Аргументы типа SS и DD
Режимы адресации

Курсовой проект "Эмулятор PDP-11"
Занятие 4 и 5

Что уже сделано

- Память RAM 64Кб, 16-битная
- Регистры 8 штук, 16-битные, pc = reg[7]
- функции чтения и записи байта и слова
- загрузка программы в эту память
- прохождение от адреса 1000 до команды halt
  - печать:
    адреса,
    слова по этому адресу,
    имени команды
  - выполнение команды do_halt — остановка

- Печатает

  001000 **01**2700: **mov**
  001002 000002: unknown
  001004 **01**2701: **mov**
  001006 000003: unknown
  001010 **06**0001: **add**
  001002 **000000**: **halt**

Первая программа — сумма 2 чисел

```
.= 1000;          ; разместим код начиная с адреса 1000
mov  #2, R0       ; R0 = 2
mov  #3, R1       ; R1 = 3
add R0, R1        ; R1 = R0 + R1
halt              ; завершение программы
```

```
0200 000c
c0
15
02
00
c1
15
03
00
01
60
00
00
```

- **Все числа восьмеричные**
- С **;** начинаются комментарии
- **.=** - псевдо команда ассемблера,
  где размещать код, который написан дальше
- По умлочанию *эталонный* эмулятор начинает выполнять программу с адреса **1000**. Наш тоже.

Файлы (octal & hex)

./as11 -o sum.o -l sum.l sum.txt

- **sum.txt**

```
.= 1000;          ; разместим код начиная с адреса 1000
mov  #2, R0       ; R0 = 2
mov  #3, R1       ; R1 = 3
add R0, R1        ; R1 = R0 + R1
halt              ; завершение программы
```

- **sum.l** **адрес** **машинный_код** ассемблер

```
000000:       .= 1000
001000:       mov #2, R0  ; R0 = 2
              012700
              000002
001004:       mov #3, R1  ; R1 = 3
              012701
```

sum.o

```
0200 000c
c0
15
02
00
c1
15
03
00
01
60
00
00
```

Запуск эталонного эмулятора

- Использование с ключом **-t** (трассировка)
  ./pdp11 -t sum.o
- Напечатает:

-T еще больше трассировки

```
---------- running ----------
001000: mov     #000002,r0          [001002]=000002
001004: mov     #000003,r1          [001006]=000003
001010: add     r0,r1               R0=000002 R1=000005
001012: halt
---------- halted ----------
r0=000002 r2=000000 r4=000000 sp=000000
r1=000005 r3=000000 r5=000000 pc=001014
psw=000000: cm=k pm=k pri=0       [4]
```

Уже работает, будем разбирать аргументы

- Наш эмулятор должен печатать похоже на эталонный эмулятор и листинг.

  адрес,
  машинный код,
  имя команды

Почему адреса только четные?
Почему после 6 идет 10?
Почему 012700 это mov?

- Напечатает:

001000 012700: **mov**
001002 000002: unknown
001004 012701: **mov**
001006 000003: unknown
001010 060001: **add**
001002 000000: **halt**

Аргументы

- ```
  .= 1000;          ; разместим код с адреса 1000
  mov  #2, R0       ; R0 = 2
  mov  #3, R1       ; R1 = 3
  add R0, R1        ; R1 = R0 + R1
  halt              ; завершение программы
  ```
- halt — аргументов нет
- mov #3, R1
  копировать **что?** число 3 **куда?** в R1.
- add R0, R1
  прочитать **что?** число из R0 и **что?** число из R1, записать сумму **куда?** в R1.

Для аргумента придется хранить и что? значение, и куда? адрес.

```
struct Argument {
    word val;
    word adr;
} ss, dd;
```

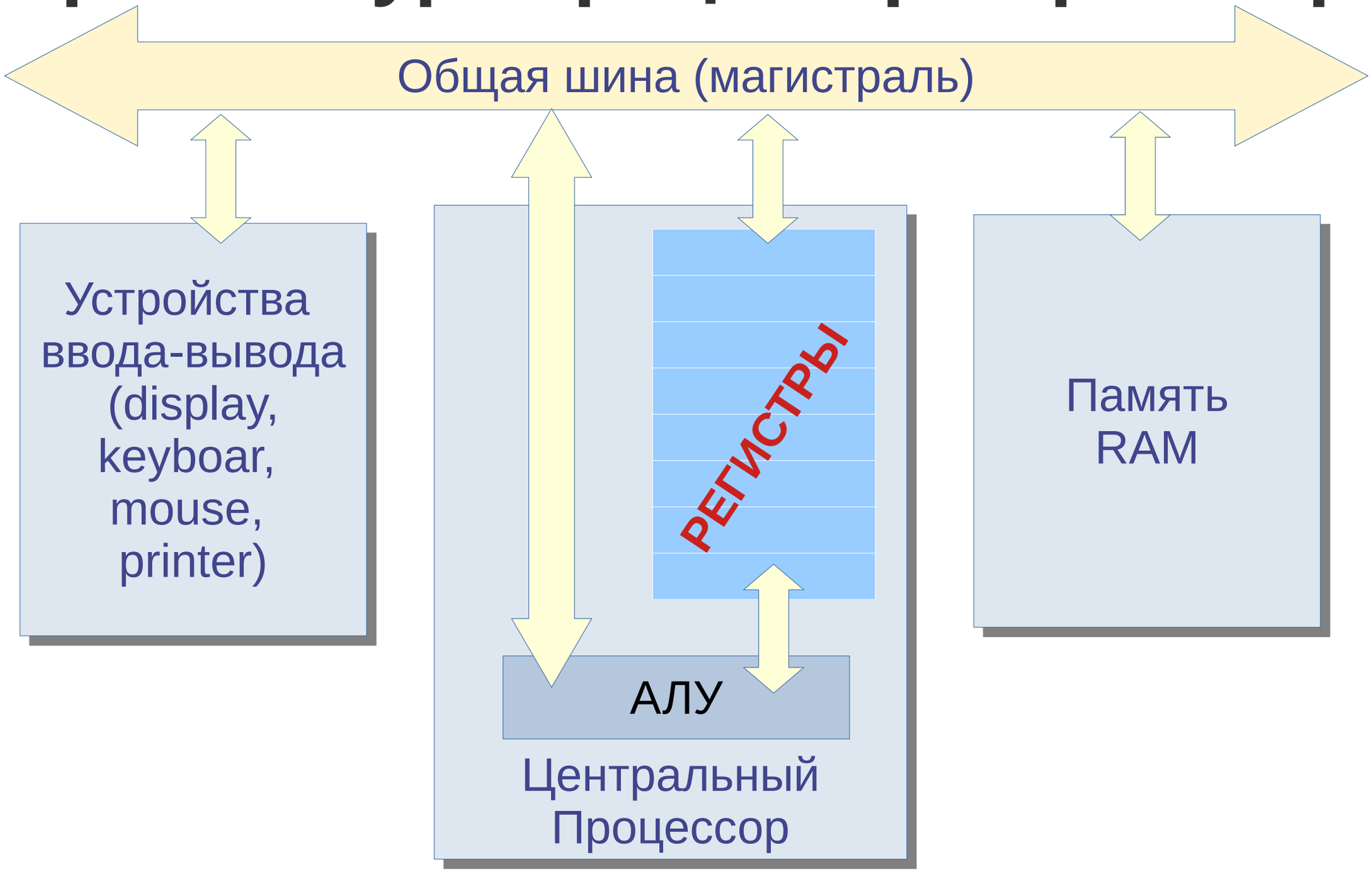
что? - значение value
куда? - адрес address
```

Аргументы SS и DD

Архитектура фон Неймана

- Узкое место — скорость передачи данных из RAM на ЦП (центральный процессор)
- В рамках ЦП сделаем память с очень малым временем доступа (регистры).
  Она дорогая, поэтому ее мало.

Архитектура процессора с регистрами



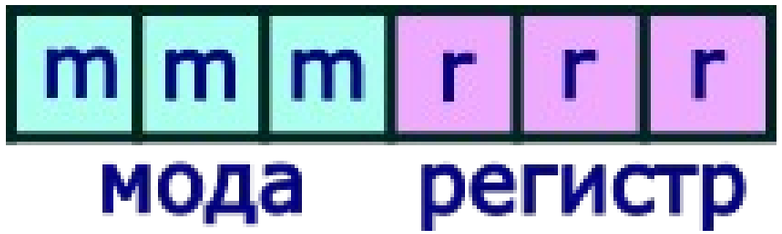
Регистры общего назначения

- RAM — Random Access Memory
- Регистры — память с наименьшим временем доступа
- Общего назначения — к R0 .. R7 обращаемся одинаково
- Нет формата обращения к половине регистра (нельзя "записать число 5 в старший байт регистра")
- 16 бит (слово)
- 8 штук, обращение R0..R7
  add R0, R1
- pc — programm counter (R7), sp — stack pointer (R6)

Операнды SS (source) и DD (destination)

- Команда с одним операндом x++ **INC** *d*
- Код операции **0052*DD***$_8$

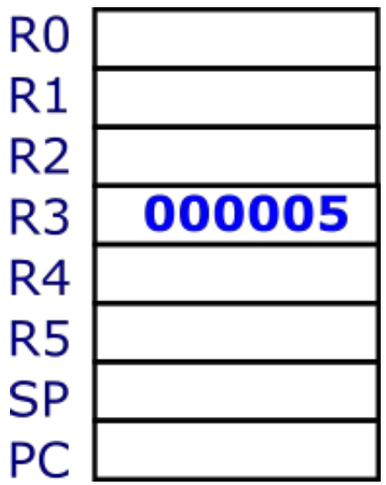
  0 000 101 010 ddd ddd



- DD — 6 бит состоит из:
  - 3 бита — номер регистра
  - 3 бита — режим адресации (мода)
- 8 регистров — сколько бит на кодирование номера регистра (от 0 до 7)?
- сколько разных режимов адресации может быть?

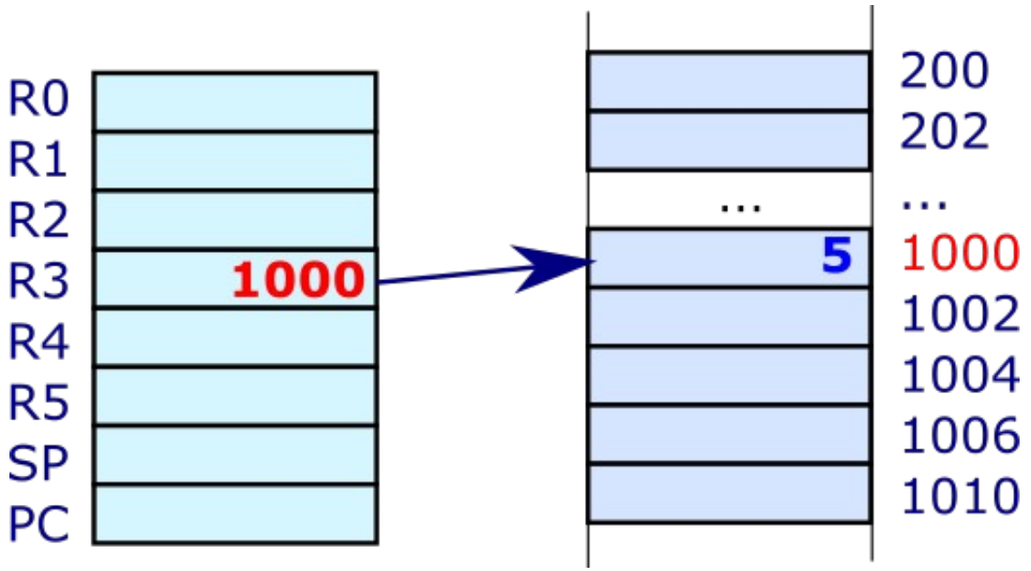
Rn Режим 0 (регистровый)

- Register (храним значение)
- INC R3
  ADD R0, R1
- Регистр содержит операнд (значение)
- Rn — обозначение
- На псевдокоде:
  adr = n
  val = reg[n]



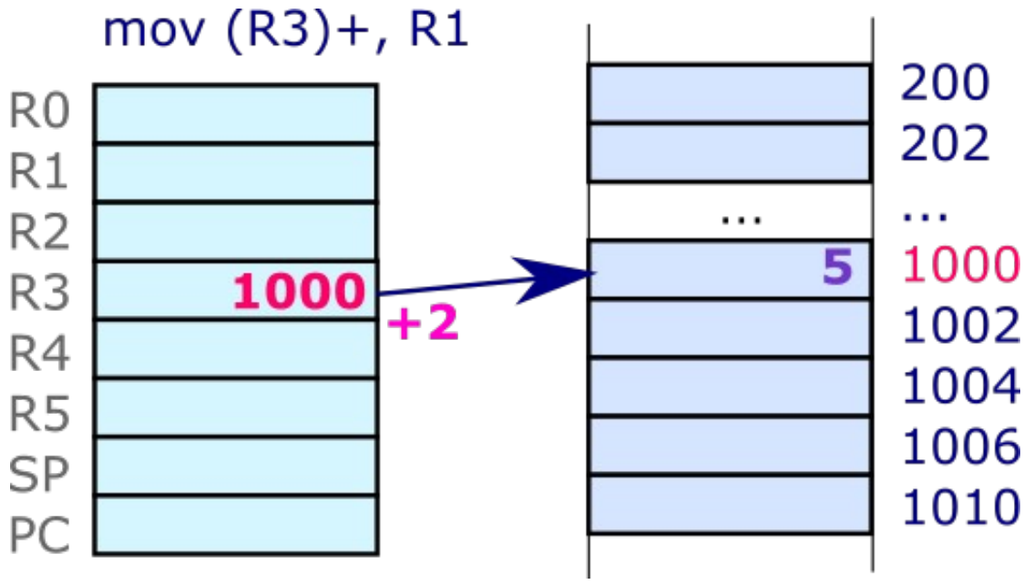
(Rn) Режим 1 (косвенно-регистровый)

- Register deferred (адрес значения)
- INC (R3)
  INC @R3
- регистр содержит адрес операнда
  В регистре лежит номер ячейки памяти, где лежит значение
- **adr** = reg[n]
  val = mem[adr]



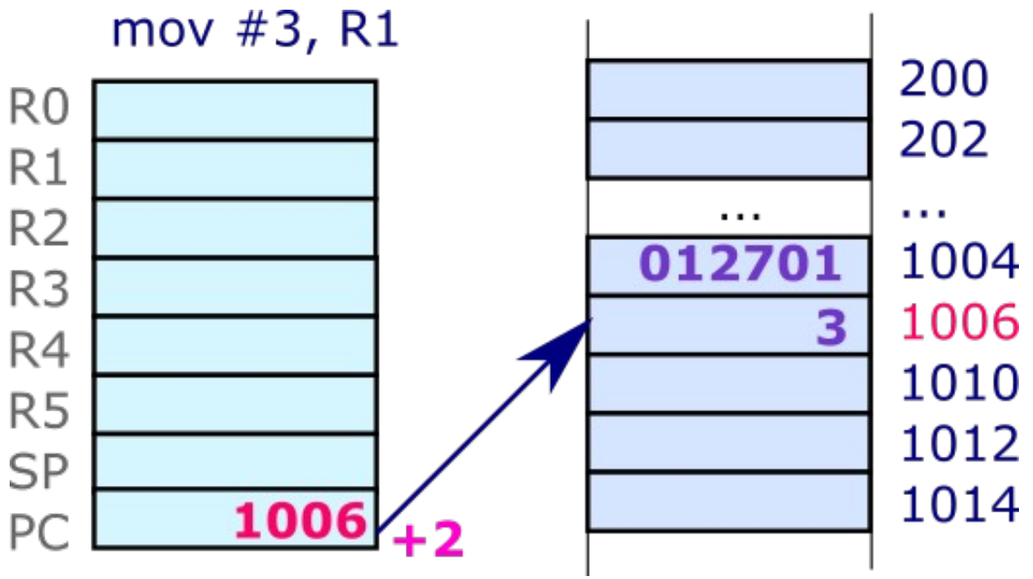
(Rn)+ Режим 2 (автоинкрементный)

- Auto-increment (перебор по массиву чисел)
- INC (R3)+
- регистр содержит адрес операнда.
  Содержимое регистра после операции увеличивается на 2 или на 1 (байтовая операция на регистрах R0-R5)
- **adr** = reg[n]
  val = mem[adr]
  reg[n] += 2 // или 1
- sp и pc всегда +2
- + **потом**



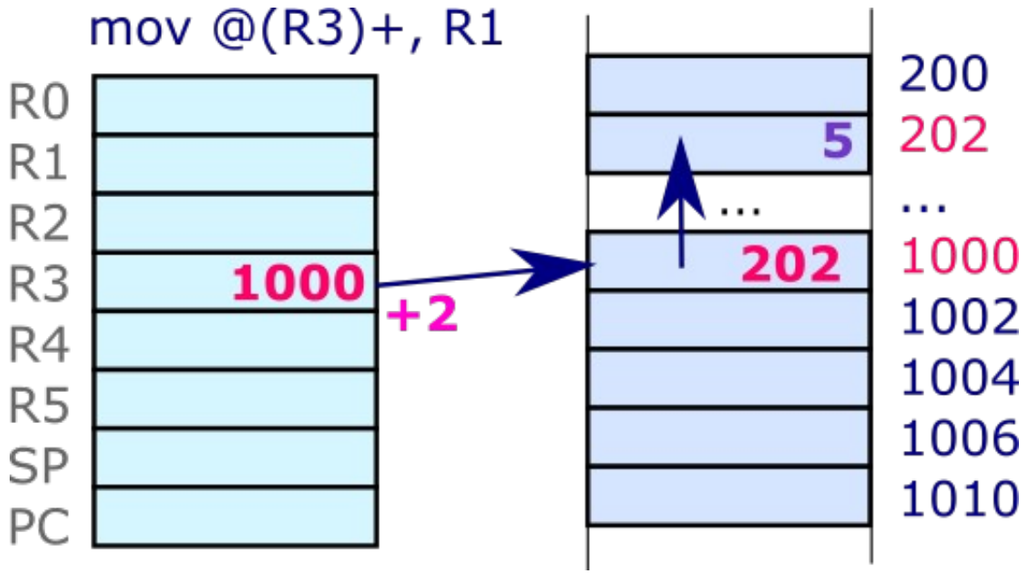
#nn Режим 2 R7 (непосредственный)

- Immediate (константа)
- MOV #3, R1
- MOV (R7)+, R1
  .WORD 3
- операнд хранится в слове, следующем за командой
- **adr** = reg[n]
  val = mem[adr]
  reg[n] += 2 // или 1
- sp и pc всегда +2
- + **потом**



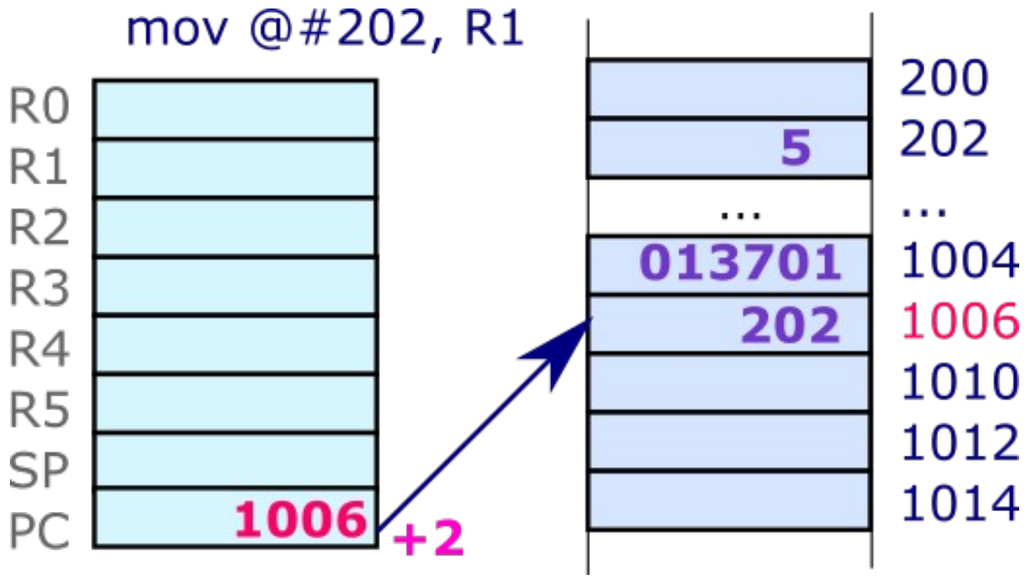
@(Rn)+ Мода 3 (косвенно автоинкрементный

- Auto-increment deferred (перебор по массиву адресов)
- INC @(R3)+
- регистр содержит адрес адреса операнда. Содержимое регистра после его использования в качестве адреса увеличивается на 2
- **adr** = reg[n]
  adr = mem[adr]
  val = mem[adr]
  reg[n] += 2
- +2 (потом!) так как адрес слова



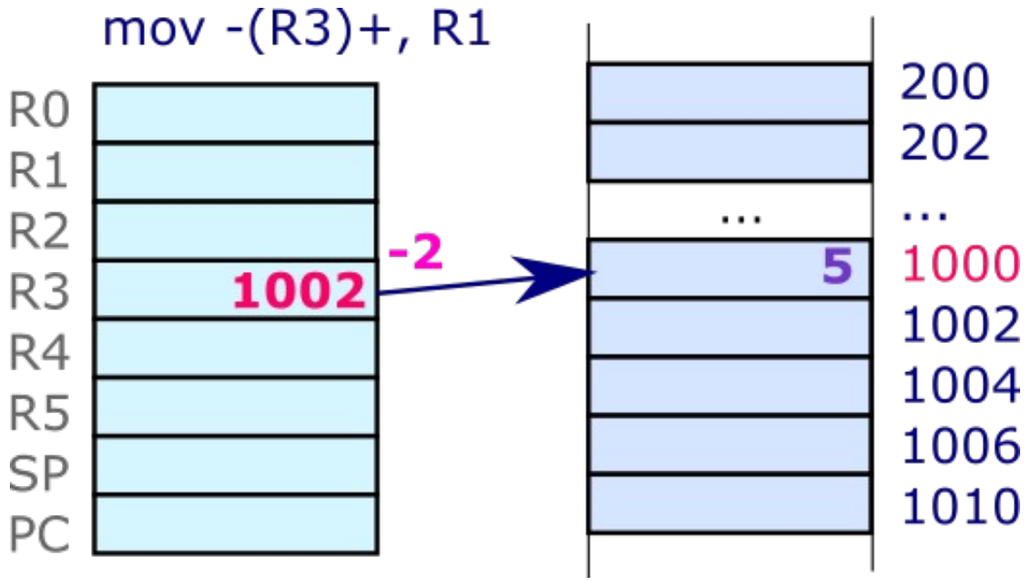
@#nn Мода 3 R7 (абсолютный)

- Absolute (адрес константы)
- mov @#202, R1
- адрес операнда хранится в слове, следующем за командой
- **adr** = reg[n]
  adr = mem[adr]
  val = mem[adr]
  reg[n] += 2
- +2 (потом!) так как адрес слова



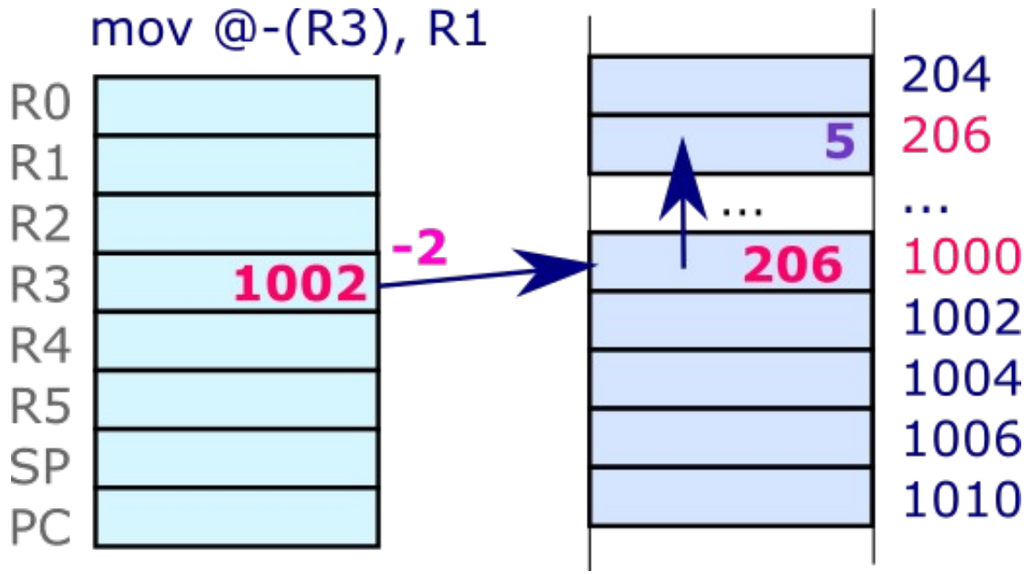
-(Rn) Режим 4 (автодекрементный)

- Auto-decrement (перебор по массиву чисел сзади)
- INC -(R3)
- Содержимое регистра до операции уменьшается на 2 или на 1 (байтовая операция на регистрах R0-R5) и используется как адрес операнда.
- **reg[n] -= 2 // или 1**
  adr = reg[n]
  val = mem[adr]
- sp и pc всегда -2
- **- сначала**



@-(Rn) Мода 5 (косвенно автодекрементный)

- Auto-decrement deferred (перебор по массиву адресов сзади)
- INC @-(R3)
- Содержимое регистра уменьшается на 2 и используется как адрес адреса операнда.
- **reg[n] -= 2**
  adr = reg[n]
  adr = mem[adr]
  val = mem[adr]
- -2 (**сразу**!) так как адрес слова

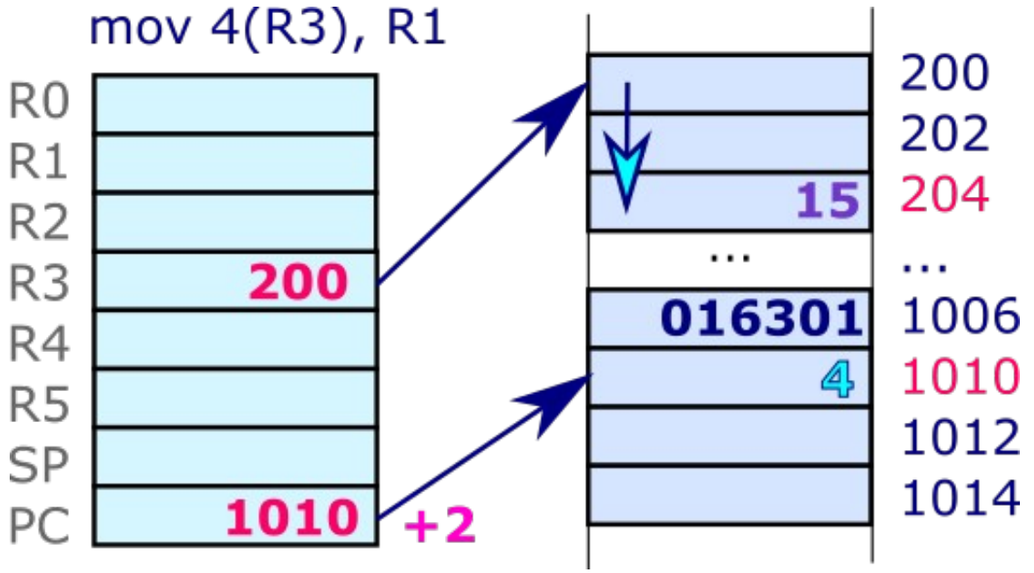


Что делает программа?

- .=1000
  MOV -(PC), -(PC)
- Как называется такой класс программ?
- Какой размер программы?

X(Rn) Мода 6 Индексный

- Index a[i]
- mov 2(R3)
- содержимое регистра складывается с числом, записанным после команды, и полученная сумма используется в качестве адреса операнда
- вычисляем сдвиг X
  X = mem[pc]
  pc += 2
  adr = X + reg[n]
  val = mem[adr]



nn Мода 6 по R7 Относительный

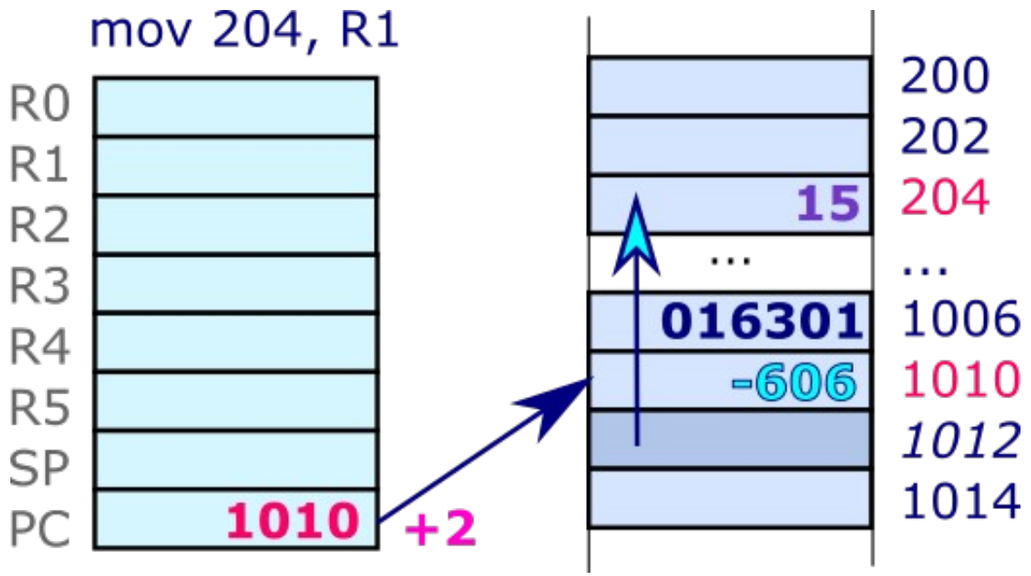
- Relative константа по адресу nn
- mov 100, R3; Число по адресу 100 положить в R3
- содержимое регистра складывается с числом, записанным после команды, и полученная сумма используется в качестве адреса операнда
- вычисляем сдвиг X

  X = mem[pc]

  pc += 2

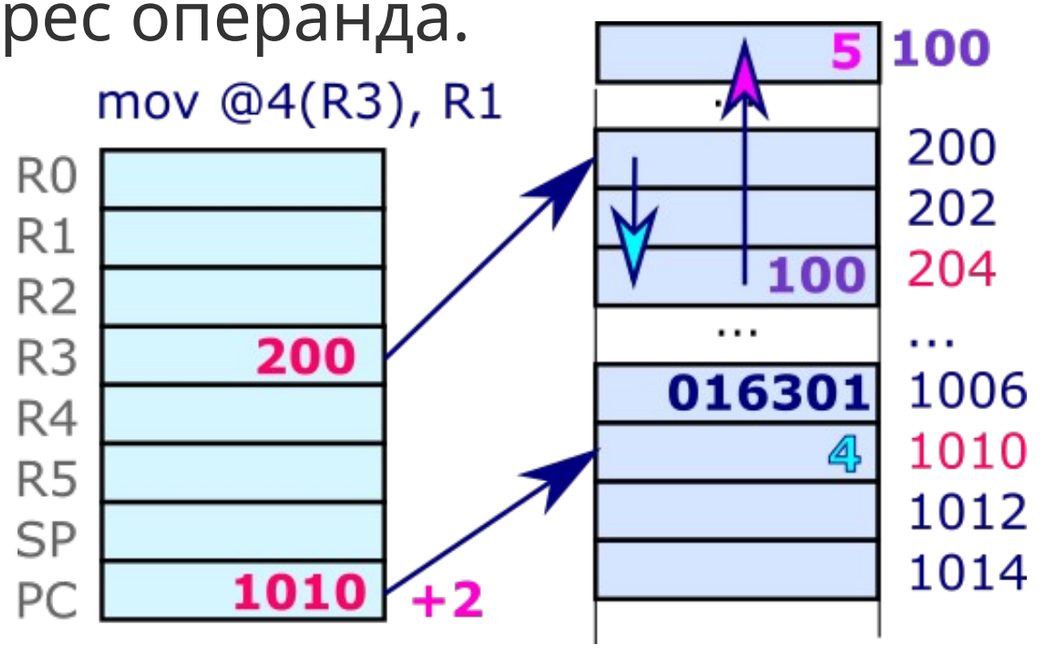
  adr = X + pc

  val = mem[adr]



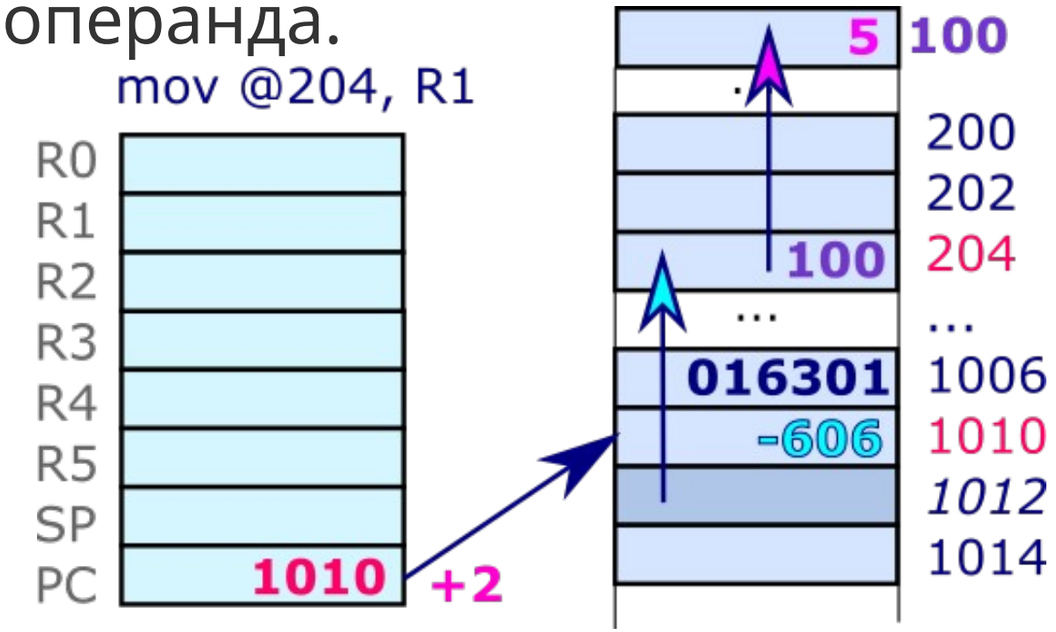
X(Rn) Мода 7 Косвенно индексный

- Index deferred *a[i]
- mov @2(R3)
- индекс лежит в очередном слове команды.
  После считывания индекса pc += 2.
  Индекс + содержимое регистра указывают на ячейку,
  которая содержит адрес операнда.
- вычисляем сдвиг X
  X = mem[pc]
  pc += 2
  adr = X + reg[n]
  adr = mem[adr]
  val = mem[adr]



@nn Мода 7 по R7 Косвенно относительный

- Relative deferred адрес константы по адресу nn
- mov @204, R1
- индекс лежит в очередном слове команды.
  После считывания индекса pc += 2.
  Индекс + содержимое pc указывают на ячейку,
  которая содержит адрес операнда.
- вычисляем сдвиг X
  X = mem[pc]
  pc += 2
  adr = X + pc
  adr = mem[adr]
  val = mem[adr]



	Мода	Пример	Значение
0	R3	inc R3	Значение в регистре R3
1	(R3)	inc (R3)	Адрес в регистре R3
2	(R3)+	inc (R3)+	Адрес в регистре R3, R3 += *2*
3	@(R3)+	inc @(R3)+	Адрес в регистре R3 содержит адрес. R3 += *2*
4	- (R3)	inc -(R3)	R3 -= *2* Адрес в регистре R3
5	@-(R3)	inc @-(R3)	R3 -= *2* Адрес в регистре R3 содержит адрес
6	2(R3)	inc 2(R3)	сложить 2 и R3, это адрес
7	@2(R3)	inc @2(R3)	сложить 2 и R3, по этому адресу лежит адрес

Моды по R7

	Мода	Пример	Значение
2	#3	mov #3, R0	Константа 3
3	@#100	mov @#100, R0	Константа по адресу 100
6	100	mov 100, R0	Константа по адресу 100
7	@100	mov @100, R0	Адрес константы лежит по адресу 100

- Работают по R7 как по другим регистрам
- Пишутся в ассемблере проще
- Абсолютная и относительная адресация

Реализация мод в эмуляторе

- Сделать обязательно 0, 1, 2, желательно 3, 4, 5
- Остальные моды — по наличию тестов (тесты)
- Добиться работоспособности программы 2+3
  Потом теста на 4 моду
- Напечатает:
  001000 012700: mov **#2 R0**
  001004 012701: mov **#3 R1**
  001010 060001: add **R0 R1**
  001002 000000: halt
  r0:3 r1:5 r2:0 r3:0 r4:0 r5:0 r6:0 r7:1014

Что уже написали

- word reg[8]; // регистры R0..R7
  #define pc reg[7] // pc это R7
- struct Arg { // аргумент SS или DD
  word val; // может быть значением
  word adr; // или адресом
  } ss, dd;
- if (это mov?) {
  ss = get_ss(w);
  dd = get_dd(w);
  do_mov( );
  }

- можно ли ss и dd читать одной функцией, назовем ее get_modereg
- важно ли в каком порядке читаем ss и dd
- записать результат в память

Arg get_modereg(word w) {

- int r = w & 7; // номер регистра
  int m = (w >> 3) & 7; // номер моды
  Arg res;
  switch (m) {
  case 0: res.adr = r; res.val = reg[r];
  trace("R%d ", r); break;
  case 1: res.adr = reg[r]; res.val = w_read(res.adr);
  trace("(R%d) ", r); break;
  case 2: res.adr = r; res.val = reg[r]; reg[r] += 2; // +1
  if (r == 7) trace("#%o ", res.val);
  ele trace("(R%d)+ ", r); break;
  default: trace("Mode %d not implemented yet", m);
  exit (1);

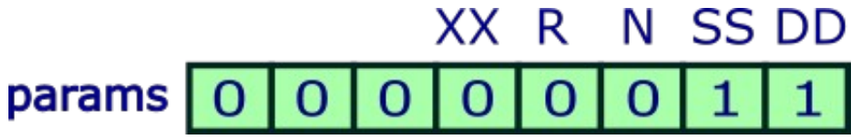
Рекомендация

- Написать функцию, которая печатает все регистры
- Вызывать ее в do_halt
- Вызывать ее сейчас после каждой операции для отладки

Связать аргументы и команды

- if (это mov?) {
  ss = get_ss(w);
  dd = get_dd(w);
  do_mov( );
  }
- В виде цикла
  if (это очередная команда?) {
  ss = get_ss(w);
  dd = get_dd(w);
  nn = get_nn(w);
  xx = get_xx(w);
  do_эта_команда();
  }
- Нельзя всегда get_ss
  потому что можно изменить
  содержимое регистров
- inc R1
  005201
  52 — это часть опкода
  а не 5 мода по R2
  (изменяет R2)

Связать аргументы и команды



- if (это очередная команда?) {
  if (у команды аргумент SS)
  ss = get_ss(w);
  if (у команды аргумент DD)
  dd = get_dd(w);
  …
  do_эта_команда();
  }
-

- #define NO_PARAMS 0
  #define HAS_DD 1
  #define HAS_SS 2
- struct Command {
  word mask;
  word opcode;
  char * name;
  char params;
  };

Продолжение

- Реализовать 4 моду и отладить тест (что он должен делать?)
  mov -(pc), -(pc)

SOB — substruct one and branch

- **SOB Rn, адрес**
- **077RNN**
  - **R** — номер регистра, 3 бита
  - **NN** — на это количество **слов** нужно уйти назад в программе
- **Rn** = **Rn** — 1 // содержимое регистра уменьшить на 1
  if (**Rn** != 0)
    goto адрес; // pc = pc — 2 * **NN**
- Используется для организации цикла со счетчиком в регистре Rn
- CLR dd 0050DD dd = 0 (clear)

Сумма чисел в массиве длиной 4 слова

```
      .= 200        ; данные располагаются с адреса 200
A:  .WORD  34, 12, -1, 66      ; массив  (A — константа равна 200)
N: .WORD  4                    ; размер массива A
    .= 1000          ; код располагается с адреса 1000
    mov #4, R1     ; заменить потом #4 на @#N и мода 3
    mov #A, R2     ; адрес начала массива в R2
    clr R0           ; R0 = 0
LOOP:
    mov (R2)+, R3     ; R3 = *R2, R2 += 2 получили слагаемое
    add R3, R0          ; слагамое добавили к сумме
    sob R1, LOOP     ; if (-- R1 != 0) goto LOOP

    halt
```

Команды PDP-11 mov, add, halt, clr, sob

- Список команд http://acm.mipt.ru/twiki/bin/view/Cintro/PDPCommandList
- SS и DD — 6-битные аргументы
- B — 0 или 1 — команда оперирует словами или байтами

Mnemonic	Opcode	NZVC	Description	Notes
ADD s, d	**06**SSDD	****	Add	d = s + d
CLRb d	**B050**DD	****	Clear	d = 0
MOVb s,d	**B1**SSDD	****	Move	d = s
HALT	**0000000**	---	Halt	
SOB r,a	**077RNN**	----	Substruct One and Branch	if (--R != 0) pc = pc - NN*2

Переведем в машинные коды

- 001012: LOOP:
- 001012 : mov (R2)+, R3 ; читаем слагаемое
  012203
- 001014: add R3, R0 ; добавляем слагаемое
  060300
- 001016: sob R1, LOOP ; if (--R1 != 0) goto LOOP;
  077103
- 001020: halt
  000000

Массив байт

- После того, как заработает, будем суммировать массив не слов, а байт
- Было:

  A: **.WORD** 34, 12, -1, 66

  N: .WORD 4

  mov — копировать слово
- Стало

  A: **.BYTE** 34, 12, -1, 66

  N: .WORD 4

  movb — копировать байт

Чтение и запись байта в регистр

- Чтение байта
  movb R1, (R3)
  Берем младший байт.
- Запись байта
  movb (R3), R1
  Положительное должно остаться положительным, отрицательное отрицательным
- Знаковое расширение: запишем в старший байт знаковый бит младшего байта