Правительство Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»
(НИУ ВШЭ)

Московский институт электроники и математики им. А.Н. Тихонова

ОТЧЕТ
О ПРАКТИЧЕСКОЙ РАБОТЕ № 2
по дисциплине «Системное программирование»
ОСНОВЫ АССЕМБЛЕРА

Студент гр. БИБ 211
_____ В.Ф. Санников
«___» _____ 2023 г.

Руководитель
Старший преподаватель
_____ В.И. Морозов
«___» _____ 2023 г.

Москва

СОДЕРЖАНИЕ

1 Задание на практическую работу

Целью работы является изучение языка описания процессорных инструкций “Ассемблер” и реализация предложенного задания на этом языке программирования. Мой порядковый номер в группе БИБ 211 это 19, соответственно мне необходимо выполнить задание из варианта номер 3:
Дан массив из 10 слов. Найти сумму остатков от деления каждого из них на 3. Результат поместить в отдельный элемент данных.

Для выполнения работы необходимо:
1) Повторно ознакомиться с теоретическим материалом из презентаций с семинаров 2 и 3.
Внимание! Все настройки, действия и манипуляции, проводимые для выполнения заданий из пункта 2, необходимо фиксировать в отчёте в виде снимков экрана или листингов вводимых команд.
2) С помощью языка программирования Ассемблер, используя любой синтаксис (Intel или AT&T) на свой выбор, необходимо написать программу, решающую задачу из варианта. При этом:

а) Решением задачи является запись в специально объявленную для этого переменную верного результата при любых допустимых входных данных.

б) Программа должна корректно завершаться, не вызывая аварийный останов.

в) Объявлять входные данные необходимо самостоятельно в секции данных. Например, если в задаче указано «дан массив из 10 байт», в секции данных необходимо самостоятельно объявить такой массив, заполнить его байтами на своё усмотрение и использовать в качестве входного. В ходе защиты может возникнуть необходимость изменения таких данных.

г) Для выполнения повторяющихся однотипных действий необходимо использовать соответствующие средства языка. Копирование и вставка одного и того же кода нужное количество раз с незначительными изменениями без необходимости является ошибкой.

д) Программа должна выдавать корректный ответ для всех видов входных данных, описанных в задании. Например, если в задании написано «дан массив из 10 байт», то входными данными может быть массив из любых 10 значений от 0 до 255. Учитывайте возможные переполнения!

е) При выполнении (в т.ч. дополнительных заданий) нельзя вызывать готовые ассемблерные подпрограммы из каких-либо библиотек и модулей.

3) Оформить отчёт в соответствии с требованиями по оформлению отчёта из SmartLMS. Не забыть прикрепить к отчёту исходный код всей программы в виде приложения. В отчёте необходимо отразить, как минимум:

а) Результаты работы программы в виде верных значений результирующих переменных.

б) Процесс сборки и запуска программы в отладчике.

в) Словесное описание алгоритма решения поставленной задачи с листингами соответствующего кода.

г) Изменения ключевых значений (в регистрах и переменных) в ходе работы программы.

4) Защитить работу на занятии. Во время защиты необходимо:

а) Продемонстрировать работу программы в отладчике (SASM, gdb или любом другом на ваш выбор).

б) При необходимости изменить входные данные и продемонстрировать, что программа отрабатывает верно на новых данных.

в) Ответить на вопросы по логике работы программы (почему и зачем была написана та или иная строка, каково её назначение и вклад в решение данной задачи, что будет, если её убрать).

г) Ответить на вопросы по теоретической части. Внимание! Вопросы по теоретической части будут браться из материалов семинаров 2 и 3.

5) Дополнительное задание 1: вывести результирующий элемент данных в консоль (в окно «вывод» в IDE), используя только команды «чистого» ассемблера (не вызывая подпрограммы, написанные не самостоятельно, наподобие printf). Даёт два дополнительных балла.

6) Дополнительное задание 2: выполнить все поставленные задачи,
написав две программы с использованием обоих синтаксисов (Intel и AT&T).
Даёт один дополнительный балл.

2 Ход работы

В ходе работы мне было необходимо написать программный код на языке Ассемблер. Я использовал синтаксис Intel, при этом вся работа проводилась в программе IDE (среда разработки), которая называется SASM (Simple Assembler). Кроме того, я выполнял данную лабораторную работу, используя операционную систему Linux Ubuntu. При использовании данной операционной системы и данной среды разработки, можно пропустить шаги установки самого ассемблера, его компилятор, а также отладчик gdb. Все эти “утилиты” входят в пакет установки SASM и работают без каких-либо проблем после запуска кода в среде разработки.
Далее необходимо разобраться в том, какие бывают виды ассемблера и чем они отличаются. Ассемблер - это низкоуровневый язык программирования, который предоставляет непосредственный доступ к аппаратным возможностям компьютера. Существует несколько различных видов ассемблера, каждый из которых имеет свои особенности и специфику. Вот краткий обзор нескольких из них:
1. NASM (Netwide Assembler):
- Один из наиболее популярных и широко используемых ассемблеров для процессоров семейства x86 и x86-64 (также известных как IA-32 и AMD64 / x86-64).
- NASM является кросс-платформенным, что позволяет писать ассемблерный код для различных операционных систем, включая Linux, Windows и macOS.
- Используется в различных проектах, включая ядро Linux.
2. GAS (GNU Assembler):
- Он является частью проекта GNU и поставляется вместе с компилятором GCC.
- GAS является стандартным ассемблером для проектов, разрабатываемых под управлением систем GNU/Linux.
- Оптимизирован для работы в составе инфраструктуры GNU.
3. FASM (Flat Assembler):
- Разработанный для работы под различными операционными системами, такими как Windows, Linux и другими.
- Отличается компактностью и скоростью сборки ассемблерных программ.

4. YASM:

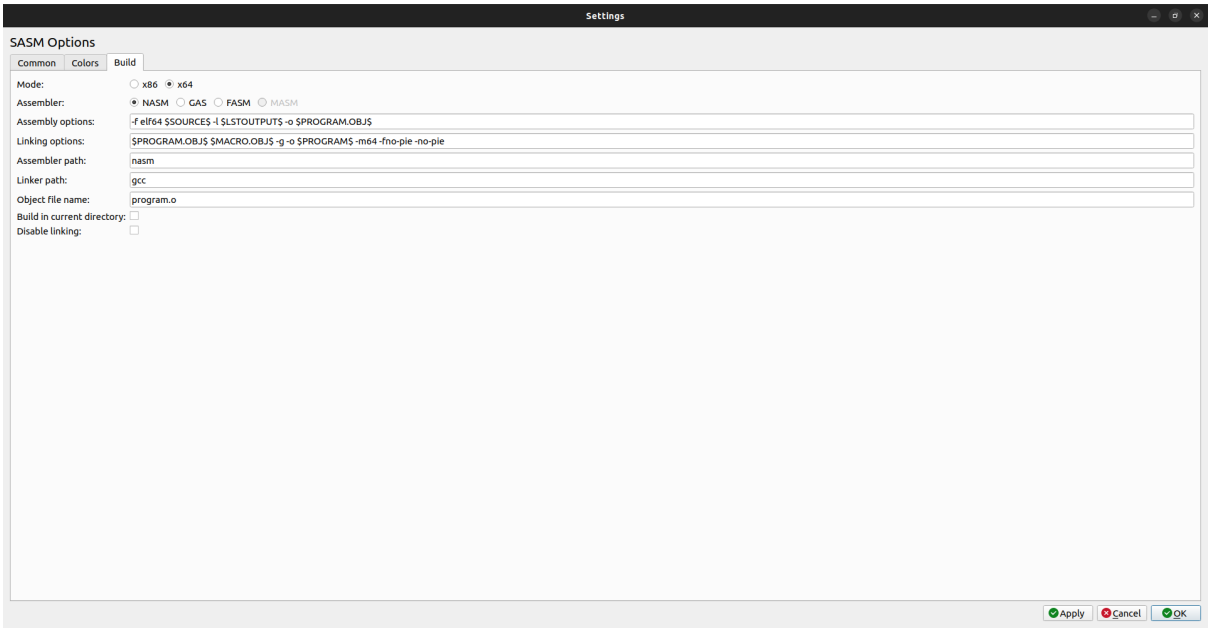
- Это NASM-совместимый ассемблер, который предлагает некоторые дополнительные оптимизации и особенности, особенно для платформы x86-64.

Теперь о разнице между архитектурами x64 и x86:

- x86 (или IA-32) – 32-битная архитектура. В ней доступно 32-битное пространство адресов и 32-битные регистры общего назначения.
- x64 (или AMD64/Intel 64) – 64-битная архитектура. Она предназначена для процессоров с поддержкой 64-битного режима. В ней доступно 64-битное пространство адресов и регистры общего назначения также увеличены до 64 бит.

Различные сборщики ассемблера могут быть связаны с конкретными архитектурами (например, NASM работает с x86 и x86-64), однако многие из них поддерживают различные архитектуры в зависимости от версии и конфигурации.

Для выполнения поставленной задачи я буду использовать assembler nasm 64 bit, создавая программный код в среде разработки SASM. В данной среде разработки есть отладчик gdb, а также окошко input и output. При выполнении программы мы взаимодействуем с ними как со входом и выходом программы. Настройки среды разработки, которые я использовал для выполнения работы, выглядят следующим образом:



Скриншот 1 – Настройки среды разработки SASM.

2.1 Программная реализация поставленной задачи

Начнём разбирать написанный код.

```
section .data
   array dq 1, 1, 1, 1, 0, 1, 0, 1, 1, 1
   result dq 0
```

Листинг 1 – Объявление данных, над которыми мы будем оперировать.

Этот код (Листинг 1) инициализирует массив из 10 элементов (каждый из которых занимает по 8 байт) и отдельную переменную для хранения результата (также занимающую 8 байт). В данном случае, массив содержит 10 элементов, как и требуется в задании.

```
section .text
global main
```

Листинг 2 – Определение секции `.text` и указание на то, что метка `main` доступна для глобального использования.

В этих строках ассемблерного кода (Листинг 2) происходит определение секции `.text` и указание на то, что метка `main` доступна для глобального использования.

1. `section .text` - Эта строка определяет секцию кода (обычно исполняемый код программы). В этой секции обычно размещаются инструкции, которые представляют собой фактический исполняемый код программы.
2. `global main` - Эта строка указывает, что метка `main` будет доступна для глобального использования. В контексте ассемблера это означает, что метка `main` может быть использована другими частями программы или внешними программами.

Такие указания помогают компилятору и операционной системе правильно организовать исполняемый код и обеспечить корректную работу программы при запуске.

```
print_uint:
  xor rcx, rcx
  mov r8, 10
```

```
.loop:
  xor rdx, rdx
  div r8
  add dl, 0x30
  dec rsp
  mov [rsp], dl
  inc rcx;
  test rax, rax
  jnz .loop

.print_chars_on_stack:
  xor rax, rax
  mov rsi, rsp;
  mov rdx, rcx
  push rcx
  mov rax, 1
  mov rdi, 1
  syscall
  pop rcx
  add rsp, rcx; when printed we can free the stack

ret
```

Листинг 3 – Реализация функции вывода результата на экран в процессе выполнения программы.

Данный модуль представляет функцию `print_uint`, которая используется для печати целого числа без знака на стандартный вывод. Взглянем на разбор по строкам:

1. `xor rcx, rcx` - Обнуление регистра `rcx`, который будет использоваться для подсчета количества цифр в числе.
2. `mov r8, 10` - Загрузка значения 10 в регистр `r8`, так как мы будем делить на 10 для извлечения цифр.
3. `.loop:` - Метка для начала цикла.
4. `xor rdx, rdx` - Очистка регистра `rdx`, который будет использоваться для хранения остатка от деления.

5. `div r8` - Деление содержимого регистров `rdx:rax` на значение в `r8`. Результат деления сохраняется в `rax`, а остаток – в `rdx`.
6. `add dl, 0x30` - Преобразование остатка в символ ASCII (0-9), добавление 48 (0x30) к остатку, чтобы получить соответствующий код ASCII.
7. `dec rsp` - Уменьшение указателя стека на 1 байт для создания места под хранение следующей цифры.
8. `mov [rsp], dl` - Запись преобразованного остатка (цифры) в стек.
9. `inc rcx` - Увеличение счетчика цифр.
10. `test rax, rax` - Проверка остатка от деления. Если остаток не равен нулю, переход к метке `.loop`.
11. `.print_chars_on_stack:` - Метка для вывода цифр со стека.
12. `xor rax, rax` - Обнуление `rax` для подготовки системного вызова.
13. `mov rsi, rsp` - Загрузка указателя стека в регистр `rsi`.
14. `mov rdx, rcx` - Загрузка количества напечатанных цифр в `rdx`.
15. `push rcx` - Сохранение значения `rcx` на стеке.
16. `mov rax, 1` - Загрузка номера системного вызова (sys_write).
17. `mov rdi, 1` - Загрузка номера файлового дескриптора stdout.
18. `syscall` - Выполнение системного вызова sys_write для вывода цифр на экран.
19. `pop rcx` - Восстановление значения `rcx` со стека.
20. `add rsp, rcx` - Освобождение памяти стека после печати цифр.
21. `ret` - Возврат из функции.

Функция `print_uint` (Листинг 3) предназначена для вывода беззнакового целого числа на экран. Она принимает значение беззнакового целого числа в регистре rax и производит его вывод путем преобразования числа в строку и последовательного вывода символов этой строки.

```
main:
    mov rbp, rsp; for correct debugging
    mov rcx, 10
    xor rsi, rsi
    xor rbx, rbx
```

Листинг 4 – Функция “main”.

Функция (Листинг 4) начинается с установки значения указателя стека (rsp) в регистр базы стека (rbp) для корректного отладочного процесса. Затем устанавливается количество итераций цикла в регистре rcx (в данном случае 10). Затем индекс текущего элемента массива (rsi) и сумма остатков (rbx) обнуляются с помощью операции XOR. Эти операции готовят переменные для использования в цикле, который будет проходить через массив и вычислять сумму остатков от деления каждого элемента массива на 3.

```
calculate_remainders:
    mov rax, [array + rsi*8]
    xor rdx, rdx
    mov rdi, 3
    div rdi
    add rbx, rdx
    add rsi, 1
    cmp rsi, rcx
    jl calculate_remainders
    mov [result], rbx
```

Листинг 5 – Функция “calculate_remainders”.

Функция `calculate_remainders` (Листинг 5) выполняет вычисление суммы остатков от деления элементов массива на 3. Она использует цикл, чтобы пройти через каждый элемент массива, выполнить деление на 3 и сохранить остаток в общей сумме остатков (хранящейся в регистре rbx).

```
    mov rax, [result]
    call print_uint
    mov rax, 60
    syscall
```

Листинг 6 – Вывод результатов и завершение программы.

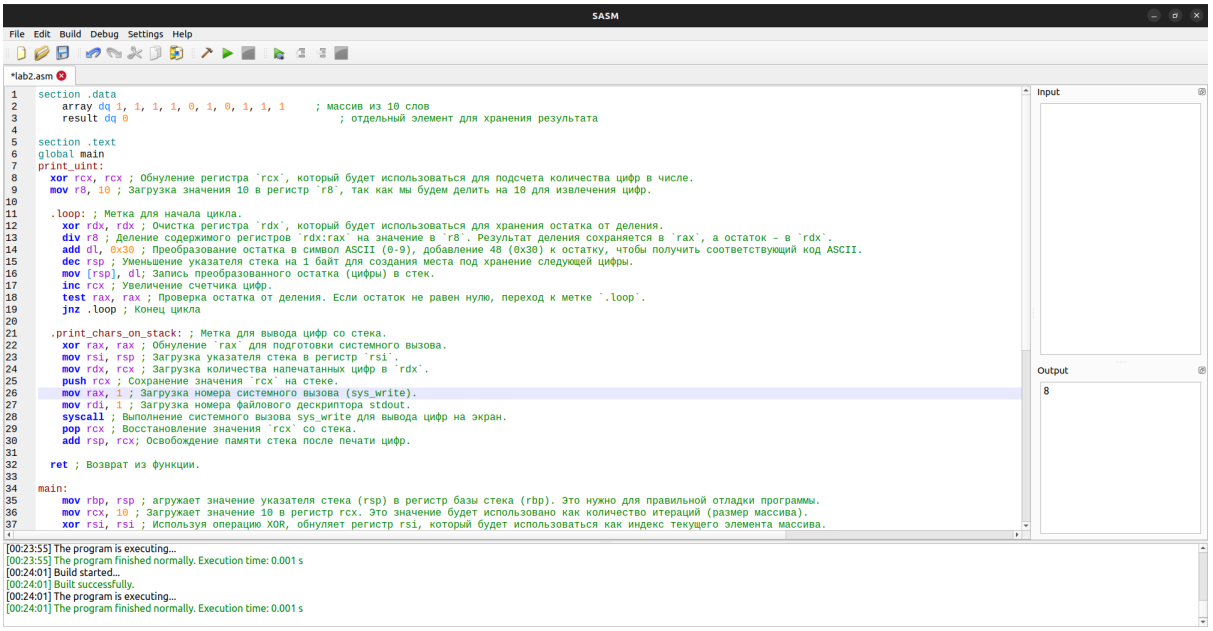
В этом коде (Листинг 6) происходит вывод результата (суммы остатков) на экран.
1. Загружается значение суммы остатков из памяти (хранящейся в элементе данных с именем "result") в регистр rax.

2. Вызывается функция "print_uint", которая предназначена для вывода беззнакового целого числа (значения из регистра rax) на экран или другой поток вывода.
3. Загружается значение 60 в регистр rax. Это значение будет использовано для системного вызова.
4. Выполняется системный вызов по номеру, который хранится в регистре rax. В зависимости от операционной системы и архитектуры, этот системный вызов будет завершать программу или выполнять другую операцию. В данном коде, предполагается, что номер 60 соответствует завершению программы.

2.2 Проверка работоспособности написанной программы

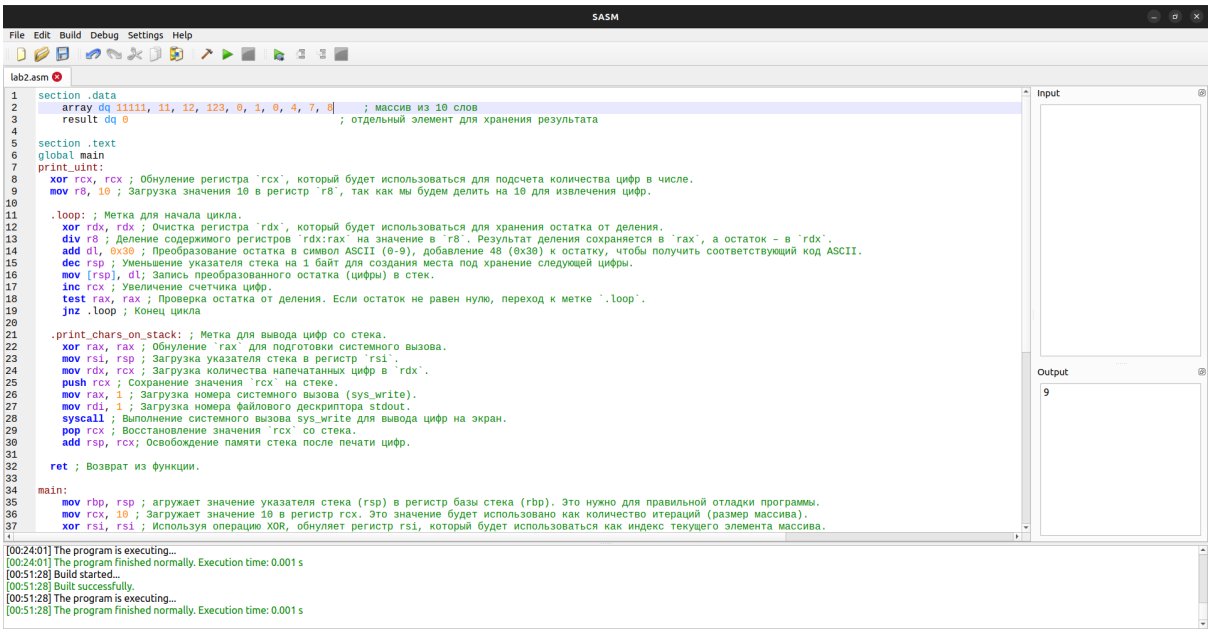
Для работы с данным программным кодом необходимо подать в программу массив из 10 чисел, чтобы на выходе получить сумму остатков от деления на 3 каждого из элементов массива. Чтобы указать, какие числа мы подаём в программу, необходимо изменить код из Листинга 1, а именно вручную изменить значения элементов массива array. При этом условие задания не накладывает особенных ограничений на размер числа в ячейке массива. Путём простого эксперимента, мне удалось выяснить, что максимальное число, которое можно подать в программу, это 18446744073709551615, то есть (2^64)-1. Если подать в программу число 18446744073709551616, то среда разработки выведет ошибку о переполнении. То есть во входной массив можно подавать числа от –((2^64)-1) до (2^64)-1, или же от -18446744073709551615 до 18446744073709551615. Это более чем удовлетворяет условию задания. Также стоит отметить, что программа не может работать с текстовыми строками, исключительно с числами.

Далее привожу для рассмотрения скриншоты, в которых показываю работу программы.

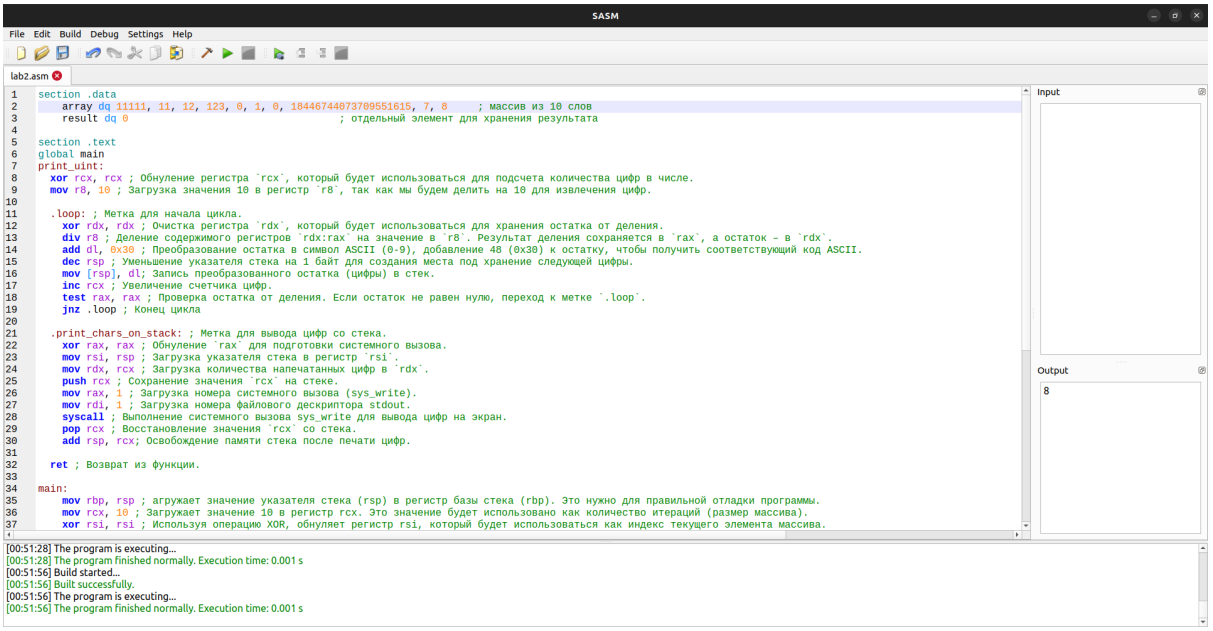


Скриншот 2 – Входной массив [1, 1, 1, 1, 0, 1, 0, 1, 1, 1]. Правильный полученный ответ 8.

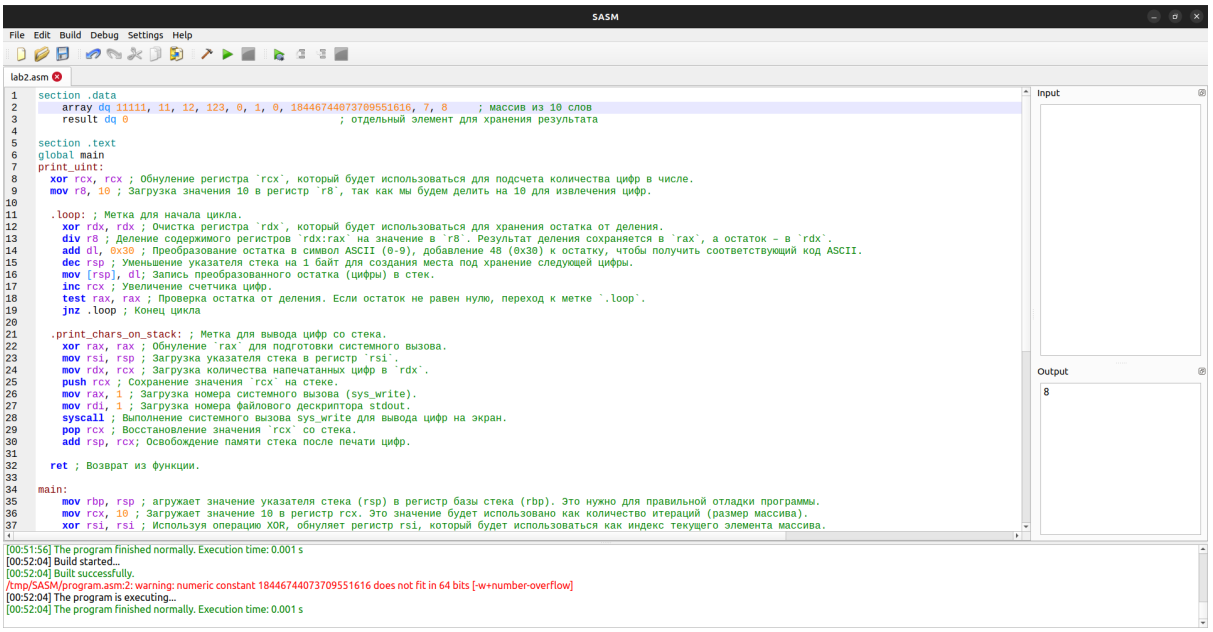
Остаток от деления единицы на тройку будет равен единице. Остаток от деления двойки на тройку будет равен двойке. Как видно на Скриншоте 2, программа абсолютно правильно считает искомое значение суммы остатков.



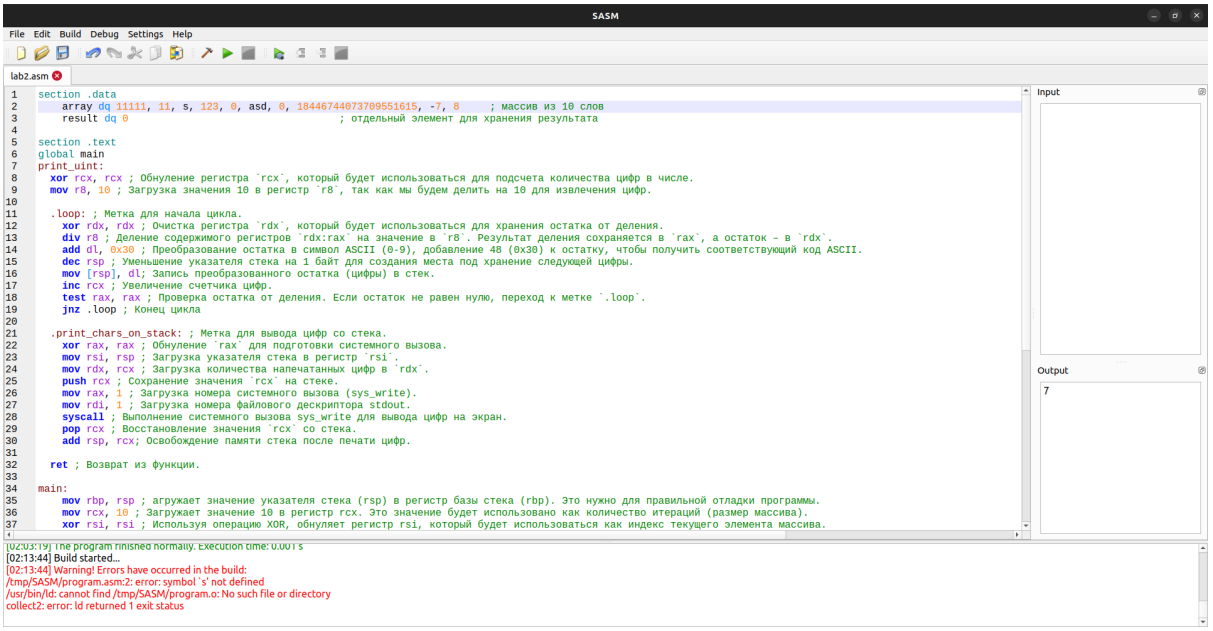
Скриншот 3 – Входной массив [11111, 11, 12, 123, 0, 1, 0, 4, 7, 8]. Правильный полученный ответ 9.



Скриншот 4 – Входной массив [11111, 11, 12, 123, 0, 1, 0, 18446744073709551615, 7, 8]. Правильный полученный ответ 8.



Скриншот 5 – Входной массив [11111, 11, 12, 123, 0, 1, 0, 18446744073709551616, 7, 8]. Ошибка переполнения. Программа выполняется, но выполняется неправильно, так как она обрезает значение 18446744073709551616.



Скриншот 6 - Входной массив [11111, 11, s, 123, 0, asd, 0, 18446744073709551616, -7, 8]. Ошибка переполнения. Ошибка входных данных.

Как мы можем заметить (Скриншот 6), в данной программе входной массив имеет следующий вид: [11111, 11, s, 123, 0, asd, 0, 18446744073709551616, -7, 8]. При этом программа правильно отработает для отрицательного значения -7, но не сможет обработать ячейки массива “s” и “asd”, так как сочтёт их за необъявленные переменные в процессе компиляции, а также не сможет корректно оперировать с числом 18446744073709551616, так как оно выходит за рамки допустимого множества чисел, с которыми мы можем оперировать в рамках данного кода.
Таким образом, мы подробно рассмотрели программную реализацию и проверили код на работоспособность, а также изучили критические моменты. Входные параметры могут варьироваться от -18446744073709551615 до 18446744073709551615, а на выходе мы можем получить число от 0 до 20, так как сумма остатков от нулевого массива будет нулевой, а при массиве, который заполнен двойками, мы получим сумму из максимальных остатков при делении на три, которая будет равна 20 при десяти элементах массива.
Прилагаю подробно расписанный код-блокнот в ПРИЛОЖЕНИИ А.

3 Выводы о проделанной работе

В результате выполнения данной лабораторной работы мне удалось успешно написать кодовую реализацию для поставленного задания, определить область входных и выходных данных, в которой может работать программа и проверить её на работоспособность и сбои. Я углубил свои познания в теоретической части и стал лучше понимать изучаемую тему.

ПРИЛОЖЕНИЕ А

Листинг полного кода на ассемблере.

```
section .data
    array dq 1, 1, 1, 1, 0, 1, 0, 1, 1, 1      ; массив из 10 слов
    result dq 0                                ; отдельный элемент для хранения результата

section .text
global main
print_uint:
  xor rcx, rcx ; Обнуление регистра rcx, который будет использоваться для подсчета количества цифр в числе.
  mov r8, 10 ; Загрузка значения 10 в регистр r8, так как мы будем делить на 10 для извлечения цифр.

  .loop: ; Метка для начала цикла.
    xor rdx, rdx ; Очистка регистра rdx, который будет использоваться для хранения остатка от деления.
    div r8 ; Деление содержимого регистров rdx:rax на значение в r8. Результат деления сохраняется в rax, а остаток – в rdx.
    add dl, 0x30 ; Преобразование остатка в символ ASCII (0-9), добавление 48 (0x30) к остатку, чтобы получить соответствующий код ASCII.
    dec rsp ; Уменьшение указателя стека на 1 байт для создания места под хранение следующей цифры.
    mov [rsp], dl; Запись преобразованного остатка (цифры) в стек.
    inc rcx ; Увеличение счетчика цифр.
    test rax, rax ; Проверка остатка от деления. Если остаток не равен нулю, переход к метке .loop.
    jnz .loop ; Конец цикла

  .print_chars_on_stack: ; Метка для вывода цифр со стека.
    xor rax, rax ; Обнуление rax для подготовки системного вызова.
    mov rsi, rsp ; Загрузка указателя стека в регистр rsi.
    mov rdx, rcx ; Загрузка количества напечатанных цифр в rdx.
```

```
    push rcx ; Сохранение значения rcx на стеке.
    mov rax, 1 ; Загрузка номера системного вызова (sys_write).
    mov rdi, 1 ; Загрузка номера файлового дескриптора stdout.
    syscall ; Выполнение системного вызова sys_write для вывода цифр на экран.
    pop rcx ; Восстановление значения rcx со стека.
    add rsp, rcx; Освобождение памяти стека после печати цифр.

  ret ; Возврат из функции.

main:
    mov rbp, rsp ; агружает значение указателя стека (rsp) в регистр базы стека (rbp). Это нужно для правильной отладки программы.
    mov rcx, 10 ; Загружает значение 10 в регистр rcx. Это значение будет использовано как количество итераций (размер массива).
    xor rsi, rsi ; Используя операцию XOR, обнуляет регистр rsi, который будет использоваться как индекс текущего элемента массива.
    xor rbx, rbx ; Используя операцию XOR, обнуляет регистр rbx, который будет использоваться для хранения суммы остатков.

calculate_remainders:
    mov rax, [array + rsi*8] ; Загружает значение текущего элемента массива в регистр rax.
                             ; Здесь используется адресация с индексом, который умножается на размер элемента массива (в данном случае 8,
                             ; так как используется 64-битная архитектура), а затем добавляется к адресу массива (предполагается,
                             ; что адрес массива сохранен в памяти перед выполнением данного кода).
    xor rdx, rdx ; Используя операцию XOR, обнуляет регистр rdx, который будет использоваться для хранения остатка от деления.
    mov rdi, 3 ; Загружает значение 3 в регистр rdi. Это значение будет использовано в качестве делителя при делении текущего элемента массива на 3.
```

div rdi ; Выполняет деление текущего элемента массива, хранящегося в регистре rax, на 3. Результат деления сохраняется в регистре rax, а остаток от деления - в регистре rdx.

add rbx, rdx ; Добавляет остаток от деления (хранящийся в регистре rdx) к общей сумме остатков (хранящейся в регистре rbx).

add rsi, 1 ; Увеличивает значение индекса текущего элемента массива (хранящегося в регистре rsi) на 1.

cmp rsi, rcx ; Сравнивает значение индекса текущего элемента массива (хранящегося в регистре rsi) с общим количеством итераций (хранящимся в регистре rcx).

jl calculate_remainders ; Если значение индекса текущего элемента массива (rsi) меньше, чем общее количество итераций (rcx), то переходит обратно к метке calculate_remainders и продолжает выполнение цикла.

mov [result], rbx ; Сохраняет значение суммы остатков (хранящейся в регистре rbx) в отдельный элемент данных с именем result. Предполагается, что в памяти уже выделено место для хранения этого значения.

; вывод результата

mov rax, [result] ; Загружает значение из элемента данных с именем result в регистр rax. Здесь предполагается, что значение суммы остатков уже сохранено в этот элемент данных.

call print_uint ; Вызывает функцию с именем print_uint, которая предназначена для вывода беззнакового целого числа (значение из регистра rax) на экран или в другой поток вывода.

mov rax, 60 ; Загружает значение 60 в регистр rax. Это значение будет использовано для системного вызова, который завершает программу.

syscall ; Выполняет системный вызов с использованием значения, загруженного в регистр rax (в данном случае значение 60).

; Этот системный вызов будет завершать программу.