Министерство науки и высшего образования Российской Федерации
Национальный научно-исследовательский университет ИТМО
Факультет программной инженерии и компьютерной техники

Лабораторная работа №3

по дисциплине

«Архитектура компьютерных систем».

Вариант

asm | stack | neum | mc -> hw | tick -> instr | struct | stream | port | cstr | prob2 | cache.

Базовый вариант. Без усложнения. Собственный язык (Kotlin).

Работу выполнил:
Афанасьев Кирилл Александрович,
Студент группы P3206.
Преподаватель:
Пенской Александр Владимирович.

Санкт-Петербург, 2024

Оглавление

Задание

Детали задания: https://gitlab.se.ifmo.ru/computer-systems/csa-rolling/-/blob/master/lab3-task.md

Язык программирования и система команд

Данные 2 раздела были объединены, поскольку ЯП – стековый ассемблер, и все его инструкции в точности повторяют систему команд процессора.

Синтаксис:

```
<line> ::= <label> <comment>? "\n"
       | <instr> <comment>? "\n"
       | <comment> "\n"

<program> ::= <line>*

<label> ::= <label_name> ":"

<instr> ::= <op0>
        | <op1> " " <label_name>
        | <op1> " " <positive_integer>

<op0> ::= "nop"
      | "word"
      | "lit"
      | "load"
      | "store"
      | "add"
      | "sub"
      | "inc"
      | "dec"
      | "drop"
      | "dup"
      | "or"
      | "and"
      | "xor"
      | "ret"
      | "in"
      | "out"
      | "halt"

<op1> ::= "jmp"
      | "jz"
      | "call"

<positive_integer> ::= [0-9]+
```

```
<integer> ::= "-"? <positive_integer>

<lowercase_letter> ::= [a-z]
<uppercase_letter> ::= [A-Z]
<letter> ::= <lowercase_letter> | <uppercase_letter>

<letter_or_number> ::= <letter> | <integer>
<letter_or_number_with_underscore> ::= <letter_or_number> | "_"

<label_name> ::= <letter> <letter_or_number_with_underscore>*

<any_letter> ::= <letter_or_number_with_underscore> | " "

<comment> ::= " "* ";" " "* <letter_or_number_with_underscore>*
```

Семантика:

Код выполняется последовательно, одна инструкция за другой.

Система команд. Она же список доступных операций:

- NOP – нет операции.
- WORD <literal> – объявить переменную в памяти.
- LIT <literal> – положить литерал на вершину стека данных.
- LOAD { address } – загрузить из памяти значение по адресу с вершины стека.
- STORE { address, element } – положить значение в память по указанному адресу.
- ADD { e1, e2 } – положить на стек результат операции сложения e2 + e1.
- SUB { e1, e2 } – положить на стек результат операции вычитания e2 – e1.
- MUL { e1, e2 } – положить на стек результат операции умножения e2 * e1.
- DIV { e1, e2 } – положить на стек результат операции деления e2 / e1.
- MOD { e1, e2 } – положить на стек результат операции взятия остатка e2 % e1.
- INC { element } – увеличить значение вершины стека на 1.
- DEC { element } – уменьшить значение вершины стека на 1.
- DROP { element } – удалить элемент из стека.
- DUP { element } – дублировать элемент на стеке.
- SWAP { e1, e2 } – поменять на стеке два элемента местами.
- OVER { e1 } [e2] – дублировать первый элемент на стеке через второй. Если в стеке только 1 элемент – поведение не определено.
- AND { e1, e2 } – положить на стек результат операции логического И e2 & e1.
- OR { e1, e2 } – положить на стек результат операции логического ИЛИ e2 | e1.
- XOR { e1, e2 } – положить на стек результат операции исключающего ИЛИ e2 ^ e1.
- JZ { element, address } – если элемент равен 0, начать исполнять инструкции по указанному адресу. Разновидность условного перехода.
- JN { element, address } – если элемент отрицательный, начать исполнять инструкции по указанному адресу. Разновидность условного перехода.
- JUMP { address } – совершить безусловный переход по указанному адресу.
- CALL { address } – начать исполнение процедуры по указанному адресу.
- RET – вернуться из процедуры в основную программу, на следующий адрес.

- IN { port } – получить данные из внешнего устройства по указанному порту.
- OUT { port, value } – отправить данные во внешнее устройство по указанному порту.
- HALT – остановка тактового генератора.

Операнды, которые берутся со стека, обозначаются { в фигурных скобках }. Непосредственное указание операнда производится <в угловых скобках>. Если операции требуется дополнительный операнд, но он не используется, он обозначен [в квадратных скобках].
Таким образом, единственный способ явного взаимодействия со стеком – операция LIT <const>. Все остальные операции работают со стеком, и берут операнды оттуда. Будьте внимательны. Если команда задействовала операнд – он будет изъят из стека без возможности возврата. Пользуйтесь командами DUP и OVER для сохранения значения на стеке.

Метки определяются на отдельной строке исходного кода:

```
label:
    word 42
```

Далее метки могут быть использованы (неважно, до или после определения) в исходном коде:

```
label_addr:
    word label    ; в памяти будет хранится адрес на ячейку памяти word 42
```

Метки не чувствительны к регистру. Повторное определение меток недопустимо. Определение меток `label` и `LaBeL` считается повторением. Использовать определённую метку можно неограниченное число раз. Транслятор поставит на место использования метки адрес той инструкции, перед которой она определена.

Любая программа обязана иметь метку `start`, указывающую на первую исполняемую команду.

Ниже приведен пример программы на ассемблере стекового процессора:

```
res:
        word 0      ; result accumulator
fac:
        dup         ; Stack: arg arg
        lit 1       ; Stack: arg arg 1
        sub         ; Stack: arg 0/pos_num
        lit break   ; Stack: arg 0/pos_num break
        swap        ; Stack: arg break 0/pos_num
        jz          ; Stack: arg
        dup         ; Stack: arg arg
        dec         ; Stack: arg (arg - 1) -> arg
        lit fac     ; Stack: [...] arg fac
        call        ; Stack: [...] res
        mul         ; Stack: res
break:
        ret         ; Stack: arg/res

start:
        lit 11      ; Stack: 11
```

```
        lit fac     ; Stack: 11 fac
        call        ; Stack: 11!
        lit res     ; Stack: 11! res_addr
        store       ; Stack: <empty>
        halt        ; halted
```

Листинг 1. Пример программы, вычисляющий факториал числа 11. Комментарии демонстрируют состояние стека данных в момент после исполнения команды.

Машинный код преобразуется в список JSON, где один элемент списка — это одна инструкция. Индекс инструкции в списке – адрес этой инструкции в памяти.

Пример машинного слова:

```
{
  "type": "io.github.zerumi.csa3.isa.MemoryCell.OperandInstruction",
  "opcode": "LIT",
  "operand": 2
},
```

где:

- type – служебная информация о типе данных для десериализации.
- opcode – строка с кодом операции
- operand – аргумент команды (обязателен для инструкций с операндом)

Система команд реализована в модуле isa.

Транслятор

CLI: `java -jar asm-1.0.jar <input_file> <target_file>`

Реализован в модуле `asm`.

Трансляция реализуется в два прохода:

1) Генерация машинного кода без адресов переходов и расчёт значений меток перехода.
   Ассемблерные мнемоники один в один транслируются в машинные команды, кроме мнемоники WORD – в ее случае в памяти инициализируется константа без какого-либо опкода. Тем не менее, WORD, наряду с инструкциями, также поддерживает метки.
2) Подстановка меток перехода в инструкции.

Организация памяти

Организация памяти:

- Машинное слово – не определено. Реализуется высокоуровневой структурой данных. Операнд – 32-битный. Интерпретируется как знаковое целое число.
- Адресация – только прямая загрузка литерала в ячейку памяти (WORD) / на вершину стека (LIT). Косвенная адресация достижима с использованием стека.
- Программа и данные хранятся в общей памяти согласно архитектуре Фон-Неймановского процессора. Программа состоит из набора инструкций, последняя инструкция – HALT. Процедуры объявляются в той же памяти, они обязаны завершаться при помощи команды RET.

- Особенность реализации: данные не могут интерпретироваться как команды. При попытке исполнить команду, которая отображена в ячейке памяти как данные, процесс моделирования будет аварийно остановлен. При попытке прочитать операнд и записать его в стек данных у команды, не имеющей операнда, поведение не определено.
- Операция записи в память перезапишет ячейку памяти как ячейку с данными. Программист имеет доступ на чтение/запись в любую ячейку памяти.
- Литералы интерпретируются как знаковые 32-разрядные числа. Константы отсутствуют.

Организация стека:

- Стек реализован в виде отдельного регистра, представляющего вершину стека (TOS) + высокоуровневой структуры данных ArrayDeque.
- Стек 32-разрядный и позволяет полностью помещать один операнд одной команды.

Модель процессора

CLI: `java -jar comp-1.0.jar [-p | --program-file <filepath>] [-i | --input-file] [-o | --output-file] [<-stdout | --log-stdout> | <-l | --log-file <filepath>>] [--memory-initial-size <size>] [--data-stack-size <size>] [--return-stack-size <size>]`
Или `java -jar comp-1.0.jar [-h | --help]`

Реализован в модуле `comp`.

Схема Datapath:

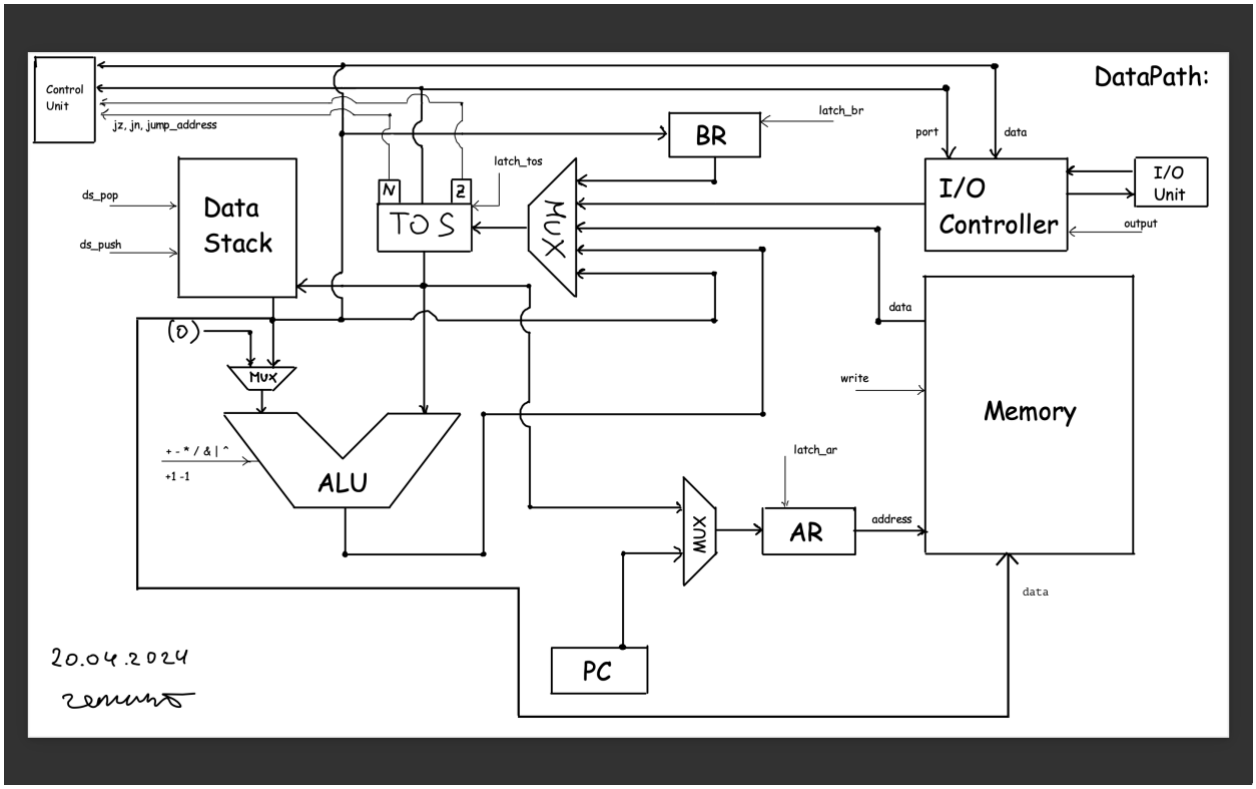

Рисунок 1. Схема Datapath.

Описание сигналов:

- data_stack_push – защелкнуть второй элемент в стеке данных значение из вершины стека.
- data_stack_pop – убрать второй элемент из стека данных.
- latch_tos – защелкнуть выбранное значение в вершину стека. Значение может быть выбрано:
    - Из АЛУ, как результат бинарной операции над вершиной стека и его вторым элементов,
    - Из стека данных (второй элемент стека),
    - Из памяти (загрузится операнд, либо данные, напоминание, при попытке защелкнуть в вершину стека значение ячейки памяти, представляющее собой инструкцию без операнда – поведение не определено),
    - Из внешнего устройства,
    - Из буферного регистра.
- latch_br – защелкнуть значение второго элемента стека данных в буферный регистр. Сделано для поддержки команд SWAP и OVER.
- memory_write – сигнал в память для перезаписи данных в ячейку. Данные берутся из второго элемента стека.
- output – сигнал контроллеру внешних устройств для записи данных во внешнее устройство.
- latch_ar – защелкнуть выбранное значение (из PC или из TOS) в регистр адреса.

Флаги:

- zero (Z) – отражает наличие нулевого значения на вершине стека.
- negative (N) – отражает наличие отрицательного значения на вершине стека.

Схема ControlUnit:

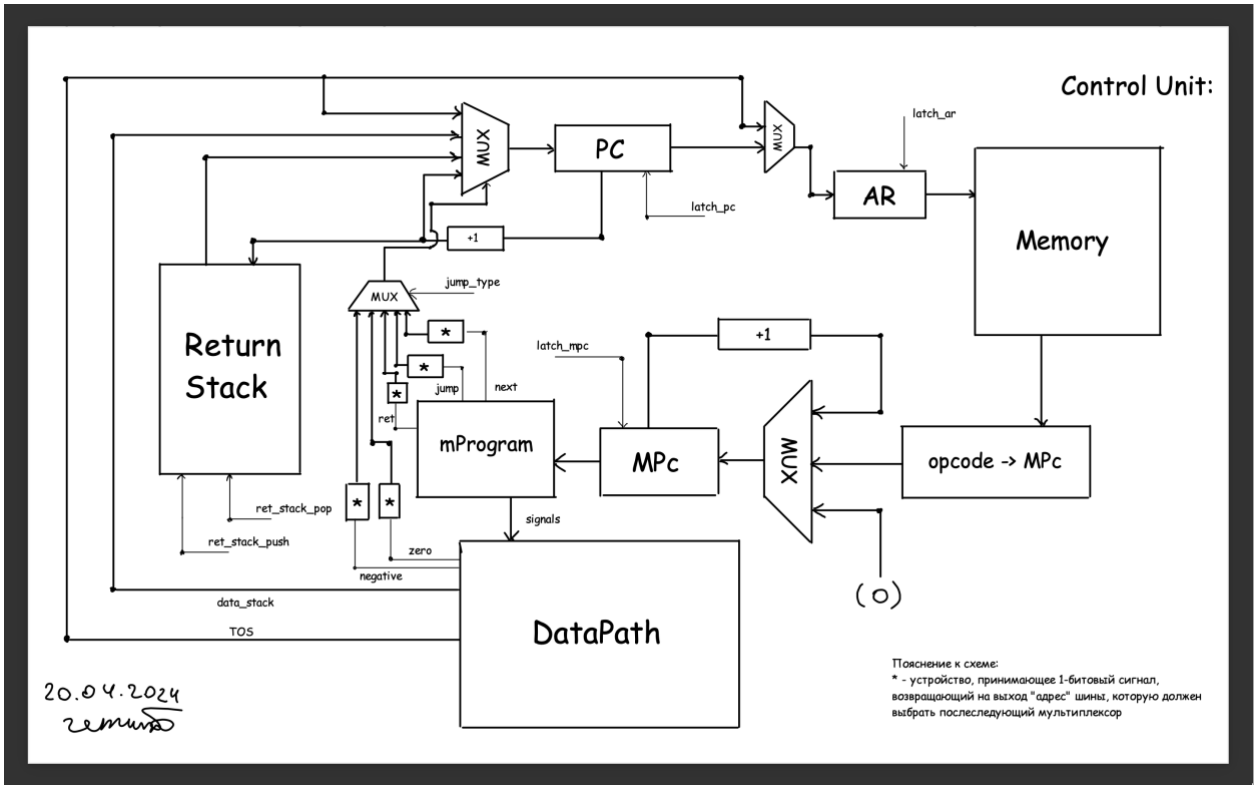

Рисунок 2. Схема ControlUnit.

Описание реализации:

- Микропрограммное управление.
- Метод dispatchMicroInstruction расшифровывает микрокод и исполняет его посигнально.
- Процесс моделирования – потактовый. Каждый шаг выводиться в файл логгирования.
- Начало симуляции происходит в функции `simulate`. Процесс моделирования продолжается до исполнения инструкции `HALT`.
- Остановка моделирования происходит в ряде случаев:
    - Попытка исполнить команду, которая представлена в памяти как данные,
    - Если поток ввода во внешнем устройстве закончился, а мы попробовали получить еще элемент,
    - Если исполнена инструкция `HALT`,
    - Если стек данных и/или стек возврата был пуст при необходимом наличии там значения (для исполнения команды)
- Переполнения стека в данной модели не предусмотрено.
- Особенность реализации: в случае множественного выбора значения для защелкивания, функциям, реализующим конкретные сигналы передается вся микроинструкция, функция, в свою очередь, берут оттуда лишь то, что им нужно.

Описание сигналов:

- latch_ar – защелкнуть выбранное значение (из PC или из TOS) в регистр адреса.
- latch_pc – защелкнуть выбранное значение из PC. Значение может быть выбрано:
    - Как инкремент предыдущего (следующая инструкция, а также JZ и JN),
    - Из стека возврата (возврат из процедуры),
    - Из вершины стека (JUMP),
    - Из второго элемента стека (JZ, JN).
- latch_MPc – защелкнуть выбранное значение счетчика микрокоманд. Значение может быть выбрано:
    - Нулем (начальная микроинструкция),
    - Как инкремент предыдущего (следующая микроинструкция),
    - С помощью устройства преобразования опкода в адрес в микропрограмме.
- ret_stack_push – защелкнуть в стеке возврата инкрементированное значение PC.
- ret_stack_pop – убрать элемент из стека возврата.

Тестирование

Тестирование выполняется при помощи golden-тестов.

1. Тестовый класс реализован в модуле `comp/src/test/integration`
2. Там же находятся ресурсы для 5 тестируемых алгоритмов:
    a. Hello – печатает на выход “Hello World!”
    b. Hello Username – печатает на выход приветствие пользователя.
    c. Cat – повторяет поток ввода на вывод
    d. Prob2 – сумма четных чисел из ряда Фибоначчи (до 4.000.000)
    e. Fac – факториал числа (демонстрация работы стека возврата)

Запустить тесты:

```
./gradlew :comp:integrationTest --tests "AlgorithmTest.catTest"
./gradlew :comp:integrationTest --tests "AlgorithmTest.helloTest"
./gradlew :comp:integrationTest --tests "AlgorithmTest.helloUserNameTest"
```

```
  ./gradlew :comp:integrationTest --tests "AlgorithmTest.prob2Test"
  ./gradlew :comp:integrationTest --tests "AlgorithmTest.facTest"
```

Обновить конфигурацию golden-тестов:

```
  ./gradlew :comp:integrationTest --tests "AlgorithmTest.catTest" -
DupdateGolden=true
  ./gradlew :comp:integrationTest --tests "AlgorithmTest.helloTest" -
DupdateGolden=true
  ./gradlew :comp:integrationTest --tests "AlgorithmTest.helloUserNameTest" -
DupdateGolden=true
  ./gradlew :comp:integrationTest --tests "AlgorithmTest.prob2Test" -
DupdateGolden=true
  ./gradlew :comp:integrationTest --tests "AlgorithmTest.facTest" -
DupdateGolden=true
```

Реализованный ci.yml в GitHub Actions:

```
name: CI

# Controls when the workflow will run
on:
  # Triggers the workflow on push or pull request events but only for the
"master" branch
  push:
    branches: [ "master" ]
  pull_request:
    branches: [ "master" ]

  # Allows you to run this workflow manually from the Actions tab
  workflow_dispatch:

# A workflow run is made up of one or more jobs that can run sequentially or
in parallel
jobs:
  # This workflow contains a single job called "build"
  lint:
    # The type of runner that the job will run on
    runs-on: ubuntu-latest

    # Steps represent a sequence of tasks that will be executed as part of
the job
    steps:
      # Checks-out your repository under $GITHUB_WORKSPACE, so your job can
access it
      - uses: actions/checkout@v3

      - name: Run DeteKt check
        uses: natiginfo/action-detekt-all@1.23.6

      - name: Run Kotlin linter
        uses: vroy/gha-kotlin-linter@v4

  build:
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v3
```

```
      - name: Gradle Build Action
        uses: gradle/gradle-build-action@v3.3.1
        with:
          gradle-version: 8.4

  test:
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v3

      - name: Golden tests
        run: |
          ./gradlew :comp:integrationTest --tests "AlgorithmTest.catTest"
          ./gradlew :comp:integrationTest --tests "AlgorithmTest.helloTest"
          ./gradlew :comp:integrationTest --tests
"AlgorithmTest.helloUserNameTest"
          ./gradlew :comp:integrationTest --tests "AlgorithmTest.prob2Test"
          ./gradlew :comp:integrationTest --tests "AlgorithmTest.facTest"
```

Использованные шаблоны:

- DeteKt All – запускает полную проверку написанного кода при помощи линтера DeteKt.
- KtLint – запускает полную проверку написанного кода при помощи линтера KtLint.
- Gradle Build Action – проверка того, что исходный код может быть собран.

Пример использования проекта:

```
zerumi@MacBook-Air-Kirill csa3-example % pwd
/Users/zerumi/Desktop/csa3-example
zerumi@MacBook-Air-Kirill csa3-example % java -jar asm-1.0.jar fac.sasm fac.json
zerumi@MacBook-Air-Kirill csa3-example % touch in.txt
zerumi@MacBook-Air-Kirill csa3-example % touch out.txt
zerumi@MacBook-Air-Kirill csa3-example % touch log.txt
zerumi@MacBook-Air-Kirill csa3-example % ls
asm-1.0.jar
comp-1.0.jar
fac.json
fac.sasm
in.txt
log.txt
out.txt
zerumi@MacBook-Air-Kirill csa3-example % java -jar comp-1.0.jar -i in.txt -o out.txt -
l log.txt -p fac.json
zerumi@MacBook-Air-Kirill csa3-example % cat log.txt
[INFO]: io.github.zerumi.csa3.comp.ControlUnit - NOW EXECUTING INSTRUCTION PC: 13 -->
LIT 11
[INFO]: io.github.zerumi.csa3.comp.ControlUnit -
TICK 0 -- MPC: 0 / MicroInstruction: LatchAR, ARSelectPC, LatchMPCounter,
MicroProgramCounterNext
Stack (size = 1): [0 | ]
Return stack (size = 0): []
PC: 13 AR: 13 BR: 0

[INFO]: io.github.zerumi.csa3.comp.ControlUnit -
TICK 1 -- MPC: 1 / MicroInstruction: LatchMPCounter, MicroProgramCounterOpcode
Stack (size = 1): [0 | ]
Return stack (size = 0): []
PC: 13 AR: 13 BR: 0
```

```
[INFO]: io.github.zerumi.csa3.comp.ControlUnit -
TICK 2 -- MPC: 3 / MicroInstruction: DataStackPush, LatchAR, ARSelectPC,
LatchMPCounter, MicroProgramCounterNext
Stack (size = 2): [0 | 0]
Return stack (size = 0): []
PC: 13 AR: 13 BR: 0

[INFO]: io.github.zerumi.csa3.comp.ControlUnit -
TICK 3 -- MPC: 4 / MicroInstruction: LatchTOS, TOSSelectMemory, LatchMPCounter,
MicroProgramCounterZero, LatchPC, PCJumpTypeNext
Stack (size = 2): [11 | 0]
Return stack (size = 0): []
PC: 14 AR: 13 BR: 0
MEMORY READ VALUE: AR: 13 ---> OperandInstruction(opcode=LIT, operand=11)

.....

[INFO]: io.github.zerumi.csa3.comp.ControlUnit - NOW EXECUTING INSTRUCTION PC: 17 -->
STORE
[INFO]: io.github.zerumi.csa3.comp.ControlUnit -
TICK 559 -- MPC: 0 / MicroInstruction: LatchAR, ARSelectPC, LatchMPCounter,
MicroProgramCounterNext
Stack (size = 3): [0 | 39916800, 0]
Return stack (size = 0): []
PC: 17 AR: 17 BR: 0

[INFO]: io.github.zerumi.csa3.comp.ControlUnit -
TICK 560 -- MPC: 1 / MicroInstruction: LatchMPCounter, MicroProgramCounterOpcode
Stack (size = 3): [0 | 39916800, 0]
Return stack (size = 0): []
PC: 17 AR: 17 BR: 0

[INFO]: io.github.zerumi.csa3.comp.ControlUnit -
TICK 561 -- MPC: 7 / MicroInstruction: LatchAR, ARSelectTOS, LatchMPCounter,
MicroProgramCounterNext
Stack (size = 3): [0 | 39916800, 0]
Return stack (size = 0): []
PC: 17 AR: 0 BR: 0

[INFO]: io.github.zerumi.csa3.comp.DataPath - MEMORY WRITTEN VALUE: AR: 0 <---
39916800
[INFO]: io.github.zerumi.csa3.comp.ControlUnit -
TICK 562 -- MPC: 8 / MicroInstruction: MemoryWrite, LatchMPCounter,
MicroProgramCounterNext
Stack (size = 3): [0 | 39916800, 0]
Return stack (size = 0): []
PC: 17 AR: 0 BR: 0

[INFO]: io.github.zerumi.csa3.comp.ControlUnit -
TICK 563 -- MPC: 9 / MicroInstruction: DataStackPop, LatchMPCounter,
MicroProgramCounterNext
Stack (size = 2): [0 | 0]
Return stack (size = 0): []
PC: 17 AR: 0 BR: 0

[INFO]: io.github.zerumi.csa3.comp.ControlUnit -
TICK 564 -- MPC: 10 / MicroInstruction: LatchTOS, TOSSelectDS, LatchMPCounter,
MicroProgramCounterNext
Stack (size = 2): [0 | 0]
Return stack (size = 0): []
PC: 17 AR: 0 BR: 0
```

```
[INFO]: io.github.zerumi.csa3.comp.ControlUnit -
TICK 565 -- MPC: 11 / MicroInstruction: DataStackPop, LatchMPCounter,
MicroProgramCounterZero, LatchPC, PCJumpTypeNext
Stack (size = 1): [0 | ]
Return stack (size = 0): []
PC: 18 AR: 0 BR: 0

[INFO]: io.github.zerumi.csa3.comp.ControlUnit - NOW EXECUTING INSTRUCTION PC: 18 -->
HALT
[INFO]: io.github.zerumi.csa3.comp.ControlUnit -
TICK 566 -- MPC: 0 / MicroInstruction: LatchAR, ARSelectPC, LatchMPCounter,
MicroProgramCounterNext
Stack (size = 1): [0 | ]
Return stack (size = 0): []
PC: 18 AR: 18 BR: 0

[INFO]: io.github.zerumi.csa3.comp.ControlUnit - [HALTED]
```

Пример тестирования исходного кода:

```
zerumi@MacBook-Air-Kirill csa3-140324-asm-stack % pwd
/Users/zerumi/IdeaProjects/csa3-140324-asm-stack
zerumi@MacBook-Air-Kirill csa3-140324-asm-stack % ./gradlew :comp:integrationTest --
tests "AlgorithmTest.facTest"
Starting a Gradle Daemon, 2 incompatible and 1 stopped Daemons could not be reused,
use --status for details

> Configure project :comp
w: file:///Users/zerumi/IdeaProjects/csa3-140324-asm-stack/comp/build.gradle.kts:72:9:
'getter for testSourceDirs: (Mutable)Set<File!>!' is deprecated. Deprecated in Java
w: file:///Users/zerumi/IdeaProjects/csa3-140324-asm-stack/comp/build.gradle.kts:72:9:
'getter for testSourceDirs: (Mutable)Set<File!>!' is deprecated. Deprecated in Java

The Kotlin Gradle plugin was loaded multiple times in different subprojects, which is
not supported and may break the build.
This might happen in subprojects that apply the Kotlin plugins with the Gradle
'plugins { ... }' DSL if they specify explicit versions, even if the versions are
equal.
Please add the Kotlin plugin to the common parent project or the root project, then
remove the versions in the subprojects.
If the parent project does not need the plugin, add 'apply false' to the plugin line.
See:
https://docs.gradle.org/current/userguide/plugins.html#sec:subprojects_plugins_dsl
The Kotlin plugin was loaded in the following projects: ':asm', ':isa'

Deprecated Gradle features were used in this build, making it incompatible with Gradle
9.0.

You can use '--warning-mode all' to show the individual deprecation warnings and
determine if they come from your own scripts or plugins.

For more on this, please refer to
https://docs.gradle.org/8.4/userguide/command_line_interface.html#sec:command_line_war
nings in the Gradle documentation.

BUILD SUCCESSFUL in 10s
15 actionable tasks: 4 executed, 11 up-to-date

The build scan was not published due to a configuration problem.
```

```
The buildScan extension 'termsOfUseAgree' value must be exactly the string 'yes'
(without quotes).
The value given was 'no'.

For more information, please see https://gradle.com/help/gradle-plugin-terms-of-use.

Alternatively, if you are using Develocity, specify the server location.
For more information, please see https://gradle.com/help/gradle-plugin-config.

| Афанасьев Кирилл Александрович | hello | 47 | - | 23 | 209 | 914 | asm |
stack | neum | mc -> hw | tick -> instr | struct | stream | port | cstr |
prob2 | cache |
| Афанасьев Кирилл Александрович | prob2 | 95 | - | 79 | 786 | 3799 | asm |
stack | neum | mc -> hw | tick -> instr | struct | stream | port | cstr |
prob2 | cache |
| Афанасьев Кирилл Александрович | fac | 23 | - | 18 | 133 | 566 | asm |
stack | neum | mc -> hw | tick -> instr | struct | stream | port | cstr |
prob2 | cache |
```

Исходный код программы

GitHub: https://github.com/Zerumi/csa3-140324-asm-stack

Вывод

Во время выполнения данной лабораторной работы я экспериментально ознакомился с устройством стекового процессора путем его моделирования. Мною был получен опыт программирования на «bare» metal (на смоделированном процессоре), я разработал алгоритмы, которые могут быть исполнены на данном процессоре (например, cat без операционной системы). Мною было проработаны 3 уровня архитектуры: уровень железа, программы и системной платформы.
Дополнительно я на практике ознакомился с Continuous Integration, разработал интеграционные golden-тесты для своего проекта, а также научился пользоваться GitHub Actions.