МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«ПОВОЛЖСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ
УНИВЕРСИТЕТ»

Кафедра ИиСП

Отчет
по лабораторной работе № 3

по дисциплине «Машинно-зависимые языки программирования»
Вариант 21

Выполнил: ст. гр. ПС-11
Маркин И. А.
Проверил: доцент, доцент кафедры ИиСП Баев А.А.

г. Йошкар-Ола
2025

Цель работы: Протестировать команды и написать простой алгоритм в Atmel Studio 7

Задания на лабораторную работу:

1)Протестировать и написать программы с помощью математических и логических операций

2)Разобраться с программами для условных переходов, а также написать к ним программму

3)Написать программу для сортировки массива вставкой

1. **Теоретические сведения**

https://trolsoft.ru/ru/avr-assembler

http://easyelectronics.ru/avr-uchebnyj-kurs-flagi.html

https://easyelectronics.ru/avr-uchebnyj-kurs-makroassembler.html

http://easyelectronics.ru/skazhu-paru-slov-o-optimizacii-koda.html

https://www.radiokot.ru/start/mcu_fpga/avr/14/

https://proglib.io/p/7-sposobov-sortirovki-massivov-na-primere-s-s-illyustraciyami-2022-04-20

2. Практическая часть

1) **Проверка математических и логических операций, работа с битами**

Тестирование программы с помощью команды Com

Мнемоника COM

Операнды Rd

Описание Побитная инверсия

Операция Rd = $FF - Rd

Флаги Z,C,N,V,S

Циклы 1

Задача: проверить изменение флагов с помощью команды Com

Текст программы:

```
reset:
     rjmp main
main:
     ldi r16, 0xFF
     out PORTB, r16
loop:
     in r16, PORTB
     com r16
     out PORTB, r16

     rjmp loop
```

Входной файл:

$log PORTB

$log SREG

$startlog Asm_Lab_Math_log_output.stim

#3

PORTB = 0

#3

PORTB = 255

#4

PORTB = 0

#4

PORTB = 255

$stoplog

$break

Выходной файл:

#4

SREG = 0x15

#1

PORTB = 0xff

#4

SREG = 0x03

#1

PORTB = 0x00

PORTB	PORTB`	SREG	Флаги
0x00	0xFF	0x15	S,N,C
0xFF	0x00	0x03	Z,C

Итог: Программа включает порт PORTB на 255, затем циклично инвертирует его состояние между 0 и 255. Логи фиксируют изменения состояния порта и флаги в SREG, подтверждая успешное переключение состояния порта.

Тесторивание программы с помощью команды Swap

Мнемоника SWAP

Операнды Rd

Описание Перестановка тетрад

Операция Rd(3..0) = Rd(7..4), Rd(7..4) =Rd(3..0)

Флаги None

Циклы 1

Задача: проверить действительно ли Swap не влияет на флаги SREG

Текст программы:

```
reset:
     rjmp main
main:
     ldi r16, 0xA5
     out PORTB, r16
loop:
     in r16, PORTB
     swap r16
     out PORTB, r16

     rjmp loop
```

Входной файл:

$log PORTB

$log SREG

$startlog Asm_Lab_Math_log_output.stim

#3

PORTB = 165

#3

PORTB = 90

$stoplog

$break

Выходной файл:

#2

PORTB = 0xa5

#3

PORTB = 0x5a

PORTB	PORTB`	SREG	Флаги
0xa5	0x5a	-	None

Итог: Программа инициализирует порт PORTB значением 0xA5. Затем, в цикле, выполняется команда swap, которая меняет старшие и младшие 4 бита, результатом чего становится 0x5A. Логи фиксируют эти изменения.

Тестирование программы с помощью команды Ror

Мнемоника ROR

Операнды Rd

Описание Циклический сдвиг вправо через C

Операция Rd(7)=C, Rd(n)=Rd(n+1),C=Rd(0)

Флаги Z,C,N,V,S

Циклы 1

Текст программы:

```
reset:
     rjmp main
main:
     ldi r16, 0x01
     out PORTB, r16
loop:
     in r16, PORTB
     ror r16
     out PORTB, r16

     rjmp loop
```

Текстовый файл:

$log PORTB

$log SREG

$startlog Asm_Lab_Math_log_output.stim

#3

PORTB = 1

#2

PORTB = 0

#5

PORTB = 128

#5

PORTB = 64

#5

PORTB = 32

#5
PORTB = 16
#5
PORTB = 8
#5
PORTB = 4
#5
PORTB = 2
#5
PORTB = 1
#5
PORTB = 0
#5
$stoplog
$break
Выходной файл:
#2
PORTB = 0x01
#1
SREG = 0x1b
#1
PORTB = 0x00
#4
SREG = 0x0c
#1
PORTB = 0x80
#4
SREG = 0x00
#1
PORTB = 0x40
#5

PORTB = 0x20

#5

PORTB = 0x10

#5

PORTB = 0x08

#5

PORTB = 0x04

#5

PORTB = 0x02

#5

PORTB = 0x01

#4

SREG = 0x1b

#1

PORTB = 0x00

PORTB	PORTB`	SREG	Флаги
0x01	-	0x1B	S,V,Z,C
0x00	0x01	0x0C	V,N
0x80	0x00	0x00	None
0x40	0x80	-	-
0x20	0x40	-	-
0x10	0x20	-	-
0x08	0x10	-	-
0x04	0x08	-	-
0x02	0x04	-	-
0x01	0x02	0x1B	S,V,Z,C
0x00	0x01	-	-

Итог: Программа инициализирует порт PORTB значением 0x01 и циклично сдвигает это значение вправо с помощью инструкции ror. Состояния порта

меняются от 0x01 до 0x80 и обратно к 0x00, и так далее. Логи фиксируют каждое изменение состояния порта и флаги в SREG.

Тестирование программы с помощью команды Adiw

Мнемоника ADIW

Операнды Rd, K6

Описание Сложить константу и слово

Операция Rdh:Rdl = Rdh:Rdl +K6═

Флаги Z,C,N,V,S

Циклы 2

Текст программы:

```
reset:
     rjmp main
main:
     ldi r24, 0x00
     ldi r25, 0x01
     out PORTB, r24
     out PORTD, r25
loop:
     out PORTB, r24
     out PORTD, r25
     adiw r24, 1

     rjmp loop
```

Входной файл:

$log PORTB

$log PORTD

$log SREG

$startlog Asm_Lab_Math_log_output.stim

#5

PORTB = 0

PORTD = 255

#6

PORTB = 1

PORTD = 255

#6

PORTB = 2

PORTD = 254

#6

PORTB = 3

PORTD = 253

#6

PORTB = 4

PORTD = 252

#6

PORTB = 5

PORTD = 251

#6

PORTB = 6

PORTD = 250

#6

PORTB = 7

PORTD = 249

#6

PORTB = 8

PORTD = 248

#6

PORTB = 9

PORTD = 247

#6

$stoplog

$break

Выходной файл:

#4

```
PORTD = 0x01
#1
PORTD = 0xff
#1
PORTD = 0x01
#5
PORTB = 0x01
PORTD = 0xff
#1
PORTD = 0x01
#5
PORTB = 0x02
PORTD = 0xfe
#1
PORTD = 0x01
#5
PORTB = 0x03
PORTD = 0xfd
#1
PORTD = 0x01
#5
PORTB = 0x04
PORTD = 0xfc
#1
PORTD = 0x01
#5
PORTB = 0x05
PORTD = 0xfb
#1
PORTD = 0x01
#5
```

PORTB = 0x06

PORTD = 0xfa

#1

PORTD = 0x01

#5

PORTB = 0x07

PORTD = 0xf9

#1

PORTD = 0x01

#5

PORTB = 0x08

PORTD = 0xf8

#1

PORTD = 0x01

#5

PORTB = 0x09

PORTD = 0xf7

#1

PORTD = 0x01

PORTB	PORTD	PORTB`	PORTC`	SREG	Флаги
0x00	0xFF	0x01	0xFF	-	-
0x01	0xFF	0x02	0x01	-	-
0x02	0xFE	0x02	0xFA	-	-
0x03	0xFD	0x03	0x01	-	-
0x04	0xFC	0x04	0xF9	-	-
0x05	0xFB	0x05	0x01	-	-
0x06	0xFA	0x06	0xF8	-	-
0x07	0xF9	0x07	0x01	-	-
0x08	0xF8	0x08	0xF7	-	-
0x09	0xF7	0x09	0x01	-	-

Итог: Программа последовательно увеличивает значение порта PORTB от 0 до 9 и уменьшает значение порта PORTD от 255 до 247. Логи фиксируют эти изменения на каждом шаге, а флаги в SREG показывают, что процесс выполняется без ошибок, так как нет признаков переноса или других флагов, влияющих на работу программы.

Тестирование программы с помощью команды Sbc

Мнемоника SBC

Операнды Rd, Rr

Описание Вычитание с переносом

Операция Rd = Rd - Rr – C

Флаги Z,C,N,V,H,S

Циклы 1

Текст программы:

```
reset:
     rjmp main
main:
     ldi r16, 0x10
     ldi r17, 0x05
     out PORTB, r16
     out PORTC, r17
loop:
     in r16, PORTB
     in r17, PORTC
     sbc r16, r17
     out PORTB, r16
     out PORTC, r17

     rjmp loop
```

Входной файл:

$log PORTB

$log PORTC

$log SREG

$startlog Asm_Lab_Math_SBC_log_output.stim

#5

PORTB = 10

PORTC = 5

#8

PORTB = 20

PORTC = 15

#8

PORTB = 0

PORTC = 255

#8

$stoplog

$break

Выходной файл:

#3

PORTB = 0x10

#1

PORTC = 0x05

#1

PORTB = 0x0a

#3

PORTB = 0x05

#5

PORTB = 0x14

PORTC = 0x0f

#1

SREG = 0x35

#1

PORTB = 0xf6

#6

PORTB = 0x00

PORTC = 0x7f

SREG = 0x34

#1

PORTB = 0xe6

#1

PORTC = 0x0f

PORTB	PORTD	PORTB`	PORTD`	SREG	Флаги
0x0A	0x05	0x10	0x05	0x35	H,S,N,C
0x14	0x0F	0x0A	0x05	0x34	H,S,N
0x00	0xFF	0x14	0x0F	0x3F	H,S,V,N,Z,C

Итоги: Программа выполняет операцию вычитания с учетом переноса на каждом цикле между значениями портов PORTB и PORTC. Логи показывают последовательность изменений на портах и соответствующие флаги в регистре SREG, что подтверждает правильность выполнения операций с учетом возможных переносов и изменений значений.

2)Условные переходы

Далее для программ с условными переходами я буду использовать команду `cp` для сравнения двух регистров

Тестирование программы с помощью команды brlt

Мнемоника BRLT

Операнды k

Описание Перейти если меньше (со знаком)

Операция f(S==1) PC = PC + k + 1

Флаги None

Циклы ½

Текст программы:

```
reset:
     rjmp main
main:
     ldi r16, 0x00
```

```
    ldi r17, 0x01
loop:
    in r16, OCR0A
    in r17, OCR0B
    cp r16, r17
    brlt result
    rjmp loop
result:
    out PORTB, r16
    rjmp loop
```

Входной файл:

$log PORTB

$log SREG

$startlog Asm_Lab_Math_log_output.stim

#3

OCR0A = 10

OCR0B = 20

#6

OCR0A = 30

OCR0B = 15

#6

OCR0A = 50

OCR0B = 50

#6

OCR0A = 5

OCR0B = 100

#8

$stoplog

$break

Выходной файл:

#5

SREG = 0x15

#3

PORTB = 0x0a

#5

SREG = 0x20

#6

SREG = 0x02

#6

SREG = 0x15

#3

PORTB = 0x05

	OCR0A	OCR0B	PORTB
1.	0x0A	0x14	0x0A
2.	0x1E	0x0F	
3.	0x32	0x32	
4.	0x05	0x64	0x05

Итоги: Программа сравнивает значения в регистрах OCR0A и OCR0B. Если значение в OCR0A меньше, чем в OCR0B, значение из OCR0A записывается в PORTB. Логи показывают, что это сравнение выполняется правильно, и значения из OCR0A записываются в PORTB, как только условие выполняется (в том числе при изменении значений в OCR0A и OCR0B).

Тестирование программы с помощью команды CPSE

Мнемоника CPSE

Операнды Rd, Rr

Описание Сравнить, пропустить если равны=

Операция if (Rd ==Rr) PC = PC 2 or 3

Флаги None

Циклы 1/2/3

Текст программы:

```
reset:
     rjmp main
main:
     ldi r16, 0x00
```

```
    ldi r17, 0x01
loop:
    in r16, OCR0A
    in r17, OCR0B
    cp r16, r17
    cpse result
    rjmp loop
result:
    out PORTB, r16
    rjmp loop
```

Входной файл:

$log PORTB

$log SREG

$startlog Asm_Lab_Math_log_output.stim

#3

OCR0A = 12

OCR0B = 18

#6

OCR0A = 25

OCR0B = 10

#6

OCR0A = 40

OCR0B = 40

#6

OCR0A = 3

OCR0B = 90

#8

$stoplog

$break

Выходной файл:

#5

SREG = 0x15

#6

SREG = 0x20

#6

SREG = 0x02

#3

PORTB = 0x28

#5

SREG = 0x35

	OCR0A	OCR0B	PORTB
1.	0x0C	0x12	-
2.	0x19	0x0A	-
3.	0x28	0x28	0x28
4.	0x03	0x5A	-

Итоги: Программа выполняет сравнение значений в регистрах OCR0A и OCR0B и записывает значение из OCR0A в PORTB только в том случае, если они равны. Логи показывают, что программа корректно обрабатывает эту логику, и значение в PORTB обновляется только тогда, когда значения в OCR0A и OCR0B равны.

Тестирование программы с помощью команды BRVC

Мнемоника BRVC

Операнды k

Описание Перейти если флаг переполнения очищен

Операция if(V==0) PC = PC + k + 1

Флаги None

Циклы 1/2

Текст программы:

```
reset:
     rjmp main
main:
     ldi r16, 0x00
     ldi r17, 0x01
loop:
     in r16, OCR0A
```

```
    in r17, OCR0B
    cp r16, r17
    brvc result
    rjmp loop
result:
    out PORTB, r16
    rjmp loop
```

Входной файл:

$log PORTB

$log SREG

$startlog Asm_Lab_Math_log_output.stim

#3

OCR0A = 12

OCR0B = 15

#6

OCR0A = 30

OCR0B = 20

#6

OCR0A = 45

OCR0B = 45

#6

OCR0A = 5

OCR0B = 50

#8

$stoplog

$break

Выходной файл:

#5

SREG = 0x35

#3

PORTB = 0x0c

#5

SREG = 0x00

#3

PORTB = 0x1e

#5

SREG = 0x02

#3

PORTB = 0x2d

	OCR0A	OCR0B	PORTB
1.	0x0C	0x0F	0x0C
2.	0x1E	0x14	0x1E
3.	0x2D	0x2D	0x2D
4.	0x05	0x32	-

Итоги: Программа выполняет сравнение значений в OCR0A и OCR0B и записывает значение из OCR0A в PORTB только в том случае, если значение в OCR0A меньше или равно значению в OCR0B. Логи показывают корректное выполнение этого процесса, и значения из OCR0A записываются в PORTB в соответствующих циклах.

Тестирование программы с помощью команды BRTS

Мнемоника BRTS

Операнды k

Описание Перейти если флаг T установлен

Операция if(T==1) PC = PC + k + 1

Флаги None

Циклы 1/2

Текст программы:

```
reset:
     rjmp main
main:
     ldi r16, 0x00
     ldi r17, 0x01
loop:
     in r16, OCR0A
```

```
    in r17, OCR0B
    cp r16, r17
    brts result
    rjmp loop
result:
    out PORTB, r16
    rjmp loop
```

Входной файл:

$log PORTB

$log SREG

$startlog Asm_Lab_Math_log_output.stim

#3

OCR0A = 12

OCR0B = 15

#6

OCR0A = 30

OCR0B = 20

#6

OCR0A = 45

OCR0B = 45

#6

OCR0A = 5

OCR0B = 50

#8

$stoplog

$break

Выходной файл:

#5

SREG = 0x35

#6

SREG = 0x00

#6

SREG = 0x02

#6

SREG = 0x15

	OCR0A	OCR0B	PORTB
1.	0x0C	0x0F	-
2.	0x1E	0x14	-
3.	0x2D	0x2D	-
4.	0x05	0x32	-

Итоги: Если не установлен флаг T, то в PORTB никаких значений не будет выводится

Тестирование программы с помощью команды CPC

Мнемоника CPC

Операнды Rd, Rr

Описание Сравнить с переносом

Операция Rd - Rr - C

Флаги Z,C,N,V,H,S

Циклы 1

Счетчик программ

Текст программы:

```
reset:
     rjmp main
main:
     ldi r16, 0x00
     ldi r17, 0x01
loop:
     in r16, OCR0A
     in r17, OCR0B
     cp r16, r17
     cpc r16, r17
     breq result
     rjmp loop
result:
     out PORTB, r16
     rjmp loop
```

Входной файл:

```
$log PORTB
$log SREG
$startlog Asm_Lab_Math_log_output.stim
#3
OCR0A = 12
OCR0B = 15
#6
OCR0A = 30
OCR0B = 20
#6
OCR0A = 45
OCR0B = 45
#6
OCR0A = 5
OCR0B = 50
#8
$stoplog
$break
```

Выходной файл:

```
#5
SREG = 0x35
#7
SREG = 0x00
#7
SREG = 0x02
#4
PORTB = 0x2d
#5
SREG = 0x15
```

	OCR0A	OCR0B	PORTB
1.	0x0C	0x0F	-
2.	0x1E	0x14	-
3.	0x2D	0x2D	0x2D
4.	0x05	0x32	-

Итоги: Программа сравнивает значения в регистрах OCR0A и OCR0B с использованием двух команд сравнения, учитывая флаг переноса. Если значения равны, программа записывает значение из OCR0A в PORTB. Логи показывают, что при совпадении значений в OCR0A и OCR0B происходит запись в PORTB, и процесс повторяется.

3)Задание с массивами

1.Программа сортировки вставкой для типа BYTE:

```
.set ARR_SIZE=10
; В сегменте данных выделим массив типа BYTE
.dseg
arr: .BYTE ARR_SIZE
.cseg
reset:
    rjmp main

main:
      ldi ZH,High(src*2)   ; Загрузка адреса 0-го
      ldi ZL,Low(src*2)    ; Элемента в рег. пару Z
      ldi YH,High(arr)     ; Загрузка адреса 0-го
      ldi YL,Low(arr)      ; Элемента в рег. пару Y
      ldi R18, ARR_SIZE  ; Загрузка размера массива в счётчик элементов
массива

copy_loop:
    lpm R0, Z+           ; Чтение из Flash
    st Y+, R0            ; Запись в RAM
    dec R18
    brne copy_loop

    rjmp insertion_sort  ; Сортировка вставками

insertion_sort:
    ldi R20, 1           ; Начинаем с первого элемента (индекс 1)

sort_outer:
    cpi R20, ARR_SIZE    ; Проверяем, дошли ли до конца массива
    brge sort_done       ; Если да, завершаем сортировку
```

```
    mov R21, R20          ; Сохраняем текущий индекс во внешнем цикле
    ldi YH, High(arr)     ; Загружаем адрес массива в Y
    ldi YL, Low(arr)
    add YL, R21           ; Переходим к текущему элементу
    ld R22, Y             ; Загружаем текущий элемент в R22

sort_inner:
    cpi R21, 0            ; Проверяем, дошли ли до начала массива
    breq insert_done      ; Если да, выходим из внутреннего цикла

    dec R21               ; Переходим к предыдущему элементу
    ldi YH, High(arr)     ; Загружаем адрес массива в Y
    ldi YL, Low(arr)
    add YL, R21           ; Переходим к предыдущему элементу
    ld R23, Y             ; Загружаем предыдущий элемент в R23

    cp R22, R23           ; Сравниваем текущий элемент с предыдущим
    brsh insert_done      ; Если текущий элемент больше или равен,
выходим(беззнаковые)

    std Y+1, R23          ; Сдвигаем предыдущий элемент вправо
    rjmp sort_inner       ; Повторяем внутренний цикл

insert_done:
    std Y+1, R22          ; Вставляем текущий элемент на правильное место
    inc R20               ; Переходим к следующему элементу
    rjmp sort_outer       ; Повторяем внешний цикл

sort_done:
    rjmp loop             ; Завершаем сортировку

loop:
    rjmp loop             ; Бесконечный цикл

.cseg
; В сегменте кода выделим массив типа BYTE
src: .db 0x02, 0x6B, 0xAE, 0x60, 0x7F, 0x69, 0xD8, 0x2B, 0x8E, 0x15
```

Примерное количество циклов 710

Было:

`0x02 0x6B 0xAE 0x60 0x7F 0x69 0xD8 0x2B 0x8E 0x15`

Стало:

`0x02 0x15 0x2B 0x60 0x69 0x6B 0x7F 0x8E 0xAE 0xD8`

Дамп памяти:

Asm_Lab_Array_mem_F_log_output:

:02001E00D1E02F

:08002000C0E0C50F68815030FB

:00000001FF

Asm_Lab_Array_mem_S_log_output:

:0A01000002152B60696B7F8EAED8EC

:00000001FF

HEX: 0x02, 0x15, 0x2B, 0x60, 0x69, 0x6B, 0x7F, 0x8E, 0xAE, 0xD8

DEC (беззнак.): 2, 21, 43, 96, 105, 107, 127, 142, 174, 216

2.Программа сортировки вставкой для Знакового байта:

```
.set ARR_SIZE=10
; В сегменте данных выделим массив типа BYTE
.dseg
arr: .BYTE ARR_SIZE
.cseg
reset:
    rjmp main

main:
      ldi ZH,High(src*2)   ; Загрузка адреса 0-го
      ldi ZL,Low(src*2)    ; Элемента в рег. пару Z
      ldi YH,High(arr)     ; Загрузка адреса 0-го
      ldi YL,Low(arr)      ; Элемента в рег. пару Y
      ldi R18, ARR_SIZE  ; Загрузка размера массива в счётчик элементов
массива

copy_loop:
    lpm R0, Z+           ; Чтение из Flash
    st Y+, R0            ; Запись в RAM
    dec R18
    brne copy_loop

    rjmp insertion_sort  ; Сортировка вставками

insertion_sort:
    ldi R20, 1           ; Начинаем с первого элемента (индекс 1)

sort_outer:
    cpi R20, ARR_SIZE    ; Проверяем, дошли ли до конца массива
    brge sort_done       ; Если да, завершаем сортировку

    mov R21, R20         ; Сохраняем текущий индекс во внешнем цикле
    ldi YH, High(arr)    ; Загружаем адрес массива в Y
    ldi YL, Low(arr)
    add YL, R21          ; Переходим к текущему элементу
    ld R22, Y            ; Загружаем текущий элемент в R22

sort_inner:
    cpi R21, 0           ; Проверяем, дошли ли до начала массива
    breq insert_done     ; Если да, выходим из внутреннего цикла

    dec R21              ; Переходим к предыдущему элементу
```

```
    ldi YH, High(arr)    ; Загружаем адрес массива в Y
    ldi YL, Low(arr)
    add YL, R21          ; Переходим к предыдущему элементу
    ld R23, Y            ; Загружаем предыдущий элемент в R23

    cp R22, R23          ; Сравниваем текущий элемент с предыдущим
    brge insert_done     ; Если текущий элемент больше или равен,
выходим(знаковые)

    std Y+1, R23         ; Сдвигаем предыдущий элемент вправо
    rjmp sort_inner      ; Повторяем внутренний цикл

insert_done:
    std Y+1, R22         ; Вставляем текущий элемент на правильное место
    inc R20              ; Переходим к следующему элементу
    rjmp sort_outer      ; Повторяем внешний цикл

sort_done:
    rjmp loop            ; Завершаем сортировку

loop:
    rjmp loop            ; Бесконечный цикл

.cseg
; В сегменте кода выделим массив типа BYTE
src: .db 0x8E, 0x6B, 0xAE, 0x02, 0x7F, 0x69, 0xD8, 0x2B, 0x60, 0x15
```

Примерное количество циклов 580

Было:

0x8E 0x6B 0xAE 0x02 0x7F 0x69 0xD8 0x2B 0x60 0x15

Стало:

0x8E 0xAE 0xD8 0x02 0x15 0x2B 0x60 0x69 0x6B 0x7F

Asm_Lab_Array_mem_F_log_output:

:02001E00D1E02F

:08002000C0E0C40F68815030FC

:00000001FF

Asm_Lab_Array_mem_S_log_output:

:0A0100008EAED802152B60696B7EC

:00000001FF

HEX: 0x8E, 0xAE, 0xD8, 0x02, 0x15, 0x2B, 0x60, 0x69, 0x6B, 0x7F

DEC (знак.): -114, -82, -40, 2, 21, 43, 96, 105, 107, 127

3.Программа сортировки вставками для слова

```
.set ARR_SIZE = 10          ; Размер массива (10 элементов)
.dseg
arr: .BYTE ARR_SIZE*4     ; Память под массив (10*4 байт)
.cseg
reset:
    rjmp main

main:
    ldi ZH, High(src*2)   ; Загрузка адреса исходного массива в Flash
    ldi ZL, Low(src*2)
    ldi YH, High(arr)     ; Загрузка адреса целевого массива в RAM
    ldi YL, Low(arr)
    ldi R20, ARR_SIZE     ; Счетчик элементов

copy_loop:
    lpm R16, Z+           ; Чтение младшего байта
    lpm R17, Z+
    lpm R18, Z+
    lpm R19, Z+           ; Чтение старшего байта (знаковый бит)
    st Y+, R16                ; Сохранение в RAM
    st Y+, R17
    st Y+, R18
    st Y+, R19
    dec R20
    brne copy_loop

    rjmp insertion_sort

insertion_sort:
    ldi R20, 4            ; i = 4 (начало второго элемента)
sort_outer:
    cpi R20, 40           ; Проверка конца массива (10 элементов * 4 = 40)
    brge sort_done        ; Выход если все элементы обработаны

    ; Загрузка текущего элемента arr[i]
    ldi YH, High(arr)
    ldi YL, Low(arr)
    add YL, R20           ; Указатель на arr[i]
    ld R19, Y+            ; Старший байт (знаковый)
    ld R18, Y+
    ld R17, Y+
    ld R16, Y+            ; Младший байт

    mov R21, R20          ; j = i

sort_inner:
    cpi R21, 0            ; Проверка начала массива (j == 0?)
    breq insert_done
```

```
    ; Загрузка предыдущего элемента arr[j-1]
    ldi YH, High(arr)
    ldi YL, Low(arr)
    mov R22, R21
    subi R22, 4           ; j-4
    add YL, R22
    ld R3, Y+             ; Старший байт arr[j-1] (знаковый)
    ld R2, Y+
    ld R1, Y+
    ld R0, Y+             ; Младший байт

    ; Знаковое сравнение arr[j-1] > arr[j]
    cp  R3, R19           ; Сравнение старших байтов (знаковый бит)
    cpc R2, R18           ; Следующие байты
    cpc R1, R17
    cpc R0, R16
    brlt no_shift         ; Если arr[j-1] < arr[j], пропустить сдвиг
    rjmp shift            ; Иначе сдвигать

shift:
    ; Сдвиг элементов вправо
    ldi YH, High(arr)
    ldi YL, Low(arr)
    add YL, R21
    st Y+, R3             ; Сохраняем arr[j-1] в arr[j]
    st Y+, R2
    st Y+, R1
    st Y+, R0
    subi R21, 4           ; j -= 4
    rjmp sort_inner

no_shift:
    rjmp insert_done

insert_done:
    ; Вставка элемента на позицию j
    ldi YH, High(arr)
    ldi YL, Low(arr)
    add YL, R21
    st Y+, R19            ; Сохраняем старший байт
    st Y+, R18
    st Y+, R17
    st Y+, R16
    subi R20, -4          ; i += 4
    rjmp sort_outer

sort_done:
    rjmp loop

loop:
    rjmp loop

; Исходные данные (знаковые числа)
.cseg
src: .dd 0x8E6BAE60, 0x7F123456, 0x11223344, 0x55667788, 0x99AABBCC,
0xAABBCCDD, 0x12345678, 0x55AA55AA, 0xFFFFFFFF, 0x00000000
```

Примерное количество циклов 1500

Было	Стало
0x8E6BAE60	0x8E6BAE60
0x7F123456	0x99AABBCC
0x11223344	0xAABBCCDD
0x55667788	0xFFFFFFFF
0x99AABBCC	0x00000000
0xAABBCCDD	0x11223344
0x12345678	0x12345678
0x55AA55AA	0x55667788
0xFFFFFFFF	0x55AA55AA
0x00000000	0x7F123456

Код для теста:

$log OCR0A

$log OCR0B

$log SREG

$startlog Asm_Lab_Array_log_output.stim

#1500

$stoplog

$memdump Asm_Lab_Array_mem_S_log_output.stim 0x0100 40 s

$memdump Asm_Lab_Array_mem_F_log_output.stim 0x001E 40 f

$break

Дамп памяти:

Asm_Lab_Array_mem_F_log_output:

:02001E00B1F738

:1000200000C044E0483264F5D1E0C0E0C40F39912B

:10003000299119910991542F5030C9F0D1E0C0E0B5

:06004000652F6450C60F9D

:00000001FF

Asm_Lab_Array_mem_S_log_output:

:1001000060AE6B8ECCBBAA99DDCCBBAAFFFFFFFF14

:10011000000000004433221178563412887766556 7

:08012000AA55AA555634127FBE

:00000001FF

HEX(ДО)	HEX(ПОСЛЕ)	DEC
0x8E6BAE60	0x60AE6B8E	-1,932,352,992
0x99AABBCC	0xCCBBAA99	-1,713,566,004
0xAABBCCDD	0xDDCCBBAA	-1,431,586,697
0xFFFFFFFF	0xFFFFFFFF	-1
0x00000000	0x00000000	0
0x11223344	0x44332211	287,454,020
0x12345678	0x87654321	305,419,896
0x55667788	0x88776655	1,431,586,696
0x55AA55AA	0xAA55AA55	1,439,578,538
0x7F123456	0x5634127F	2,131,869,782

Микроконтроллеры AVR хранят данные в памяти в формате little-endian, где младший байт числа располагается по младшему адресу.

Пример:

0x11223344 В Flash хранится как 44 33 22 11 (по возрастанию адресов).

Это и привело к тому, что на выходе мы получаем обратные значения по сравнению с начальным массивом.

Выводы: В ходе проведения данной лабораторной работы я научился писать несложные программы с использованием различных команд и условных переходов, а также работать с ОЗУ.