

Архитектура вычислительных систем.  
Домашнее задание 1

Фролов-Буканов Виктор Дмитриевич БПИ-228

Сентябрь 2023

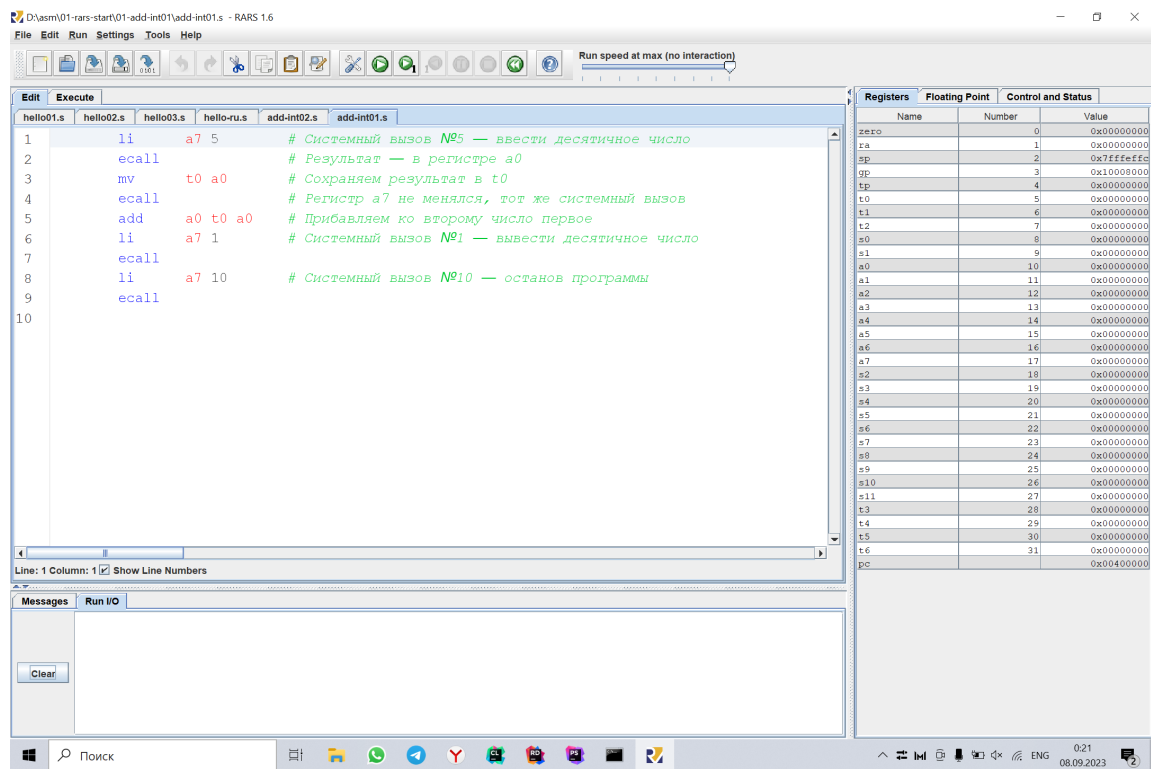


Figure 1: 1 скрин

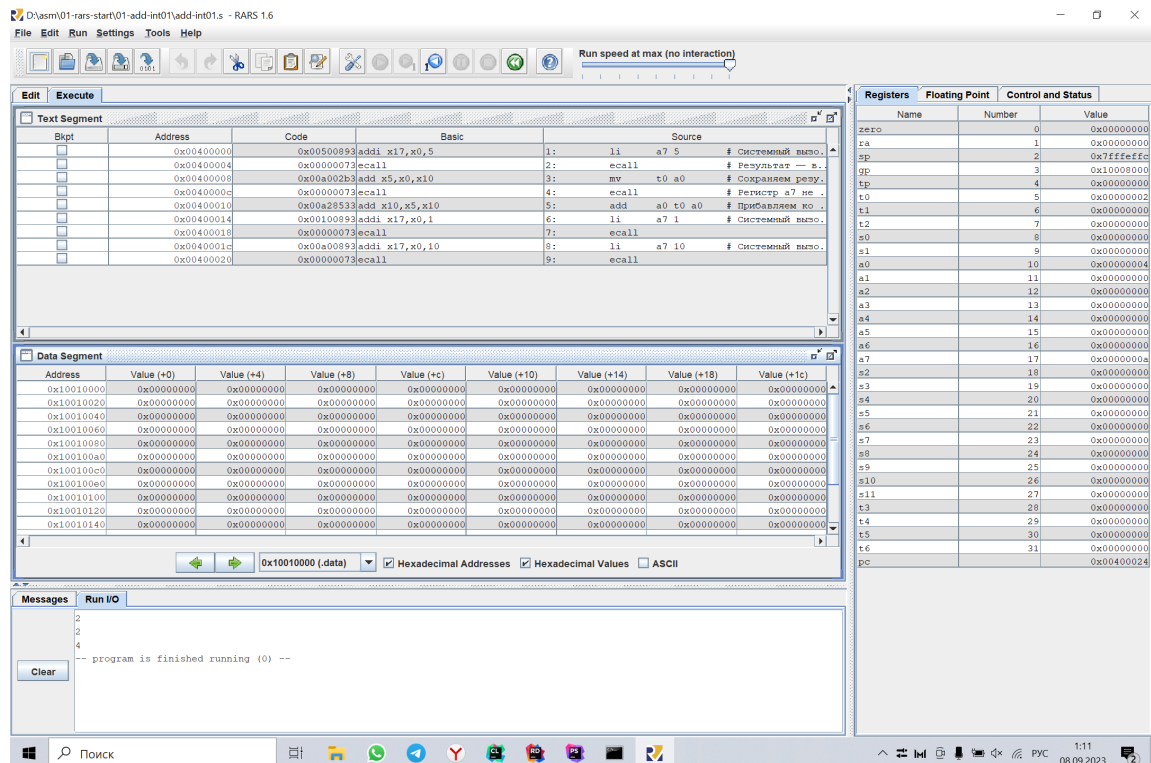


Figure 2: 2 скрин

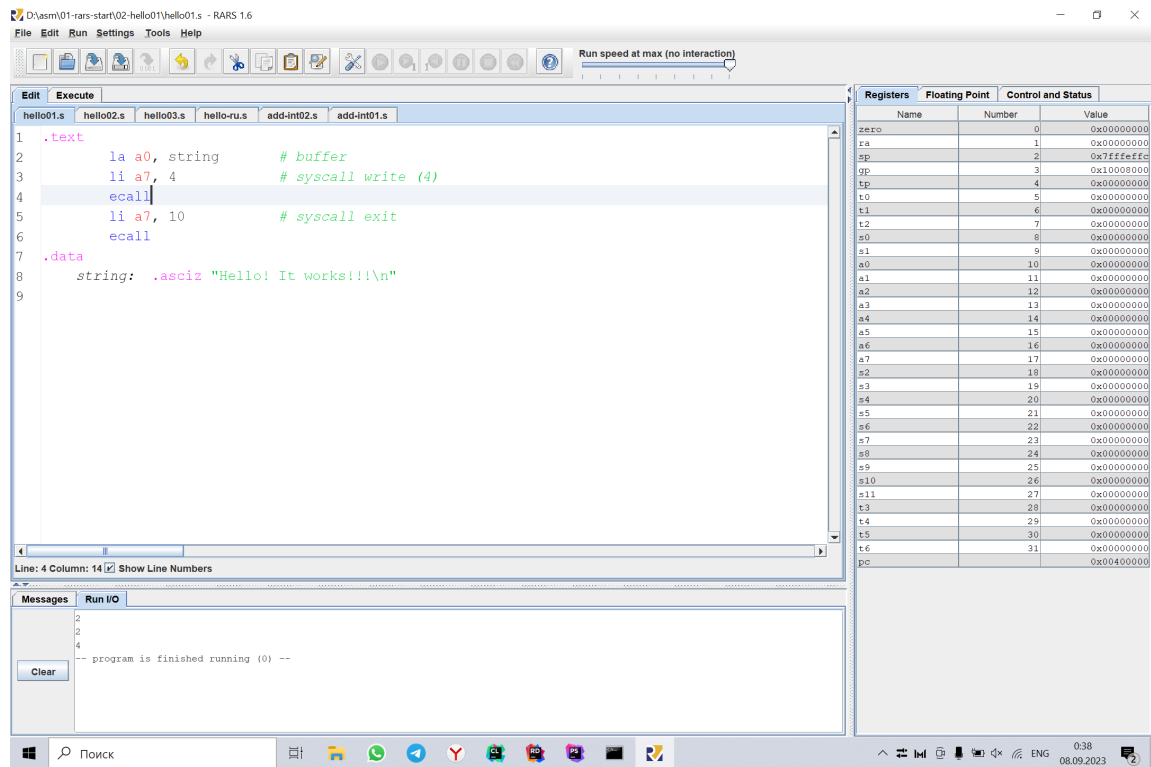


Figure 3: 3 скрин

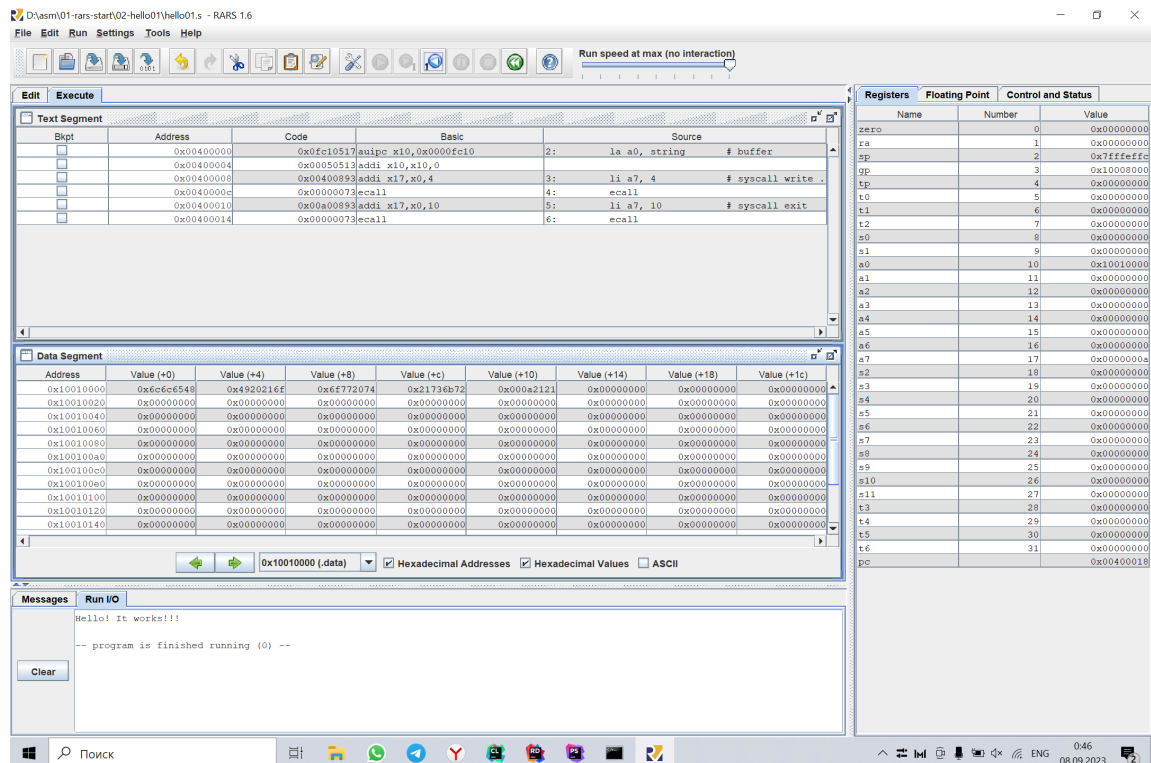


Figure 4: 4 скрин

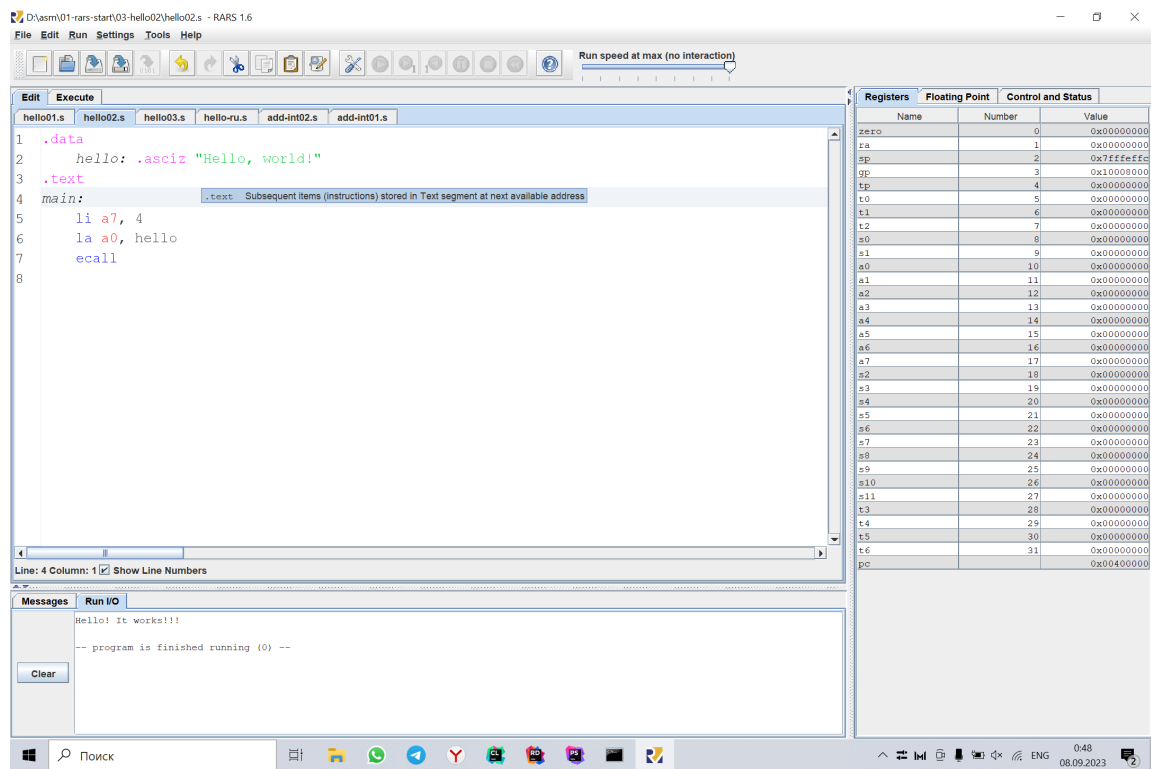


Figure 5: 5 скрин

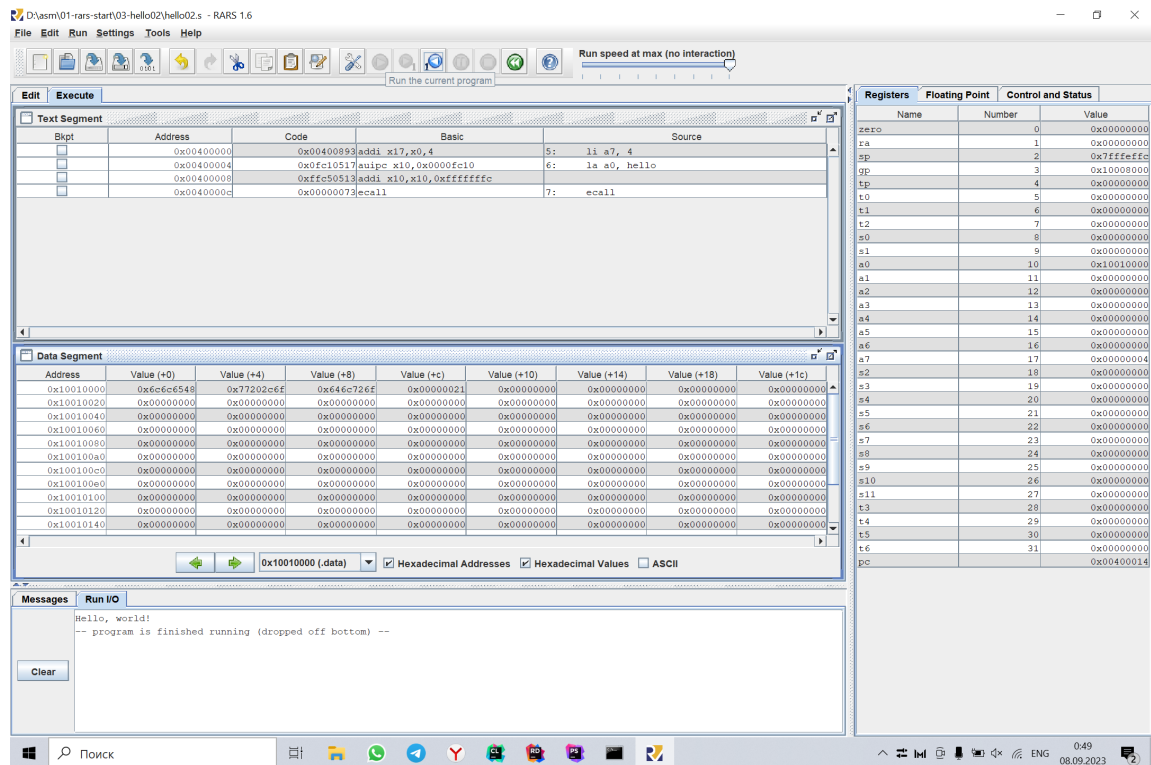


Figure 6: 6 скрин

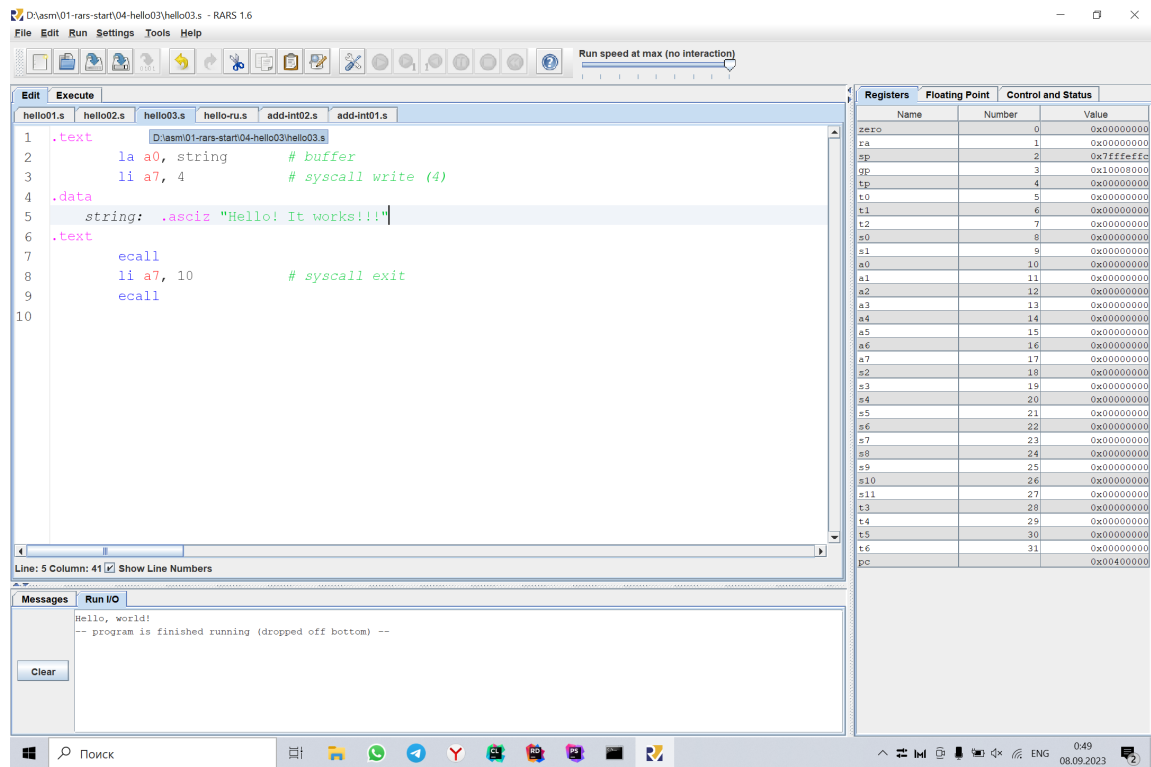


Figure 7: 7 скрин

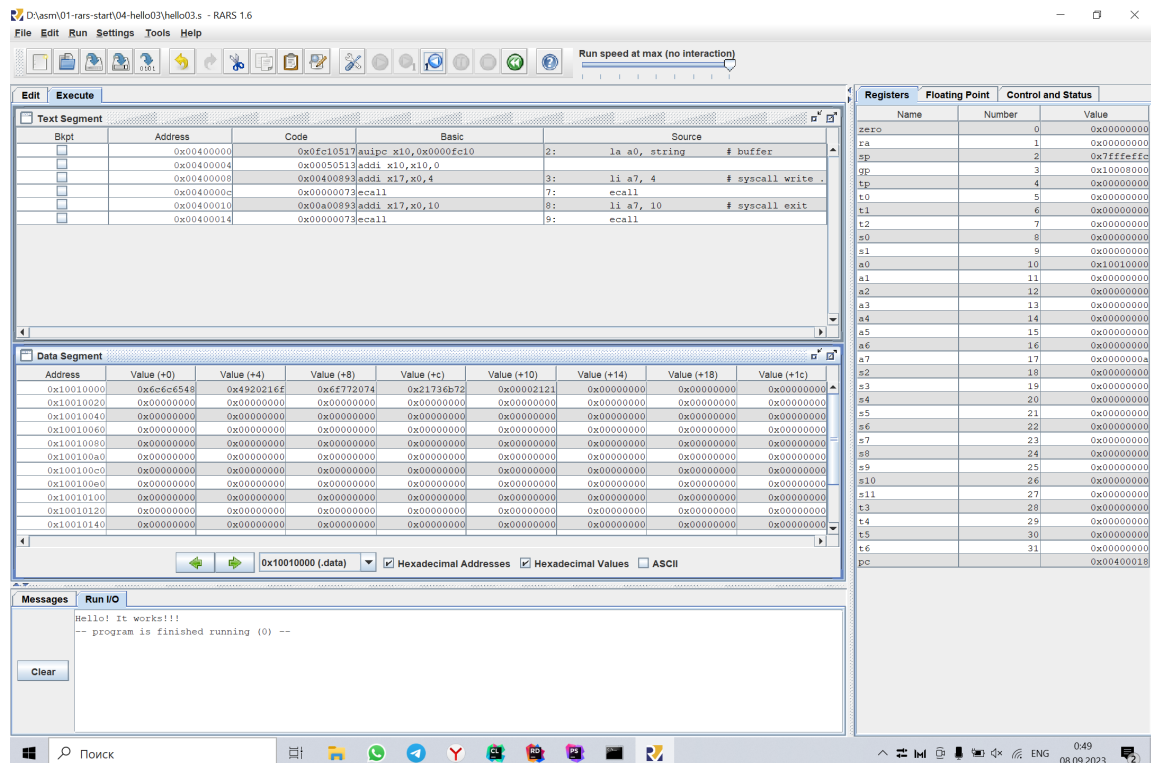
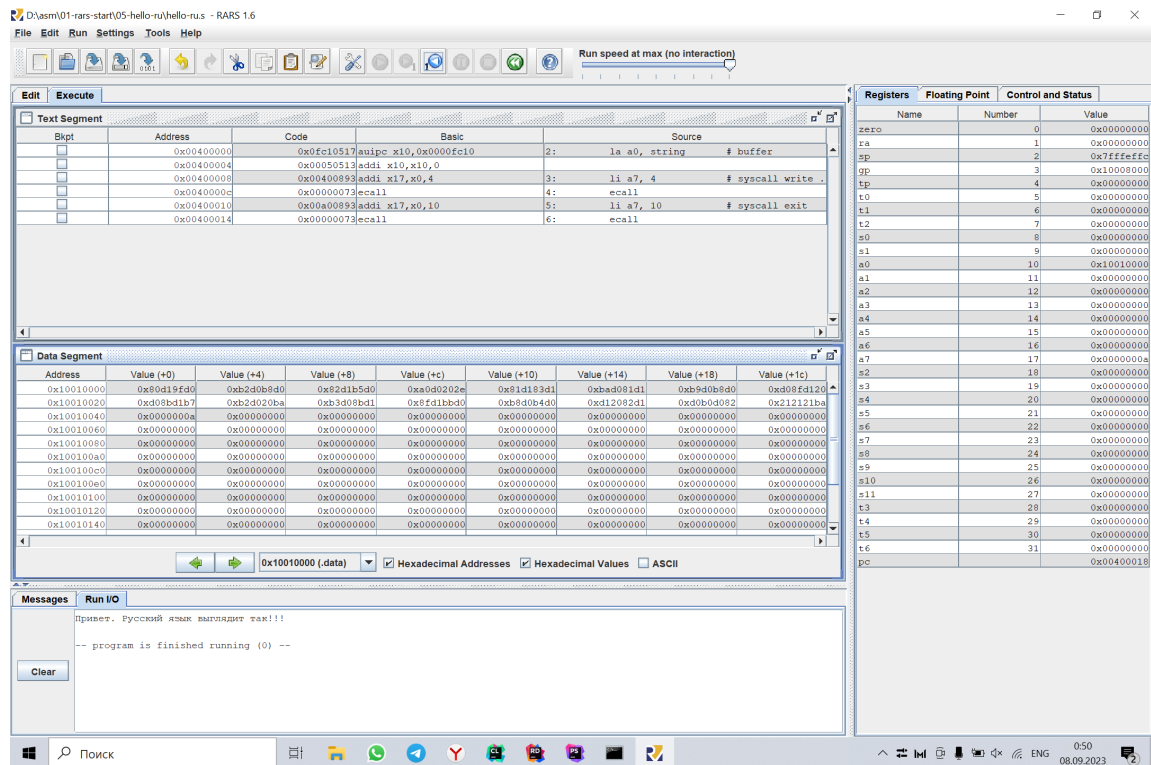
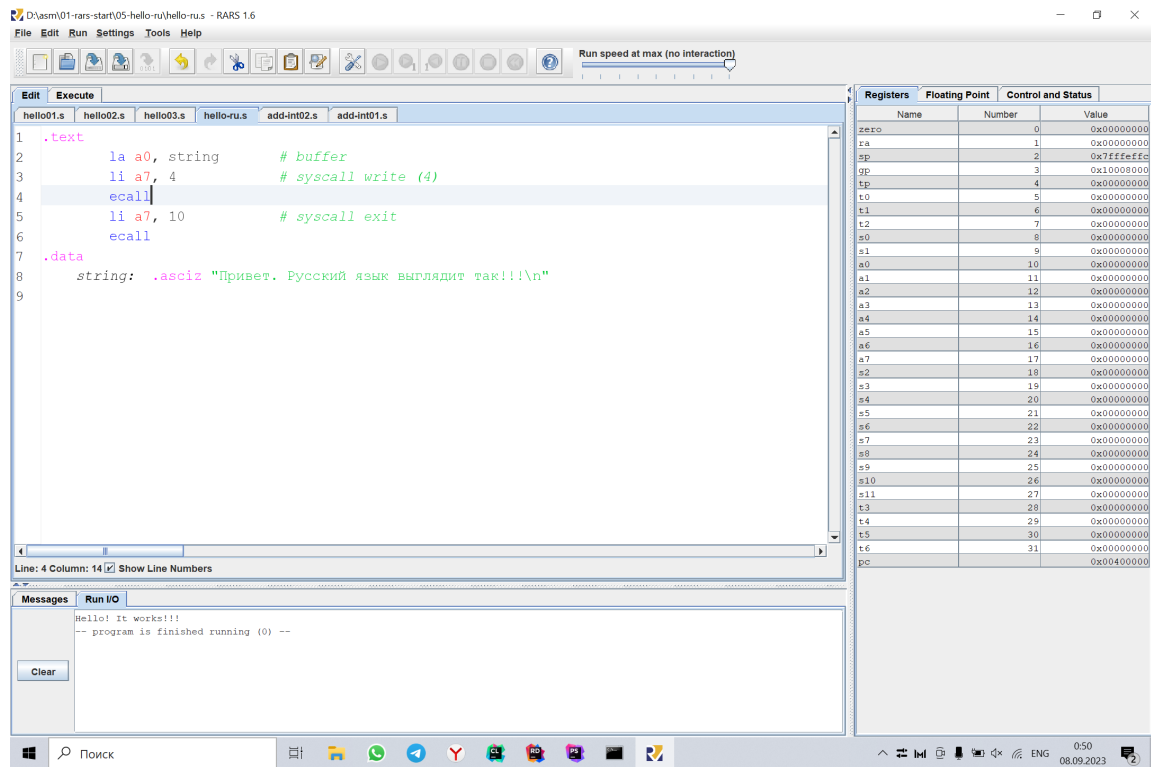


Figure 8: 8 скрин



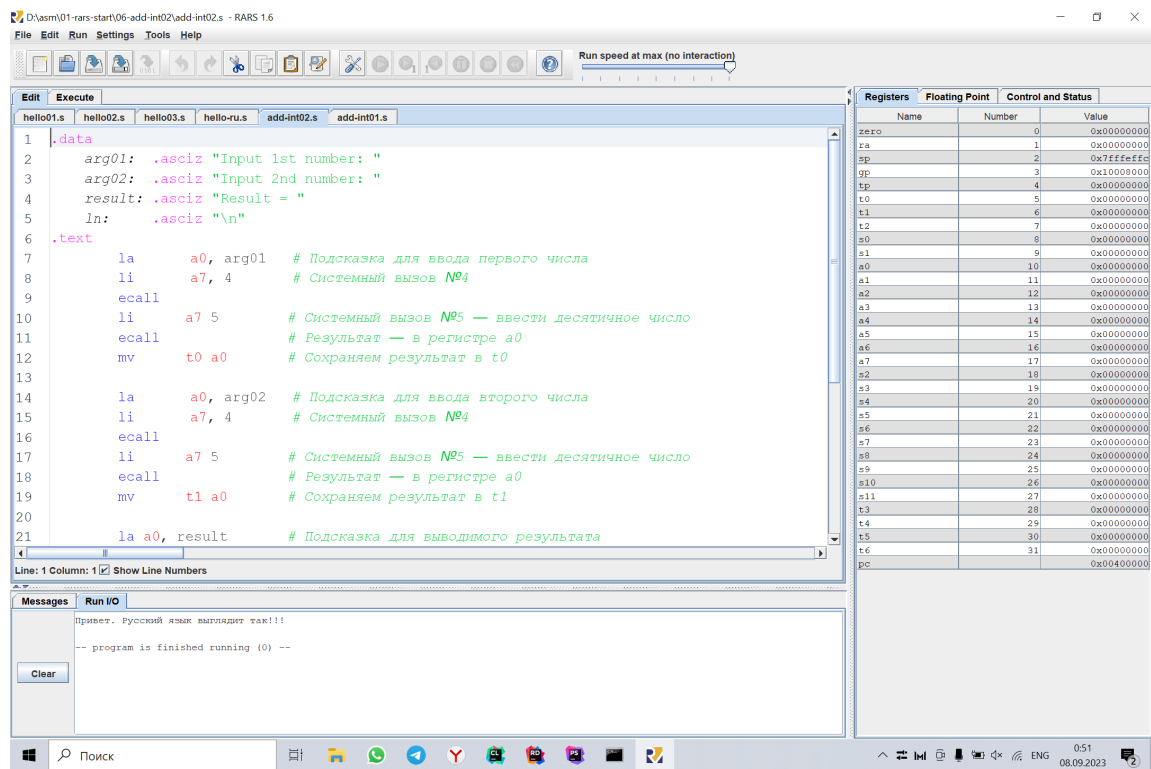


Figure 11: 11 скрин

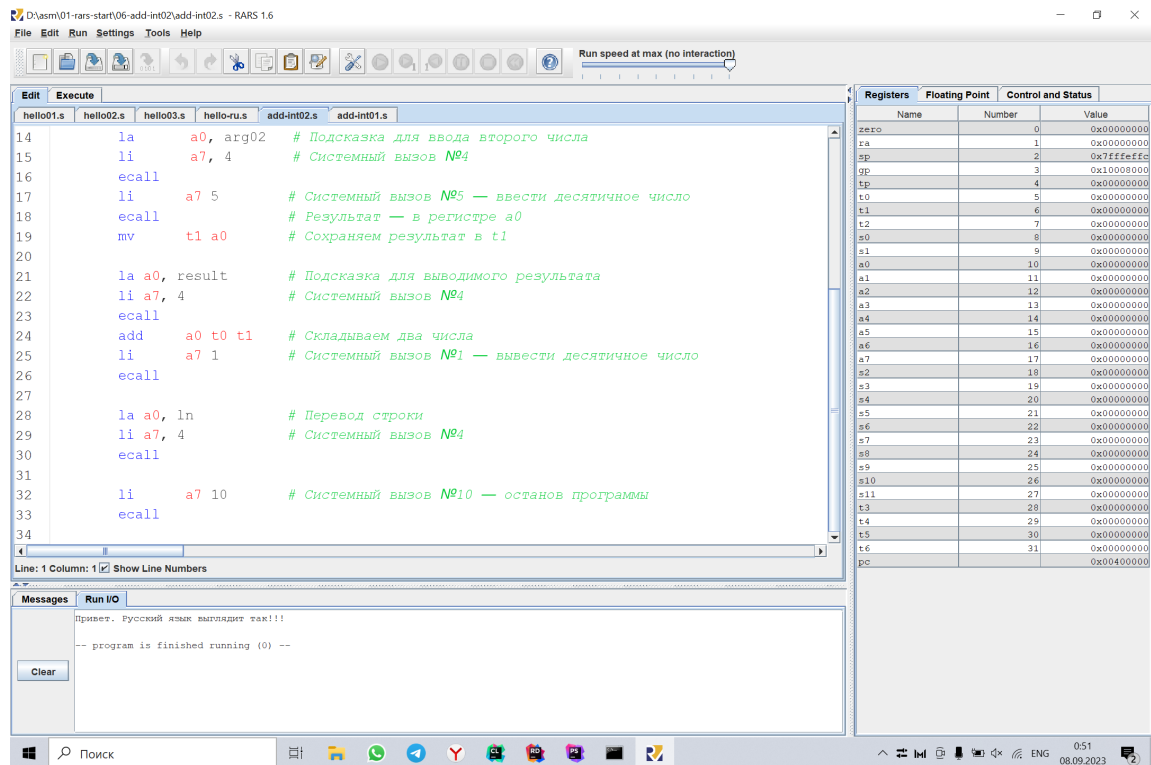


Figure 12: 12 скрин

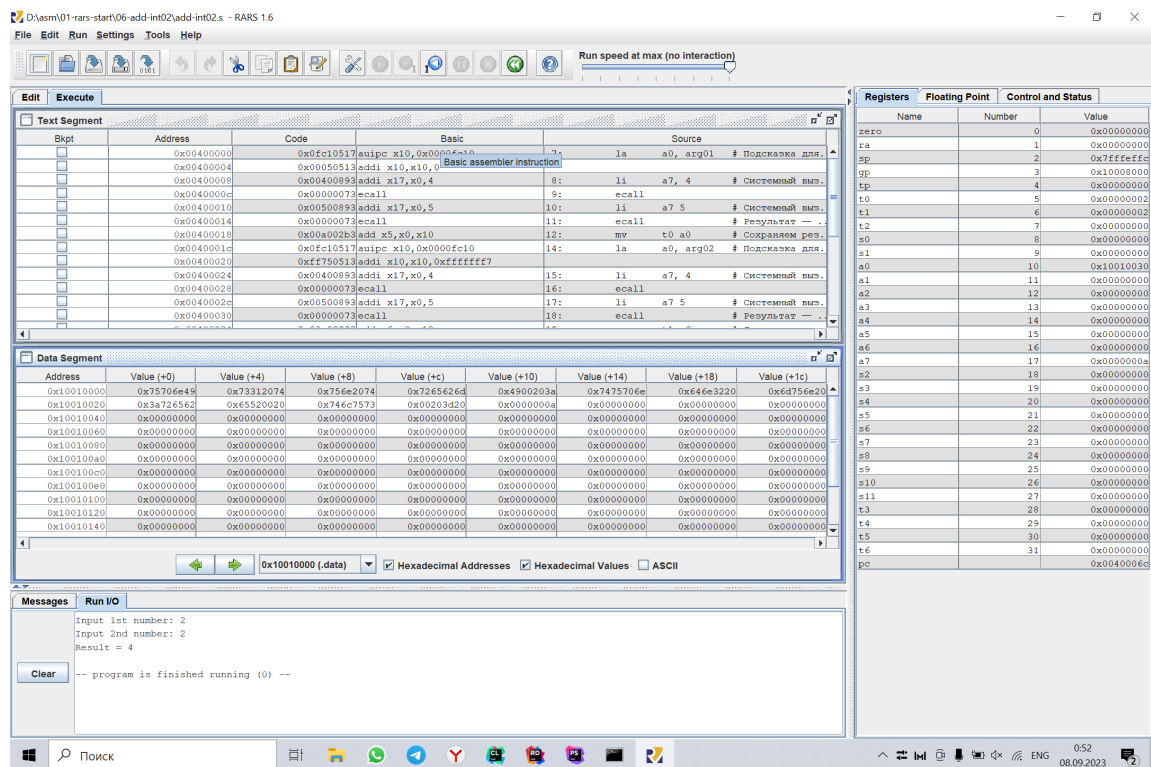


Figure 13: 13 скрин

Все 13 скриншотов выше демонстрируют работу эмулятора с программами, размещенными в LMS в теме первого семинара



# 1 Какие команды являются псевдокомандами?

В качестве примера возьмём самую первую программу (*add – int01.s*) на 2 странице (1 и 2 скрины).

*Псевдокомандами* являются команды, не имеющие непосредственного аналога в машинном коде процессора с архитектурой *RISC–V*. В результате компиляции псевдокоманды переводятся в базовые, которые позже переводятся в машинный код. Псевдокоманда теоретически может состоять из нескольких базовых. Это как бы удобное сокращение, синтаксический сахар, предоставляющий программисту более удобное использование языка ассемблера. В данном случае, это команды **li a7 5, mv t0 a0, li a7 1 и li a7 10**. На 2 скриншоте можно видеть соответствие, как код из столбика source переводится в код в столбике basic (в базовые инструкции), которые после переводятся в столбик code. Сравнивая столбики basic и source, можно видеть, например, что команда **li a7 5** заменяется на **addi x17, x0, 5**. При этом команда **addi t1, t2, -100** - является базовой командой (о чём можно удостовериться в help - скриншот 14), из чего следует вывод, что **li** - это новая команда, но не базовая (поэтому и псевдокоманда), которая внутренне реализована через базовую команду **addi**. Аналогично можно заметить про остальные псевдокоманды **li**, а также про псевдокоманду **mv**, которой нет в списке базовых команд, но которая реализована через базовую команду **add**. Таким образом, мы пришли к выводу, что команды **li a7 5, mv t0 a0, li a7 1 и li a7 10** являются псевдокомандами (о чём дополнительно можно также удостовериться в help - скрин 15)

| Basic Instructions  | Extended (pseudo) Instructions  | Directives | Syscalls | Exceptions | Macros |
|---------------------|---|------------|----------|------------|--------|
| add t1,t2,t3        | Addition: set t1 to (t2 plus t3)  |            |          |            |        |
| addi t1,t2,-100     | Addition immediate: set t1 to (t2 plus signed 12-bit immediate)                 |            |          |            |        |
| and t1,t2,t3        | Bitwise AND : Set t1 to bitwise AND of t2 and t3                                |            |          |            |        |
| andi t1,t2,-100     | Bitwise AND immediate : Set t1 to bitwise AND of t2 and sign-extended 12-bit im |            |          |            |        |
| auipc t1,100000     | Add upper immediate to pc: set t1 to (pc plus an upper 20-bit immediate)        |            |          |            |        |
| beq t1,t2,label     | Branch if equal : Branch to statement at label's address if t1 and t2 are equal |            |          |            |        |
| bge t1,t2,label     | Branch if greater than or equal: Branch to statement at label's address if t1 i |            |          |            |        |
| bgeu t1,t2,label    | Branch if greater than or equal to (unsigned): Branch to statement at label's a |            |          |            |        |
| blt t1,t2,label     | Branch if less than: Branch to statement at label's address if t1 is less than  |            |          |            |        |
| bltu t1,t2,label    | Branch if less than (unsigned): Branch to statement at label's address if t1 is |            |          |            |        |
| bne t1,t2,label     | Branch if not equal : Branch to statement at label's address if t1 and t2 are n |            |          |            |        |
| csrrc t0, fcsr, t1  | Atomic Read/Clear CSR: read from the CSR into t0 and clear bits of the CSR acco |            |          |            |        |
| csrrci t0, fcsr, 10 | Atomic Read/Clear CSR Immediate: read from the CSR into t0 and clear bits of th |            |          |            |        |
| csrrs t0, fcsr, t1  | Atomic Read/Set CSR: read from the CSR into t0 and logical or t1 into the CSR   |            |          |            |        |
| csrrsi t0, fcsr, 10 | Atomic Read/Set CSR Immediate: read from the CSR into t0 and logical or a const |            |          |            |        |
| csrrw t0, fcsr, t1  | Atomic Read/Write CSR: read from the CSR into t0 and write t1 into the CSR      |            |          |            |        |
| csrrwi t0, fcsr, 10 | Atomic Read/Write CSR Immediate: read from the CSR into t0 and write a constant |            |          |            |        |
| div t1,t2,t3        | Division: set t1 to the result of t2/t3   |            |          |            |        |
| divu t1,t2,t3       | Division: set t1 to the result of t2/t3 using unsigned division                 |            |          |            |        |

Figure 14: 14 скрин

| Basic Instructions   | Extended (pseudo) Instructions   | Directives | Syscalls | Exceptions | Macros |
|----------------------|--|------------|----------|------------|--------|
| lh t1,-100           | Load Halfword : Set t1 to sign-extended 16-bit value from effective memory halfw |            |          |            |        |
| lh t1,10000000       | Load Halfword : Set t1 to sign-extended 16-bit value from effective memory halfw |            |          |            |        |
| lh t1,label          | Load Halfword : Set t1 to sign-extended 16-bit value from effective memory halfw |            |          |            |        |
| lhu t1,(t2)          | Load Halfword Unsigned : Set t1 to zero-extended 16-bit value from effective me  |            |          |            |        |
| lhu t1,-100          | Load Halfword Unsigned : Set t1 to zero-extended 16-bit value from effective me  |            |          |            |        |
| lhu t1,10000000      | Load Halfword Unsigned : Set t1 to zero-extended 16-bit value from effective me  |            |          |            |        |
| lhu t1,label         | Load Halfword Unsigned : Set t1 to zero-extended 16-bit value from effective mem |            |          |            |        |
| li t1,-100           | Load Immediate : Set t1 to 12-bit immediate (sign-extended)                      |            |          |            |        |
| li t1,10000000       | Load Immediate : Set t1 to 32-bit immediate                                      |            |          |            |        |
| lui t1,%hi(label)    | Load Upper Address : Set t1 to upper 20-bit label's address                      |            |          |            |        |
| lw t1,%lo(label)(t2) | Load from Address  |            |          |            |        |
| lw t1,(t2)           | Load Word : Set t1 to contents of effective memory word address                  |            |          |            |        |
| lw t1,-100           | Load Word : Set t1 to contents of effective memory word address                  |            |          |            |        |
| lw t1,10000000       | Load Word : Set t1 to contents of effective memory word address                  |            |          |            |        |
| lw t1,label          | Load Word : Set t1 to contents of memory word at label's address                 |            |          |            |        |
| mv t1,t2             | MoVe : Set t1 to contents of t2  |            |          |            |        |
| neg t1,t2            | NEGate : Set t1 to negation of t2  |            |          |            |        |
| nop                  | NO OPERATION   |            |          |            |        |
| not t1,t2            | Bitwise NOT (bit inversion)  |            |          |            |        |

Figure 15: 15 скрин

## 2 Описать типы форматов команд для одной из представленных программ

В качестве примера вновь возьмём самую первую программу (*add – int01.s*) на 2 странице (1 и 2 скрины).

| Команда      | Тип      |
|--------------|----------|
| li a7 5      | Store    |
| mv t0 a0     | Register |
| add a0 t0 a0 | Register |
| li a7 1      | Store    |
| li a7 10     | Store    |

Стоит отметить, что команда **li t1,-100** реализована через **addi t1,t2,-100**, которая имеет тип регистр-регистр-непосредственное значение (store), поэтому команду **li** можно отнести к типу store. Аналогично **mv t1,t2** реализована через **add t1,t2,t3**, имеющую тип register, поэтому команду **mv** отнесём к типу register

## 3 Какие системные вызовы используются в изученных программах.

1. li a7 1 - системный вызов для вывода в консоль integer, хранимого в регистре a0
2. li a7 5 - системный вызов для считывания integer из консоли и помещения его в регистр a0

3. `li a7 10` - системный вызов для завершения работы программы
4. `li a7 4` - системный вызов для вывода в консоль `string`, хранимого в регистре `a0`