Эмулятор MIX

Yellow Rabbit

10 января 2009 г.

Содержание

1 Введение

Это эмулятор эмулятор машины MIX, мифического компьютера, придуманного Кнут'ом [1] для его изумительной книги. По большому счёту это просто игрушка, на которой я разбирался с *литературным программированием* [2], битовыми полями:), эмуляторами, TEX'ом, GCC и программированием под Linux вообще.

В процессе игры использовались:

- транслятор *noweb* version 2.11 описанный в [2];
- компилятор GCC version 4.1.2 20061115 (prerelease);
- "украшатель" кода C++ dpp version 0.2.1 ©Dan Schmidt <dfan@alum.mit.edu>;
- любимый ViM version 7.0 (с плагином Project version 1.4.1 ©Blumer aricvim@charter.net);
- издательская система LATEX2e <2003/12/01>.

2 MIX

Архитектура компьютера MIX не очень похожа на архитектуру компьютера IBM PC, к которому я привык. Например, отрицательные числа представляются как модуль и знак, а не в дополнительном коде. Есть два нуля: $+0$ и -0. Отсутствуют стек и программный счётчик.

Честно говоря, я почувствовал себя неуютно, обнаружив, что *настоящие программисты* обходятся без стека :).

2.1 Реализация

Слово компьютера MIX состоит из 5-ти байт и знака. Байт хранит 64 различных значения, знак — два. 64 значения — это 2^6, в двойное слово моего Pentium'а можно записать 2^{32} значений. Таким образом, попробуем упаковать все части слова MIX в двойное слово Pentium. Обращаться к слову можно как к целому (*asInt*) или по частям (*sign*, *b1*, *b2*, *b3*, *b4*, *b5*). Обращение *asInt* предназначено для тех, кто знает, что делает — здесь нужно внимательно следить за знаковым битом *sign*!

2 ⟨Слово MIX 2⟩≡ (10d) 3a▷

```
union mix_word{
 public:
  struct{
   unsigned int b5:6;
   unsigned int b4:6;
   unsigned int b3:6;
   unsigned int b2:6;
   unsigned int b1:6;
   unsigned int sign:1;
  };
  unsigned int asInt;
 ⟨Шаблоны операций со словом MIX 8d⟩
 enum exception{E_OVER};
```

```
};
```
Uses mix_word.

3a ⟨Слово MIX 2⟩+≡ (10d) ◁2 3b▷
```
const mix_word operator+(const mix_word& a,const mix_word& b);
```
Uses mix_word and operator+.

При выполнении некоторых операций со словами MIX возможно переполнение и т.д. В этих случаях буду генерировать исключение.

3b ⟨Слово MIX 2⟩+≡ (10d) ◁3a
```
class mix_word_exception{
 public:
  mix_word::exception code;
  mix_word val;
  mix_word_exception(mix_word::exception c,mix_word v){
   code = c;
   val = v;
  }
};
```
Defines:
mix_word, used in chunks 2, 3, 5, 7–10, 12a, 20a, 26–31, 38a, 39a, and 43.

MIX имеет два полнословных регистра A и X, шесть индексных регистров $I1$-6, регистр J[1], флагов сравнения *flagC* и переполнения *flagV*. Регистры $I1$-6 и J состоят из двух байт и знака. Оперативная память состоит из четырёх тысяч слов.

3c ⟨Константы класса машины 3c⟩≡ (3d) 4b▷
```
static const int RAM_SIZE = 4000;
```
Defines:
RAM_SIZE, used in chunk 7d.

3d ⟨Класс машины MIX 3d⟩≡ (10d)
```
class mix{
 public:
 ⟨Константы класса машины 3c⟩
 ⟨Функции класса машины 5b⟩
 public:
  mix_word rA,rX,rI[7],rJ;
  mix_word* ram;
  enum eCompare{less,equal,greater} flagC;
  bool flagV;
 protected:
 ⟨Внутренние функции и переменные класса машины 4a⟩
};
```
Defines:
mix, used in chunks 5–7, 11a, 12c, 18, 20, 23–30, 32, 33, 35, 36a, 38–40, 42c, and 43.
Uses mix_word.

[1] Применение регистра J весьма специфично и будет прояснено далее.

Замечу, что хотя индексных регистров всего шесть, память резервируется для семи — *rI*[7], сделал я это для того, чтобы всегда иметь под рукой нулевой индексный регистр. Он обнуляется в конструкторе. И больше нигде не записывается.

3e ⟨Инициализация специального индексного регистра 3e⟩≡ (7d)
```
rI[0].setInt(0);
```
Uses `setInt`.

2.2 Устройства ввода/вывода

Машина MIX оснащена богатым набором устройств ввода/вывода:

№устройства	Периферийное устройство	Размер блока (слов)
t	Накопитель на магнитной ленте ($0 <= t <= 7$)	100
d	Диск или барабан ($8 <= d <= 15$)	100
16	Устройство чтения перфокарт	16
17	Перфоратор	16
18	Принтер	24
19	Терминал ввода данных	14
20	Перфолента	14

t и d представляют собой просто файлы. Они не создаются автоматически при запуске эмулятора. Необходимые задержки возможно реализовать таймерами.

4a ⟨Внутренние функции и переменные класса машины 4a⟩≡ (3d) 5a▷
```
fstream* tape[8];
fstream* disk[8];
```

4b ⟨Константы класса машины 3c⟩+≡ (3d) ◁3c 8b▷
```
static const int IO_BYTES_PER_WORD = 6;    // 5 байт и знак
static const int TAPE_BLOCK_SIZE = 100;
static const int PRINTER_BLOCK_SIZE = 24;
```

4c ⟨Инициализация устройств ввода/вывода 4c⟩≡ (7d)
```
std::_Ios_Openmode mode = ios::in | ios::out | ios::binary;
tape[0] = new fstream("tape0",mode); tape[1] = new fstream("tape1",mode);
tape[2] = new fstream("tape2",mode); tape[3] = new fstream("tape3",mode);
tape[4] = new fstream("tape4",mode); tape[5] = new fstream("tape5",mode);
tape[6] = new fstream("tape6",mode); tape[7] = new fstream("tape7",mode);
disk[0] = new fstream("disk0",mode); disk[1] = new fstream("disk1",mode);
disk[2] = new fstream("disk2",mode); disk[3] = new fstream("disk3",mode);
disk[4] = new fstream("disk4",mode); disk[5] = new fstream("disk5",mode);
disk[6] = new fstream("disk6",mode); disk[7] = new fstream("disk7",mode);
```

4d ⟨Освобождение устройств ввода/вывода 4d⟩≡ (7d)
```
delete tape[0]; delete tape[1]; delete tape[2]; delete tape[3];
delete tape[4]; delete tape[5]; delete tape[6]; delete tape[7];
delete disk[0]; delete disk[1]; delete disk[2]; delete disk[3];
delete disk[4]; delete disk[5]; delete disk[6]; delete disk[7];
```

2.3 Эмулятор

Самая главная функция эмулятора — выполнить одну операцию MIX-машины. Так как программного счётчика у нас нет, то вроде бы и неясно, откуда брать саму операцию. Пусть слово, содержащее операцию, будет передаваться как параметр. Функция возвращает *NO_ADDR*, если в результате выполнения операции никаких предположений о следующей операции нет, *M_ADDR*, если адрес следующей операции является эффективным адресом текущей команды (от эмулятора требуется занести в регистр *J* адрес текущей команды) или *M_NOJ_ADDR* - то же, но регистр *J* трогать не нужно. Особый случай — останов по инструкции HLT, в этом случае возвращается значение *STOP_ADDR*.

5a ⟨Внутренние функции и переменные класса машины 4a⟩+≡ (3d) ◁4a 5d▷
```
eOpReturnCode nextOpAddr;
```
Uses eOpReturnCode.

5b ⟨Функции класса машины 5b⟩≡ (3d) 7c▷
```
eOpReturnCode doOp(mix_word op);
```
Uses doOp, eOpReturnCode, and mix_word.

5c ⟨Реализация функций машины MIX 5c⟩≡ (7b) 6e▷
```
mix::eOpReturnCode mix::doOp(mix_word op){
  nextOpAddr = NO_ADDR;
  ⟨Разбор слова операции 6a⟩
  ⟨Выполнение операции 6f⟩
  return nextOpAddr;
}
```
Defines:
doOp, used in chunks 5b, 12c, and 20c.
Uses eOpReturnCode, mix, and mix_word.

Выделяем из слова операции составляющие и запоминаем их в отдельных переменных, чтобы потом было проще писать функции выполнения операций.

Вот структура слова операции MIX:

Номер байта	0	1	2	3	4	5
Назначение	±	A	A	I	F	C

Здесь:

AA константа, обычно адрес;

I номер индексного регистра;

F модификация кода операции/спецификация нужного поля;

C собственно код операции.

Помещаю их в отдельные переменные

5d ⟨Внутренние функции и переменные класса машины 4a⟩+≡ (3d) ◁5a 6c▷
```
int I,F,C,AA,M,Sign;
```

6a ⟨Разбор слова операции 6a⟩≡ (5c) 6b▷
```
Sign = op.sign;
I = op.b3;
F = op.b4;
C = op.b5;
AA = (op.b1≪6) | op.b2;
AA = op.sign?-AA:AA;
```

В MIX для каждой операции осуществляется индексирование, то есть адрес *AA*, указанный непосредственно в слове, складывается с содержимым индексного регистра *I*. Кнут[1] обозначает результат этого действия как *M*, пусть будет так.

6b ⟨Разбор слова операции 6a⟩+≡ (5c) ◁6a
```
M = rI[I].getInt() + AA;
```
Uses getInt.

Теперь, когда параметры подготовлены, переходим непосредственно к выполнению. Адреса всех функций, выполняющих необходимые манипуляции с памятью и регистрами машины MIX, собраны в таблицу. Индексом в этой таблице служит код операции.

Функции, реализующие операции машины MIX, я сделал статическими членами класса машины по той простой причине, что не могу сообразить, как сделать массив адресов нестатических функций. По этой причине пришлось также сделать их (эти функции) принимающими один параметр — адрес экземпляра машины MIX.

6c ⟨Внутренние функции и переменные класса машины 4a⟩+≡ (3d) ◁5d 6d▷
```
typedef void (*OP_FUNC)(mix*);
static OP_FUNC ops[64];
```
Defines:
ops, used in chunks 6f and 7a.
Uses mix.

Вот пример функции, реализующей операцию NOP:

6d ⟨Внутренние функции и переменные класса машины 4a⟩+≡ (3d) ◁6c 23c▷
```
static void op_nop(mix* ptr);
```
Uses mix.

6e ⟨Реализация функций машины MIX 5c⟩+≡ (7b) ◁5c 6g▷
```
void mix::op_nop(mix* ptr){  /* ничего не делаем */  }
```
Uses mix.

Таким образом, выполнение операции просто сводится к следующему:

6f ⟨Выполнение операции 6f⟩≡ (5c)
```
ops[C](this);
```
Uses ops.

6g ⟨Реализация функций машины MIX 5c⟩+≡ (7b) ◁6e 24a▷
⟨Инициализация массива операций 7a⟩

7a ⟨Инициализация массива операций 7a⟩≡ (6g)

```
mix::OP_FUNC mix::ops[64]={
mix::op_nop, mix::op_add, mix::op_sub, mix::op_mul, mix::op_div,mix::op_hlt, mix::op_sla,
mix::op_move, mix::op_lda, mix::op_ld1, mix::op_ld2, mix::op_ld3, mix::op_ld4, mix::op_ld5,
mix::op_ld6, mix::op_ldx, mix::op_ldan, mix::op_ld1n, mix::op_ld2n, mix::op_ld3n,
mix::op_ld4n, mix::op_ld5n, mix::op_ld6n, mix::op_ldxn, mix::op_sta, mix::op_st1,
mix::op_st2, mix::op_st3, mix::op_st4, mix::op_st5, mix::op_st6, mix::op_stx, mix::op_stj,
mix::op_stz, mix::op_jbus, mix::op_ioc, mix::op_in, mix::op_out, mix::op_jred, mix::op_jmp,
mix::op_ja, mix::op_ji, mix::op_ji, mix::op_ji, mix::op_ji, mix::op_ji, mix::op_ji,
mix::op_jx, mix::op_inca, mix::op_inc1, mix::op_inc1, mix::op_inc1, mix::op_inc1,
mix::op_inc1, mix::op_inc1, mix::op_incx, mix::op_cmpa, mix::op_cmp1, mix::op_cmp1,
mix::op_cmp1, mix::op_cmp1, mix::op_cmp1, mix::op_cmp1, mix::op_cmpx
};
```

Uses mix and ops.

2.4 Другие кусочки реализации машины

Здесь опишем оставшиеся функции класса **mix** и функции работы со словами **mix_word**.

7b ⟨mix.cpp 7b⟩≡

```
using namespace std;
#include <iostream>
#include <fstream>
#include "mix.h"
⟨Конструктор и деструктор 7d⟩
⟨Реализация операций со словом MIX 8c⟩
⟨Реализация функций машины MIX 5c⟩
```

Uses mix.

Конструктор и деструктор нужны, чтобы выделить и освободить реальную оперативную память под виртуальную память машины MIX — мне как-то не по душе, если память будет выделяться на стеке.

7c ⟨Функции класса машины 5b⟩+≡ (3d) ◁5b 8a▷

```
mix(void);
~mix(void);
```

Uses mix.

7d ⟨Конструктор и деструктор 7d⟩≡ (7b)

```
mix::mix(void){
 ram = new mix_word[RAM_SIZE];
 ⟨Инициализация специального индексного регистра 3e⟩
 ⟨Инициализация устройств ввода/вывода 4c⟩
}

mix::~mix(void){
 delete[] ram;
 ⟨Освобождение устройств ввода/вывода 4d⟩
}
```

Uses mix, mix_word, and RAM_SIZE.

Эта вспомогательная функция возвращает результат вычисления эффиктивного адреса команды.

8a ⟨Функции класса машины 5b⟩+≡ (3d) ◁7c
```
int getM(void) {return M;}
```

8b ⟨Константы класса машины 3c⟩+≡ (3d) ◁4b
```
enum eOpReturnCode {NO_ADDR,BAD_ADDR,M_ADDR,M_NOJ_ADDR,STOP_ADDR};
```
Defines:
eOpReturnCode, used in chunks 5 and 20.

Следующие фукции облегчают манипулирование словами MIX. Получение слова MIX из целого числа со знаком и наоборот:

8c ⟨Реализация операций со словом MIX 8c⟩≡ (7b) 8e▷
```
void mix_word::setInt(const int val){
  if(0 > val){
    asInt = -val;
    sign = 1;
  }else{
    asInt = val;
    sign = 0;
  }
}
int mix_word::getInt(void) const{
  return (sign?-(0x3fffffff & asInt):asInt);
}
```
Defines:
getInt, used in chunks 6b, 9, 10, 18, 19b, 27b, 28a, 32, 33a, 35, 37a, and 42c.
setInt, used in chunks 3e, 9, 10, 12c, 13a, 20, 27–31, 36b, and 42c.
Uses mix_word.

Получить абсолютное значение как слово MIX.

8d ⟨Шаблоны операций со словом MIX 8d⟩≡ (2) 8f▷
```
mix_word abs(void);
```
Uses abs and mix_word.

8e ⟨Реализация операций со словом MIX 8c⟩+≡ (7b) ◁8c 9a▷
```
mix_word mix_word::abs(void){
  mix_word t = *this;
  t.sign = 0;
  return t;
}
```
Defines:
abs, used in chunks 8–10, 26–28, 37a, and 40a.
Uses mix_word.

Получить поле слова MIX как целое слово. Аргумент fld представляет собой $l*8+r$, где l — номер левого поля, если $l\equiv 0$, то учитываем также и знак, иначе знак принимается +, r — номер правого поля.

8f ⟨Шаблоны операций со словом MIX 8d⟩+≡ (2) ◁8d 9d▷
```
mix_word getField(int fld);
```
Uses getField and mix_word.

9a ⟨Реализация операций со словом MIX 8c⟩+≡ (7b) ◁8e 9b▷
```
mix_word mix_word::getField(int fld){
  int tInt = std::abs(getInt()), l = (fld ≫ 3), r = (fld & 7);
  mix_word t;
  t.setInt(0);
  if( ¬l ∧ ¬r){
    t.sign = sign;
    return t;
  }
```
Defines:
getField, used in chunks 8f, 24, 26–28, 32, and 33a.
Uses abs, getInt, mix_word, and setInt.

Машина MIX не проверяет спецификацию поля на осмысленность, поэтому предполагаем её корректность, то есть $l \leq r$.

9b ⟨Реализация операций со словом MIX 8c⟩+≡ (7b) ◁9a 9c▷
```
  tInt ≫= 6∗(5 - r);
  if(¬l){
    t.setInt(tInt);
    t.sign = sign;
    return t;
  }
```
Uses setInt.

Нужно урезать левую часть.

9c ⟨Реализация операций со словом MIX 8c⟩+≡ (7b) ◁9b 9e▷
```
  tInt &= (0x3fffffff ≫ 6∗(4 - (r - l)));
  t.setInt(tInt);
  return t;
}
```
Uses setInt.

Записать часть слова MIX в другое слово. Меняются только поля, указанные в спецификации поля *fld*.

9d ⟨Шаблоны операций со словом MIX 8d⟩+≡ (2) ◁8f 10e▷
```
void setField(mix_word src,int fld);
```
Uses mix_word and setField.

9e ⟨Реализация операций со словом MIX 8c⟩+≡ (7b) ◁9c 10a▷
```
void mix_word::setField(mix_word src,int fld){
  int tInt = std::abs(src.getInt()), l = (fld ≫ 3), r = (fld & 7);
  if(¬l ∧ ¬r){
    sign = src.sign;
    return;
  }
```
Defines:
setField, used in chunks 9d, 25, and 26.
Uses abs, getInt, and mix_word.

Запоминаем знак слова назначения и меняем его при необходимости.

10a ⟨Реализация операций со словом MIX 8c⟩+≡ (7b) ◁9e 10b▷

```
int new_sign = sign;
if(¬l){
  new_sign = src.sign;
  ++l;
 }
int mask = 0x3fffffff ≫ 6 ∗ (4 - (r - l));
tInt &= mask;
tInt ≪= 6 ∗ (5 - r);
mask ≪= 6 ∗ (5 - r);
mask = ∼mask & 0x3fffffff;
setInt((std::abs(getInt()) & (mask)) | tInt );
sign = new_sign;
}
```

Uses abs, getInt, and setInt.

Сложение двух слов MIX. В случае переполнения генерируем исключение.

10b ⟨Реализация операций со словом MIX 8c⟩+≡ (7b) ◁10a 10c▷

```
const mix_word operator+(const mix_word& a,const mix_word& b){
  mix_word t;
  int i = a.getInt() + b.getInt();
```

Defines:
operator+, used in chunk 3a.
Uses getInt and mix_word.

При переполнении считаем, что регистр A имеет слева лишние разряды, куда происходит перенос. Поэтому генерируя исключение передаём также правые разряды результата сложения.

10c ⟨Реализация операций со словом MIX 8c⟩+≡ (7b) ◁10b

```
  if(0x40000000 & abs(i)){
    t.setInt(0x3fffffff & abs(i));
    t.sign = a.sign;
    throw mix_word_exception(mix_word::E_OVER,t);
   }
  t.setInt(i);
  return t;
 }
```

Uses abs, mix_word, and setInt.

10d ⟨mix.h 10d⟩≡

```
#ifndef MIX_H
#define MIX_H
#include <iostream>
 ⟨Слово MIX 2⟩
 ⟨Класс машины MIX 3d⟩
#endif
```

10e ⟨Шаблоны операций со словом MIX 8d⟩+≡ (2) ◁9d

```
void setInt(int val);
int getInt(void) const;
```

Uses getInt and setInt.

3 Основная часть программы

Программа исполняет команды оператора в цикле. Если был указано имя файла при запуске программы, то этот файл загружается и исполняется как список команд оператора.

11a ⟨mix-emu.cpp 11a⟩≡

```
⟨Подключение заголовочных файлов 23a⟩
⟨Свои типы 13b⟩
⟨Константы 11b⟩
⟨Прототипы функций 11c⟩
// экземпляр машины
mix emu;
// программа
int main(int argc, char* argv[]){
 optind = 0;
 int nextOpt;
 do{
  nextOpt = getopt_long(argc,argv,short_options,long_options,NULL);
  if('h' ≡ nextOpt ∨ '?' ≡ nextOpt){
   cerr ≪ "Usage: " ≪ argv[0] ≪ " [filename]" ≪ endl;
   return 1;
  }
 } while(-1 ≠ nextOpt);
 // Если задано имя файла
 if(optind ≠ argc){
  ifstream input(argv[optind]);
  if(input.fail()){ cerr ≪ "Can't open `" ≪ argv[optind] ≪ "'" ≪ endl; return 2;}
  mainCycle(input);
 }
 cout ≪ "MIX emulator v 0.1 Copyright (c) Yellow Rabbit 2008." ≪ endl;
 mainCycle(cin);
 return 0;
}
⟨Обработка потока команд 12a⟩
⟨Вспомогательные функции 13c⟩
```

Uses emu, main, mainCycle, and mix.

Структуры для обработки опций командной строки.

11b ⟨Константы 11b⟩≡ (11a) 14b▷

```
const char* const short_options = "h";
const struct option long_options[]={
 {"help",no_argument,NULL,0},{NULL,0,0,0}
};
```

Defines:
emu, used in chunks 11–13 and 18–20.
main, used in chunk 11a.

Выполняются команды. считываемые из заданного потока.

11c ⟨Прототипы функций 11c⟩≡ (11a) 17b▷

```
void mainCycle(istream& input);
```

Uses mainCycle.

12a ⟨Обработка потока команд 12a⟩≡ (11a) 12b▷

```
void mainCycle(istream& input){
  eCmd cmd;
  mix_word mix_op;
  int ip = 0;    // текущий адрес операции (псевдопрограммный счётчик)
  // Части команды оператора
  int op,sign,aa,idx,fld,param0,param1;

  while(CMD_END ≠ (cmd = getCommand(input,op,sign,aa,idx,fld,param0,param1))){
   switch(cmd){
    case CMD_QUIT: exit(0);
    case CMD_STEP: step(ip); break;
    case CMD_DUMPREG: dumpReg(); break;
    case CMD_DUMPMEM: dumpMemory(param0,param1); break;
    case CMD_EDITMEM: editMem(param0,param1); break;
```

Uses dumpMemory, dumpReg, eCmd, editMem, getCommand, mainCycle, mix_word, and step.

Запуск на выполнение программы с указанного адреса.

12b ⟨Обработка потока команд 12a⟩+≡ (11a) ◁12a 12c▷

```
    case CMD_RUN: run(param0,param1,ip); break;
```

Uses run.

Нужно выполнить операцию машины, поэтому собираем слово MIX, как если бы оно было прочитано из памяти.

12c ⟨Обработка потока команд 12a⟩+≡ (11a) ◁12b 13a▷

```
    case CMD_DOOP:
     mix_op.setInt(aa ≪ 18);
     mix_op.sign = sign;
     mix_op.b5 = op; mix_op.b4 = fld; mix_op.b3 = idx;
     if(mix::M_ADDR ≡ emu.doOp(mix_op)){
      cout ≪ "next addr :" ≪ emu.getM() ≪ endl;
     }
     break;
```

Uses doOp, emu, mix, and setInt.

Ассемблирование одной операции. Записываем получившийся код машины MIX в память по указанному адресу.

13a ⟨Обработка потока команд 12a⟩+≡ (11a) ◁ 12c

```
      case CMD_ASM:
        {
          int addr = param0;
          cmd = getCommand(input,op,sign,aa,idx,fld,param0,param1);
          mix_op.setInt(aa ≪ 18);
          mix_op.sign = sign;
          mix_op.b5 = op; mix_op.b4 = fld; mix_op.b3 = idx;
          emu.ram[addr] = mix_op;
          break;
        }
      default: cout ≪ cmd ≪ "'" ≪ param0 ≪ "'" ≪ param1 ≪ endl;
    }
  }
}
```

Defines:
`mainCycle`, used in chunks 11 and 12a.
Uses `emu`, `getCommand`, and `setInt`.

13b ⟨Свои типы 13b⟩≡ (11a)

```
⟨Коды возврата getCommand 17a⟩
⟨Словарная статья 14a⟩
```

13c ⟨Вспомогательные функции 13c⟩≡ (11a)

```
⟨Получить команду 15a⟩
⟨Найти команду 14c⟩
⟨Реализация команд эмулятора 18⟩
```

4 Команды эмулятора

4.1 Разбор ввода оператора

Эмулятор понимает несколько команд, которые читаются из стандартного ввода. Кроме команд собственно эмулятора, таких как просмотр памяти, регистров и т.д., распознаются мнемоники операций машины MIX.

Все возможные команды храняться в словаре. Словарная статья состоит из имени команды (в нижнем регистре), кода команды, числа параметров и кода операции. Код операции равен -1 для команд эмулятору и полю *C* для мнемоник MIX.

14a ⟨Словарная статья 14a⟩≡ (13b)
```
struct vocEntry{
  string name;
  eCmd code;
  int argc;
  int op;
};
```
Defines:
  `vocEntry`, used in chunk 14.
Uses `eCmd`.

Словарь представляет собой массив статей.

14b ⟨Константы 11b⟩+≡ (11a) ◁11b
```
static vocEntry vocabulary[]={
  {"quit",CMD_QUIT,0,-1},
  {"dr",CMD_DUMPREG,0,-1},
  ⟨Команды оператора 21c⟩
  ⟨Мнемоники операций MIX 22⟩
};
```
Defines:
  `vocabulary`, used in chunks 14c and 15b.
Uses `vocEntry`.

Просто пробегаем по всему словарю и справниваем имена команд с переданной строкой. Никакой оптимизации не предусмотрено — число команд невелико. Функция возвращает индекс найденной словарной статьи или -1, если статья не найдена.

14c ⟨Найти команду 14c⟩≡ (13c)
```
int findCommand(string sCmd){
  for(int i = 0; i < (sizeof(vocabulary)÷sizeof(vocEntry)); ++i)
    if(sCmd ≡ vocabulary[i].name) return i;
  return -1;
}
```
Defines:
  `findCommand`, used in chunks 15b and 17b.
Uses `vocabulary` and `vocEntry`.

Выбираем из входного потока команду и пытаемся разобрать её на составляющие.

15a ⟨Получить команду 15a⟩≡ (13c) 15b▷

```
eCmd getCommand(istream& input,int& op,int& sign,int& aa,int& idx,int& fld,
                int& param0,int& param1){
  string sCmd;
  input ≫ sCmd;
  // приводим к нижнему регистру
  for (int i = 0; i < sCmd.length(); ++i)
    sCmd[i] = tolower(sCmd[i]);
```

Defines:
getCommand, used in chunks 12a, 13a, and 17b.
Uses eCmd.

Ищем команду в словаре, если нашли, то проверям код операции — если он отличен от -1, то имеем дело с мнемоникой и нужно её разбирать. Если же это команда эмулятору, то считываем её параметры.

15b ⟨Получить команду 15a⟩+≡ (13c) ◁15a

```
  int iVoc;
  if(-1 ≠ (iVoc = findCommand(sCmd))){
    if(-1 ≡ vocabulary[iVoc].op){
      switch(vocabulary[iVoc].argc){
        case 2: input ≫ param0 ≫ param1; break;
        case 1: input ≫ param0; break;
        default: break;
      }
    }else{
      ⟨Разбираем параметры мнемоники 16⟩
      op = vocabulary[iVoc].op;
    }
    return vocabulary[iVoc].code;
  }
  return CMD_ERROR;
}
```

Uses findCommand and vocabulary.

Мнемонка имеет формат `OP [-]ADDRESS,I(L:R)`, все поля обязательны к указанию даже для команд, не имеющих параметров (например NOP). *isBuf* нужен мне для перевода строки в число, переменная *sLine* урезается по мере разбора параметров мнемоники.

16 ⟨Разбираем параметры мнемоники 16⟩≡ (15b)

```
string sLine;
istringstream isBuf;
int pos;
input ≫ sLine ;
// знак
sign = 0;
if('-' ≡ sLine[0]){
  sLine[0] = '+';
  sign = 1;
}
// ADDRESS (aa)
pos = sLine.find(',');
isBuf.str(sLine.substr(0,pos));
isBuf ≫ aa;
sLine.erase(0,1 + pos);
// I (idx)
pos = sLine.find('(');
isBuf.clear();
isBuf.str(sLine.substr(0,pos));
isBuf ≫ idx;
sLine.erase(0,1 + pos);
// F L:R (fld)
int l,r;
pos = sLine.find(':');
isBuf.clear();
isBuf.str(sLine.substr(0,pos));
isBuf ≫ l;
sLine.erase(0,1 + pos);
pos = sLine.find(')');
isBuf.clear();
isBuf.str(sLine.substr(0,pos));
isBuf ≫ r;
sLine.erase(0,1 + pos);
fld = 8 ∗ l + r;
```

17a ⟨Коды возврата getCommand 17a⟩≡ (13b)

```
enum eCmd{
  CMD_ERROR,    // ошибочная команда
  CMD_QUIT,     // выход из программы
  CMD_DUMPREG,    // показать регистры
  CMD_DUMPMEM,    // показать область памяти
  CMD_DOOP,     // выполнить операцию MIX
  CMD_ASM,    // ассемблировать команду по адресу
  CMD_RUN,    // выполнить программу
  CMD_EDITMEM,    // записать число в память
  CMD_STEP,    // выполнить одну операцию MIX
  CMD_END    // конец входного файла
};
```

Defines:
eCmd, used in chunks 12a, 14a, 15a, and 17b.

17b ⟨Прототипы функций 11c⟩+≡ (11a) ◁ 11c 19a ▷

```
eCmd getCommand(istream& input,int& op,int& sign,int& aa,int& idx,int& fld,
        int& param0,int& param1);
int findCommand(string sCmd);
```

Uses eCmd, findCommand, and getCommand.

4.2 Список команд оператора

dr Вывести на экран содержимое всех регистров машины MIX.

18 ⟨Реализация команд эмулятора 18⟩≡ (13c) 19b ▷

```
void dumpReg(void){
  // A
  cout << "A  :  " << setfill(' ') << setw(11) << emu.rA.getInt() << "  |"
       << "  " << (emu.rA.sign?"-":"+")
       << "  " << setfill('0') << setw(2) << emu.rA.b1
       << "  " << setfill('0') << setw(2) << emu.rA.b2
       << "  " << setfill('0') << setw(2) << emu.rA.b3
       << "  " << setfill('0') << setw(2) << emu.rA.b4
       << "  " << setfill('0') << setw(2) << emu.rA.b5 << endl;
  // X
  cout << "X  :  " << setfill(' ') << setw(11) << emu.rX.getInt() << "  |"
       << "  " << (emu.rX.sign?"-":"+")
       << "  " << setfill('0') << setw(2) << emu.rX.b1
       << "  " << setfill('0') << setw(2) << emu.rX.b2
       << "  " << setfill('0') << setw(2) << emu.rX.b3
       << "  " << setfill('0') << setw(2) << emu.rX.b4
       << "  " << setfill('0') << setw(2) << emu.rX.b5 << endl;
  // J
  cout << "J  :  " << setfill(' ') << setw(11) << emu.rJ.getInt() << "  |"
       << "  " << (emu.rJ.sign?"-":"+")
       << "  " << setfill('0') << setw(2) << emu.rJ.b1
       << "  " << setfill('0') << setw(2) << emu.rJ.b2
       << "  " << setfill('0') << setw(2) << emu.rJ.b3
       << "  " << setfill('0') << setw(2) << emu.rJ.b4
       << "  " << setfill('0') << setw(2) << emu.rJ.b5 << endl;
  // I1-6
  for(int i = 1; i <= 6; ++i){
    cout << "I" << i << ":  "
         << setfill(' ') << setw(11) << emu.rI[i].getInt() << "  |"
         << "  " << (emu.rI[i].sign?"-":"+")
         << "  " << setfill('0') << setw(2) << emu.rI[i].b4
         << "  " << setfill('0') << setw(2) << emu.rI[i].b5 << endl;
  }
  // Флаги
  cout << "C  :";
  switch(emu.flagC){
    case mix::less: cout << "Less"; break;
    case mix::equal: cout << "Equal"; break;
    case mix::greater: cout << "Greater"; break;
  }
  cout << "   V  :";
  if(emu.flagV) cout << "True";
  else cout << "False";
```

```
  cout ≪ endl;
 }
```

Defines:
dumpReg, used in chunks 12a, 19a, and 21b.
Uses emu, getInt, and mix.

19a ⟨Прототипы функций 11c⟩+≡ (11a) ◁17b 19c▷

```
 void dumpReg(void);
```

Uses dumpReg.

dm <addr> <n> Вывести на экран содержимое <n> слов памяти машины MIX начиная с ячейки <addr>.

19b ⟨Реализация команд эмулятора 18⟩+≡ (13c) ◁18 20a▷

```
 void dumpMemory(int addr, int count){
  while(0 < count--){
   cout ≪ setfill(' ') ≪ setw(5) ≪ addr ≪ " :"
       ≪ setfill(' ') ≪ setw(11) ≪ emu.ram[addr].getInt() ≪ " |"
       ≪ " " ≪ (emu.ram[addr].sign?"-":"+")
       ≪ " " ≪ setfill('0') ≪ setw(2) ≪ emu.ram[addr].b1
       ≪ " " ≪ setfill('0') ≪ setw(2) ≪ emu.ram[addr].b2
       ≪ " " ≪ setfill('0') ≪ setw(2) ≪ emu.ram[addr].b3
       ≪ " " ≪ setfill('0') ≪ setw(2) ≪ emu.ram[addr].b4
      ≪ " " ≪ setfill('0') ≪ setw(2) ≪ emu.ram[addr].b5 ≪ endl;
   ++addr;
  }
 }
```

Defines:
dumpMemory, used in chunks 12a and 19c.
Uses emu and getInt.

19c ⟨Прототипы функций 11c⟩+≡ (11a) ◁19a 19d▷

```
 void dumpMemory(int addr, int count);
```

Uses dumpMemory.

run <addr0> <addr1> Выполняем программу с адреса <addr0>, останов по операции HLT или по достижении <addr1>.

19d ⟨Прототипы функций 11c⟩+≡ (11a) ◁19c 20b▷

```
 void run(int addr, int stopAddr, int& ip);
```

Uses run.

20a ⟨Реализация команд эмулятора 18⟩+≡ (13c) ◁19b 20c▷

```
void run(int addr, int stopAddr, int& ip){
  mix_word op;
  while(stopAddr ≠ addr){
    if(mix::STOP_ADDR ≡ oneOperation(addr)){
      cout ≪ endl ≪ "-hlt at " ≪ addr ≪ endl; addr = stopAddr; break;
      break;
    }
  }
  ip = addr;
}
```

Defines:
run, used in chunks 12b, 19d, and 21c.
Uses mix, mix_word, and oneOperation.

Вспомогательная функция, выполняет одну операцию по адресу *ip*.

20b ⟨Прототипы функций 11c⟩+≡ (11a) ◁19d 20d▷

```
mix::eOpReturnCode oneOperation(int& ip);
```

Uses eOpReturnCode, mix, and oneOperation.

20c ⟨Реализация команд эмулятора 18⟩+≡ (13c) ◁20a 20e▷

```
mix::eOpReturnCode oneOperation(int& ip){
  mix::eOpReturnCode code = emu.doOp(emu.ram[ip]);
  switch(code){
    case mix::NO_ADDR: ++ip; break;
    case mix::M_ADDR: emu.rJ.setInt(1 + ip);
    case mix::M_NOJ_ADDR: ip = emu.getM(); break;
  }
  return code;
}
```

Defines:
oneOperation, used in chunks 20 and 21b.
Uses doOp, emu, eOpReturnCode, mix, and setInt.

em <addr> <n> Пишем число <n> в ячейку памяти по адресу <addr>.

20d ⟨Прототипы функций 11c⟩+≡ (11a) ◁20b 21a▷

```
void editMem(int addr, int value);
```

Uses editMem.

20e ⟨Реализация команд эмулятора 18⟩+≡ (13c) ◁20c 21b▷

```
void editMem(int addr, int value){
  emu.ram[addr].setInt(value);
}
```

Defines:
editMem, used in chunks 12a and 20d.
Uses emu and setInt.

t Выполнить одну операцию машины MIX, адрес которой находится в *ip*. После выполнения вывести содержимое регистров и скорректированный адрес.

21a ⟨Прототипы функций 11c⟩+≡ (11a) ◁20d
```
void step(int& ip);
```
Uses `step`.

21b ⟨Реализация команд эмулятора 18⟩+≡ (13c) ◁20e
```
void step(int& ip){
  oneOperation(ip);
  dumpReg();
  cout ≪ "ip : " ≪ ip ≪ endl;
}
```
Defines:
`step`, used in chunks 12a and 21a.
Uses `dumpReg` and `oneOperation`.

asm <addr> <mnem> Скомпилировать одну операцию машины MIX <mnem> по адресу <addr>.

quit Выйти из эмулятора.

4.3 Словарь команд и операций

В словарной статье указываем имя команды, число параметров и константу -1 в качестве кода операции.

21c ⟨Команды оператора 21c⟩≡ (14b)
```
{"dm",CMD_DUMPMEM,2,-1},
{":q",CMD_QUIT,0,-1},
{"asm",CMD_ASM,1,-1},
{"run",CMD_RUN,2,-1},
{"em",CMD_EDITMEM,2,-1},
{"t",CMD_STEP,0,-1},
{"end",CMD_END,0,-1},
```
Uses `run`.

В словарных статьях для мнемоник машины MIX указывается код операции или, другими словами, число в поле *C*.

22 ⟨Мнемоники операций MIX 22⟩≡ (14b)

```
#define MNEM(name,op_code){name,CMD_DOOP,0,op_code}
 MNEM("nop",0),MNEM("add",1),MNEM("sub",2),MNEM("mul",3),
 MNEM("div",4),MNEM("hlt",5),MNEM("num",5),MNEM("char",5),
 MNEM("sla",6),MNEM("sra",6),MNEM("slax",6),MNEM("srax",6),
 MNEM("slc",6),MNEM("src",6),MNEM("move",7),MNEM("lda",8),
 MNEM("ld1",9),MNEM("ld2",10),MNEM("ld3",11),MNEM("ld4",12),
 MNEM("ld5",13),MNEM("ld6",14),MNEM("ldx",15),MNEM("ldan",16),
 MNEM("ld1n",17),MNEM("ld2n",18),MNEM("ld3n",19),MNEM("ld4n",20),
 MNEM("ld5n",21),MNEM("ld6n",22),MNEM("ldxn",23),MNEM("sta",24),
 MNEM("st1",25),MNEM("st2",26),MNEM("st3",27),MNEM("st4",28),
 MNEM("st5",29),MNEM("st6",30),MNEM("stx",31),MNEM("stj",32),
 MNEM("stz",33),MNEM("jbus",34),MNEM("ioc",35),MNEM("in",36),
 MNEM("out",37),MNEM("jred",38),MNEM("jmp",39),MNEM("jsj",39),
 MNEM("jov",39),MNEM("jnov",39),MNEM("jl",39),MNEM("je",39),
 MNEM("jg",39),MNEM("jge",39),MNEM("jne",39),MNEM("jle",39),
 MNEM("jan",40),MNEM("jaz",40),MNEM("jap",40),MNEM("jann",40),
 MNEM("janz",40),MNEM("janp",40),MNEM("j1n",41),MNEM("j1z",41),
 MNEM("j1p",41),MNEM("j1nn",41),MNEM("j1nz",41),MNEM("j1np",41),
 MNEM("j2n",42),MNEM("j2z",42),MNEM("j2p",42),MNEM("j2nn",42),
 MNEM("j2nz",42),MNEM("j2np",42),MNEM("j3n",43),MNEM("j3z",43),
 MNEM("j3p",43),MNEM("j3nn",43),MNEM("j3nz",43),MNEM("j3np",43),
 MNEM("j4n",44),MNEM("j4z",44),MNEM("j4p",44),MNEM("j4nn",44),
 MNEM("j4nz",44),MNEM("j4np",44),MNEM("j5n",45),MNEM("j5z",45),
 MNEM("j5p",45),MNEM("j5nn",45),MNEM("j5nz",45),MNEM("j5np",44),
 MNEM("j6n",46),MNEM("j6z",46),MNEM("j6p",46),MNEM("j6nn",46),
 MNEM("j6nz",46),MNEM("j6np",46),MNEM("jxn",47),MNEM("jxz",47),
 MNEM("jxp",47),MNEM("jxnn",47),MNEM("jxnz",47),MNEM("jxnp",47),
 MNEM("inca",48),MNEM("deca",48),MNEM("enta",48),MNEM("enna",48),
 MNEM("inc1",49),MNEM("dec1",49),MNEM("ent1",49),MNEM("enn1",49),
 MNEM("inc2",50),MNEM("dec2",50),MNEM("ent2",50),MNEM("enn2",50),
 MNEM("inc3",51),MNEM("dec3",51),MNEM("ent3",51),MNEM("enn3",51),
 MNEM("inc4",52),MNEM("dec4",52),MNEM("ent4",52),MNEM("enn4",52),
 MNEM("inc5",53),MNEM("dec5",53),MNEM("ent5",53),MNEM("enn5",53),
 MNEM("inc6",54),MNEM("dec6",54),MNEM("ent6",54),MNEM("enn6",54),
 MNEM("incx",55),MNEM("decx",55),MNEM("entx",55),MNEM("ennx",55),
 MNEM("cmpa",56),MNEM("cmp1",57),MNEM("cmp2",58),MNEM("cmp3",59),
 MNEM("cmp4",60),MNEM("cmp5",61),MNEM("cmp6",62),MNEM("cmpx",63)
```

5 Файлы заголовков

23a ⟨Подключение заголовочных файлов 23a⟩≡ (11a) 23b▷

```
using namespace std;
#include <fstream>
#include <iostream>
#include <sstream>
#include <iomanip>
#include <string>
#include <getopt.h>
```

Нам нужны шаблоны и своих классов

23b ⟨Подключение заголовочных файлов 23a⟩+≡ (11a) ◁23a

```
#include "mix.h"
```

Uses mix.

6 Приложения

6.1 Прототипы функций, реализующих операции машины MIX

23c ⟨Внутренние функции и переменные класса машины 4a⟩+≡ (3d) ◁6d 39a▷

```
static void op_add(mix*);static void op_sub(mix*);static void op_mul(mix*);
static void op_div(mix*);static void op_hlt(mix*);static void op_sla(mix*);
static void op_move(mix*);static void op_lda(mix*);static void op_ld1(mix*);
static void op_ld2(mix*);static void op_ld3(mix*);static void op_ld4(mix*);
static void op_ld5(mix*);static void op_ld6(mix*);static void op_ldx(mix*);
static void op_ldan(mix*);static void op_ld1n(mix*);static void op_ld2n(mix*);
static void op_ld3n(mix*);static void op_ld4n(mix*);static void op_ld5n(mix*);
static void op_ld6n(mix*);static void op_ldxn(mix*);static void op_sta(mix*);
static void op_st1(mix*);static void op_st2(mix*);static void op_st3(mix*);
static void op_st4(mix*);static void op_st5(mix*);static void op_st6(mix*);
static void op_stx(mix*);static void op_stj(mix*);static void op_stz(mix*);
static void op_jbus(mix*);static void op_ioc(mix*);static void op_in(mix*);
static void op_out(mix*);static void op_jred(mix*);static void op_jmp(mix*);
static void op_ja(mix*);static void op_ji(mix*);
static void op_jx(mix*);static void op_inca(mix*);
static void op_inc1(mix*);static void op_incx(mix*);static void op_cmpa(mix*);
static void op_cmp1(mix*);static void op_cmpx(mix*);
```

Uses mix.

6.2 Функции, реализующие операции машины MIX

6.2.1 Команды загрузки

LDA Содержимое *A* заменяется на содержимое *M*. Знак используется, если он является частью поля *F*, в противном случае используется +. По мере загрузки содержимое сдвигается в правую часть регистра.

24a ⟨Реализация функций машины MIX 5c⟩+≡ (7b) ◁6g 24b▷

```
void mix::op_lda(mix* ptr){
 ptr→rA = ptr→ram[ptr→M].getField(ptr→F);
 }
```

Uses `getField` and `mix`.

LDX Идентична *LDA*, только загружается регистр *X*.

24b ⟨Реализация функций машины MIX 5c⟩+≡ (7b) ◁24a 24c▷

```
void mix::op_ldx(mix* ptr){
 ptr→rX = ptr→ram[ptr→M].getField(ptr→F);
}
```

Uses `getField` and `mix`.

LD1 – LD6 Загрузить индексный регистр. Учитываем, что для индексного регистра байты 1, 2 и 3 всегда равны 0.

24c ⟨Реализация функций машины MIX 5c⟩+≡ (7b) ◁24b 25a▷

```
void mix::op_ld1(mix* ptr){
 ptr→rI[1] = ptr→ram[ptr→M].getField(ptr→F);
 ptr→rI[1].b1 = ptr→rI[1].b2 = ptr→rI[1].b3 = 0;
}
void mix::op_ld2(mix* ptr){
 ptr→rI[2] = ptr→ram[ptr→M].getField(ptr→F);
 ptr→rI[2].b1 = ptr→rI[2].b2 = ptr→rI[2].b3 = 0;
}
void mix::op_ld3(mix* ptr){
 ptr→rI[3] = ptr→ram[ptr→M].getField(ptr→F);
 ptr→rI[3].b1 = ptr→rI[3].b2 = ptr→rI[3].b3 = 0;
}
void mix::op_ld4(mix* ptr){
 ptr→rI[4] = ptr→ram[ptr→M].getField(ptr→F);
 ptr→rI[4].b1 = ptr→rI[4].b2 = ptr→rI[4].b3 = 0;
}
void mix::op_ld5(mix* ptr){
 ptr→rI[5] = ptr→ram[ptr→M].getField(ptr→F);
 ptr→rI[5].b1 = ptr→rI[5].b2 = ptr→rI[5].b3 = 0;
}
void mix::op_ld6(mix* ptr){
 ptr→rI[6] = ptr→ram[ptr→M].getField(ptr→F);
 ptr→rI[6].b1 = ptr→rI[6].b2 = ptr→rI[6].b3 = 0;
}
```

Uses `getField` and `mix`.

LDAN Загрузить регистр *A* с обратным знаком.

25a ⟨Реализация функций машины MIX 5c⟩+≡ (7b) ◁24c 25b▷

```
void mix::op_ldan(mix* ptr){
  op_lda(ptr);
  ptr→rA.sign = 1 - ptr→rA.sign;
}
```

Uses mix.

LDXN Аналогично *LDAN*, только загружается регистр *X*.

25b ⟨Реализация функций машины MIX 5c⟩+≡ (7b) ◁25a 25c▷

```
void mix::op_ldxn(mix* ptr){
  op_ldx(ptr);
  ptr→rX.sign = 1 - ptr→rX.sign;
}
```

Uses mix.

LD1N – LD6N С обратным знаком загружаем индексные регистры.

25c ⟨Реализация функций машины MIX 5c⟩+≡ (7b) ◁25b 25d▷

```
void mix::op_ld1n(mix* ptr){ op_ld1(ptr); ptr→rI[1].sign = 1 - ptr→rI[1].sign;}
void mix::op_ld2n(mix* ptr){ op_ld2(ptr); ptr→rI[2].sign = 1 - ptr→rI[2].sign;}
void mix::op_ld3n(mix* ptr){ op_ld3(ptr); ptr→rI[3].sign = 1 - ptr→rI[3].sign;}
void mix::op_ld4n(mix* ptr){ op_ld4(ptr); ptr→rI[4].sign = 1 - ptr→rI[4].sign;}
void mix::op_ld5n(mix* ptr){ op_ld5(ptr); ptr→rI[5].sign = 1 - ptr→rI[5].sign;}
void mix::op_ld6n(mix* ptr){ op_ld6(ptr); ptr→rI[6].sign = 1 - ptr→rI[6].sign;}
```

Uses mix.

6.2.2 Команды записи в память

STA Записываем регистр *A* в поле слова памяти. Знак трогаем только в том случае, если он является частью спецификации поля.

25d ⟨Реализация функций машины MIX 5c⟩+≡ (7b) ◁25c 25e▷

```
void mix::op_sta(mix* ptr){
  ptr→ram[ptr→M].setField(ptr→rA,ptr→F);
}
```

Uses mix and setField.

STX Идентична *STX*, только выгружается регистр *X*.

25e ⟨Реализация функций машины MIX 5c⟩+≡ (7b) ◁25d 26a▷

```
void mix::op_stx(mix* ptr){
  ptr→ram[ptr→M].setField(ptr→rX,ptr→F);
}
```

Uses mix and setField.

STJ Сохранить регистр *J*, знак всегда будет +.

26a ⟨Реализация функций машины MIX 5c⟩+≡ (7b) ◁25e 26b▷

```
void mix::op_stj(mix* ptr){
  ptr→ram[ptr→M].setField(ptr→rJ.abs(),ptr→F);
}
```

Uses abs, mix, and setField.

STZ Записать в память нуль со знаком +. Используем индексный регистр 0, благо он всегда пуст.

26b ⟨Реализация функций машины MIX 5c⟩+≡ (7b) ◁26a 26c▷

```
void mix::op_stz(mix* ptr){
  ptr→ram[ptr→M].setField(ptr→rI[0],ptr→F);
}
```

Uses mix and setField.

ST1 – ST6 Выгрузить индексный регистр. Учитываем, что для индексного регистра байты 1, 2 и 3 всегда равны 0.

26c ⟨Реализация функций машины MIX 5c⟩+≡ (7b) ◁26b 26d▷

```
void mix::op_st1(mix* ptr){ ptr→ram[ptr→M].setField(ptr→rI[1],ptr→F);}
void mix::op_st2(mix* ptr){ ptr→ram[ptr→M].setField(ptr→rI[2],ptr→F);}
void mix::op_st3(mix* ptr){ ptr→ram[ptr→M].setField(ptr→rI[3],ptr→F);}
void mix::op_st4(mix* ptr){ ptr→ram[ptr→M].setField(ptr→rI[4],ptr→F);}
void mix::op_st5(mix* ptr){ ptr→ram[ptr→M].setField(ptr→rI[5],ptr→F);}
void mix::op_st6(mix* ptr){ ptr→ram[ptr→M].setField(ptr→rI[6],ptr→F);}
```

Uses mix and setField.

6.2.3 Арифметические команды

ADD Содержимое поля слова памяти добавляется к содержимому регистра *A*. При переполнении устанавливается флажок.

26d ⟨Реализация функций машины MIX 5c⟩+≡ (7b) ◁26c 27a▷

```
void mix::op_add(mix* ptr){
  try{
    ptr→rA = ptr→rA + ptr→ram[ptr→M].getField(ptr→F);
  }catch(mix_word_exception e_mix){
    if(mix_word::E_OVER ≡ e_mix.code){
      ptr→rA = e_mix.val;
      ptr→flagV = true;
    }
  }
}
```

Uses getField, mix, and mix_word.

SUB Вычесть из регистра A содержимое памяти.

27a ⟨Реализация функций машины MIX 5c⟩+≡ (7b) ◁26d 27b▷

```
void mix::op_sub(mix* ptr){
  mix_word t;
  t = ptr→ram[ptr→M].getField(ptr→F);
  t.sign = 1 - t.sign;
  try{
    ptr→rA = ptr→rA + t;
  }catch(mix_word_exception e_mix){
    if(mix_word::E_OVER ≡ e_mix.code){
      ptr→rA = e_mix.val;
      ptr→flagV = true;
    }
  }
}
```

Uses `getField`, `mix`, and `mix_word`.

MUL Произведение регистра A и слова памяти располагается в регистрах A и X, младшая часть помещается в X, старшая часть — в A. Если знаки множителей одинаковы, то произведение имеет знак +, иначе — -.

27b ⟨Реализация функций машины MIX 5c⟩+≡ (7b) ◁27a 28a▷

```
void mix::op_mul(mix* ptr){
  long long int t;
  int sign = ptr→ram[ptr→M].getField(ptr→F).sign;
  sign ⊕= ptr→rA.sign;

  t = (long long)(ptr→rA.abs().getInt()) * (long long)(ptr→ram[ptr→M].getField(ptr→F).abs().getInt());
  ptr→rX.setInt(0x3fffffff & t);
  t ≫= 30;
  ptr→rA.setInt(0x3fffffff & t);
  ptr→rA.sign = ptr→rX.sign = sign;
}
```

Uses `abs`, `getField`, `getInt`, `mix`, and `setInt`.

DIV Делим *A*:*X* на слово памяти. Частное помещается в *A*, а остаток — в *X*. Знак остатка определяется знаком *A*. Знак частного определяется также как и при умножении. Переполнение и попытка деления на нуль устанавливают флажок.

28a ⟨Реализация функций машины MIX 5c⟩+≡ (7b) ◁27b 28b▷

```
void mix::op_div(mix* ptr){
  mix_word word_a = ptr→rA;
  unsigned int a = word_a.abs().getInt();
  mix_word word_b = ptr→rX;
  unsigned int b = word_b.abs().getInt();
  mix_word word_c = ptr→ram[ptr→M].getField(ptr→F);
  unsigned int c = word_c.abs().getInt();
  if(¬c ∨ (a > c)){
    ptr→flagV = true;
    return;
  }
  ptr→rA.setInt((((long long)a ≪ 30) + (long long)b) ÷ (long long)c);
  ptr→rX.setInt((((long long)a ≪ 30) + (long long)b) % (long long)c);
  ptr→rX.sign = word_a.sign;
  ptr→rA.sign = word_a.sign ⊕ word_c.sign;
}
```

Uses abs, getField, getInt, mix, mix_word, and setInt.

6.2.4 Команды операций с адресами

Так они названы у Кнутта. Операции работают, рассматривая *AA* не как адрес, а как константу. Код одинаков для нескольких операций, они различаются полем *F*.

28b ⟨Реализация функций машины MIX 5c⟩+≡ (7b) ◁28a 28c▷

```
void mix::op_inca(mix* ptr){
```

Uses mix.

INCA Увеличиваем содержимое регистра *A* на эффективный адрес M, то есть *A* += *AA* + содержимое индексного регистра.

28c ⟨Реализация функций машины MIX 5c⟩+≡ (7b) ◁28b 29a▷

```
  if(0 ≡ ptr→F){
    mix_word t;
    t.setInt(ptr→M);
    ptr→rA = ptr→rA + t;
    return;
  }
```

Uses mix_word and setInt.

DECA Уменьшаем содержимое регистра *A* на эффективный адрес M, то есть *A* -= *AA* + содержимое индексного регистра.

29a ⟨Реализация функций машины MIX 5c⟩+≡ (7b) ◁28c 29b▷

```
if(1 ≡ ptr→F){
  mix_word t;
  t.setInt(-ptr→M);
  ptr→rA = ptr→rA + t;
  return;
 }
```

Uses `mix_word` and `setInt`.

ENTA Загружаем в регистр *A* эффективный адрес M, то есть *A* = *AA* + содержимое индексного регистра. Здесь непонятно: у Кнута сказано, что операция *ENTA* 0,1 загружает в регистр *A* содержимое регистра *I1*, только знак устанавливается +.

29b ⟨Реализация функций машины MIX 5c⟩+≡ (7b) ◁29a 29c▷

```
if(2 ≡ ptr→F){
  ptr→rA.setInt(ptr→M);
  if(¬ptr→AA) ptr→rA.sign = ptr→Sign;
  return;
 }
```

Uses `setInt`.

ENNA Загружаем в регистр *A* эффективный адрес M с обратным знаком, то есть *A* = -(*AA* + содержимое индексного регистра).

29c ⟨Реализация функций машины MIX 5c⟩+≡ (7b) ◁29b 29d▷

```
if(3 ≡ ptr→F){
  ptr→rA.setInt(ptr→M);
  if(¬ptr→AA) ptr→rA.sign = ptr→Sign;
  ptr→rA.sign = 1 - ptr→rA.sign;
  return;
 }
}
```

Uses `setInt`.

Операции, подобные выше приведённым *ENTA*, *ENNA*, *INCA* и *DECA*, есть и для индексных регистров, однако я реализую их несколько по другому. Все эти команды будет обрабатывать одна функция, используя тот индексный регистр, на который указывает код операции. Для этого используем *idx*, как номер в массиве индексных регистров *rI*.

29d ⟨Реализация функций машины MIX 5c⟩+≡ (7b) ◁29c 30a▷

```
void mix::op_inc1(mix* ptr){
 int idx = ptr→C - 48;
```

Uses `mix`.

INCn Увеличиваем содержимое индексного регистра *In* на эффективный адрес M, то есть *In* += *AA* + содержимое индексного регистра.

30a ⟨Реализация функций машины MIX 5c⟩+≡ (7b) ◁29d 30b▷

```
if(0 ≡ ptr→F){
  mix_word t;
  t.setInt(ptr→M);
  ptr→rI[idx] = ptr→rI[idx] + t;
  return;
 }
```

Uses `mix_word` and `setInt`.

DECn Уменьшаем содержимое индексного регистра *In* на эффективный адрес M, то есть *In* -= *AA* + содержимое индексного регистра.

30b ⟨Реализация функций машины MIX 5c⟩+≡ (7b) ◁30a 30c▷

```
if(1 ≡ ptr→F){
  mix_word t;
  t.setInt(-ptr→M);
  ptr→rI[idx] = ptr→rI[idx] + t;
  return;
 }
```

Uses `mix_word` and `setInt`.

ENTn Загружаем в регистр *In* эффективный адрес M, то есть *In* = *AA* + содержимое индексного регистра.

30c ⟨Реализация функций машины MIX 5c⟩+≡ (7b) ◁30b 30d▷

```
if(2 ≡ ptr→F){
  ptr→rI[idx].setInt(ptr→M);
  if(¬ptr→AA) ptr→rI[idx].sign = ptr→Sign;
  return;
 }
```

Uses `setInt`.

ENNn Загружаем в регистр *In* эффективный адрес M с обратным знаком, то есть *In* = -*AA* + содержимое индексного регистра).

30d ⟨Реализация функций машины MIX 5c⟩+≡ (7b) ◁30c 30e▷

```
if(3 ≡ ptr→F){
  ptr→rI[idx].setInt(ptr→M);
  if(¬ptr→AA) ptr→rI[idx].sign = ptr→Sign;
  ptr→rI[idx].sign = 1 - ptr→rI[idx].sign;
  return;
 }
}
```

Uses `setInt`.

30e ⟨Реализация функций машины MIX 5c⟩+≡ (7b) ◁30d 31a▷

```
void mix::op_incx(mix* ptr){
```

Uses `mix`.

INCX Увеличиваем содержимое регистра *X* на эффективный адрес M, то есть *X* += *AA* + содержимое индексного регистра.

31a ⟨Реализация функций машины MIX 5c⟩+≡ (7b) ◁30e 31b▷

```
if(0 ≡ ptr→F){
  mix_word t;
  t.setInt(ptr→M);
  ptr→rX = ptr→rX + t;
  return;
 }
```

Uses `mix_word` and `setInt`.

DECX Уменьшаем содержимое регистра *X* на эффективный адрес M, то есть *X* -= *AA* + содержимое индексного регистра.

31b ⟨Реализация функций машины MIX 5c⟩+≡ (7b) ◁31a 31c▷

```
if(1 ≡ ptr→F){
  mix_word t;
  t.setInt(-ptr→M);
  ptr→rX = ptr→rX + t;
  return;
 }
```

Uses `mix_word` and `setInt`.

ENTX Загружаем в регистр *X* эффективный адрес M, то есть *X* = *AA* + содержимое индексного регистра. Здесь непонятно: у Кнута сказано, что операция *ENTX* 0,1 загружает в регистр *X* содержимое регистра *I1*, только знак устанавливается +.

31c ⟨Реализация функций машины MIX 5c⟩+≡ (7b) ◁31b 31d▷

```
if(2 ≡ ptr→F){
  ptr→rX.setInt(ptr→M);
  if(¬ptr→AA) ptr→rX.sign = ptr→Sign;
  return;
 }
```

Uses `setInt`.

ENNX Загружаем в регистр *X* эффективный адрес M с обратным знаком, то есть *X* = -(*AA* + содержимое индексного регистра).

31d ⟨Реализация функций машины MIX 5c⟩+≡ (7b) ◁31c 32a▷

```
if(3 ≡ ptr→F){
  ptr→rX.setInt(ptr→M);
  if(¬ptr→AA) ptr→rX.sign = ptr→Sign;
  ptr→rX.sign = 1 - ptr→rX.sign;
  return;
 }
}
```

Uses `setInt`.

6.2.5 Команды сравнения

При выполнении всех команд сравнения MIX справнивается величина, содержащаяся в регистре, с величиной в памяти. Затем для флага сравнения *flagC* устанавливается значение *less* (меньше), *equal* (равно) или *greater* (больше) в зависимости от того, будет ли содержащееся в регистре значение меньше, равно или больше значения, содержащегося в памяти. При этом нуль со знаком "-" считается равным нулю со знаком "+".

CMPA Заданное поле *rA* сравнивается с тем же полем *CONTENTS*(*M*). Если знак не входит в спецификацию поля, то числа рассматриваются как беззнаковые. Если *F*(0:0), то результатом всегда будет равенство.

32a ⟨Реализация функций машины MIX 5c⟩+≡ (7b) ◁31d 32b▷

```
void mix::op_cmpa(mix* ptr){
  if(¬ptr→F){
    ptr→flagC = equal;
    return;
  }
  int r = ptr→rA.getField(ptr→F).getInt();
  int m = ptr→ram[ptr→M].getField(ptr→F).getInt();
  ptr→flagC = equal;
  if(r < m) ptr→flagC = less;
  if(r > m) ptr→flagC = greater;
}
```

Uses `getField`, `getInt`, and `mix`.

CMPn Байты 1, 2 и 3 индексных регистров считаются нулевыми, поэтому при сравнении с *F* равным (1:3) результатом не может быть *greater*.

32b ⟨Реализация функций машины MIX 5c⟩+≡ (7b) ◁32a 33a▷

```
void mix::op_cmp1(mix* ptr){
  if(¬ptr→F){
    ptr→flagC = equal;
    return;
  }
  int idx = ptr→C - 57;
  int r = ptr→rI[idx].getField(ptr→F).getInt();
  int m = ptr→ram[ptr→M].getField(ptr→F).getInt();
  ptr→flagC = equal;
  if(r < m) ptr→flagC = less;
  if(r > m) ptr→flagC = greater;
}
```

Uses `getField`, `getInt`, and `mix`.

CMPX

33a ⟨Реализация функций машины MIX 5c⟩+≡ (7b) ◁32b 33b▷

```
void mix::op_cmpx(mix* ptr){
  if(¬ptr→F){
    ptr→flagC = equal;
    return;
  }
  int r = ptr→rX.getField(ptr→F).getInt();
  int m = ptr→ram[ptr→M].getField(ptr→F).getInt();
  ptr→flagC = equal;
  if(r < m) ptr→flagC = less;
  if(r > m) ptr→flagC = greater;
}
```

Uses `getField`, `getInt`, and `mix`.

6.2.6 Команды перехода

Команды обычно выполняются последовательно. При выполнении команды перехода порядок нарушается — следующая команда выбирается из ячейки по адресу *M*. Адрес ячейки, следующей *за* командой перехода сохраняется в регистре *J*. Это происходит в том случае, если переход действительно происходит.

33b ⟨Реализация функций машины MIX 5c⟩+≡ (7b) ◁33a 33c▷

```
void mix::op_jmp(mix* ptr){
  switch(ptr→F){
```

Uses `mix`.

JMP Безусловный переход, следующая команда адресуется *M*.

33c ⟨Реализация функций машины MIX 5c⟩+≡ (7b) ◁33b 33d▷

```
    case 0:
      ptr→nextOpAddr = M_ADDR;
      break;
```

JSJ Безусловный переход, следующая команда адресуется *M*. В отличие от *JMP* содержимое регистра *J* не меняется.

33d ⟨Реализация функций машины MIX 5c⟩+≡ (7b) ◁33c 33e▷

```
    case 1:
      ptr→nextOpAddr = M_NOJ_ADDR;
      break;
```

JOV Если для флага переполнения установлено значение **true**, то он переключается в *flase* и осуществляется безусловный переход JMP.

33e ⟨Реализация функций машины MIX 5c⟩+≡ (7b) ◁33d 34a▷

```
    case 2: break;
```

JNOV Если для флага переполнения установлено значение **false**, то осуществляется безусловный переход JMP.

34a ⟨Реализация функций машины MIX 5c⟩+≡ (7b) ◁33e 34b▷
```
case 3: break;
```

JL Переход по значению флага сравнения *less*, значение флага не изменяется.

34b ⟨Реализация функций машины MIX 5c⟩+≡ (7b) ◁34a 34c▷
```
case 4:
  if(less ≡ ptr→flagC) ptr→nextOpAddr = M_ADDR;
  break;
```

JE Переход по значению флага сравнения *equal*, значение флага не изменяется.

34c ⟨Реализация функций машины MIX 5c⟩+≡ (7b) ◁34b 34d▷
```
case 5:
  if(equal ≡ ptr→flagC) ptr→nextOpAddr = M_ADDR;
  break;
```

JG Переход по значению флага сравнения *greater*, значение флага не изменяется.

34d ⟨Реализация функций машины MIX 5c⟩+≡ (7b) ◁34c 34e▷
```
case 6:
  if(greater ≡ ptr→flagC) ptr→nextOpAddr = M_ADDR;
  break;
```

JGE Переход по значению флага сравнения *equal* или *greater*, значение флага не изменяется.

34e ⟨Реализация функций машины MIX 5c⟩+≡ (7b) ◁34d 34f▷
```
case 7:
  if(less ≠ ptr→flagC) ptr→nextOpAddr = M_ADDR;
  break;
```

JNE Переход по значению флага сравнения *less* или *greater*, значение флага не изменяется.

34f ⟨Реализация функций машины MIX 5c⟩+≡ (7b) ◁34e 34g▷
```
case 8:
  if(equal ≠ ptr→flagC) ptr→nextOpAddr = M_ADDR;
  break;
```

JLE Переход по значению флага сравнения *less* или *equal*, значение флага не изменяется.

34g ⟨Реализация функций машины MIX 5c⟩+≡ (7b) ◁34f 35a▷
```
case 9:
  if(greater ≠ ptr→flagC) ptr→nextOpAddr = M_ADDR;
  break;
 }
}
```

JAN, JAZ, JAP, JANN, JANZ, JANP Перейти, если в регистре *A* отрицательное, нулевое, положительное, неотрицательное, ненулевое, неположительное значение. “Положительным” считается значение больше нуля, “неположительным” — нуль и отрицательное.

35a ⟨Реализация функций машины MIX 5c⟩+≡ (7b) ◁34g 35b▷

```
void mix::op_ja(mix* ptr){
 switch(ptr→F){
  case 0: if(0 > ptr→rA.getInt()) ptr→nextOpAddr = M_ADDR; break;
  case 1: if(0 ≡ ptr→rA.getInt()) ptr→nextOpAddr = M_ADDR; break;
  case 2: if(0 < ptr→rA.getInt()) ptr→nextOpAddr = M_ADDR; break;
  case 3: if(0 ≤ ptr→rA.getInt()) ptr→nextOpAddr = M_ADDR; break;
  case 4: if(0 ≠ ptr→rA.getInt()) ptr→nextOpAddr = M_ADDR; break;
  case 5: if(0 ≥ ptr→rA.getInt()) ptr→nextOpAddr = M_ADDR; break;
 }
}
```

Uses `getInt` and `mix`.

JXN, JXZ, JXP, JXNN, JXNZ, JXNP Перейти, если в регистре *X* отрицательное, нулевое, положительное, неотрицательное, ненулевое, неположительное значение. “Положительным” считается значение больше нуля, “неположительным” — нуль и отрицательное.

35b ⟨Реализация функций машины MIX 5c⟩+≡ (7b) ◁35a 35c▷

```
void mix::op_jx(mix* ptr){
 switch(ptr→F){
  case 0: if(0 > ptr→rX.getInt()) ptr→nextOpAddr = M_ADDR; break;
  case 1: if(0 ≡ ptr→rX.getInt()) ptr→nextOpAddr = M_ADDR; break;
  case 2: if(0 < ptr→rX.getInt()) ptr→nextOpAddr = M_ADDR; break;
  case 3: if(0 ≤ ptr→rX.getInt()) ptr→nextOpAddr = M_ADDR; break;
  case 4: if(0 ≠ ptr→rX.getInt()) ptr→nextOpAddr = M_ADDR; break;
  case 5: if(0 ≥ ptr→rX.getInt()) ptr→nextOpAddr = M_ADDR; break;
 }
}
```

Uses `getInt` and `mix`.

JnN, JnZ, JnP, JnNN, JnNZ, JnNP Перейти, если в регистре *In* отрицательное, нулевое, положительное, неотрицательное, ненулевое, неположительное значение. “Положительным” считается значение больше нуля, “неположительным” — нуль и отрицательное.

35c ⟨Реализация функций машины MIX 5c⟩+≡ (7b) ◁35b 36a▷

```
void mix::op_ji(mix* ptr){
 int idx = ptr→C - 40;
 switch(ptr→F){
  case 0: if(0 > ptr→rI[idx].getInt()) ptr→nextOpAddr = M_ADDR; break;
  case 1: if(0 ≡ ptr→rI[idx].getInt()) ptr→nextOpAddr = M_ADDR; break;
  case 2: if(0 < ptr→rI[idx].getInt()) ptr→nextOpAddr = M_ADDR; break;
  case 3: if(0 ≤ ptr→rI[idx].getInt()) ptr→nextOpAddr = M_ADDR; break;
  case 4: if(0 ≠ ptr→rI[idx].getInt()) ptr→nextOpAddr = M_ADDR; break;
  case 5: if(0 ≥ ptr→rI[idx].getInt()) ptr→nextOpAddr = M_ADDR; break;
 }
}
```

Uses `getInt` and `mix`.

6.2.7 Команды преобразования

Достаточно “сложные” функции преобразования в/из символьного представления чисел.

36a ⟨Реализация функций машины MIX 5c⟩+≡ (7b) ◁35c 36b▷

```
void mix::op_hlt(mix* ptr){
 switch(ptr→F){
```

Uses mix.

NUM Преобразовать символьную строку в регистрах A и X в число в регистре A. При переполнении сохраняется остаток от деления на b^5, где b — размер байта.

36b ⟨Реализация функций машины MIX 5c⟩+≡ (7b) ◁36a 37a▷

```
  case 0:
   {
    long long multi = 1;
    long long result = 0;
    result += multi * (ptr→rX.b5 % 10); multi *= 10;
    result += multi * (ptr→rX.b4 % 10); multi *= 10;
    result += multi * (ptr→rX.b3 % 10); multi *= 10;
    result += multi * (ptr→rX.b2 % 10); multi *= 10;
    result += multi * (ptr→rX.b1 % 10); multi *= 10;
    result += multi * (ptr→rA.b5 % 10); multi *= 10;
    result += multi * (ptr→rA.b4 % 10); multi *= 10;
    result += multi * (ptr→rA.b3 % 10); multi *= 10;
    result += multi * (ptr→rA.b2 % 10); multi *= 10;
    result += multi * (ptr→rA.b1 % 10);
    result %= 64 * 64 * 64 * 64 * 64;
    int sign = ptr→rA.sign;
    ptr→rA.setInt(result);
    ptr→rA.sign = sign;

   }
   break;
```

Uses setInt.

CHAR Преобразовать число без знака из регистра *A* в десятибайтовую строку в регистрах *A* и *X*.

37a ⟨Реализация функций машины MIX 5c⟩+≡ (7b) ◁36b 37b▷

```
case 1:
 {
 int num = abs(ptr→rA.getInt());
 ptr→rX.b5 = 30 + (num % 10); num ÷= 10;
 ptr→rX.b4 = 30 + (num % 10); num ÷= 10;
 ptr→rX.b3 = 30 + (num % 10); num ÷= 10;
 ptr→rX.b2 = 30 + (num % 10); num ÷= 10;
 ptr→rX.b1 = 30 + (num % 10); num ÷= 10;
 ptr→rA.b5 = 30 + (num % 10); num ÷= 10;
 ptr→rA.b4 = 30 + (num % 10); num ÷= 10;
 ptr→rA.b3 = 30 + (num % 10); num ÷= 10;
 ptr→rA.b2 = 30 + (num % 10); num ÷= 10;
 ptr→rA.b1 = 30 + (num % 10);
 }
 break;
```

Uses `abs` and `getInt`.

HLT Полный останов машины.

37b ⟨Реализация функций машины MIX 5c⟩+≡ (7b) ◁37a 38a▷

```
 case 2:
  ptr→nextOpAddr = STOP_ADDR;
  break;
 }
}
```

6.2.8 Команды ввода/вывода

IN Передача информации из заданного устройства ввода в область памяти начиная с M. Номер устройства указывается F, число слов соответствует размеру блока для данного устройства.

38a ⟨Реализация функций машины MIX 5c⟩+≡ (7b) ◁37b 38b▷

```
void mix::op_in(mix* ptr){
  int addr = ptr→M;
  if(0 ≤ ptr→F ∧ 7 ≥ ptr→F){
    for(int i=0; i < TAPE_BLOCK_SIZE; ++i){
      mix_word tmp;
      char ch;
      *(ptr→tape[ptr→F]) ≫ ch; tmp.sign = ch;
      *(ptr→tape[ptr→F]) ≫ ch; tmp.b1 = ch;
      *(ptr→tape[ptr→F]) ≫ ch; tmp.b2 = ch;
      *(ptr→tape[ptr→F]) ≫ ch; tmp.b3 = ch;
      *(ptr→tape[ptr→F]) ≫ ch; tmp.b4 = ch;
      *(ptr→tape[ptr→F]) ≫ ch; tmp.b5 = ch;
      ptr→ram[addr++] = tmp;
    }
    return;
  }
}
```

Uses `mix` and `mix_word`.

OUT Передача информации из памяти начиная с ячейки M на устройство F.

38b ⟨Реализация функций машины MIX 5c⟩+≡ (7b) ◁38a 39b▷

```
void mix::op_out(mix* ptr){
  int addr = ptr→M;
```

Uses `mix`.

Вывод на ленточные накопители

38c ⟨Реализация фукнций машины MIX 38c⟩≡

```
  if(0 ≤ ptr→F ∧ 7 ≥ ptr→F){
    for(int i=0; i < TAPE_BLOCK_SIZE; ++i){
      char ch;
      ch = ptr→ram[addr].sign; *(ptr→tape[ptr→F]) ≪ ch;
      ch = ptr→ram[addr].b1; *(ptr→tape[ptr→F]) ≪ ch;
      ch = ptr→ram[addr].b2; *(ptr→tape[ptr→F]) ≪ ch;
      ch = ptr→ram[addr].b3; *(ptr→tape[ptr→F]) ≪ ch;
      ch = ptr→ram[addr].b4; *(ptr→tape[ptr→F]) ≪ ch;
      ch = ptr→ram[addr].b5; *(ptr→tape[ptr→F]) ≪ ch;
      ++addr;
    }
    return;
  }
```

Вывод на принтер (экран). Для перекодировки используется вспомогательная функция. которая возвращает строку из пяти символов, соответствующих слову MIX.

39a ⟨Внутренние функции и переменные класса машины 4a⟩+≡ (3d) ◁23c
```
static string makeString(mix_word w);
```
Uses mix_word.

39b ⟨Реализация функций машины MIX 5c⟩+≡ (7b) ◁38b 39c▷
```
if(17 ≡ ptr→F){
 for(int i=0; i< PRINTER_BLOCK_SIZE; ++i)
  cout ≪ makeString(ptr→ram[addr++]);
 cout ≪ endl;
 return;
}
}
```

IOC Управление вводом/выводом. В F указывается номер устройства, дальнейшие действия зависят от типа устройства.

39c ⟨Реализация функций машины MIX 5c⟩+≡ (7b) ◁39b 39d▷
```
void mix::op_ioc(mix* ptr){
 switch(ptr→F){
```
Uses mix.

Магнитная лента. Если M равно нулю, то лента перематывается в начало, если M меньше нуля, то лента перематывается на M блоков назад, иначе на M блоков вперёд.

39d ⟨Реализация функций машины MIX 5c⟩+≡ (7b) ◁39c 39e▷
```
 case 0: case 1: case 2: case 3: case 4: case 5: case 6: case 7:
  if(¬ptr→M) ptr→tape[ptr→F]→seekg(0,ios::beg);
  else ptr→tape[ptr→F]→seekg(IO_BYTES_PER_WORD * TAPE_BLOCK_SIZE * ptr→M,ios::cur);
  break;
```

39e ⟨Реализация функций машины MIX 5c⟩+≡ (7b) ◁39d 39f▷
```
 }
}
```

JRED Переход, если устройство готово. Без таймеров задержек устройства готовы всегда.

39f ⟨Реализация функций машины MIX 5c⟩+≡ (7b) ◁39e 39g▷
```
void mix::op_jred(mix* ptr){
 ptr→nextOpAddr = M_ADDR;
}
```
Uses mix.

JBUS Переход, если устройство занято. Без таймеров задержек устройства не бывают заняты.

39g ⟨Реализация функций машины MIX 5c⟩+≡ (7b) ◁39f 40a▷
```
void mix::op_jbus(mix* ptr){
}
```
Uses mix.

6.2.9 Другие команды

Сдвиги реализованы не самым оптимальным способом — не использованы возможности Pentium. Перед командами сдвига нормализуем число байт, на которое сдвигаем.

40a ⟨Реализация функций машины MIX 5c⟩+≡ (7b) ◁39g 40b▷

```
void mix::op_sla(mix* ptr){
  int shift = abs(ptr→M);
  shift = (10 < shift)?10:shift;
  switch(ptr→F){
```

Uses `abs` and `mix`.

SLA Сдвинуть содержимое регистра *A* на *M* байт влево. *M* должно быть неотрицательным.

40b ⟨Реализация функций машины MIX 5c⟩+≡ (7b) ◁40a 40c▷

```
    case 0:
      shift = (5 < shift)?5:shift;
      for(int i=0; i<shift; ++i){
       ptr→rA.b1 = ptr→rA.b2;
       ptr→rA.b2 = ptr→rA.b3;
       ptr→rA.b3 = ptr→rA.b4;
       ptr→rA.b4 = ptr→rA.b5;
       ptr→rA.b5 = 0;
       }
      break;
```

SRA Сдвинуть содержимое регистра *A* на *M* байт вправо. *M* должно быть неотрицательным.

40c ⟨Реализация функций машины MIX 5c⟩+≡ (7b) ◁40b 41a▷

```
    case 1:
      shift = (5 < shift)?5:shift;
      for(int i=0; i<shift; ++i){
       ptr→rA.b5 = ptr→rA.b4;
       ptr→rA.b4 = ptr→rA.b3;
       ptr→rA.b3 = ptr→rA.b2;
       ptr→rA.b2 = ptr→rA.b1;
       ptr→rA.b1 = 0;
       }
      break;
```

SLAX Сдвинуть содержимое регистров *A* и *X* на *M* байт влево. *M* должно быть неотрицательным.

41a ⟨Реализация функций машины MIX 5c⟩+≡ (7b) ◁40c 41b▷

```
case 2:
 for(int i=0; i<shift; ++i){
  ptr→rA.b1 = ptr→rA.b2;
  ptr→rA.b2 = ptr→rA.b3;
  ptr→rA.b3 = ptr→rA.b4;
  ptr→rA.b4 = ptr→rA.b5;
  ptr→rA.b5 = ptr→rX.b1;
  ptr→rX.b1 = ptr→rX.b2;
  ptr→rX.b2 = ptr→rX.b3;
  ptr→rX.b3 = ptr→rX.b4;
  ptr→rX.b4 = ptr→rX.b5;
  ptr→rX.b5 = 0;
  }
 break;
```

SRAX Сдвинуть содержимое регистров *A* и *X* на *M* байт вправо. *M* должно быть неотрицательным.

41b ⟨Реализация функций машины MIX 5c⟩+≡ (7b) ◁41a 42a▷

```
case 3:
 for(int i=0; i<shift; ++i){
  ptr→rX.b5 = ptr→rX.b4;
  ptr→rX.b4 = ptr→rX.b3;
  ptr→rX.b3 = ptr→rX.b2;
  ptr→rX.b2 = ptr→rX.b1;
  ptr→rX.b1 = ptr→rA.b5;
  ptr→rA.b5 = ptr→rA.b4;
  ptr→rA.b4 = ptr→rA.b3;
  ptr→rA.b3 = ptr→rA.b2;
  ptr→rA.b2 = ptr→rA.b1;
  ptr→rA.b1 = 0;
  }
 break;
```

SLC Сдвинуть содержимое регистра *A* циклично влево. *M* должно быть неотрицательным.

42a ⟨Реализация функций машины MIX 5c⟩+≡ (7b) ◁41b 42b▷

```
    case 4:
     for(int i=0; i<shift; ++i){
      int tmp = ptr→rA.b1;
      ptr→rA.b1 = ptr→rA.b2;
      ptr→rA.b2 = ptr→rA.b3;
      ptr→rA.b3 = ptr→rA.b4;
      ptr→rA.b4 = ptr→rA.b5;
      ptr→rA.b5 = tmp;
      }
     break;
```

SRC Сдвинуть содержимое регистра *A* циклично вправо. *M* должно быть неотрицательным.

42b ⟨Реализация функций машины MIX 5c⟩+≡ (7b) ◁42a 42c▷

```
    case 5:
     for(int i=0; i<shift; ++i){
      int tmp = ptr→rA.b5;
      ptr→rA.b5 = ptr→rA.b4;
      ptr→rA.b4 = ptr→rA.b3;
      ptr→rA.b3 = ptr→rA.b2;
      ptr→rA.b2 = ptr→rA.b1;
      ptr→rA.b1 = tmp;
      }
     break;
   }
  }
```

MOVE Количество слов, определяемое *F*, перемещается начиная с ячейки *M* в другие ячейки, адрес первой из которых находиться в регистре *I1*. Перемещение осуществляется по одному слову за раз, и к концу выполнения операции значение в регистре *I1* увеличивается на *F*. Если *F* равно нулю, то ничего не происходит.

42c ⟨Реализация функций машины MIX 5c⟩+≡ (7b) ◁42b 43▷

```
 void mix::op_move(mix* ptr){
  if(¬ptr→F) return;
  int dst = ptr→rI[1].getInt();
  int src = ptr→M;
  int i = ptr→F;
  while(i−−)
   ptr→ram[dst++] = ptr→ram[src++];
  ptr→rI[1].setInt(dst);
 }
```

Uses `getInt`, `mix`, and `setInt`.

6.2.10 Преобразование символов в машине MIX

43 ⟨Реализация функций машины MIX 5c⟩+≡ (7b) ◁42c

```
string makeChar(int byte){
  static string ch[]={" ","A","B","C","D","E","F","G","H","I","?","J","K","L","M","N","O",
                      "P","Q","R","?","?","S","T","U","V","W","X","Y","Z","0","1","2","3",
                      "4","5","6","7","8","9",".",",","(",")","+","-","*","/","=","$","<",
                      ">","@",";",":","'"};
  return ch[byte];
}
string mix::makeString(mix_word w){
  string str="";
  str += makeChar(w.b1);
  str += makeChar(w.b2);
  str += makeChar(w.b3);
  str += makeChar(w.b4);
  str += makeChar(w.b5);
  return str;
}
```

Uses `mix` and `mix_word`.

6.3 Секции кода

⟨mix-emu.cpp 11a⟩ 11a
⟨mix.cpp 7b⟩ 7b
⟨mix.h 10d⟩ 10d
⟨Внутренние функции и переменные класса машины 4a⟩ 3d, 4a, 5a, 5d, 6c, 6d, 23c, 39a
⟨Вспомогательные функции 13c⟩ 11a, 13c
⟨Выполнение операции 6f⟩ 5c, 6f
⟨Инициализация массива операций 7a⟩ 6g, 7a
⟨Инициализация специального индексного регистра 3e⟩ 3e, 7d
⟨Инициализация устройств ввода/вывода 4c⟩ 4c, 7d
⟨Класс машины MIX 3d⟩ 3d, 10d
⟨Коды возврата getCommand 17a⟩ 13b, 17a
⟨Команды оператора 21c⟩ 14b, 21c
⟨Константы 11b⟩ 11a, 11b, 14b
⟨Константы класса машины 3c⟩ 3c, 3d, 4b, 8b
⟨Конструктор и деструктор 7d⟩ 7b, 7d
⟨Мнемоники операций MIX 22⟩ 14b, 22
⟨Найти команду 14c⟩ 13c, 14c
⟨Обработка потока команд 12a⟩ 11a, 12a, 12b, 12c, 13a
⟨Освобождение устройств ввода/вывода 4d⟩ 4d, 7d
⟨Подключение заголовочных файлов 23a⟩ 11a, 23a, 23b
⟨Получить команду 15a⟩ 13c, 15a, 15b
⟨Прототипы функций 11c⟩ 11a, 11c, 17b, 19a, 19c, 19d, 20b, 20d, 21a
⟨Разбираем параметры мнемоники 16⟩ 15b, 16
⟨Разбор слова операции 6a⟩ 5c, 6a, 6b
⟨Реализация команд эмулятора 18⟩ 13c, 18, 19b, 20a, 20c, 20e, 21b
⟨Реализация операций со словом MIX 8c⟩ 7b, 8c, 8e, 9a, 9b, 9c, 9e, 10a, 10b, 10c
⟨Реализация фукнций машины MIX 38c⟩ 38c
⟨Реализация функций машины MIX 5c⟩ 5c, 6e, 6g, 7b, 24a, 24b, 24c, 25a, 25b, 25c, 25d, 25e, 26a, 26b, 26c, 26d, 27a, 27b, 28a, 28b, 28c, 29a, 29b, 29c, 29d, 30a, 30b, 30c, 30d, 30e, 31a, 31b, 31c, 31d, 32a, 32b, 33a, 33b, 33c, 33d, 33e, 34a, 34b, 34c, 34d, 34e, 34f, 34g, 35a, 35b, 35c, 36a, 36b, 37a, 37b, 38a, 38b, 39b, 39c, 39d, 39e, 39f, 39g, 40a, 40b, 40c, 41a, 41b, 42a, 42b, 42c, 43
⟨Свои типы 13b⟩ 11a, 13b
⟨Словарная статья 14a⟩ 13b, 14a
⟨Слово MIX 2⟩ 2, 3a, 3b, 10d
⟨Функции класса машины 5b⟩ 3d, 5b, 7c, 8a
⟨Шаблоны операций со словом MIX 8d⟩ 2, 8d, 8f, 9d, 10e

6.4 Определения

`abs`: 8d, 9a, 9e, 10a, 10c, 26a, 27b, 28a, 37a, 40a
`doOp`: 5b, 12c, 20c
`dumpMemory`: 12a, 19c
`dumpReg`: 12a, 19a, 21b
`eCmd`: 12a, 14a, 15a, 17b
`editMem`: 12a, 20d

Список литературы

[1] Д.Э. Кнут: *Искусство программирования для ЭВМ*, Том 1–3.

[2] Norman Ramsey: *Literate programming simplified*.