0						
	<<		0	0	0	0	1	0	0	0		↓	↓	↓	↓	
			0	0	0	1	0	0	0	0						
5)		-	0	1	1	1	0	0	0	0		0	0	1	0	<<
	< 0		**1**	0	0	0	0	0	0	0		**0**	**1**	**0**	**0**	Частное
		+	0	1	1	1	0	0	0	0						
	<<		0	0	0	1	0	0	0	0						
			0	**0**	**0**	**1**	0	0	0	0	0					
	Остаток		↑	↑	↑	↑										

Данный алгоритм называется алгоритмом деления с восстановлением остатка. Итоговый алгоритм выглядит так (n - количество разрядов делителя и частного):

1) Сдвиг делителя на n бит влево. Настройка счётчика цикла на количество итераций равное n+1. Если старший бит делителя равен 0, то сдвигать его на 1 бит влево и каждый раз добавлять 1 к счётчику итераций, пока старший бит делителя не станет равен 1.

2) Делимому присвоить разность делимого и делителя.

3) Сдвиг частного на 1 бит влево.

4) Если результат неотрицательный, то переход к действию 6. Иначе выполнение по порядку.

5) К делимому добавить делитель. Безусловный переход к действию 7.

6) К частному прибавить 1.

7) Сдвиг делимого на 1 бит влево. Если счётчик цикла не обнулился, то переход к действию 2. Иначе выполнение по порядку.

Алгоритм деления без восстановления остатка отличается от предыдущего лишь тем, что вычитание делителя происходит не всегда, а только если тестовое вычитание без присвоения дало неотрицательный результат. Итоговый алгоритм выглядит так (n - количество разрядов делителя и частного):

1) Сдвиг делителя на n бит влево. Настройка счётчика цикла на количество итераций равное n+1. Если старший бит делителя равен 0, то сдвигать его на 1 бит влево и каждый раз добавлять 1 к счётчику итераций, пока старший бит делителя не станет равен 1.

2) Сдвиг частного на 1 бит влево.

3) Из делимого без присвоения вычитается делитель. Если результат отрицательный, то переход к действию 6. Иначе выполнение по порядку.

4) К частному прибавить 1.

5) Делимому присвоить разность делимого и делителя.

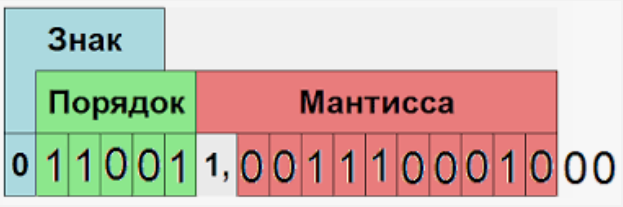
6) Сдвиг делимого на 1 бит влево. Если счётчик цикла не обнулился, то переход к действию 2. Иначе выполнение по порядку.

Для увеличения точности вычислений нужно увеличить количество итераций. С каждым новым сдвигом делимого влево, то есть после его искусственного умножения на 2, так же искусственно в 2 раза увеличивается частное. Поставив воображаемую запятую после выполнения основных итераций, можно выполнять дополнительные итерации, с каждой из них в 2 раза увеличивая точность результата. Увеличение точности числа ограничено лишь размером регистра, в котором оно хранится.

Для выполнения операций деления чисел со знаком, анализируются знаки двух исходных чисел. Наличие знака является единицей в старшем бите. Операцией сложения по модулю два между битами знаков делимого и делителя определяется знак частного. Отрицательные входные числа переводятся из дополнительного кода (ДК), выполняется деление по одному из описанных выше алгоритмов. После получения частного, оно переводится в ДК, если его знак равен 1. Аналогично для умножения целых чисел со знаком.

7.3 Действия над числами в формате плавающей точки

Общие сведения о представлении дробных чисел в формате плавающей точки (ПТ) представлены в пункте 1.3. Рассмотрим операции сложения, вычитания, умножения и деления на примере формата половинной точности. На рисунке наглядно показана структура этого формата.



Порядок в половинной точности пятибитный. Значит по формуле $2^{n-1}-1$, где n - количество разрядов, выделенное на порядок, можно вычислить значение, от которого идёт отсчёт смещения в половинной точности. $2^{5-1}-1 = 15$. Когда порядок равен 15, вся мантисса показывает полностью дробную часть числа (без учёта погрешности 10-битной мантиссы). Такое число будет иметь значение в таком диапазоне $1 \leq x < 2$. Для указания более больших/маленьких чисел к порядку нужно добавлять/вычитать смещение. Для увеличения диапазона в 2 раза к порядку прибавляется 1, для уменьшения вычитается.

Сложение и вычитание.

Для начала наложением маски выделяются мантиссы чисел, после чего к ним добавляется число 0400_h для восстановления единицы перед мантиссой. Для дальнейших вычислений нужно перевести числа в формат с фиксированной точкой. То есть оба числа должны представлять из себя мантиссы (с единицей перед ними), расположенные таким образом, чтобы их порядки совпадали. Для этого нужно сдвинуть меньшее число, подстраивая его под порядок большего числа. Например, для чисел 11,001 и 0,0101 мантиссы с единицами перед ними выглядят так: 11001 и 10100. Причём старшие единицы записаны у них в одном и том же разряде. Порядок второго числа на 3 меньше первого, поэтому второе число нужно сдвинуть на 3 бита вправо. Таким образом получатся модули чисел, приведённых к общему порядку. Для приведения к общему порядку вычисляется разность двух порядков. Если разность отрицательная, то осуществляются сдвиг первого числа вправо на 1 бит и инкремент разности порядков до тех пор, пока она не станет нулевой (когда отрицательное число в результате циклического сложения становится нулевым или положительным, в момент перехода устанавливается единица в флаг C, но флаг Z при этом никогда не станет равным единице, так как в момент перехода этому помешает флаг C, а дальше при продолжении сложения результат уже не будет нулевым). Если же разность порядков получилось положительной, то влево нежно сдвигать уже второе число с декрементом этой разности, пока она не станет нулевой. Изначально нулевую разность порядков можно обрабатывать как отдельный случай, либо переходить на тот же цикл, что и для положительной разности, но тогда в начале этого цикла должна быть отдельная проверка на 0 до декремента, по которой можно осуществить выход из цикла. Имея модули чисел, приведённых к общему порядку, остаётся только при необходимости перевести их в дополнительный код и выполнить операцию сложения. В алгоритме сложения в дополнительный код переводятся все числа, бит знака которых в ПТ равен 1. В алгоритме вычитания в дополнительный код переводится уменьшаемое, если у него бит знака равен 1, и вычитаемое, если у него бит знака равен 0. Если после сложения чисел

результат получился в дополнительном коде, то в бит знака результата в ПТ вписывается 1, а число переводится в прямой код. По итогу получился модуль итогового числа, который нужно преобразовать в мантиссу. На мантиссу отведено первый 10 бит перед которыми стоит старшая единица числа, которая в ПТ не вписывается. Так как после сложения старшая единица может оказаться не на десятом (считая от нуля) бите, то необходимо выполнить операцию сдвига, меняя порядок итогового числа в зависимости от количества и видов сдвигов. Старшая единица может оказаться либо в одиннадцатом, либо в любом более младшем бите. Эта единица не может быть в более старших битах, так как максимальные числа при сложении 1111…11, 1111…11 дадут максимальный результат 11111…10. Уменьшиться разряд старшей единицы может при вычитании чисел. Если старшая единица осталась в десятом бите, то итоговым порядком будет больший порядок двух исходных чисел, так как обе мантиссы были приведены именно к этому порядку. Если единица оказалась в одиннадцатом бите, то нужно сдвинуть число на 1 бит вправо с увеличением итогового порядка на 1. Если единица оказалась в десятом и более младших битах, то можно запустить цикл сдвига числа влево и проверки на наличие единицы в десятом бите. Так как единицу нужно получить именно в десятом бите и при входе в этот цикл она уже может там быть, то проверку на эту единицу с выходом из цикла в случае успеха следует расположить в начале тела цикла. При каждом сдвиге числа влево на 1 бит из итогового порядка будет вычитаться 1. Если же в результате сложения получился 0, то есть нет единиц, которые можно поместить в десятый бит, то есть циклы сдвигов будут выполняться бесконечно, то нужно перед циклом сдвига проверять результат на 0 (так как это может получиться только при сложении числа в прямом коде с равным ему по модулю числом в дополнительном коде, то флаг нуля загорится не в момент этого сложения, так как этому помешает флаг C, поэтому проверку можно сделать отдельным тактом, сохранив мантиссу в РЗУ и проверив её ещё раз, что позволяет сбросить флаг C и анализировать уже по флагу Z). Обработка нулей и бесконечностей в ПТ описана в конце этого пункта. Бит знака итогового числа определён при анализе результата сложения на отрицательность. Порядок вычислен в результате нормализации мантиссы (приведения её к виду, в котором в десятом бите располагается старшая единица). Мантисса после нормализации обрезается до десятого бита (то есть убирается старшая единица. Этого достаточно, чтобы собрать итоговый результат.

Умножение и деление.

Операции умножения и деления не требуют приведения мантисс к общему порядку. Знак определяется операцией исключающего или знаков двух входных чисел. Порядок при умножении определяется суммой смещений (они могут быть как положительными, так и отрицательными) и добавления этой суммы к точке отсчёта порядков (в половинной точности это число 15). Другой способ вычисления этого порядка - сумма двух исходных порядков и вычитание из них 15. При делении из смещения делимого вычитается смещение делителя и к результату добавляется 15. Другой способ вычисления этого порядка - разность двух исходных порядков и добавление к ним 15. К мантиссам приписывается старшая единица и по любому из алгоритмов умножения (пункт 7.1) и деления (пункт 7.2) вычисляется результат, который приводится к виду со старшей единицей на десятом бите, которая потом обрезается. Нормализация мантисс в операциях умножения и деления не приводит к изменению порядка. Итоговый порядок в этих операциях зависит только от исходных порядков, а не от результата.

При вычислении умножения произведение может выйти за пределы 16-битного регистра. От произведения нужно сохранить только 10 бит после старшей единицы. Для избегания потери этих бит, СЧП на каждой итерации можно сдвигать вправо на 1 бит, сохраняя старшую его часть в RGQ. Младшая часть произведения всё равно может потеряться из-за ограниченности длин мантисс в ПТ, но хотя бы сохранятся старшие 16 бит произведения, которые уже можно выдвигать из RGQ, подгоняя старшую единицу в десятый бит. При выполнении операции деления в счётчик количества итераций так же вписывается размер делимого плюс 1. При желании в этот счётчик можно вписать любое число в зависимости от желаемой точности результата. В целочисленном делении более строгие ограничения на этот счётчик, так как в формате целых чисел не должно быть дробной части. После нахождения частного в ПТ, число сдвигается до старшей единицы в десятом бите.

Обработка нулей и бесконечностей в ПТ.

Сложение равных по модулю, но разных по знаку чисел, умножение на 0, деление нуля или деление на бесконечность, получение очень маленьких (около нулевых) результатов могут приводить к необходимости представления нуля в ПТ. В ПТ есть 2 представления нуля: положительный 0 (знак, порядок и мантисса нулевые); отрицательный 0 (знак единичный, порядок и мантисса нулевые).

При сложении равных по модулю, но разных по знаку чисел, проверка на 0 происходит при получении результата этого сложение. В момент сдвига результата влево и вычитания из порядка единицы может получиться порядок, равный нулю. Это значит, что число меньше, чем то, которое может храниться в данном формате. В таком случае можно либо вывести минимальный ненулевое число (знак по ситуации, порядок равен 1, мантисса равна 0), либо вывести 0.

При умножении на 0, делении нуля или делении на бесконечность, проверка на 0 происходит перед вычислением порядка и мантиссы. Если хотя бы одно из входных чисел равно 0, то вычисляется знак и выводится 0 с соответствующим знаком. Если указанных выше особых случаев нет, но порядок произведения оказался неположительным, то можно либо вывести минимальное ненулевое число, либо 0.

Умножение на бесконечность, деление бесконечности или деление на 0, получение очень больших результатов могут приводить к необходимости представления бесконечности в ПТ. Бесконечность представлена порядком, полностью заполненным единицами, нулевой мантиссой и любым знаком в зависимости от вида бесконечности. Максимальное небесконечное число записывается порядком со всеми единицами минус 1 и мантиссой полностью заполненной нулями. Проверка на бесконечность аналогична проверке на 0.

У неопределённостей нет представления в ПТ, поэтому они обрабатываются как ошибки.

7.4 Алгоритмы нахождения остатков от деления

Для быстрого нахождения остатков от деления, числа не обязательно делить. Для этого есть особые алгоритмы. Для чисел записанных в системе счисления (СС) с основанием s есть следующие свойства:

При делении их на s-1 остаток от деления равен остатку от деления на s-1 суммы цифр числа. Рассмотрим пример для 9-ичной СС. Поделим на 8 число 731_9. Для этого найдём сумму цифр

1+3+7=11. Остаток от деления 11 на 8 равен **3**. Переведём исходное число в десятичную СС. $731_9 = 1 + 3*9 + 7*9*9 = 595_{10}$. При делении 595 на 8 получается целая часть 74, а остаток **3**.

При делении чисел на s+1 остаток от деления равен остатку от деления разности суммы цифр чётных разрядов (считая с нулевого) и суммы цифр нечётных разрядов. Если разность получилась отрицательной, то к ней нужно добавлять s+1, пока не получится положительный результат. Рассмотрим пример для того же числа, но уже делённого на 10. (1+7)-3 или 1-3+7=**5**. При делении этого же числа в 10-ичной СС получается целая часть 59, а остаток **5**.

Рассмотрим пример для деления на s+1, когда получается отрицательная разность. Найдём остаток от деления 16-ичного числа $F2E5_h$ на 17. 5-E+2-F = 5-14+2-15 = -22. Добавим 17 к -22 столько раз, сколько нужно, чтобы число получилось неотрицательным. -22+17*2=**12**. Переведём $F2E5_h$ в 10-ичную СС. $5 + 14*16 + 2*16*16 + 15*16*16*16 = 62181_{10}$. При делении этого числа на 17 получается целая часть 3657 и остаток **12**.

Чуть больше информации об этом алгоритме. Пусть есть число из n разрядов с основанием СС s. $a_n a_{n-1} \dots a_3 a_2 a_1 a_0 = a_n*s^n + a_{n-1}*s^{n-1} + \dots + a_3*s^3 + a_2*s^2 + a_1*s^1 + a_0*s^0$

Поделим каждое слагаемое на s-1 и найдём его остаток.

$a_0*\frac{s^0}{s-1} = \frac{a_0}{s-1}$, где целая часть 0, остаток от деления a_0

$a_1*\frac{s^1}{s-1} = a_1*\frac{s-1+1}{s-1} = a_1*(\frac{s-1}{s-1} + \frac{1}{s-1}) = a_1 + \frac{a_1}{s-1}$, где целая часть a_1, остаток от деления a_1

Из последней строки можно вывести правило $\frac{s}{s-1} = 1 + \frac{1}{s-1}$

$a_2*\frac{s^2}{s-1} = s*a_2*\frac{s}{s-1} = sa_2*(1 + \frac{1}{s-1}) = sa_2 + a_2*\frac{s}{s-1} = sa_2 + a_2*(1 + \frac{1}{s-1}) = sa_2 + a_2 + \frac{a_2}{s-1}$, где целая часть $sa_2 + a_2$, остаток от деления a_2

$a_3*\frac{s^3}{s-1} = s^2a_3*\frac{s}{s-1} = s^2a_3*(1 + \frac{1}{s-1}) = s^2a_3 + sa_3*\frac{s}{s-1} = s^2a_3 + sa_3*(1 + \frac{1}{s-1}) = s^2a_3 + sa_3 + a_3*\frac{s}{s-1} = s^2a_3 + sa_3 + a_3 + \frac{a_3}{s-1}$, где целая часть $s^2a_3 + sa_3 + a_3$, остаток от деления a_3

Несложно заметить закономерность, по которой с каждым новым слагаемым изменяется целая часть и остаток от деления. Аналогично предыдущим решениям получим результат для последнего слагаемого.

$a_n*\frac{s^n}{s-1} = s^{n-1}a_n + s^{n-2}a_n + \dots + s^3a_n + s^2a_n + sa_n + a_n + \frac{a_n}{s-1}$, где остаток от деления a_n

Таким образом, от каждого i-го слагаемого после деления на s-1 получается остаток a_i. Просуммировав остатки от каждого слагаемого, получим, что остаток от деления числа с основанием СС s на s-1 равен остатку от деления на s-1 суммы цифр этого числа.

Поделим каждое слагаемое на s+1 и найдём его остаток.

$a_0*\frac{s^0}{s+1} = \frac{a_0}{s+1}$, где целая часть 0, остаток от деления a_0

$a_1*\frac{s^1}{s+1} = a_1*\frac{s+1-1}{s+1} = a_1*(\frac{s+1}{s+1} + \frac{-1}{s+1}) = a_1 + \frac{-a_1}{s+1}$, где целая часть a_1, остаток от деления $-a_1$

Из последней строки можно вывести правило $\frac{s}{s+1} = 1 + \frac{-1}{s+1}$

$a_2*\frac{s^2}{s+1} = s*a_2*\frac{s}{s+1} = sa_2*(1 + \frac{-1}{s+1}) = sa_2 + a_2*\frac{-s}{s+1} = sa_2 - a_2*(1 + \frac{-1}{s+1}) = sa_2 - a_2 + \frac{a_2}{s+1}$, где целая часть $sa_2 - a_2$, остаток от деления a_2

$a_3*\frac{s^3}{s+1} = s^2a_3*\frac{s}{s+1} = s^2a_3*(1 + \frac{-1}{s+1}) = s^2a_3 + sa_3*\frac{-s}{s+1} = s^2a_3 - sa_3*(1 + \frac{-1}{s+1}) = s^2a_3 - sa_3 + a_3*\frac{s}{s+1} = s^2a_3 - sa_3 + a_3 + \frac{-a_3}{s+1}$, где целая часть $s^2a_3 - sa_3 + a_3$, остаток от деления $-a_3$

Несложно заметить закономерность, по которой с каждым новым слагаемым изменяется целая часть и остаток от деления. Аналогично предыдущим решениям получим результат для последнего слагаемого.

Для чётного n: $a_n*\frac{s^n}{s+1} = s^{n-1}a_n - s^{n-2}a_n + \ldots + s^3a_n - s^2a_n + sa_n - a_n + \frac{a_n}{s+1}$, где остаток от деления a_n

Для нечётного n: $a_n*\frac{s^n}{s+1} = s^{n-1}a_n - s^{n-2}a_n + \ldots - s^3a_n + s^2a_n - sa_n + a_n + \frac{-a_n}{s+1}$ где остаток от деления $-a_n$

Таким образом, от каждого i-го слагаемого после деления на s+1 получается остаток $(-1)^ia_i$ (при индексации с нуля). Просуммировав остатки от каждого слагаемого, получим, что остаток от деления числа с основанием СС s на s+1 равен остатку от деления на s+1 суммы цифр этого числа, где цифры с нечётным индексом (при индексации с нуля) взяты со знаком минус.

Рассмотрим более сложный случай - деление на s-k, где k - любое действительное число.

$a_0*\frac{s^0}{s-k} = \frac{a_0}{s-k}$, где целая часть 0, остаток от деления a_0

$a_1*\frac{s^1}{s-k} = a_1*\frac{s-k+k}{s-k} = a_1*(\frac{s-k}{s-k} + \frac{k}{s-k}) = a_1 + \frac{ka_1}{s-k}$, где целая часть a_1, остаток от деления ka_1

Из последней строки можно вывести правило $\frac{s}{s-k} = 1 + \frac{k}{s-k}$

$a_2*\frac{s^2}{s-k} = s*a_2*\frac{s}{s-k} = sa_2*(1 + \frac{k}{s-k}) = sa_2 + ka_2*\frac{s}{s-k} = sa_2 + ka_2*(1 + \frac{k}{s-k}) = sa_2 + ka_2 + \frac{k^2a_2}{s-k}$, где целая часть $sa_2 + ka_2$, остаток от деления k^2a_2

$a_3*\frac{s^3}{s-k} = s^2a_3*\frac{s}{s-k} = s^2a_3*(1 + \frac{k}{s-k}) = s^2a_3 + ksa_3*\frac{s}{s-k} = s^2a_3 + ksa_3*(1 + \frac{k}{s-k}) = s^2a_3 + ksa_3 + k^2a_3*\frac{s}{s-k} = s^2a_3 + ksa_3 + k^2a_3 + \frac{k^3a_3}{s-k}$, где целая часть $s^2a_3 + ksa_3 + k^2a_3$, остаток от деления k^3a_3

Несложно заметить закономерность, по которой с каждым новым слагаемым изменяется целая часть и остаток от деления. Аналогично предыдущим решениям получим результат для последнего слагаемого.

$a_n*\frac{s^n}{s-k} = s^{n-1}a_n + ks^{n-2}a_n + \ldots + k^{n-4}s^3a_n + k^{n-3}s^2a_n + k^{n-2}sa_n + k^{n-1}a_n + \frac{k^na_n}{s-k}$, где остаток от деления k^na_n

Таким образом, от каждого i-го слагаемого после деления на s-k получается остаток k^ia_i (при индексации от нуля). Просуммировав остатки от каждого слагаемого, получим, что остаток от деления числа с основанием СС s на s-k равен остатку от деления на s-k суммы цифр этого числа, каждая из которых умножена на k в степени номера этой цифры (от нуля).

7.5 Алгоритм перевода из двоично-десятичного в двоичный код

В двоично-десятичном коде число, записанное в 16-ичной системе счисления, воспринимается как десятичное. Пусть имеется двоично-десятичное число 4567. Это 16-ичное число (7 + 6*16 + 5*16*16 + 4*16*16*16), которое воспринимается как десятичное (7 + 6*10 + 5*10*10 + 4*10*10*10). Чтобы и процессор, работающий на двоичном коде, так же

воспринимал это число, нужно выполнить несколько действий. Каждый разряд двоично-десятичного кода занимает 4 бита. Для соответствия ожидаемого и реального значения нужно перевести число по формуле: ((a*10+b)*10+c)*10+d, где a - старшая тетрада, d - младшая. Оптимальный способ замены операции умножения на 10 выглядит так: (x>>2+x)>>1. Перевод в двоичный код можно реализовать циклом, в котором на каждой итерации выделяется старшая тетрада двоично-десятичного кода, умножается на 10, к ней добавляется следующая тетрада.

1) Загрузка в RACT количество итераций цикла минус 1 (для 4-тетрадного числа нужно 4 итерации в этом алгоритме).

2) Выделение старшей тетрады (она должна оказаться в отдельном регистре в четырёх младших разрядах).

3) Сдвиг исходного числа на 4 бита влево.

4) Умножение регистра, в котором будет хранится итоговый результат, на 10.

5) Добавление тетрады, полученной в действии 2, к регистру промежуточного результата.

6) Декремент RACT, переход к действию 2.

Приложение 1. Все значения полей микрокоманд

Пункт 3.1

Поля A/B	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
РОН	AX	CX	DX	BX	SP	BP	SI	DI	CS	SS	DS	ES	IP	PSW	RGK	RW

Пункт 6.5

Поле MA/MB	**0**	**1**	**2**	**3**
Источник адресов регистров РЗУ	Поле RGMK A или B	Поле RGK reg1	Поле RGK reg2	Поле RGK r/m

Раздел 4. Поле MEM выполняется после действия в АЛУ, но перед записью в РЗУ.

Поле MEM	**4**	**5**	**6**	**7**
	Чтение байта	Чтение слова	Запись байта	Запись слова

Пункт 3.2. При выборе операнда RGR берётся то значение, которое было там на начало такта.

Поле SRC	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Операнд R	0000	RGA	RGA	RGA	RGA*2	CONST	CONST	CONST
Операнд S	0000	RGB	RGQ	RGR	RGB	RGB	RGR	RGQ

Пункт 3.4

Поле SH	**Операция**
0	Без сдвига
1	АС АЛУ вправо
2	ЛС АЛУ вправо
3	АС АЛУ, RGQ вправо
4	ЛС АЛУ, RGQ вправо
6	RGQ⇐ALU (загрузка из ALU в RGQ без изменения значения АЛУ)
8	ЛС АЛУ влево
A	ЛС АЛУ, RGQ влево
E	Расширение знака (дублирование старшего бита младшего байта АЛУ на каждый бит старшего байта, что позволяет расширить однобайтовое число со знаком на всё слово)

Пункт 3.4

Поле N равно количеству разрядов от 0 до f, на которые осуществляется сдвиг.

Пункт 3.2

Поле ALU	**Операция АЛУ**	**Флажки**				**Поле ALU**	**Операция АЛУ**	**Флажки**			
		N	Z	V	C			N	Z	V	C
0	На всех выходах «0»	0	1	0	0	8	Умножение на 2 бита	+	+	+	+
1	S - R - 1 + C0	+	+	+	+	9	R & S	+	+	0	0
2	R - S - 1 + C0	+	+	+	+	A	R & $\overline{S}$	+	+	0	0
3	R + S + C0	+	+	+	+	B	$\overline{R \& S}$	+	+	0	0
4	S + C0	+	+	+	+	C	R V S	+	+	0	0
5	$\overline{S}$ + C0	+	+	+	+	D	$\overline{R \vee S}$	+	+	0	0
6	R + C0	+	+	+	+	E	R ⊕ S	+	+	0	0
7	$\overline{R}$ + C0	+	+	+	+	F	$\overline{R \oplus S}$	+	+	0	0

Пункт 3.2

Поле CCX равно 1 при необходимости заменить константу C0 на 1. При CCX=0, C0=0.

Пункт 3.3

Поле F равно 1 при необходимости сохранить флажки из RFI (значения флажков на текущем такте) в RFD (регистр длительного хранения флажков).

Пункт 3.5. Поле DST выполняется в последнюю очередь.

Поле DST	**0**	**1**	**2**	**3**	**4**
Источник	Без записи	RGR	RGRL	RGRH	SDA
Приемник	Без записи	РЗУ	РЗУН	РЗУL	РЗУ

Раздел 4. Поле WM выполняется после действия в АЛУ, но перед действием над ОП.

Поле WM	**0**	**1**	**2**	**3**
Источник	Без записи	SDA	SDA	RGB
Приемник	Без записи	RGW	ARAM	ARAM

Пункт 5.2

Поле JFI состоит из трёх бит, расположенных в той же последовательности: J (jump) - безусловный переход при J=1, условный переход при J=0; F (flag) - бит, ответственный за регистр, из которого брать флаги. RFI (значения флажков на текущем такте) при F=0, RFD (регистр длительного хранения флажков) при F=1; I (inversion) - инверсия условия при I=1, сохранение условия неизменным при I=0.

Пункт 5.2

Поле CC	**Вид перехода**	**Условие перехода**
0	JP, JNP*	P=1
1	JZ, JNZ*	Z=1
2	JS, JNS*	N=1
3	JO, JNO*	V=1
4	JC, JNC*	C=1
5	JL, JNL*	$N \oplus V=1$
6	JLE, JNLE*	$Z \vee (N \oplus V)=1$
7	JBE, JNBE*	$C \vee Z=1$

Пункты 5.2-5.4, 6.5

Поле CHA	**Мнемоника**	**X=0**		**X=1**		**RACT**
		Y	**Стек**	**Y**	**Стек**	
0	JZ	0	Очистка	0	Очистка	Хранение
1	CJS	CMK	Хранение	CONST	Загрузка	Хранение
2	JMAP	PA	Хранение	PA	Хранение	Хранение
3	CJP	CMK	Хранение	CONST	Хранение	Хранение
4	RPCT	CMK	Хранение	CONST	Хранение	Декремент
5	CRTN	CMK	Хранение	Стек	Выгрузка	Хранение
6	LDCT	CMK	Хранение	CMK	Хранение	Загрузка
7	CONT	CMK	Хранение	CMK	Хранение	Хранение

Пункты 3.2, 5.3-5.4

Поле CONST может принимать любое 16-битное значение. Его назначение зависит от других полей микрокоманды. Число из CONST может быть операндом для АЛУ, адресом перехода или значением, загружаемым в RACT.