Постановка задачи
Функции работы с памятью

Курсовой проект "Эмулятор PDP-11"
Занятие 1 и 2

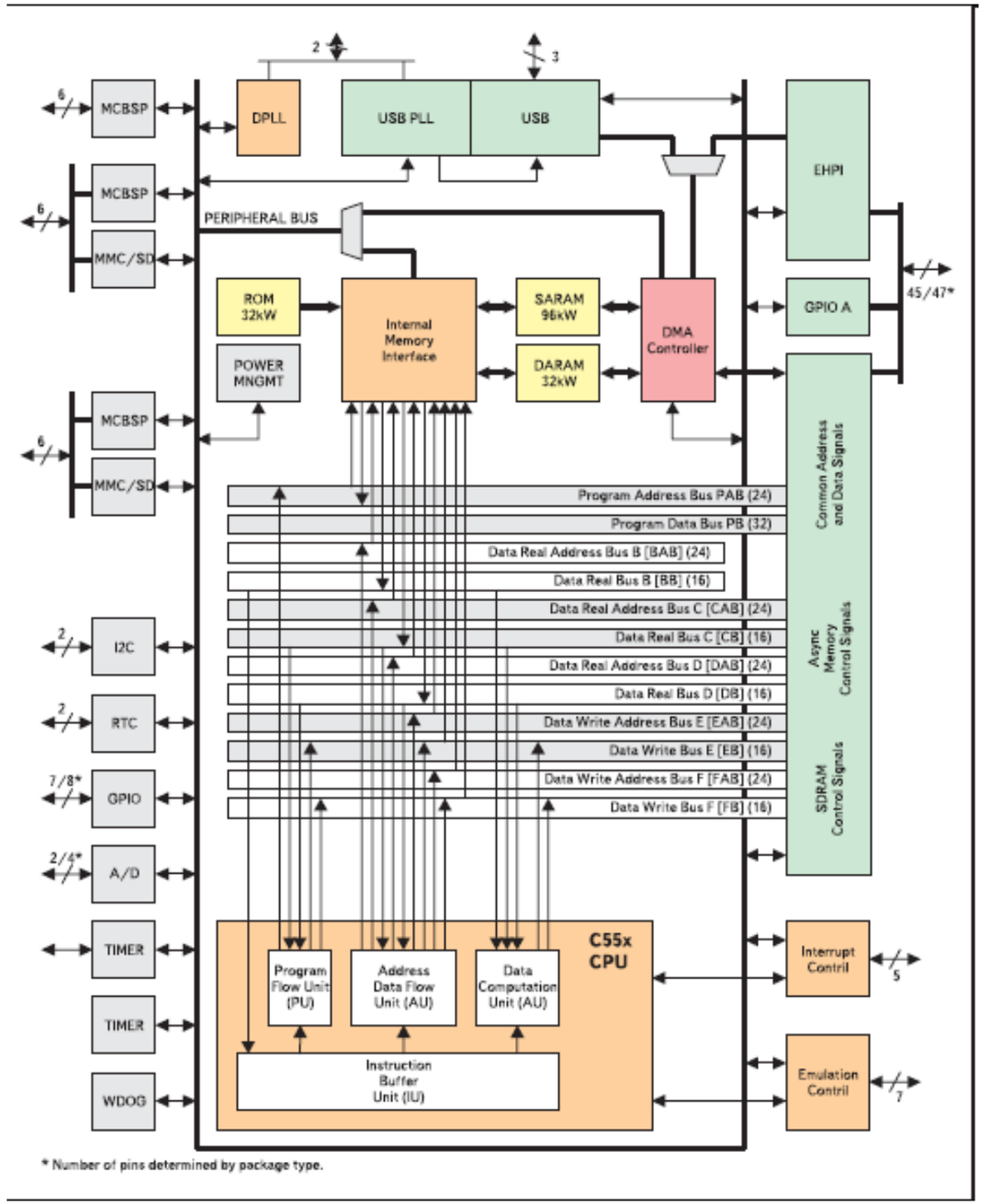
Архитектура Фон Неймана

- Реальная архитектура — сложнее
  - несколько ЦП
  - несколько шин и тп
- Гарвардская архитектура
  (отдельная память под программу и под данные)

Реальность сложнее

- Упрощеная схема

  Архитектура процессора TMS320VC5509



Классификация языков

- Машинный код 000011101101
- Ассемблер mov ax, b
- Высокого уровня C, Java, Python
-
- Ассемблер PDP-11
  - самый простой
  - Система команд самая красивая
  - Разбирать будем не ассемблер, а его байт-код
- Зачем нужны эмуляторы старых машин
- Выполнение не на железе, а на симуляторе

Компилируемый / интерпретируемый

- Компиляция до уровня
  - Машинный код 000011101101
    - быстрое исполнение
    - неоднозначное обратное преобразование
  - Байт-код
    - нужен интерпретатор кода
      - **вирутальная машина** (будем писать)
      - JVM
- Интерпретируемые языки python

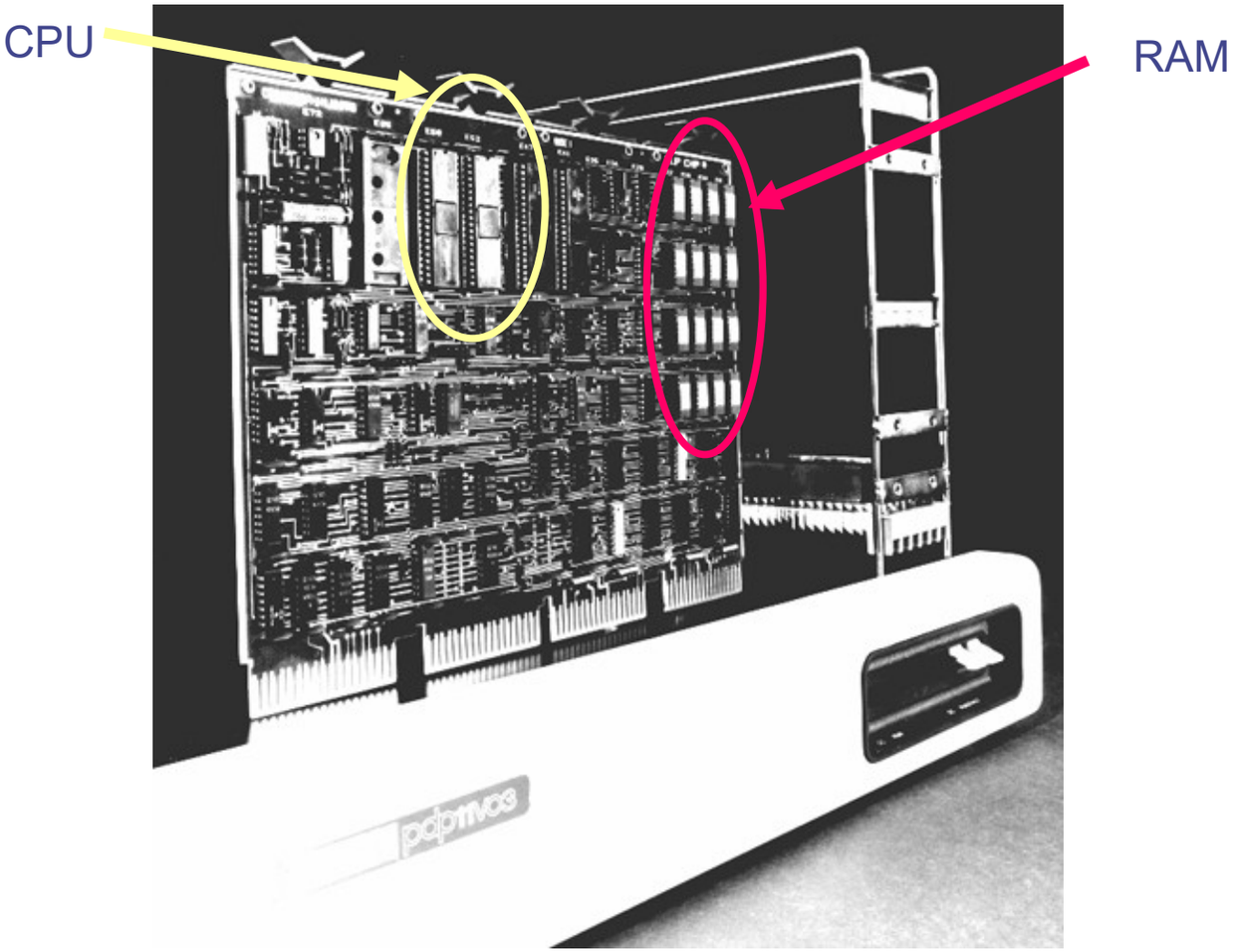
Задача

- Написать интерпретатор подмножества машинных команд компьютера PDP-11
- Есть
  - документация, список команд (acm.mipt.ru / язык Си / секретные материалы)
  - компилятор из ассемблера в машинный код
  - эталонный эмулятор (сравниваем результат выполнения теста на нем и на вашем эмуляторе)
  - набор программ на ассемблере PDP-11 (тесты)
  - git

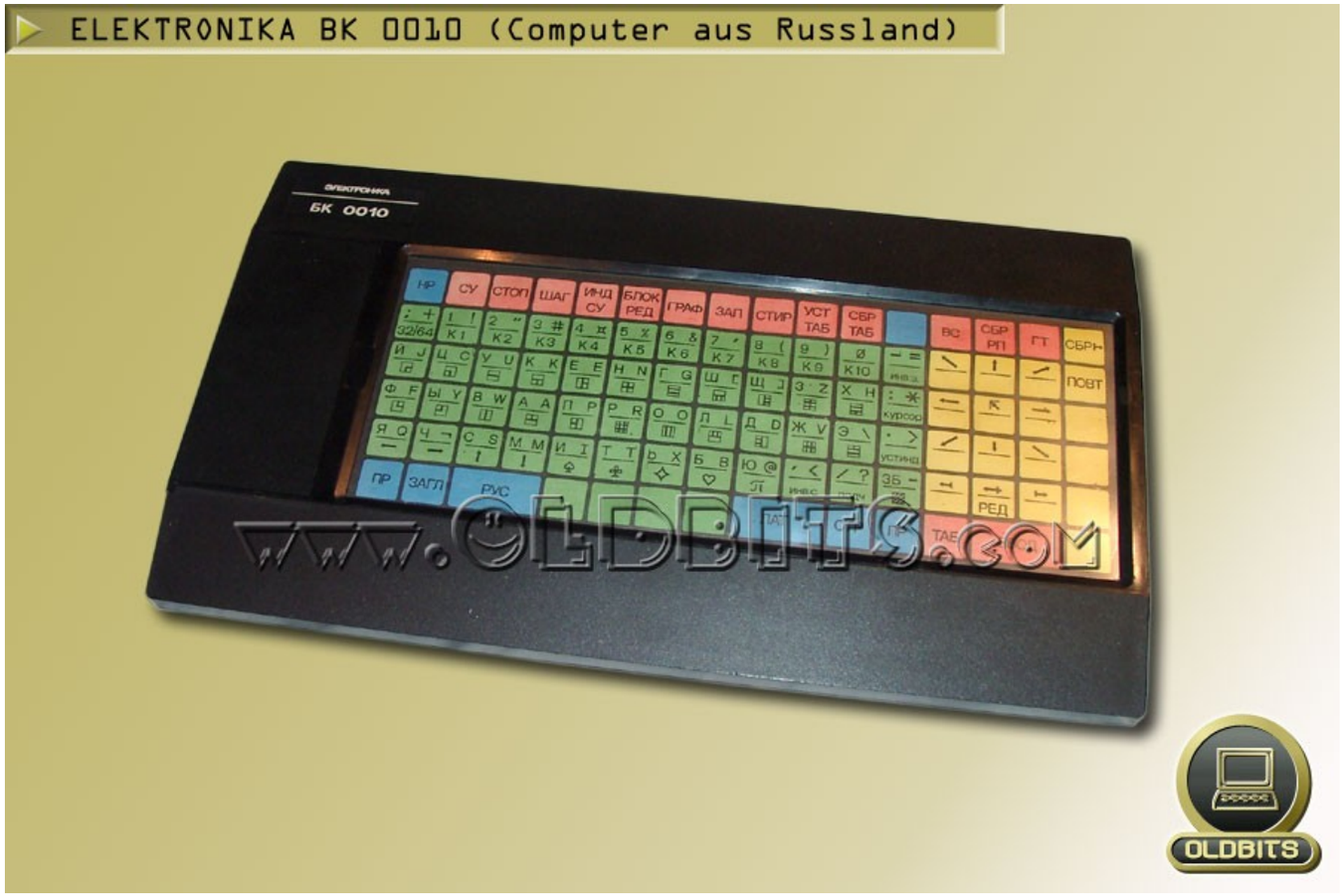
Как это выглядело

- Мини-ЭВМ PDP-11/40 графдисплей, "посадка на луну"

CPU & RAM



И его клоны

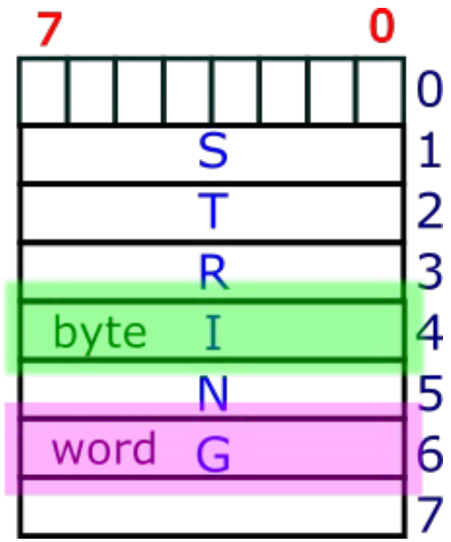


- Пленочная клавиатура,
  можно менять пленку для разных приложений

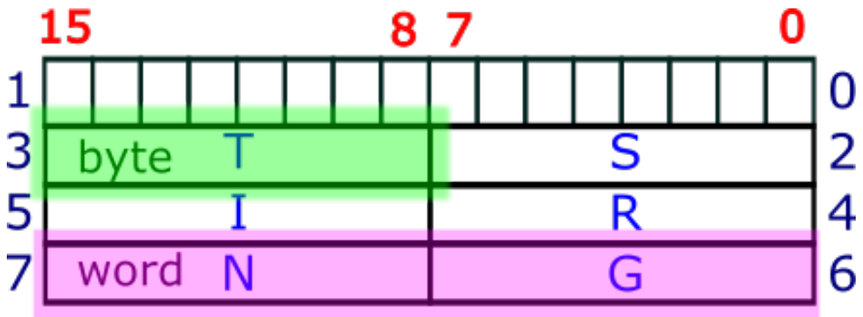
Термины

- Бит — 0 или 1
- Байт — наименьшая адресуемая единица памяти
- Октет — 8 бит
- Разрядность (памяти, процессора, шины и тп)
  8, 16 (20, 24), 32, 64 бит
- Машинное слово — ячейка памяти, участвующее в качестве операнда в инструкции процессора
- Адресация — побайтовая или словами
- Litle endian — big endian

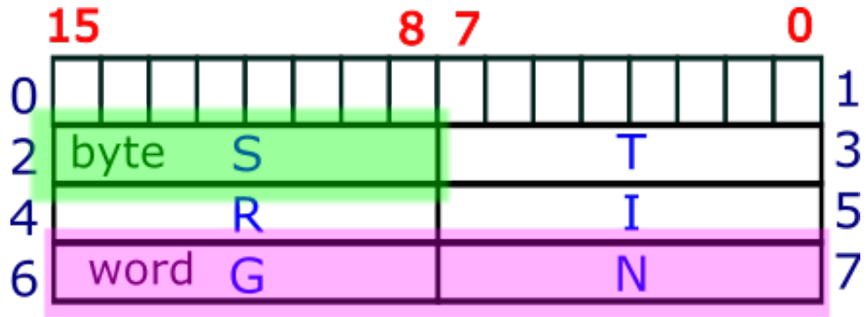
Термины в картинках



- 8 битная с побайтовой адресацией

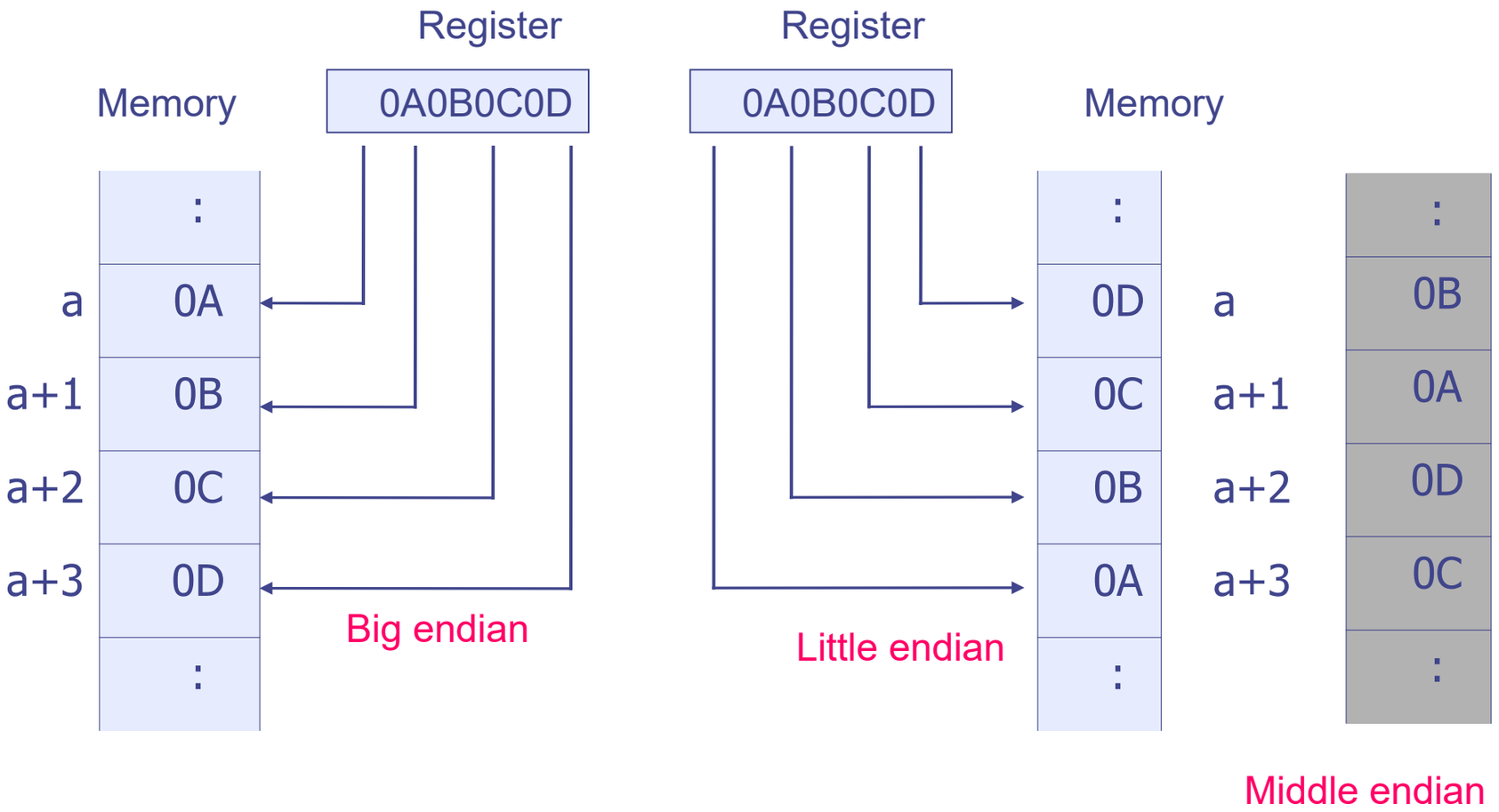


- 16-битная little endian PDP-11, Intel



- 16-битная big endian network

Big, little and mixed



- Программно переключаемый порядок (bi-endian, bytesexual)

Побитовые операции

- Применяются только к целочисленным операндам
- **&** побитовое AND
- **I** побитвое OR
- **^** побитовый XOR
- **~** отрицание (тильда над ё, под Esc)
- **>>** сдвиг вправо (на неотрицательное)
- **<<** сдвиг влево (на неотрицательное)

Смещение >> и <<

- x = 7; 00000111
  y = x << 3; 00111000
  всегда дополняется нулями

- unsinged short x =7; 00000111
  z = x >> 2; 00000001

- int x = 7; 00000111
  z = x >> 2; 00000001

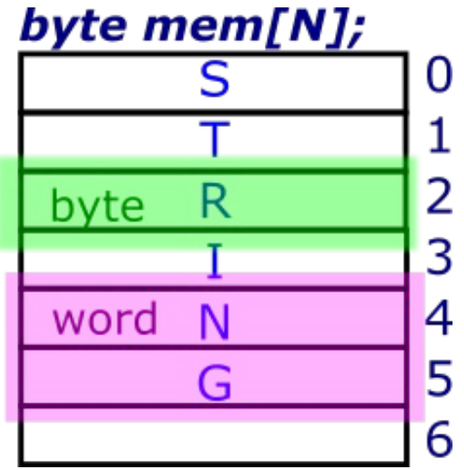
- int x = -7; 11111001
  z = x >> 2; 11111110
  дополняется знаковым битом

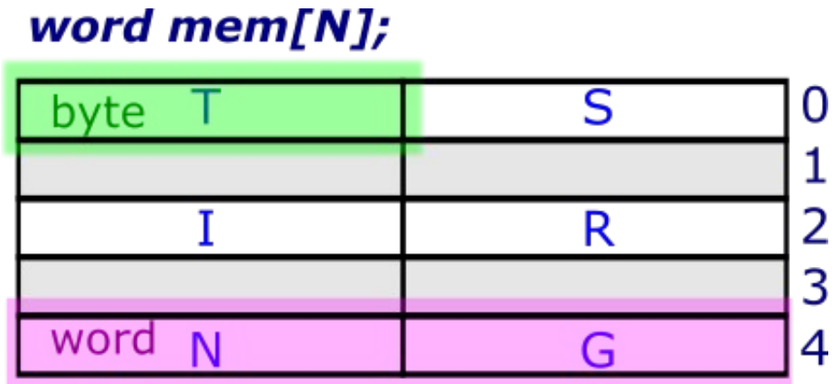
Начинаем писать программу Шаг 1

- Определить типы
  typedef *существующий тип* **новое_имя ;**
  typedef *???* **byte ;** **// 8 bit**
  typedef *???* **word ;** **// 16 bit**
  typedef *word* Adress ; **// 64 Kb**
- 64 Kb = 8*8*2^{10} = 2^3*2^3*2^{10} = 2^{16}
- **#include "pdp11.h"**

Шаг 2: объявляем массив RAM

- RAM — random access memory (память произвольного доступа) 64 Kb
- mem - глобальная переменная — нужна всем
- Вариант 1
  byte mem [MEMSIZE];
- Вариант 2;
  word mem [MEMSIZE];





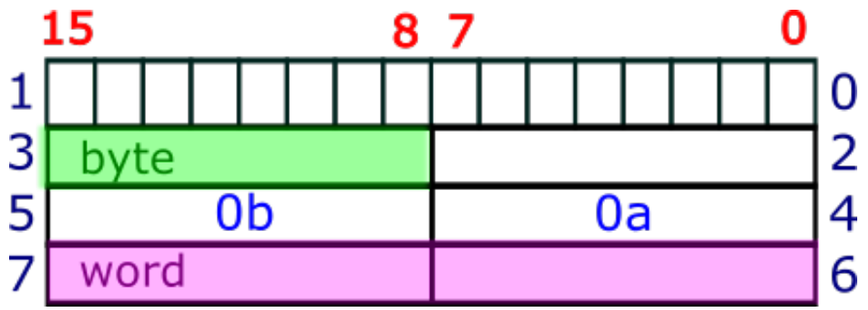
Шаг 3. Пишем API функций работы с памятью

- Писать прямо в память не будем — можем ошибиться. Реализуем функции чтения и записи в память:
- **void b_write** (**Adress adr**, **byte b**);
  // пишем байт b по адресу adr
- **byte b_read** (**Adress adr**);
  // читаем байт по адресу adr
- **void w_write** (**Adress adr**, **word w**);
  // пишем слово w по адресу adr
- **word w_read** (**Adress adr**);
  // читаем слово по адресу adr

Шаг 4. Тесты для этих функций

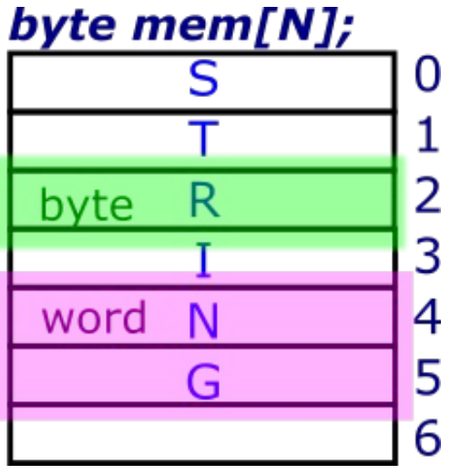
- Пишем байт — читаем байт
  Пишем слово — читаем слово
  Пишем 2 байта — читаем 1 слово
  Пишем 1 слово — читаем 2 байта

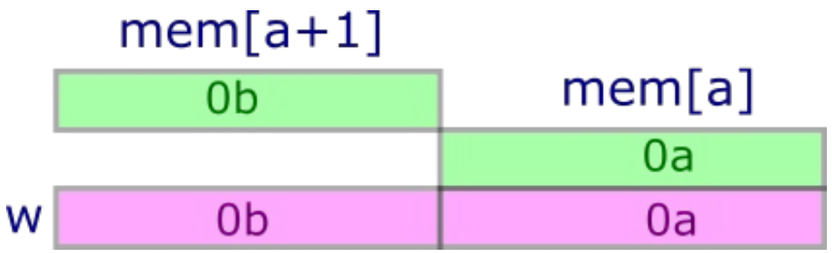
- Adress a = 4;
  byte b1 = 0x0b, b0 = 0x0a;
  word w = 0x0b0a;
  b_write (a, b0);
  b_write (a+1, b1);
  word wres = w_read (a);
  printf("%04hx=%02hhx%02hhx\n", wres, b1, b0);
  assert(wres == w);



Шаг 5-1. Реализация функций и отладка

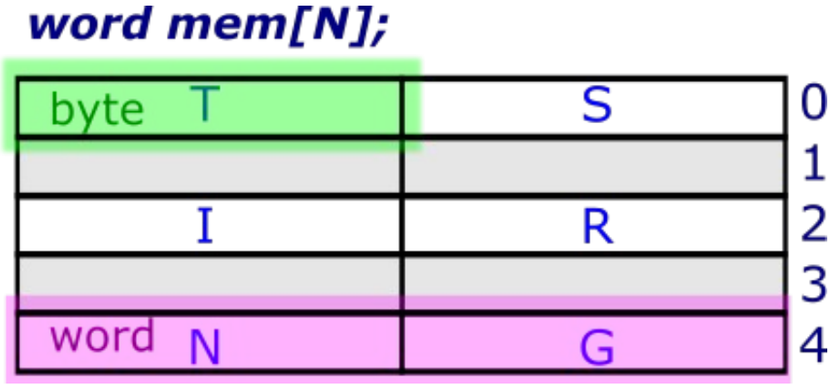
- Вариант 1. byte mem[ ];
- void b_read (Adress adr) {
  return mem [adr];
  }
- byte b_write (Adress adr, byte b) {
  mem [adr] = b;
  }
- word w_read (Adress adr) {
  word w = 0;
  // пишем код
  return w;
  }

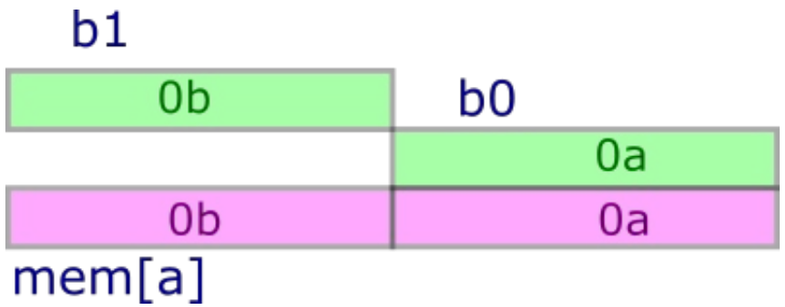




Шаг 5-2. Реализация функций и отладка

- Вариант 2. word mem[ ];
- void w_read (Adress adr) {
  return mem [adr];
  }
- word w_write (Adress adr, word d) {
  mem [adr] = w;
  }
- byte b_read (Adress adr) {
  byte b;
  // пишем код
  return b;
  }





Шаг 6. Загрузка программы в память

- void **load_file** ( );
- **только 16-ричные числа** %x
- ./pdp.exe < test.o
- Несколько блоков
- 1 блок состоит из

адрес_начала_блока **N**
байт1
байт2
…
байтN

```
0200 000a
c0
65
7a
00
3f
10
f8
01
00
00
0400 0002
76
0f
```

Шаг 7: Чтение из файла

- До этого можно сдать решения в контест pdp_rw, pdp_load.
- Дальше — тестируем сами.
- Чтение данных из файла
  (имя файла можно заранее зафиксировать)
- void **load_file** (const char * filename);
- **load_file** ("/home/taty/pdp11/tests/sum.o");

Шаг 7. Имя файла в агрументах программы

- Было

  ./pdp.exe < test.o

- Стало

  ./pdp.exe test.o

- int main (int **argc**, char * **argv** [ ]) {

  ...

  }

Шаг 8: функции trace и debug

- Напишите функции печати
- отладочная печать
  debug(const char * format, ...);
  + запускается ключом -d
  ./pdp.exe -d test.o
- трассировка выполнения
  вызывается толко когда указан ключ -t
  ./pdp.exe -t test.o
  trace(const char * format, ...);

Шаг 8 (альтернатива)

- Написать одну функцию
  void log(int log_level, const char * format, ...);
- Определить уровни логирования
  #define ERROR 0
  #define INFO 1
  #define TRACE 2
  #define DEBUG 3
- Использование:
  log (INFO, "Load file %s\n", "test.o");
  log (DEBUG, "%d : %s\n", __LINE__, __FUNCTION__);