

```
.TAIL 1 ''
.TAIL 0 ''
.HEAD 1 ''
.HEAD 0 ''
```

- Сибирское отделение АН СССР -

- Институт систем информатики -

[illegible]

Новосибирск - 90

```
.PAGE
.TAIL 1 '@A%62d@a'
```

СОДЕРЖАНИЕ

Введение	4
Часть I. Виртуальная Модуля-2 машина	6
1. Составные части виртуальной машины	6
1.1. Процессор	6
1.2. Арифметический стек	7
1.3. Процесс и его дескриптор	7
1.4. Память	8
2. Структуры в памяти	8
2.1. Модуль, готовый к исполнению	8
2.2. Сегмент кода и процедурная таблица	9
2.3. Область глобальных данных	9
2.4. Строковый пул	10
2.5. Область связей	10
2.6. Процедурный стек	11
2.6.1. Область локальных данных.	12
2.7. Еще раз о дескрипторе процесса	13
3. М-код	13
3.1. Способы адресации.	13
3.2. Непосредственные операнды.	14
3.3. Функции и назначение команд.	15
4. Прерывания	16
4.1. Типы прерываний	16
4.2. Вектора прерываний	17
4.3. Пространство векторов прерываний	17
4.4. Маска прерываний	17
Часть II. Интерпретатор М-кода	18
Часть III. Описание системы команд	32
Часть IV. Иллюстрации к архитектуре процессоров	87
1. Операторы	88
1.1. Присваивание	88
1.2. Доступ к глобальным переменным	88
1.3. Доступ к внешним переменным	89
1.4. Условный оператор (IF)	89
1.5. Оператор цикла (LOOP)	89
1.6. Оператор цикла (REPEAT)	90
1.7. Оператор цикла (FOR)	91
1.8. Оператор выбора (CASE)	92
2. Процедуры	93
2.1. Описание и вызов примитивной процедуры	93
2.2. Работа с локалами процедуры	94
2.3. Вложенные процедуры	94
2.4. Вызов внешней процедуры	95
2.5. Размещение мультимножеств	96
2.6. Работа с процедурными значениями	97
2.7. Передача параметров	98
2.8. Вызов функции (на непустом стеке)	100
3. Выражения	101
3.1. Индексация словных массивов	101
3.2. Индексация байтовых массивов	101
3.3. Индексация байтовых массивов с контролем границ	102
3.4. Проверка принадлежности диапазону	103

3.5. Работа с объектами типа BITSET.	103
3.6. Команды ANDJP и ORJP	103
Индекс-таблица системы команд.	104
.PAGE	
.HEAD 1 '@УАРХИТЕКТУРА	
.HEAD 0 '@УАРХИТЕКТУРА	

ВВЕДЕНИЕ@u'
ВВЕДЕНИЕ@u'

ВВЕДЕНИЕ

Архитектура процессоров семейства КРОНОС ориентирована на поддержку языков высокого уровня (Си, Модула-2, Паскаль, Оккам), что дает возможность реализовать новейшие концепции в области использования ЭВМ. Наличие 32-разрядного машинного слова позволяет использовать процессоры семейства для решения вычислительных задач. Широкое адресное пространство (до 2 миллиардов слов) дает возможность создания виртуальной памяти для объектно-ориентированных моделей вычислений и тем самым поддерживает разработку систем искусственного интеллекта. Наличие механизма прерываний по событиям, по синхронизации процессов, а также компактность кода программ позволяет с уверенностью утверждать, что процессоры семейства КРОНОС могут успешно применяться в системах реального времени.

Любой процессор семейства может быть использован как в составе отдельной ЭВМ, так и в мультипроцессорных комплексах.

В настоящее время семейство включает три разработки: 2.2, 2.5, 2.6. У всех процессоров одна система команд; они полностью совместимы программно и различаются лишь по внутреннему функциональному устройству, быстродействию и конструктивному исполнению.

Логику функционирования всех блоков процессора реализует блок микропрограммного управления. Две шины данных объединяют арифметико-логическое устройство, блок регистров, быстрый аппаратный стек на 7 слов, устройства выборки команд и ввода/вывода.

Двухшинная внутренняя структура процессоров позволяет выполнять бинарные операции на стеке (сложение, вычитание, логические И, ИЛИ и т.д.) за один такт. Таким образом, за один такт исполняется большинство команд, что отвечает основным идеям RISC-архитектуры. Микропрограммное управление упрощает устройство процессоров и дает возможность реализовать сложные команды типа вызова процедуры.

Все узлы процессоров выполнены на советских ТТЛ и ТТЛШ микросхемах широкого применения серий 155, 531, 1802, 1804, 589, 556.

Процессор	Кронос 2.2	Кронос 2.5	Кронос 2.6
Конструктив	Электроника-60	Intel	Евромеханика
Количество плат	1	2	2@10f2)@4f

Шина	Q-bus 22	Multibus-1	локальная@10f3)@4f
Ширина микрокоманды, бит	56	64	64
Объем микропрограмм, Кслов	2	2	4
Прямо адресуемая память	4 Мбайт	2,5 Мбайт@10f1)@4f	8 Гбайт
Тактовая частота, мГц	4	3	3
Число простых операций над стеком, млн/сек	0,6	1	1,5

@10f1)@4f Существенным отличием Кронос 2.5 является наличие локальной памяти объемом 0,5-2 Мбайт - в зависимости от применяемых микросхем. Остальная память - на шине Multibus-1 (до 1 Мбайт).

@10f2)@4fВ минимальный комплект входят: плата обрабатывающего тракта (АЛУ, стек, регистры), плата микропрограммного управления, плата локальной памяти (0.5-2 Мбайт), плата адаптера шины ввода/вывода.

@10f3)@4fВсе устройства объединены локальной синхронной 32-разрядной шиной. Сам процессор не зависит от конкретной шины ввода/вывода. Настройка на конкретную шину производится с помощью соответствующего адаптера. К локальной шине могут быть добавлены платы памяти, адаптера межпроцессорной связи, контроллера к локальной сети и накопителя на магнитных дисках, плата памяти кода (при разделении плат кода и данных), bitmap-дисплея и т.д..

.PAGE

.HEAD 1 '@УАРХИТЕКТУРА

ВИРТУАЛЬНАЯ МОДУЛА-2 МАШИНА@u'

.HEAD 0 '@УАРХИТЕКТУРА

ВИРТУАЛЬНАЯ МОДУЛА-2 МАШИНА@u'

ЧАСТЬ I. ВИРТУАЛЬНАЯ МОДУЛА-2 МАШИНА

Виртуальная Модула-2 машина (BM2M) и ее программный и аппаратный интерпретаторы обеспечивают процесс исполнения программ. В данном разделе будет охарактеризован М-код, пояснена его интерпретация и проиллюстрированы отдельные команды.

Механизм ввода/вывода существенно зависит от конфигурации конкретного оборудования и здесь не рассматривается.

Основные отличия BM2M от традиционных машин:

1) вычисление выражений на быстром стеке небольшой аппаратно фиксированной глубины. Освобождение этого стека

(копирование в память) при вызове процедур-функций;

2) разделение кода и области данных любого процесса, следствием чего является повторно-входимость всех программ и даже их отдельных частей (модулей);

3) отказ от абсолютной адресации в сегменте кода. Наличие таблицы смещений начал процедур упрощает вызовы процедур;

4) развитые виды адресации отражают понятия современных языков программирования. Имеется адресация локальных, глобальных, внешних объектов и объектов статически вложенных процедур;

5) специальные команды упрощают реализацию циклов, вызовов, выбирающих операторов и других сложных конструкций;

6) таблица раздельно загруженных модулей позволяет организовать динамическое связывание и загрузку программ;

7) имеются команды для работы с мультизначениями.

Для понимания этого раздела требуется знакомство с языком программирования Модуля-2. В дальнейшем считается, что читатель знаком со следующими понятиями Модуля-2:

@АПРОГРАММА

МОДУЛЬ

ИМПОРТ-ЭКСПОРТ ОБ'ЕКТОВ

ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ

ПРОЦЕДУРА

ЛОКАЛЬНАЯ ПРОЦЕДУРА

ЛОКАЛЬНЫЕ ПЕРЕМЕННЫЕ@a

1. Составные части виртуальной машины

ВМ2М состоит из процессора, стека для хранения обрабатываемых данных и вычисления выражений (арифметического стека), дескриптора исполняемого процесса и памяти.

1.1. Процессор

@АПроцессор@a ВМ2М - устройство для интерпретации инструкций управления (@АМ-кода@a).

1.2. Арифметический стек

@АА-стек@a (арифметический стек, стек выражений, expression stack) - быстрый стек небольшой, аппаратно фиксированной глубины, шириной в одно слово (здесь и далее: ширина машинного слова 32 бита), над которым определены операции помещения слова на стек (Push) и снятия слова с верхушки стека со счеркиванием (Pop). Переполнение и исчерпание стека отслеживаются аппаратно с возбуждением соответствующего прерывания.

```
|-----| \
|-----|  |
```

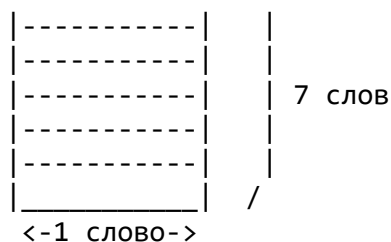


Рис.1. А-стек (стек выражений)

1.3. Процесс и его дескриптор

В каждый момент времени ВМ2М выполняет некоторый @Апроцесс@a. Процесс - это последовательность операций при выполнении программы и данные, используемые этими операциями.

@АДескриптор процесса@a - 8 слов в памяти, содержащих указатели на специальные информационные структуры, связанные с процессом. Эти указатели в совокупности определяют весь контекст процесса, т.е. всю информацию, необходимую для его исполнения виртуальной машиной.



Рис.2. Дескриптор процесса

В терминах Модуль-2 дескриптор процесса представляет собой запись:

```
TYPE Process_Descriptor = RECORD
    G, L, PC, M, S, H, T, RFE: WORD;
END;
```

Каждое из полей дескриптора имеет специальное назначение, описанное ниже. В дальнейшем поля дескриптора процесса для удобства изложения называются регистрами процессора (например, "Process_Descriptor.G" будем называть "G-регистр").

Примечание. Существующие реализации ВМ2М (процессоры семейства КРОНОС) действительно имеют специализированные регистры, во время работы содержащие копии соответствующих полей дескриптора текущего процесса. При переключении процессов текущее содержимое этих регистров копируется в память по адресу, содержащемуся в Р-регистре.

1.4. Память

@АПамять@a в ВМ2М представляется линейной последовательностью слов размером 32 бита. Каждому слову сопоставлен тридцатиоднобитный номер - адрес слова. Адресуется только все слово целиком. Адреса бывают только положительные.

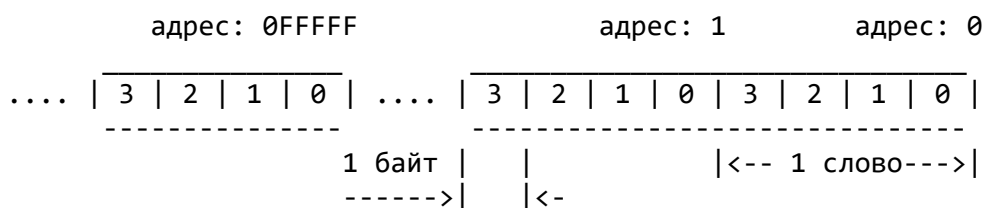


Рис.3. Память BM2M.

Обратите внимание, что на всех рисунках по традиции младшие адреса справа, а старшие - слева. Поэтому все изображенные ниже структуры располагаются справа налево.

2. Структуры в памяти

Во время работы BM2M в памяти размещены некоторые структуры, устройство и взаимодействие которых описаны в этом разделе.

2.1. Модуль, готовый к исполнению

Каждый из скомпилированных и загруженных в память модулей состоит из сегмента кода, области глобальных данных, области структурных констант, то есть строк, массивов, записей (строковый пул), области связей с внешними модулями (при наличии в модуле импорта объектов).

2.2. Сегмент кода и процедурная таблица

@АСегмент@a кода занимает непрерывный кусок памяти и имеет следующую структуру. В начале сегмента находится процедурная таблица, занимающая до 256 слов. Процедурная таблица устанавливает соответствие между номером процедуры и байтовым смещением от начала сегмента до начала кода соответствующей процедуры (в нулевом слове - смещение 0-й процедуры, в n-м - смещение n-й). Внутри модуля процедура идентифицируется своим номером в процедурной таблице модуля. При исполнении любой процедуры модуля указатель на начало сегмента кода содержится в F-регистре процессора. В другом регистре - PC (Programm Counter)- содержится байтовое смещение от начала сегмента кода до байта, содержащего следующую за исполняемой в данный момент команду.

сюда указывает F-регистр

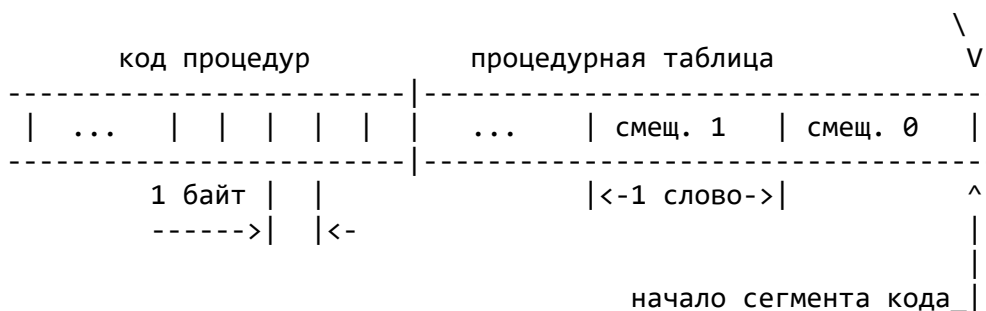


Рис.4. Сегмент кода

2.3. Область глобальных данных

@АОбласть глобальных данных@ одного модуля занимает непрерывный кусок памяти и представляется последовательностью слов. При этом однословные данные хранятся в них непосредственно, а структурные данные могут быть представлены в виде ссылки на специально отведенные для них области памяти.

На начало области глобальных данных текущего модуля указывает G-регистр процессора. Таким образом, доступ к глобальным переменным модуля организуется индексно по содержимому G-регистра, т.е. глобальное слово с номером Z находится в ячейке памяти с адресом $[G]+Z$ (здесь и далее запись $[REG]$ обозначает содержимое регистра "REG"). Первые два слова области глобальных данных заняты специальной информацией: в нулевом слове области содержится указатель на начало сегмента кода данного модуля (копия F-регистра), в 1-м слове - указатель на начало строкового пула.

.PAGE

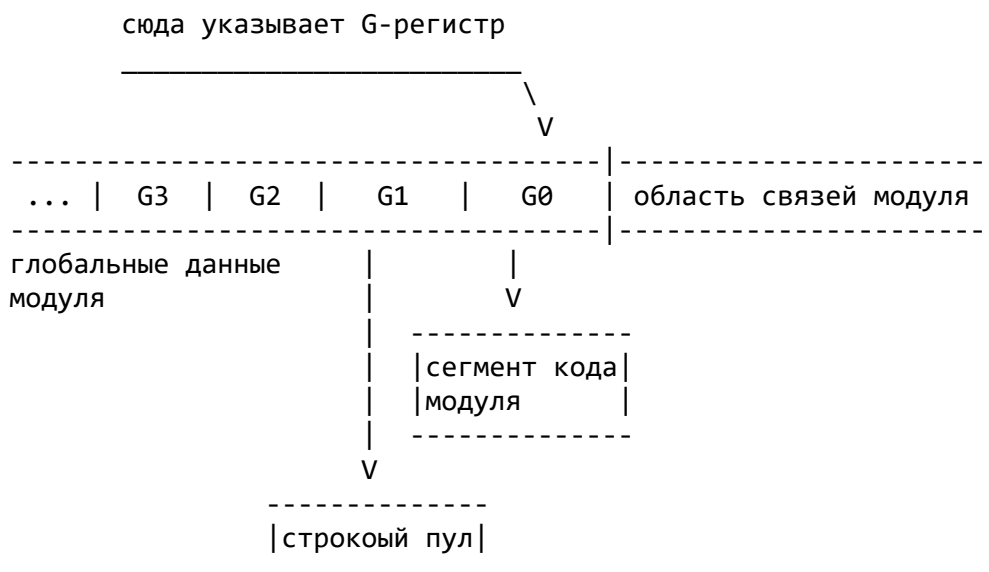


Рис.5. Область глобальных данных модуля. Стрелками изображены указатели.

	V	\emptyset	\dots	i
область глобальных данных модуля	адрес ссылки на ОГД модуля \emptyset	\dots	адрес ссылки на ОГД модуля i	
			V	V
		\dots	адрес начала	

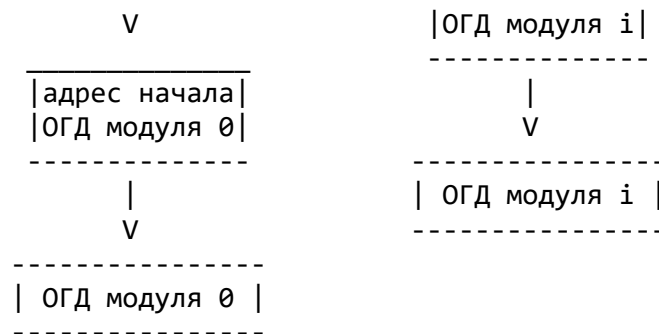


Рис.7. Область связей модуля

2.6. Процедурный стек

Процедурный стек (@AP-стек@a) занимает непрерывный кусок памяти и используется для размещения @Алокальных данных процедур@a, организации вызовов и возвратов из процедур и размечен тремя регистрами процессора. S-регистр указывает на вершину Р-стека, т.е. первое свободное на Р-стеке слово. Н-регистр указывает на последнее слово области памяти, отведенной под Р-стек (предел увеличения [S]). L-регистр указывает на начало области локальных данных процедуры, исполняемой в данный момент. Перекрывание регистров S и Н (т.е. ситуация, когда [S] >= [Н]) называется переполнением Р-стека и отслеживается аппаратно с возбуждением соответствующего прерывания.

.PAGE

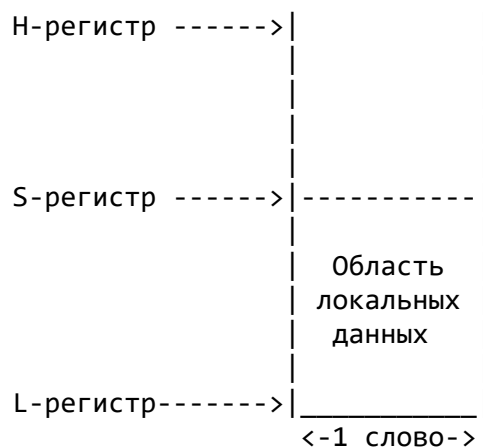


Рис.8. Р-стек (процедурный стек) и его разметка. Стрелками изображены указатели.

2.6.1. Область локальных данных

Область @Алокальных данных@a текущей процедуры располагается

на Р-стеке и представляет собой последовательность слов, содержащих локальные переменные непосредственно или ссылки на начала участков памяти, отведенных для переменных структурных типов. Доступ к локальным данным осуществляется индексно по L-регистру, т.е. локальное слово с номером i находится по адресу $[L]+i$ в памяти. Специальные команды облегчают доступ к локальным словам с номерами 4..255.

Первые 4 локальных слова с номерами 0..3 заняты специальной информацией: в нулевом локальном слове содержится указатель на начало области локальных данных объемлющей процедуры (процедуры, внутри которой текущая описана как локальная процедура), или указатель на область глобальных данных вызывающего модуля (если процедура была вызвана из внешнего модуля) - так называемая @Астатическая цепочка@a. 1-е локальное слово указывает на начало области локальных данных процедуры, из которой произошел вызов текущей (@Адинамическая цепочка@a). 2-е локальное слово содержит значение PC возврата. 3-е локальное слово оставлено для возможного дальнейшего использования (RFE - Reserved for Future Extension).

.PAGE

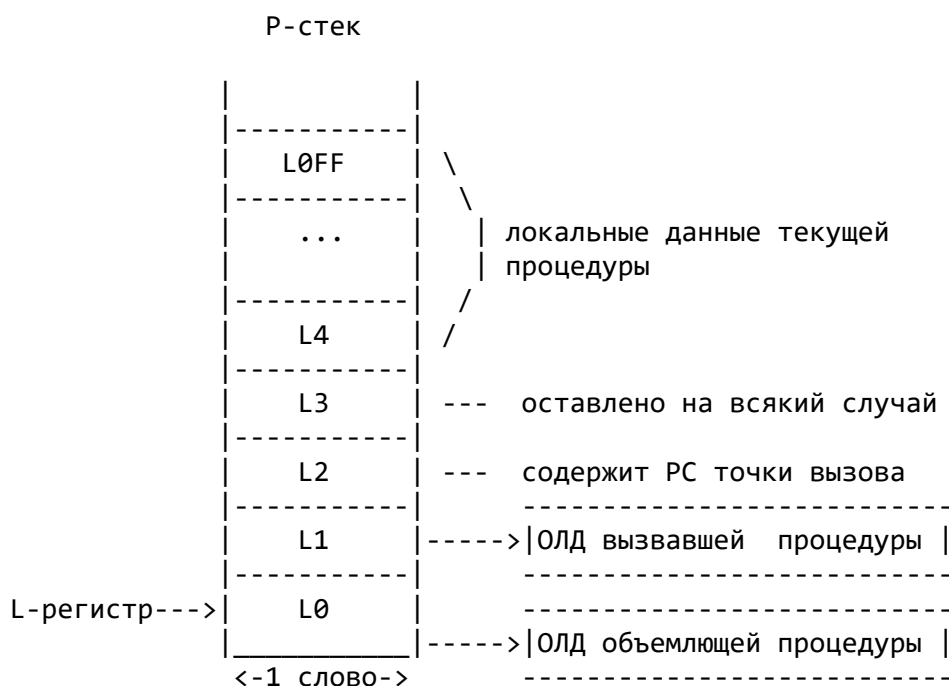


Рис.9. Область локальных данных исполняемой процедуры. Стрелками изображены указатели.

2.7. Еще раз о дескрипторе процесса

Итак, суммируем информацию о дескрипторе процесса. Он занимает непрерывный кусок памяти размером восемь слов, содержащих следующее:

- 1) G - адрес области глобальных данных;

- 2) L - адрес локальных данных;
- 3) PC - байтовое смещение команды, подлежащей исполнению;
- 4) M - маска прерываний;
- 5) S - адрес вершины процедурного стека;
- 6) H - адрес границы процедурного стека;
- 7) T - причина прерывания;
- 8) RFE - для дальнейших расширений.

3. M-код

Код VM2M (@AM-код@a) представляет собой поток байтов, т.е. код команды всегда занимает один байт. Команды VM2M не содержат в себе адресных полей, но по коду команды всегда точно известно, где находятся операнды.

3.1. Способы адресации

Различные команды адресуют данные несколькими способами, которые иллюстрирует таблица.

Способ адресации	Как вычисляется адрес
локальный	$[L] + N$
глобальный	$[G] + N$
внешний	$DFT[M]^{\wedge} + N$
промежуточный	$[L]^{\wedge} + N$ или $[[L]^{\wedge}]^{\wedge} + N \dots$
косвенный	$[POP()]^{\wedge} + N,$

где M - номер внешнего модуля, N - индекс объекта, символ " \wedge " обозначает операцию разыменования указателя.

Промежуточный способ адресации предназначен для работы с объектами статически объемлющей процедуры.

3.2. Непосредственные операнды

Команды могут содержать от 4 бит до 4 байт непосредственных операндов. Непосредственные операнды - операнды, следующие в сегменте кода непосредственно за кодом команды, причем по коду операции всегда однозначно известно, сколько байт непосредственных операндов следует за ним. При извлечении данных из потока команд и самих команд счетчик команд (PC) увеличивается на 1..5 байта соответственно.

M-код имеет четыре типа непосредственных операндов:

3.2.1. Полубайтовый операнд: последние 4 бита команды интерпретируются как непосредственный операнд. Так, например, имеются команды

LI0, LI1, ..., LI15:

(* Load Immediate - загрузка непосредственная с кодами 0,1, ..., 15 и семантикой: *)

```

    Push(IR MOD 16)
(* где IR - регистр кода команды *)

    SLW4,...,SLW15:
(* Store Local Word *)
    MEM[L + IR MOD 16] := Pop();
(* MEM - физическая или логическая память *)
Таким образом, тело процедуры

```

```

PROCEDURE p; CONST c=2;
    VAR i: INTEGER;
BEGIN i:=c;
END p;

```

реализуется в две следующие однобайтовые команды:

```

LI2
SLW4.

```

3.2.2. Байтовый операнд:

Например,

```
LIB:
```

```

(* Load Immediate Byte *)
    Push(Next());
(* Next() - операция выборки одного байта из кода *)

```

При замене описания константы в предыдущем примере на следующее:

```
c=153
```

породится код

```

LIB
153
SLW4

```

3.2.3. Двухбайтовый операнд - представленный следующей за кодом парой байтов.

3.2.4. Словный операнд - представленный следующей за кодом четверкой байтов.

3.2.5. Существуют команды, имеющие несколько операндов различной длины.

3.3. Функции и назначение команд

Команды ВМ2М условно подразделяются на следующие группы: арифметико-логические, команды организации управления, команды для работы с А-стеком, вспомогательные.

Команды для работы со стеком содержат инструкции загрузки на стек и записи в память вершины стека.

Все арифметико-логические операции работают над одним или двумя верхними элементами стека и помещают результат на стек.

Команды для организации управления представлены обычными командами переходов, условных и безусловных, специальными командами для организации операторов FOR и CASE, набором

инструкций вызова и возврата из процедуры. Сюда же относятся команды TRAP - программного прерывания и TRANSFER - переключения процессов, аналог переключения сопрограмм, дающий мощное средство для организации мультипрограммной работы и синхронизации.

BM2M имеет набор вспомогательных команд, облегчающих обработку мультизначений, параметров процедуры, команды ввода/вывода и др..

.PAGE

4. Прерывания

Прерывание - прекращение выполнения текущей команды или текущей последовательности команд для обработки некоторого события; событие может быть вызвано командой или сигналом от внешнего устройства. Прерывание позволяет обработать возникшее событие специальной программой и вернуться к прерванной программе.

А.Борковский
Англо-русский словарь
по программированию и информатике

В BM2M прерывания осуществляются как переключения процессов с занесением в Т-регистр причины прерывания (для прерываний, возбуждаемых процессором и связанных с процессом). Производится переключение с прерванного процесса на процесс обработки прерывания. Обработываемый процесс определяется номером прерывания.

4.1. Типы прерываний

Прерывания могут быть вызваны как аппаратными, так и программными средствами. Примером аппаратного прерывания может служить прерывание от таймера, которое производится 50 раз в секунду.

Программными средствами могут быть вызваны такие прерывания, как переполнение целого, выход за границы массива и т.д..

Прерывание полностью определяется своим номером. Перечислим прерывания и их номера:

- 01h таймер;
- 02h останов процессора;
- 03h обращение к отсутствующей памяти;
- 04h авария питания;
- 05h ошибка процессора;
- 06h ошибка ввода вектора прерывания;
- 07h нереализованная команда;
- 08h по вызову процедуры (П2.2);

09h	по возврату из процедуры (П2.2);
0Bh	трассировка (прерывание по каждой команде); (П2.5,П2.6)
40h	переполнение Р-стека ($S > H$);
41h	переполнение целого или деление на 0;
42h	переполнение вещественного;
43h	исчерпание вещественного;
44h	переполнение адреса (П2.2);
49h	INVLD команда;
4Ah	выход из диапазона;
4Bh	неверный параметр команды (аппаратный ASSERT);
4Ch	исчерпание или переполнение стека выражений (П2.2).

Если в скобках указан номер модели процессора, то соответствующее прерывание реализовано только для этой модели.

4.2. Вектора прерываний

Каждому прерыванию соответствует пара слов, в первом из которых лежит указатель на дескриптор обрабатываемого процесса, а во втором - адрес слова, в которое должен быть занесен адрес дескриптора прерванного процесса. Эта пара называется вектором прерывания.

4.3. Пространство векторов прерываний

Адрес вектора прерывания может быть вычислен по номеру этого прерывания умножением на 2, с одним исключением. Все прерывания с номером, большим 3Fh, происходят по вектору 3Fh.

Таким образом, множество векторов прерываний занимает непрерывный кусок памяти с абсолютного адреса 2h (прерывания нумеруются с единицы) по адрес 7Fh, и называется пространством векторов прерываний.

4.4. Маска прерываний

Некоторые из прерываний могут быть запрещены. Это делается, например, с целью программной обработки причины прерывания без возбуждения такового. Разрешение прерывания осуществляется выставлением соответствующего бита в маске прерываний. Маска прерываний хранится в М-регистре процессора. При этом выставление 0-го бита разрешает прерывания от внешних устройств, 1-го бита - прерывания от таймера, 31-го бита - программные прерывания (все прерывания с номером, большим или равным 3Fh).

Более полное и подробное представление о VM2M можно получить, ознакомившись с программой интерпретатора М-кода,

которая также является спецификацией микропрограмм процессоров
Кронос.

.PAGE

.HEAD 1 '@УАРХИТЕКТУРА

ИНТЕРПРЕТАТОР М-КОДА@u'

.HEAD 0 '@УАРХИТЕКТУРА

ИНТЕРПРЕТАТОР М-КОДА@u'

ЧАСТЬ II. ИНТЕРПРЕТАТОР М-КОДА

(*

	00	20	40	60	80	A0	C0	E0
00	LI0	LLW	LXB	LSW0		LSS	MOVE	INCL
01	LI1	LGW	LXW	LSW1	QUIT	LEQ	**CHKNIL	EXCL
02	LI2	LEW	LGW2	LSW2	GETM	GTR	LSTA	*INL
03	LI3	LSW	LGW3	LSW3	SETM	GEQ	COMP	*QUOT
04	LI4	LLW4	LGW4	LSW4	TRAP	EQU	GB	INC1
05	LI5	LLW5	LGW5	LSW5	TRA	NEQ	GB1	DEC1
06	LI6	LLW6	LGW6	LSW6	TR	ABS	CHK	INC
07	LI7	LLW7	LGW7	LSW7	IDLE	NEG	CHKZ	DEC
08	LI8	LLW8	LGW8	LSW8	ADD	OR	ALLOC	STOT
09	LI9	LLW9	LGW9	LSW9	SUB	AND	ENTR	LODT
0A	LI0A	LLW0A	LGW0A	LSW0A	MUL	XOR	RTN	LXA
0B	LI0B	LLW0B	LGW0B	LSW0B	DIV	BIC	NOP	LPC
0C	LI0C	LLW0C	LGW0C	LSW0C	SHL	IN	CX	**BBU
0D	LI0D	LLW0D	LGW0D	LSW0D	SHR	BIT	CI	**BBP
0E	LI0E	LLW0E	LGW0E	LSW0E	ROL	NOT	CF	**BBLT
0F	LI0F	LLW0F	LGW0F	LSW0F	ROR	MOD	CL	**PDX
10	LIB	SLW	SXB	SSW0	IO0	DECS	CL0	SWAP
11	LID	SGW	SXW	SSW1	IO1	DROP	CL1	LPA
12	LIW	SEW	SGW2	SSW2	IO2	LODFV	CL2	LPW
13	LIN	SSW	SGW3	SSW3	IO3	STORE	CL3	SPW
14	LLA	SLW4	SGW4	SSW4	IO4	STOFV	CL4	SSWU
15	LGA	SLW5	SGW5	SSW5	*ARRCMP	COPT	CL5	**RCHK
16	LSA	SLW6	SGW6	SSW6	*WM	CPCOP	CL6	**RCHZ
17	LEA	SLW7	SGW7	SSW7	*BM	PCOP	CL7	**CM
18	JFLC	SLW8	SGW8	SSW8	FADD	*FOR1	CL8	*CHKBX
19	JFL	SLW9	SGW9	SSW9	FSUB	*FOR2	CL9	*BMG
1A	JFSC	SLW0A	SGW0A	SSW0A	FMUL	*ENTC	CL0A	ACTIV
1B	JFS	SLW0B	SGW0B	SSW0B	FDIV	*XIT	CL0B	USR
1C	JBLC	SLW0C	SGW0C	SSW0C	FCMP	ADDP	CL0C	SYS
1D	JBL	SLW0D	SGW0D	SSW0D	FABS	JMP	CL0D	**NII
1E	JBSC	SLW0E	SGW0E	SSW0E	FNEG	ORJP	CL0E	DOT
1F	JBS	SLW0F	SGW0F	SSW0F	FFCT	ANDJP	CL0F	INVLD

* -- реализовано только на П2.2.

** -- не реализовано на П2.2.

(c) COPYRIGHT Kronos Research Group

1985,1986,1987,1989

Последнее исправление

Oct-1989

.PAGE

MODULE Kronos_Interpreter;

(* Leo 27-Nov-85. (c) Kronos *)

(* Ned 30-Aug-87. (c) KRONOS *)

(* Этот интерпретатор является спецификацией аппаратуры. *)

```
FROM SYSTEM    IMPORT  ADDRESS, WORD, ADR;  
FROM KRONOS    IMPORT  ROR, ROL, SHR, SHL;
```

```
TYPE CPUs = (Kronos2_2, Kronos2_5, Kronos2_6);
```

```
VAR cpu: CPUs;
```

```
CONST ESdepth = 7; (* глубина стека выражений *)
```

```
TYPE
```

```
  BYTE      = [0..255];  
  WORD16    = [0..0FFFFh];  
  PC_Range  = WORD16;  
  CodePtr   = POINTER TO ARRAY PC_Range OF BYTE;
```

```
VAR
```

```
  PC:      PC_Range;      (* счетчик команд *)  
  IR:      [0..0FFh];     (* регистр команды *)  
  F :      CodePtr;       (* адрес сегмента кода *)  
  G :      ADDRESS;       (* адрес сегмента глобальных данных *)  
  L :      ADDRESS;       (* адрес сегмента локальных данных *)  
  S :      ADDRESS;       (* адрес вершины Р-стека *)  
  H :      ADDRESS;       (* граница Р-стека *)  
  P :      ADDRESS;       (* адрес дескриптора процесса *)  
  M :      BITSET;        (* маска прерываний *)  
  Ipt:     BOOLEAN;       (* запрос на прерывание *)  
  IptNo:   WORD16;        (* номер прерывания *)
```

```
CONST (* номера битов в слове L2 (см. рис.) *)
```

```
  ExternalBit = 1Fh;  
  ChangeMaskBit = 1Eh;
```

```
CONST
```

```
  NonVectBit = 1Fh;
```

```
(* бит, маскирующий программные прерывания *)
```

```
(* Замечание. Регистр H задает заниженную на ESdepth+1 слов  
  границу стека. Это необходимо для сохранения стека  
  выражений при переключении процессов (см. SaveExpStack).  
*)
```

```
VAR MEM: ARRAY ADDRESS OF WORD;
```

```
  ByteMEM: ARRAY OF BYTE; (* Наложено на MEM *)
```

```
MODULE InstructionFetch;
```

```
  IMPORT F, PC, WORD, WORD16, BYTE, MEM, ADDRESS;  
  EXPORT Next, Next2, Next4, GetPc;
```

```
  PROCEDURE Next(): BYTE;
```

```
BEGIN INC(PC); RETURN INTEGER(F^[PC-1])
END Next;
```

```
PROCEDURE Next2(): WORD16;
BEGIN RETURN Next()+Next()*100h
END Next2;
```

```
PROCEDURE Next4(): WORD;
BEGIN RETURN Next2()+Next2()*10000h
END Next4;
```

```
PROCEDURE GetPc(procno: INTEGER): INTEGER;
(* Выдает PC начала процедуры *)
BEGIN RETURN MEM[ADDRESS(F)+procno]
END GetPc;
```

```
END InstructionFetch;
```

```
MODULE Mask;
IMPORT M, NonVectBit, BYTE;
EXPORT NotMasked;
```

```
PROCEDURE NotMasked(N: BYTE): BOOLEAN;
BEGIN
  IF (N>=0Fh) & (N<3Fh) THEN RETURN (0 IN M)
  ELSIF (N< 0Fh) & (N>0) THEN RETURN (0 IN M) & (N IN M)
  ELSIF (N =3Fh) THEN RETURN (NonVectBit IN M)
  ELSE ASSERT(FALSE)
  END
END NotMasked;
```

```
END Mask;
```

```
MODULE ExpressionStack;
IMPORT WORD, ESdepth, Ipt, IptNo;
EXPORT Push, Pop, Empty;
```

```
VAR A: ARRAY [0..ESdepth-1] OF WORD; sp: [0..ESdepth];
```

```
PROCEDURE Push(X: WORD);
BEGIN A[sp]:=X;
  IF sp<ESdepth THEN INC(sp) ELSE Ipt:=TRUE; IptNo:=4Ch END;
END Push;
```

```
PROCEDURE Pop(): INTEGER;
BEGIN
  IF sp=0 THEN Ipt:=TRUE; IptNo:=4Ch ELSE DEC(sp) END;
  RETURN A[sp];
END Pop;
```

```
PROCEDURE Empty(): BOOLEAN;
BEGIN RETURN sp=0 END Empty;
```

```
BEGIN sp:=0 END ExpressionStack;
```

```

MODULE ProcessSupport;
  IMPORT PC,G,F,H,L,S,P,M, MEM, NotMasked, ESdepth
    , CodePtr, WORD16, ADDRESS;
  FROM ExpressionStack IMPORT Pop, Push, Empty;

  EXPORT SaveExpStack, RestoreExpStack, Transfer, TRAP;

  PROCEDURE SaveExpStack;
    VAR c: CARDINAL; (* счетчик глубины стека *)
  BEGIN c:=0;
    WHILE NOT Empty() DO MEM[S]:=Pop(); INC(S); INC(c) END;
    MEM[S]:=c; INC(S);
  END SaveExpStack;

  PROCEDURE RestoreExpStack;
    VAR c: CARDINAL; (* счетчик глубины стека *)
  BEGIN DEC(S); c:=MEM[S];
    WHILE c>0 DO DEC(c); DEC(S); Push(MEM[S]) END;
  END RestoreExpStack;

  PROCEDURE SaveRegs;
  BEGIN SaveExpStack;
    MEM[P+0]:=G; MEM[P+1]:=L;
    MEM[P+2]:=PC; MEM[P+3]:=CARDINAL(M);
    MEM[P+4]:=S; MEM[P+5]:=H+ESdepth+1;
  END SaveRegs;

  PROCEDURE RestoreRegs;
  BEGIN
    G:=MEM[P+0]; F:=CodePtr(MEM[G]);
    L:=MEM[P+1]; PC:=MEM[P+2]; M:=BITSET(MEM[P+3]);
    S:=MEM[P+4]; H:=MEM[P+5]-ESdepth-1;
    RestoreExpStack;
  END RestoreRegs;

  PROCEDURE Transfer(pFrom,pTo: ADDRESS);
    VAR j: CARDINAL;
  BEGIN (* заметьте - pFrom может быть равно pTo *)
    j:=MEM[pTo]; SaveRegs; MEM[pFrom]:=P; MEM[1]:=P;
    P:=j; RestoreRegs; MEM[0]:=P;
  END Transfer;

  PROCEDURE TRAP(N: WORD16);
  BEGIN MEM[P+6]:=N;
    IF N>3Fh THEN N:=3Fh END;
    IF NotMasked(N) THEN Transfer(N*2, MEM[N*2+1]) END;
  END TRAP;

END ProcessSupport;

(* Маркировка P-стека перед вызовом процедуры *)

PROCEDURE Mark(X: ADDRESS; External: BOOLEAN);

```

```

VAR i: ADDRESS;
BEGIN i:=S;
  MEM[S]:=X; INC(S); (* статическая цепочка *)
  MEM[S]:=L; INC(S); (* динамическая цепочка *)
  IF External THEN MEM[S]:=WORD(BITSET(PC)+{ExternalBit})
  ELSE
    MEM[S]:=PC
  END; INC(S,2); L:=i;
END Mark;

PROCEDURE ioP2_2;
BEGIN
  ASSERT(IR=90h IN {0..7});
  (* см. документацию на П2.2 *)
END ioP2_2;

PROCEDURE ioP2_5;
BEGIN
  ASSERT(IR=90h IN {0..7});
  (* Запросы на В/В передаются другим процессорам через общую
    память. См. LABTAM 3000 manuals.
  *)
END ioP2_5;

PROCEDURE ioP2_6;
BEGIN ASSERT(IR=90h IN {0..7});
  (* Смотря на какой шине. *)
END ioP2_6;

(* Рабочие переменные интерпретатора: *)

VAR i,j,k: CARDINAL;  X,Y : REAL;
    v,w : BITSET;     a,b : CHAR;
    adr,adr1,sz,hi,low: CARDINAL;
    src,trg: INTEGER;
    dyn: POINTER TO
      RECORD
        adr : ADDRESS;
        high: INTEGER;
      END;

PROCEDURE ConsolMicroProgram;
BEGIN
  (* Пультовая микропрограмма позволяет произвести начальную
    загрузку программы и по команде оператора "Go" выполняет
    Transfer(0,1).
  *)
END ConsolMicroProgram;

PROCEDURE Interpret;
BEGIN
  CASE IR OF
    00h..0Fh: (* LI0..LI0F Load Immediate *) Push(IR MOD 10h);

    |10h: (* LIB Load Immediate Byte *) Push(Next())

```

```

|11h: (* LID Load Immediate Double byte *) Push(Next2())
|12h: (* LIW Load Immediate Word *) Push(Next4())
|13h: (* LIN Load Immediate NIL *) Push(NIL)
|14h: (* LLA Load Local Address *) Push(L+Next())
|15h: (* LGA Load Global Address *) Push(G+Next())
|16h: (* LSA Load Stack Address *) Push(Pop()+Next())
|17h: (* LEA Load External Address *)
      i:=G-Next()-1; (* индекс в DFT модуля.*)
      adr:=MEM[i]; (* указатель на элемент большой DFT *)
      Push(MEM[adr]+Next())
|18h: (* JLFC Jump Long Forward Condition *)
      IF Pop()=0 THEN PC:=Next2()+PC
      ELSE INC(PC,2) END
|19h: (* JLF Jump Long Forward *) PC:=Next2()+PC;
|1Ah: (* JSFC Jump Short Forward Condition *)
      IF Pop()=0 THEN PC:=Next()+PC
      ELSE INC(PC) END
|1Bh: (* JSF Jump Short Forward *) PC:=Next()+PC;
|1Ch: (* JLBC Jump Long Back Condition *)
      IF Pop()=0 THEN PC:=-Next2()+PC
      ELSE INC(PC,2) END
|1Dh: (* JLB Jump Long Back *) PC:=-Next2()+PC;
|1Eh: (* JSBC Jump Short Back Condition *)
      IF Pop()=0 THEN PC:=-Next()+PC
      ELSE INC(PC) END
|1Fh: (* JSB Jump Short Back *) PC:=-Next()+PC;
|20h: (* LLW Load Local Word *) Push(MEM[L+Next()])
|21h: (* LGW Load Global Word *) Push(MEM[G+Next()])
|22h: (* LEW Load External Word *)
      i:=G-Next()-1; adr:=MEM[MEM[i]]; (* external G *)
      Push(MEM[adr+Next()])
|23h: (* LSW Load Stack addressed Word *)
      Push(MEM[Pop()+Next()])
|24h..2Fh: (* LLW0..LLW0F Load Local Word *)
      Push(MEM[L+IR MOD 10h])
|30h: (* SLW Store Local Word *) MEM[L+Next()]:=Pop()
|31h: (* SLG Store Global Word *) MEM[G+Next()]:=Pop()
|32h: (* SEW Store External Word *)
      i:=G-Next()-1; adr:=MEM[MEM[i]]; (* external G *)
      MEM[adr+Next()]:=Pop()
|33h: (* SSW Store Stack addressed Word *)
      i:=Pop(); MEM[Pop()+Next()]:=i
|34h..3Fh: (* SLW0..SLW0F Store Local Word *)
      MEM[L+IR MOD 10h]:=Pop()

|40h: (* LXB Load Indexed Byte *)
      i:=Pop(); Push(ByteMEM[Pop()*4+i]);
|41h: (* LXW Load Indexed Word *)
      i:=Pop(); Push(MEM[Pop()+i])
|42h..4Fh: (* LGW02..LGW0F Load Global Word *)
      Push(MEM[G+IR MOD 10h])
|50h: (* SXB Store Indexed Byte *)
      j:=Pop(); i:=Pop(); ByteMEM[Pop()*4+i]:=j;
|51h: (* SXW Store Indexed Word *)

```

```

        j:=Pop(); i:=Pop(); MEM[Pop()+i]:=j
|52h..5Fh: (* SGW02..SGW0F Store Global Word *)
        MEM[G+IR MOD 10h]:=Pop()

|60h..6Fh: (* LSW00..LSW0F Load Stack addressed Word *)
        Push(MEM[Pop()+IR MOD 10h])
|70h..7Fh: (* SSW00..SSW0F Store Stack addressed Word *)
        i:=Pop(); MEM[Pop()+IR MOD 10h]:=i

|80h: TRAP(7h);
|81h: (* QUIT Stop processor *) ConsolMicroProgram
|82h: (* GETM Get Mask *) Push(M)
|83h: (* SETM Set Mask *) M:=BITSET(Pop());
|84h: (* TRAP interrupt simulation *) TRAP(Pop())
|85h: (* TRA Transfer control between process *)
        i:=Pop(); Transfer(Pop(),i)
|86h: (* TR Test & Reset *)
        i:=Pop(); Push(MEM[i]); MEM[i]:=0
|87h: (* IDLE IDLE process *)
        DEC(PC); REPEAT (* не занимая шины *) UNTIL Ipt
(* В следующих шести командах, а также в командах
MOD,NEG,ABS,FOR2,INC,DEC,INC1,DEC1 в случае переполнения
возбуждается прерывание с IptNo=41h.
*)
|88h: (* ADD integer ADD *) Push(Pop()+Pop())
|89h: (* SUB integer SUB *) i:=Pop(); Push(Pop()-i);
|8Ah: (* MUL integer MUL *) Push(Pop()*Pop())
|8Bh: (* DIV integer DIV *) i:=Pop(); Push(Pop() DIV i)
|8Ch: (* SHL integer SHift Left *)
        i:=Pop(); Push(SHL(Pop(),i))
|8Dh: (* SHR integer SHift Right *)
        i:=Pop(); Push(SHR(Pop(),i))

|8Eh: (* ROL word ROTate Left *)
        i:=Pop() MOD 20h; Push(ROL(Pop(),i))
|8Fh: (* ROR word ROTate Right *)
        i:=Pop() MOD 20h; Push(ROR(Pop(),i))
|90h..94h: (* io section *)
        CASE cpu OF
            |Kronos2_2: ioP2_2
            |Kronos2_5: ioP2_5
            |Kronos2_6: ioP2_6
        ELSE
        END

|95h: (* ARRCMP array compare *)
        sz:=Pop(); adr:=Pop(); adr1:=Pop();
        IF sz<0 THEN Push(sz); TRAP(4Fh)
        ELSIF sz=0 THEN Push(adr1); Push(adr1)
        ELSE LOOP
            IF (adr^ # adr1^ ) OR (sz=1) THEN
                Push(adr1); Push(adr); EXIT
            END;
            DEC(sz); INC(adr); INC(adr1);

```

```

        END;
    END;

|96h: (* WM      word move *)
      sz:=Pop(); f:=Pop(); t:=Pop();
      IF t>f THEN
        t:=t+sz-1; f:=f+sz-1;
        WHILE sz>0 DO
          MEM[t]:=MEM[f]; DEC(t); DEC(f); DEC(sz)
        END
      ELSE
        WHILE sz>0 DO
          MEM[t]:=MEM[f]; INC(t); INC(f); DEC(sz)
        END
      END;

|97h: (* BM      bit move *)
      sz:=Pop(); -- размер пересылаемой области в битах
      i:=Pop(); src:=Pop(); -- смещение и адрес источника
      j:=Pop(); trg:=Pop(); -- смещение и адрес приемника
      src:=src*32+i;
      trg:=trg*32+j;
      IF src>=trg THEN                n:=+1
      ELSE INC(src,sz-1); INC(trg,sz-1); n:=-1
      END;
      FOR k:=0 TO sz-1 DO
        i:=trg DIV 32; j:=trg MOD 32;
        IF (src MOD 32) IN BITSET(MEM[src DIV 32]) THEN
          MEM[i]:=INTEGER( BITSET(MEM[i]) + {j} )
        ELSE
          MEM[i]:=INTEGER( BITSET(MEM[i]) - {j} )
        END;
        INC(src,n); INC(trg,n)
      END

```

(* В следующих восьми командах (за исключением FCMP) при исчезновении порядка или переполнении возбуждаются прерывания с IptNo=42h или 43h соответственно.

*)

```

|98h: (* FADD  Float ADD *) Push(REAL(Pop())+REAL(Pop()))
|99h: (* FSUB  Float SUB *)
      X:=REAL(Pop()); Push(REAL(Pop())-X)
|9Ah: (* FMUL  Float MUL *) Push(REAL(Pop())*REAL(Pop()))
|9Bh: (* FDIV  Float DIV *)
      X:=REAL(Pop()); Push(REAL(Pop())/X)
|9Ch: (* FCMP  Float CoMPare *)
      X:=REAL(Pop()); Y:=REAL(Pop());
      IF X<Y THEN Push(1); Push(0)
      ELIF X>Y THEN Push(0); Push(1)
      ELSE Push(0); Push(0) END
|9Dh: (* FABS  Float ABS *) X:=REAL(Pop());
      IF X<0.0 THEN Push(-X) ELSE Push(X) END
|9Eh: (* FNEG  Float NEG *) Push(-REAL(Pop()))
|9Fh: (* FFCT  Float FunCTions *) i:=Next();

```

```

IF i=0 THEN Push(FLOAT(INTEGER(Pop())))
ELIF i=1 THEN Push(TRUNC( REAL(Pop())))
ELSE DEC(PC); TRAP(7h)
END;

|0A0h: (* LSS int LeSS *) i:=Pop(); Push(Pop(<i)
|0A1h: (* LEQ int Less or Equal *) i:=Pop(); Push(Pop(<=i)
|0A2h: (* GTR int GreaTeR *) i:=Pop(); Push(Pop(>i)
|0A3h: (* GEQ int Greater or Equal *) i:=Pop(); Push(Pop(>=i)
|0A4h: (* EQU int EQUal *) Push(Pop()==Pop())
|0A5h: (* NEQ int Not Equal *) Push(Pop()#Pop())
|0A6h: (* ABS int ABSolute value *) Push(ABS(Pop()))
|0A7h: (* NEG int NEGate *) Push(-Pop())

|0A8h: (* OR logical bit per bit OR *)
v:=BITSET(Pop()); w:=BITSET(Pop()); Push(w+v)
|0A9h: (* AND logical bit per bit AND *)
v:=BITSET(Pop()); w:=BITSET(Pop()); Push(w*v)
|0AAh: (* XOR logical bit per bit XOR *)
v:=BITSET(Pop()); w:=BITSET(Pop()); Push(w/v)
|0ABh: (* BIC logical bit per bit BIT Clear *)
v:=BITSET(Pop()); w:=BITSET(Pop()); Push(w-v)
|0ACh: (* IN membership to bitset *)
v:=BITSET(Pop()); Push(Pop() IN v)
|0ADh: (* BIT setBIT *) i:=Pop();
IF (i<0) OR (i>=20h) THEN TRAP(4Ah)
ELSE w:={}; INCL(w,i); Push(w) END

|0AEh: (* NOT boolean NOT (not bit per bit!) *) Push(Pop()==0)
|0AFh: (* MOD integer MODulo *) i:=Pop(); Push(Pop() MOD i)

|0B0h: (* DECS DECriment S register (reverse to ALLOC) *)
DEC(S,Pop())
|0B1h: (* DROP *) i:=Pop();
|0B2h: (* LODF reLOaD expr. stack after Function return *)
i:=Pop(); RestoreExpStack; Push(i)
|0B3h: (* STORE STORE expr. stack before function call *)
IF S+ESdepth+1>H THEN DEC(PC); TRAP(40h)
ELSE SaveExpStack
END
|0B4h: (* STOFV STOfV expr. stack with Formal function Value
on top before function call (см. команду CF)
*)
IF S+ESdepth+2>H THEN DEC(PC); TRAP(40h)
ELSE i:=Pop(); SaveExpStack; MEM[S]:=i; INC(S) END
|0B5h: (* COPT COpy Top of expr. stack *)
i:=Pop(); Push(i); Push(i)
|0B6h: (* CPCOP Character array Parameter COpy *)
i:=Pop(); (* High *) sz:=(i+4) DIV 4;
IF S+sz>H THEN Push(i); DEC(PC); TRAP(40h)
ELSE MEM[L+Next()]:=S; adr:=Pop();
WHILE sz>0 DO MEM[S]:=MEM[adr]; INC(S); INC(adr) END
END
|0B7h: (* PCOP structure Parameter allocate and COpy *)

```



```

i:=Pop(); (* High *) sz:=i+1;
IF S+sz>H THEN Push(i); DEC(PC); TRAP(40h)
ELSE MEM[L+Next()]:=S; adr:=Pop();
  WHILE sz>0 DO MEM[S]:=MEM[adr]; INC(S); INC(adr) END
END
|0B8h: (* FOR1 enter FOR statment *)
IF S+2>H THEN DEC(PC); TRAP(40h)
ELSE sz:=Next(); (* =0 up; #0 down *)
  hi:=Pop(); low:=Pop(); adr:=Pop(); k:=Next2()+PC;
  IF ((sz=0) & (low<=hi)) OR ((sz#0) & (low>=hi)) THEN
    MEM[adr]:=low;
    MEM[S]:=adr; INC(S); MEM[S]:=hi; INC(S);
  ELSE (* цикл не исполняется не разу *) PC:=k
  END
END
|0B9h: (* FOR2 end of FOR statment *)
hi:=MEM[S-1]; adr:=MEM[S-2]; sz:=Next();
IF sz>7Fh THEN sz:=7Fh-sz END; (* шаг [-128..127] *)
k:=-Next2()+PC; i:=MEM[adr]+sz;
IF ((sz>=0) & (i>hi)) OR ((sz<0) & (i<hi)) THEN
  DEC(S,2); (* terminate *)
ELSE MEM[adr]:=i; PC:=k (* continue *)
END
|0BAh: (* ENTC ENTER Case statment *)
IF S+1>H THEN DEC(PC); TRAP(40h)
ELSE PC:=Next2()+PC; (* jump to case table *)
  k:=Pop(); low:=Next2(); hi:=Next2();
  MEM[S]:=PC + 2*(hi-low) + 4; INC(S);
  (* PC для выхода *)
  IF (k>=low) & (k<=hi) THEN
    PC:=PC+2*(k-low+1) (* jump into case table *)
  END;
  PC:=-Next2()+PC (* jump back to variant's code *)
END
|0BBh: (* XIT eXIT from case or control structure *)
DEC(S); PC:=MEM[S]
|0BCh: (* ADDPC add to program counter *)
Push(Pop()+PC);
|0BDh: (* JMP *)
PC:=Pop();
|0BEh: (* ORJP short circuit OR Jump *)
IF Pop()#0 THEN Push(1); PC:=Next()+PC
ELSE INC(PC) END
|0BFh: (* ANDJP short circuit AND Jump *)
IF Pop()=0 THEN Push(0); PC:=Next()+PC
ELSE INC(PC)
END

|0C0h: (* MOVE MOVE block *) sz:=Pop();
i:=Pop(); j:=Pop();
WHILE sz>0 DO
  MEM[j]:=MEM[i]; INC(i); INC(j); DEC(sz)
END
|0C1h: (* CHKNIL check address for NIL *)

```

```

        i:=Pop(); Push(i);
        IF i=NIL THEN DEC(PC); TRAP(41h) END
|0C2h: (* LSTA Load STring Address *)
        Push(MEM[G+1]+Next2());
|0C3h: (* COMP COMPare strings *) i:=Pop()*4; j:=Pop()*4;
        REPEAT a:=CHAR(ByteMEM[i]); b:=CHAR(ByteMEM[j]);
            INC(i); INC(j)
        UNTIL (a=0c) OR (b=0c) OR (a#b); Push(a); Push(b)
|0C4h: (* GB Get procedure Base n level down *)
        i:=L; k:=Next();
        WHILE k>0 DO i:=MEM[i]; DEC(k) END; Push(i)
|0C5h: (* GB1 :=Pop(); i:=Pop(); Push(i);
|0C6h: (* CHK array boundary CHecK *)
        hi:=Pop(); low:=Pop(); i:=Pop(); Push(i);
        IF (i<low) OR (i>hi) THEN
            Push(low); Push(hi); TRAP(4Ah)
        END
|0C7h: (* CHKZ array boundary CHecK (low=Zero) *)
        hi:=Pop(); i:=Pop(); Push(i);
        IF (i<0) OR (i>hi) THEN Push(hi); TRAP(4Ah) END
|0C8h: (* ALLOC ALLOCate block *) sz:=Pop();
        IF S+sz>H THEN Push(sz); DEC(PC); TRAP(40h)
        ELSE Push(S); INC(S,sz) END
|0C9h: (* ENTR ENTer procedure *) sz:=Next();
        IF S+sz>H THEN DEC(PC,2); TRAP(40h)
        ELSE INC(S,sz) END
|0CAh: (* RTN ReTurN from procedure *)
        S:=L; L:=MEM[S+1];
        PC:=WORD(BITSET(MEM[S+2])*{0..0Fh});
        IF ExternalBit IN BITSET(MEM[S+2]) THEN
            (* external called *)
            G:=MEM[S]; F:=CodePtr(MEM[G])
        END;
|0CBh: (* NOP No OPeration *)
|0CCh: (* CX Call eXternal *)
        IF S+4<=H THEN j:=MEM[G-Next()-1];
            i:=Next(); Mark(G,TRUE);
            G:=MEM[j]; F:=CodePtr(MEM[G]); PC:=GetPc(i);
        ELSE DEC(PC); TRAP(40h) END
|0CDh: (* CI Call procedure at Intermediate level *)
        IF S+4<=H THEN
            i:=Next(); Mark(Pop(),FALSE); PC:=GetPc(i);
        ELSE DEC(PC); TRAP(40h) END
|0CEh: (* CF Call Formal procedure *)
        IF S+3<=H THEN i:=MEM[S-1]; DEC(S); Mark(G,TRUE);
            k:=i DIV 1000000h; i:=i MOD 1000000h;
            G:=MEM[i]; F:=CodePtr(MEM[G]); PC:=GetPc(k);
        ELSE DEC(PC); TRAP(40h) END
|0CFh: (* CL Call Local procedure *)
        IF S+4<=H THEN i:=Next(); Mark(L,FALSE); PC:=GetPc(i);
        ELSE DEC(PC); TRAP(40h) END
|0D0h..0DFh: (* CL0..CL0F Call Local procedure *)
        IF S+4<=H THEN Mark(L,FALSE); PC:=GetPc(IR MOD 10h);
        ELSE DEC(PC); TRAP(40h) END

```

```

|0E0h: (* INCL INCLude in set *)
        i:=Pop(); j:=Pop() + i DIV 32;
        MEM[j]:=INTEGER( BITSET(MEM[j]) + {i MOD 32} );
|0E1h: (* EXCL EXCLude from set *)
        i:=Pop();
        j:=Pop() + i DIV 32;
        MEM[j]:=INTEGER( BITSET(MEM[j]) - {i MOD 32} );
|0E2h: (* INL membership IN Long set *)
        k:=Pop(); j:=Pop(); i:=Pop();
        IF (i<0) OR (i>=k) THEN Push(0)
        ELSE
            Push( (i MOD 32) IN BITSET(MEM[j+i DIV 32]) );
        END;
|0E3h: (* QUOT QUOTient commands *)
        i:=Pop(); j:=Pop();
        CASE Next() OF
            |0: Push( j SHRQ i ); -- деление на степень двойки
            |1: Push( j QOU i ); -- деление с округлением к нулю
            |2: Push( j ANDQ i ); -- модуль по степени двойки
            |3: Push( j REM i ); -- модуль
        ELSE TRAP(7); DEC(PC,2)
        END;
|0E4h: (* INC1 INCrement by 1 *) INC(MEM[Pop()])
|0E5h: (* DEC1 DECrement by 1 *) DEC(MEM[Pop()])
|0E6h: (* INC INCrement *) i:=Pop(); INC(MEM[Pop()],i)
|0E7h: (* DEC DECrement *) i:=Pop(); DEC(MEM[Pop()],i)
|0E8h: (* STOT STOre Top on proc stack *)
        IF S+1>H THEN DEC(PC); TRAP(40h)
        ELSE MEM[S]:=Pop(); INC(S)
        END
|0E9h: (* LODT LOaD Top of proc stack *)
        DEC(S); Push(MEM[S])
|0EAh: (* LXA Load indeXed Address *)
        sz:=Pop(); i:=Pop(); adr:=Pop(); Push(adr+i*sz)
|0EBh: (* LPC Load Procedure Constant *)
        i:=Next(); j:=Next(); Push(j*1000000h+MEM[G-i-1])

```

(* Следующие 3 команды работают с вырезками битов. Битовый адрес - это пара (адрес,битовое смещение). Битовое смещение может быть больше 32. Вырезка может переходить границы слова.

*)

```

|0ECh: (* BBU Bit Block Unpack *)
        sz:=Pop();
        IF (sz<1) OR (sz>32) THEN
            Push(sz); DEC(PC); TRAP(4Ah)
        END;
        i:=Pop(); adr:=Pop();
        (* j:=битовая вырезка длиной sz, начиная с
           битового адреса (adr,i)
        *)
        Push(j);
|0EDh: (* BBP Bit Block Pack *)
        j:=Pop(); sz:=Pop();

```

```

    IF (sz<1) OR (sz>32) THEN
        Push(sz); DEC(PC); TRAP(4Ah)
    END;
    i:=Pop(); adr:=Pop();
    (* Упаковывает sz младших битов из j по битовому
       адресу (adr,i)
    *)
|0EEh: (* BBLT Bit BLock Transfer *)
    sz:=Pop(); (* Длина может любой, больше 0 *)
    i:=Pop(); adr:=Pop();
    j:=Pop(); adr1:=Pop();
    (* Переслать биты (adr,i) -> (adr1,j) длиной sz *)
    (* Куски не должны перекрываться!!! *)
|0EFh: (* PDX Prepare Dynamic index *)
    i:=Pop();
    dyn:=Pop();
    Push(dyn^.adr); Push(i);
    IF (i<0) OR (i>dyn^.high)
    THEN TRAP(4Ah)
    END;
|0F0h: (* SWAP *)
    i:=Pop(); j:=Pop(); Push(i); Push(j)
|0F1h: (* LPA Load Parameter Address *)
    Push(L-Next()-1);
|0F2h: (* LPW Load Parameter WORD *)
    Push(MEM[L-Next()-1]);
|0F3h: (* SPW Store Parameter WORD *)
    MEM[L-Next()-1]:=Pop();
|0F4h: (* SSWU Store Stack Word Undestructive *)
    i:=Pop(); MEM[Pop()]:=i; Push(i)
|0F5h: (* RCHK range CHeck *)
    hi:=Pop(); low:=Pop(); i:=Pop(); Push(i);
    Push( (low<=i) & (i<=hi) );
|0F6h: (* RCHZ range check (low=Zero) *)
    hi:=Pop(); i:=Pop(); Push(i);
    Push( (0<=i) & (i<=hi) );
|0F7h: (* CM Call procedure from dynamic Module *)
    IF S+4<=H THEN
        i:=Next();
        DEC(S); j:=MEM[S];
        Mark(G,TRUE);
        G:=j; F:=CodePtr(MEM[G]); PC:=GetPc(i);
    ELSE DEC(PC); TRAP(40h)
    END;
|0F8h: (* CHKBX CHeck BoX *)
    p0:=Pop(); p1:=Pop();
    x00:=INTEGER(p0^) MOD 10000h;
    y00:=INTEGER(p0^<<16) MOD 10000h;
    INC(p0);
    x01:=INTEGER(p0^) MOD 10000h;
    y01:=INTEGER(p0^<<16) MOD 10000h;
    x10:=INTEGER(p1^) MOD 10000h;
    y10:=INTEGER(p1^<<16) MOD 10000h;
    INC(p1);

```

```

x11:=INTEGER(p1^)      MOD 10000h;
y11:=INTEGER(p1^<<16) MOD 10000h;
Push((x10<=x01) & (y10<=y01) & (x00<=x11) & (y00<=y11));
|0F9h: (* BMG Bit Map Graphic commands *)
CASE Next() OF
  |0: IN_RECT -- IN RECTangle?
  |1: DVL     -- Display Vertical Lines
  |2: BBLT_G  -- BBLT-loGic
  |3: DCH     -- Display CHar
  |4: CLP     -- CLiPping
  |5: DLN     -- Display LiNe
  |6: CRC     -- CiRCus
  |7: ARC     -- ARC
  |8: TRF     -- TRiangle Filling
  |9: CRF     -- CiRcus Filling
ELSE
  Trap(49h)
END
|0FAh: (* ACTIVE process *) Push(P)
|0FBh: (* USR User defined functions *) i:=Next(); (* *)
|0FCh: (* SYS SYStem rarely functions *)
CASE Next() OF
  |00h: (* PID Processor IDent {2,5,6} *)
        (* Push(PID) *)
        (* Остальные могут быть различными в разных моделях *)
  |02h: (* PM Processor Model *)
ELSE TRAP(7h)
END;
|0FDh: (* NII Never Implemented Instruction *) TRAP(7h);
|0FFh: (* INVLD INVaLiD Operation *) TRAP(49h)
ELSE (* unimplemented instruction *) TRAP(7h)
END (*CASE*)
END Interpret;

BEGIN (* "Power On" *)
(* cpu:=Kronos2_2 | Kronos2_5 | Kronos2_6 *)
ORIGIN(ByteMEM,ADR(MEM),SIZE(MEM));
Conso1MicroProgram;
LOOP
  IF Ipt THEN TRAP(IptNo) END;
  IR:=Next();
  Interpret;
END (*LOOP*)
END Kronos_Interpreter.

```