Министерство образования и науки РФ

Федеральное государственное автономное
образовательное учреждение высшего образования
«Национальный исследовательский университет ИТМО»

Факультет программной инженерии и компьютерной техники

Лабораторная работа 3

вариант 19

по дисциплине

«Системное программное обеспечение»

Выполнил:

Кустарев Иван Палович 309681

Преподаватель:

Кореньков Юрий Дмитриевич

Санкт-Петербург, 2025

Цель:

Реализовать формирование линейного кода в терминах некоторого набора инструкций посредством анализа графа потока управления для набора подпрограмм. Полученный линейный код вывести в мнемонической форме в выходной текстовый файл.

Задачи:

Подготовка к выполнению по одному из двух сценариев:

1. Составить описание виртуальной машины с набором инструкций и моделью памяти по варианту
    a. Изучить нотацию для записи определений целевых архитектур
    b. Составить описание ВМ в соответствии с вариантом
        i. Описание набор регистров и банков памяти
        ii. Описать набор инструкций: для каждой инструкции задать структуру операционного кода, содержащего описание операндов и набор операций, изменяющих состояние ВМ
            1. Описать инструкции перемещения данных и загрузки констант
            2. Описать инструкции арифметических и логических операций
            3. Описать инструкции условной и безусловной передачи управления
            4. Описать инструкции ввода-вывода с использованием скрытого регистра в качестве порта ввода-вывода
        iii. Описать набор мнемоник, соответствующих инструкциям ВМ
    c. Подготовить скрипт для запуска ассемблированного листинга с использованием описания ВМ:
        i. Написать тестовый листинг с использованием подготовленных мнемоник инструкций
        ii. Задействовать транслятор листинга в бинарный модуль по описанию ВМ
        iii. Запустить полученный бинарный модуль на исполнение и получить результат работы
        iv. Убедиться в корректности функционирования всех инструкций ВМ

2. Выбрать и изучить прикладную архитектуру системы команд существующей ВМ
    a. Для выбранной ВМ:
        i. Должен существовать готовый эмулятор (например qemu)
        ii. Должен существовать готовый тулчейн (набор инструментов разработчика): компилятор Си, ассемблер и дизассемблер, линковщик, желательно отладчик
    b. Согласовать выбор ВМ с преподавателем
    c. Изучить модель памяти и набор инструкций ВМ
    d. Научиться использовать тулчейн (собирать и запускать программы из листинга)
    e. Подготовить скрипт для запуска ассемблированного листинга с использованием эмулятора
        i. Написать тестовый листинг с использованием инструкций ВМ
        ii. Задействовать ассемблер и компоновщик из тулчейна
        iii. Запустить бинарный модуль на исполнение и получить результат его работы

Порядок выполнения:

1. Описать структуры данных, необходимые для представления информации об элементах образа программы (последовательностях инструкций и данных), расположенных в памяти
    a. Для каждой инструкции – имя мнемоники и набор операндов в терминах данной ВМ
    b. Для элемента данных – соответствующее литеральное значение или размер экземпляра типа данных в байтах
2. Реализовать модуль, формирующий образ программы в линейном коде для данного набора подпрограмм
    a. Программный интерфейс модуля принимает на вход структуру данных, содержащую графы потока управления и информацию о локальных переменных и сигнатурах для набора подпрограмм, разработанную в задании 2 (п. 1.a, п. 2.b)
    b. В результате работы порождается структура данных, разработанная в п. 1, содержащая описание образа программы в памяти: набор именованных элементов данных и набор именованных фрагментов линейного кода, представляющих собой алгоритмы подпрограмм
    c. Для каждой подпрограммы посредством обхода узлов графа потока управления в порядке топологической сортировки (начиная с узла, являющегося первым базовым блоком алгоритма подпрограммы), сформировать набор именованных групп инструкций, включая пролог и эпилог подпрограммы (формирующие и разрушающие локальное состояние подпрограммы)
    d. Для каждого базового блока в составе графа потока управления сформировать группу инструкций, соответствующих операциям в составе дерева операций
    e. Использовать имена групп инструкций для формирования инструкций перехода между блоками инструкций, соответствующих узлам графа потока управления, в соответствии с дугами в нём
3. Доработать тестовую программу, разработанную в задании 2 для демонстрации работоспособности созданного модуля
    a. Добавить поддержку аргумента командной строки для имени выходного файла, вывод информации о графах потока управления сделать опциональных
    b. Использовать модуль, разработанный в п. 2 для формирования образа программы на основе информации, собранной в результате работы модуля, созданного в задании 2 (п. 2.b)
    c. Для сформированного образа программы в линейном коде вывести в выходной файл ассемблерный листинг, содержащий мнемоническое представление инструкций и данных, как они описаны в структурах данных (п. 1), построенных разработанным модулем (пп. 2.c-e)
    d. Проверить корректность решения посредством сборки сгенерированного листинга и запуска полученного бинарного модуля на эмуляторе ВМ (см. подготовка п. 1.c или п. 2.e)
4. Результаты тестирования представить в виде отчета, в который включить:
    a. В части 3 привести описание разработанных структур данных
    b. В части 4 описать программный интерфейс и особенности реализации разработанного модуля
    c. В части 5 привести примеры исходных текстов, соответствующие ассемблерные листинги и примера вывода запущенных тестовых программ

Выполнение:

Данная лабораторная работа реализует функционал создания листингов на основе предыдущих лабораторных работ. Для создания листингов использовались следующие структуры, сохраненные в файле заголовков:

```
//
// Created by Ivan on 26.01.2025.
//
#ifndef SPO_LAB1_LISTING_H
#define SPO_LAB1_LISTING_H

#include "main.h"

typedef struct ListingNode ListingNode;
typedef struct ValuePlaceAssociation ValuePlaceAssociation;

struct ListingNode {
   ExecutionNode *node;
   char *label;
   int checked;
};

struct ValuePlaceAssociation {
   char *name;
   char *type;
   int shiftPosition;
};

void placeLabels(Array *funExecutions);

void printListing(Array *funExecutions, FILE *listingFile);

#endif //SPO_LAB1_LISTING_H
```

Обход графа, полученного из модулей, созданных в предыдущей лабораторной работе происходит дважды. Во время первого обхода расставляются метки, если к выражению нужно будет совершить доступ через команду перехода. Во второй обход листинг записывается в файл.

Структура в памяти следующая:

- в самом начале лежат команды
- стек растет с конца
- куча растет сверху вниз, начинается после команд

Для аллокации памяти в куче используется линейный аллокатор.

В памяти у для каждого блока данных выделяется 64 бита на значение и 64 бита на тип данных. Поскольку типизация статическая - при выражениях с разными типами берётся тип первого операнда.

Конфигурации виртуальной машины:

```
architecture spo309681 {

  registers:
    storage SP[16]; /* Стековый указатель */
    storage IP[16]; /* Указатель на текущую инструкцию */
    storage BP[16]; /* Указатель на базовый адрес фрейма стека */
    storage HP[16]; /* Указатель на кучу */
    storage IN_PORT[8]; /* Порт ввода */
    storage OUT_PORT[8]; /* Порт вывода */

    storage EX[16]; /* ДЛЯ ПРОВЕРКИ ОШИБОК */

  memory:
    range ram [0x0000..0xFFFF] {
      cell = 8;
      endianess = big-endian;
      granularity = 0;
    }

  instructions:
    encode imm64 field = immediate [64];
    encode imm16 field = immediate [16];

    instruction push = { 0000 0000 0000 0001, imm64 as value } {
      SP = SP - 8;
      ram:8[SP] = value;
      IP = IP + 10;
    };

    instruction pop = { 0000 0000 0000 0010 } {
      SP = SP + 8;
      IP = IP + 2;
    };

    instruction load_bp = { 0000 0000 0000 0100, imm16 as shift } {
      if (shift >> 15) == 1 then {
//          без & 0x0FFFF: Error: Access out of addressable space range: 0x8ffc0 vs ram[0x0..0x0ffff]
        SP = SP - 8;
        ram:8[SP] = ram:8[(BP + ((~shift + 1) + 1) * 8) & 0x0FFFF];
        SP = SP - 8;
        ram:8[SP] = ram:8[(BP + (~shift + 1) * 8) & 0x0FFFF];
        IP = IP + 4;
      } else {
        SP = SP - 8;
        ram:8[SP] = ram:8[BP - (shift * 8)];
        SP = SP - 8;
```

```
        ram:8[SP] = ram:8[BP - ((shift + 1) * 8)];
        IP = IP + 4;
    }
};

instruction load = { 0000 0000 0000 0101 } {
    let addr = ram:8[SP];
    ram:8[SP + 8] = ram:8[addr];
    ram:8[SP] = ram:8[addr + 8];
    IP = IP + 2;
};

instruction save_bp = { 0000 0000 0000 0110, imm16 as shift } {
    ram:8[BP - ((shift + 1) * 8)] = ram:8[SP];
    SP = SP + 8;
    ram:8[BP - (shift * 8)] = ram:8[SP];
    SP = SP + 8;
    IP = IP + 4;
};

instruction save = { 0000 0000 0000 0111 } {
    let value = ram:8[SP];
    SP = SP + 8;
    let type = ram:8[SP];
    SP = SP + 8;
    let addr = ram:8[SP];
    ram:8[addr] = type;
    ram:8[addr + 8] = value;
    IP = IP + 2;
};

instruction load_in = { 0000 0000 0000 1000 } {
    SP = SP - 8;
    ram:8[SP] = 1;
    SP = SP - 8;
    ram:8[SP] = IN_PORT;
    IP = IP + 2;
};

instruction save_out = { 0000 0000 0000 1001 } {
    OUT_PORT = ram:8[SP];
    SP = SP -16;
    IP = IP + 2;
};

instruction sum = { 0000 0000 0001 0000 } {
    let b = ram:8[SP];
    SP = SP + 8;
    let bt = ram:8[SP];
    SP = SP + 8;
    let a = ram:8[SP];
    let at = ram:8[SP+8];
    let value = a + b;
    ram:8[SP] = value;
    IP = IP + 2;
};
```

```
instruction sub = { 0000 0000 0010 0000 } {
    let b = ram:8[SP];
    SP = SP + 8;
    let bt = ram:8[SP];
    SP = SP + 8;
    let a = ram:8[SP];
    let at = ram:8[SP+8];
    let value = a - b;
    ram:8[SP] = value;
    IP = IP + 2;
};

instruction mul = { 0000 0000 0011 0000 } {
    let b = ram:8[SP];
    SP = SP + 8;
    let bt = ram:8[SP];
    SP = SP + 8;
    let a = ram:8[SP];
    let at = ram:8[SP+8];
    let value = a * b;
    ram:8[SP] = value;
    IP = IP + 2;
};

instruction div = { 0000 0000 0100 0000 } {
    let b = ram:8[SP];
    SP = SP + 8;
    let bt = ram:8[SP];
    SP = SP + 8;
    let a = ram:8[SP];
    let at = ram:8[SP+8];
    let value = a / b;
    ram:8[SP] = value;
    IP = IP + 2;
};

instruction mod = { 0000 0000 0101 0000 } {
    let b = ram:8[SP];
    SP = SP + 8;
    let bt = ram:8[SP];
    SP = SP + 8;
    let a = ram:8[SP];
    let at = ram:8[SP+8];
    let value = a % b;
    ram:8[SP] = value;
    IP = IP + 2;
};

instruction op_and = { 0000 0000 0110 0000 } {
    let b = ram:8[SP];
    SP = SP + 8;
    let bt = ram:8[SP];
    SP = SP + 8;
    let a = ram:8[SP];
    let at = ram:8[SP+8];
```

```
    let value = a & b;
    ram:8[SP] = value;
    IP = IP + 2;
};

instruction op_or = { 0000 0000 0111 0000 } {
    let b = ram:8[SP];
    SP = SP + 8;
    let bt = ram:8[SP];
    SP = SP + 8;
    let a = ram:8[SP];
    let at = ram:8[SP+8];
    let value = a | b;
    ram:8[SP] = value;
    IP = IP + 2;
};

instruction lt = { 0000 0000 1000 0000 } {
    let b = ram:8[SP];
    SP = SP + 8;
    let bt = ram:8[SP];
    SP = SP + 8;
    let a = ram:8[SP];
    let at = ram:8[SP+8];
    if (a < b) then ram:8[SP] = 1; else ram:8[SP] = 0;
    IP = IP + 2;
};

instruction lte = { 0000 0000 1001 0000 } {
    let b = ram:8[SP];
    SP = SP + 8;
    let bt = ram:8[SP];
    SP = SP + 8;
    let a = ram:8[SP];
    let at = ram:8[SP+8];
    if (a <= b) then ram:8[SP] = 1; else ram:8[SP] = 0;
    IP = IP + 2;
};

instruction gt = { 0000 0000 1010 0000 } {
    let b = ram:8[SP];
    SP = SP + 8;
    let bt = ram:8[SP];
    SP = SP + 8;
    let a = ram:8[SP];
    let at = ram:8[SP+8];
    if (a > b) then ram:8[SP] = 1; else ram:8[SP] = 0;
    IP = IP + 2;
};

instruction gte = { 0000 0000 1011 0000 } {
    let b = ram:8[SP];
    SP = SP + 8;
    let bt = ram:8[SP];
    SP = SP + 8;
    let a = ram:8[SP];
```

```
    let at = ram:8[SP+8];
    if (a >= b) then ram:8[SP] = 1; else ram:8[SP] = 0;
    IP = IP + 2;
};

instruction eq = { 0000 0000 1100 0000 } {
    let b = ram:8[SP];
    SP = SP + 8;
    let bt = ram:8[SP];
    SP = SP + 8;
    let a = ram:8[SP];
    let at = ram:8[SP+8];
    if (a == b) then ram:8[SP] = 1; else ram:8[SP] = 0;
    ram:8[SP+8] = 1;
    IP = IP + 2;
};

instruction neq = { 0000 0000 1101 0000 } {
    let b = ram:8[SP];
    SP = SP + 8;
    let bt = ram:8[SP];
    SP = SP + 8;
    let a = ram:8[SP];
    let at = ram:8[SP+8];
    if (a != b) then ram:8[SP] = 1; else ram:8[SP] = 0;
    ram:8[SP+8] = 1;
    IP = IP + 2;
};

instruction jz = { 0000 0001 0000 0000, imm64 as addr } {
    if (ram:8[SP] == 0) then IP = addr; else IP = IP + 10;
    SP = SP + 16;
};

instruction jnz = { 0000 0010 0000 0000, imm64 as addr } {
    if (ram:8[SP] != 0) then IP = addr; else IP = IP + 10;
    SP = SP + 16;
};

instruction jmp = { 0000 0011 0000 0000, imm64 as addr } {
    IP = addr;
};

instruction call = { 0001 0000 0000 0000, imm64 as addr } {
    SP = SP - 8;
    ram:8[SP] = BP;
    SP = SP - 8;
    ram:8[SP] = IP + 10;
    BP = SP;
    IP = addr;
};

instruction ret = { 0010 0000 0000 0000, imm64 as arg_number } {
    SP = BP;
    IP = ram:8[SP];
    SP = SP + 8;
```

```
        let oldbp = ram:8[SP];
        SP = SP + 8;
        SP = SP + arg_number * 16;
        SP = SP - 8;
        EX = BP - 8;
        BP = BP - 8;
        ram:8[SP] = ram:8[BP];
        SP = SP - 8;
        BP = BP - 8;
        ram:8[SP] = ram:8[BP];
        BP = oldbp;
    };

    instruction alloc = { 1110 0000 0000 0000, imm64 as byte_size } {
        SP = SP - 8;
        ram:8[SP] = 4;
        SP = SP - 8;
        ram:8[SP] = HP;
        HP = HP + byte_size * 8;
        IP = IP + 10;
    };

    instruction init = { 1111 0000 0000 0000, imm64 as code_end_addr } {
        HP = code_end_addr;
        IP = IP + 10;
    };

    instruction hlt = { 1111 1111 1111 1111 } {};

mnemonics:
    mnemonic PUSH for push(value) "{1}";
    mnemonic POP for pop();
    mnemonic LOAD_BP for load_bp(shift) "{1}";
    mnemonic SAVE_BP for save_bp(shift) "{1}";
    mnemonic LOAD for load();
    mnemonic LOAD_IN for load_in();
    mnemonic SAVE_OUT for save_out();
    mnemonic SAVE for save();

    mnemonic SUM for sum();
    mnemonic SUB for sub();
    mnemonic MUL for mul();
    mnemonic DIV for div();
    mnemonic MOD for mod();
    mnemonic BIT_AND for op_and();
    mnemonic OR for op_or();
    mnemonic LT for lt();
    mnemonic LTE for lte();
    mnemonic GT for gt();
    mnemonic GTE for gte();
    mnemonic EQ for eq();
    mnemonic NEQ for neq();

    mnemonic JZ for jz(addr) "{1}";
    mnemonic JNZ for jnz(addr) "{1}";
    mnemonic JMP for jmp(addr) "{1}";
```

```
    mnemonic CALL for call(addr) "{1}";
    mnemonic RET for ret(arg_number) "{1}";

    mnemonic ALLOC for alloc(byte_size) "{1}";
    mnemonic INIT for init(code_end_addr) "{1}";
    mnemonic HLT for hlt();
}
```

Команды для запуска:

```
.\Portable.RemoteTasks.Manager.exe -ul <login> -up <userpassword> -w -id -s AssembleDebug definitionFile architecture.pdsl archName spo309681 asmListing test-list.asm sourcesDir output

.\Portable.RemoteTasks.Manager.exe -ul <login> -up <userpassword> -g 1c560d56-9622-49e1-9523-11e66802175e -r out.ptptb -o out.ptptb

.\Portable.RemoteTasks.Manager.exe -ul <login> -up <userpassword> -il -q -w -s ExecuteBinaryWithInteractiveInput stdinRegStName IN_PORT stdoutRegStName OUT_PORT definitionFile architecture.pdsl archName spo309681 binaryFileToRun out.ptptb codeRamBankName ram ipRegStorageName IP finishMnemonicName HLT
```

Результаты:

```
[section ram]
INIT code_end_addr
CALL main
POP
HLT
sum:
PUSH 0
PUSH 0
LOAD_BP -4
LOAD_BP -2
SUM
SAVE_BP 1
RET 2
sub:
PUSH 0
PUSH 0
LOAD_BP -4
LOAD_BP -2
SUB
SAVE_BP 1
RET 2
mul:
PUSH 0
PUSH 0
LOAD_BP -4
LOAD_BP -2
MUL
SAVE_BP 1
RET 2
div:
PUSH 0
PUSH 0
LOAD_BP -4
LOAD_BP -2
DIV
SAVE_BP 1
RET 2
read_num:
PUSH 0
PUSH 0
PUSH 0
PUSH 0
PUSH 0
PUSH 0
PUSH 0
PUSH 0
PUSH 1
PUSH 0
SAVE_BP 5
PUSH 1
PUSH 1
SAVE_BP 7
label2:
```

```
PUSH 1
PUSH 1
JNZ label1
label4:
LOAD_BP 5
SAVE_BP 1
RET 0
label1:
LOAD_IN
SAVE_BP 3
LOAD_BP 3
PUSH 1
PUSH 13
EQ
JNZ label3
LOAD_BP 5
PUSH 1
PUSH 10
MUL
LOAD_BP 3
PUSH 1
PUSH 48
SUB
SUM
SAVE_BP 5
LOAD_BP 7
PUSH 1
PUSH 1
SUM
SAVE_BP 7
JMP label2
label3:
LOAD_IN
JMP label4
number_length:
PUSH 0
PUSH 0
PUSH 0
PUSH 0
PUSH 0
PUSH 0
PUSH 1
PUSH 1
SAVE_BP 5
LOAD_BP -2
PUSH 1
PUSH 10
DIV
SAVE_BP 3
label6:
LOAD_BP 3
PUSH 1
PUSH 0
NEQ
JNZ label5
LOAD_BP 5
```

```
SAVE_BP 1
RET 1
label5:
LOAD_BP 3
PUSH 1
PUSH 10
DIV
SAVE_BP 3
LOAD_BP 5
PUSH 1
PUSH 1
SUM
SAVE_BP 5
JMP label6
print_num:
PUSH 0
PUSH 0
PUSH 0
PUSH 0
PUSH 0
PUSH 0
PUSH 0
PUSH 0
LOAD_BP -2
SAVE_BP 3
LOAD_BP -2
CALL number_length
PUSH 1
PUSH 1
SUB
SAVE_BP 5
PUSH 1
PUSH 1
SAVE_BP 7
label10:
LOAD_BP 5
PUSH 1
PUSH 0
NEQ
JNZ label9
label8:
LOAD_BP 7
PUSH 1
PUSH 0
NEQ
JNZ label7
PUSH 1
PUSH 10
SAVE_OUT
PUSH 1
PUSH 13
SAVE_OUT
PUSH 1
PUSH 0
SAVE_BP 1
RET 1
```

```
label7:
LOAD_BP 3
LOAD_BP 7
DIV
PUSH 1
PUSH 48
SUM
SAVE_OUT
LOAD_BP 3
LOAD_BP 7
MOD
SAVE_BP 3
LOAD_BP 7
PUSH 1
PUSH 10
DIV
SAVE_BP 7
JMP label8
label9:
LOAD_BP 7
PUSH 1
PUSH 10
MUL
SAVE_BP 7
LOAD_BP 5
PUSH 1
PUSH 1
SUB
SAVE_BP 5
JMP label10
main:
PUSH 0
PUSH 0
PUSH 0
PUSH 0
PUSH 0
PUSH 0
PUSH 0
PUSH 0
PUSH 0
PUSH 0
label12:
PUSH 1
PUSH 1
JNZ label11
SAVE_BP 1
RET 0
label11:
CALL read_num
SAVE_BP 3
LOAD_IN
SAVE_BP 7
LOAD_IN
LOAD_IN
CALL read_num
SAVE_BP 5
```

```
LOAD_BP 7
PUSH 3
PUSH 43
EQ
JNZ label19
label20:
LOAD_BP 7
PUSH 3
PUSH 45
EQ
JNZ label17
label18:
LOAD_BP 7
PUSH 3
PUSH 42
EQ
JNZ label15
label16:
LOAD_BP 7
PUSH 3
PUSH 47
EQ
JNZ label13
label14:
LOAD_BP 9
CALL print_num
JMP label12
label13:
LOAD_BP 3
LOAD_BP 5
CALL div
SAVE_BP 9
JMP label14
label15:
LOAD_BP 3
LOAD_BP 5
CALL mul
SAVE_BP 9
JMP label16
label17:
LOAD_BP 3
LOAD_BP 5
CALL sub
SAVE_BP 9
JMP label18
label19:
LOAD_BP 3
LOAD_BP 5
CALL sum
SAVE_BP 9
JMP label20
code_end_addr:
```

Вывод:

В результате выполнения лабораторной работы познакомился с конфигурацией бинарных команд для виртуальной машины, принципами работы и конфигурации виртуальной машины. Ознакомился с процессом формирования листингов на основе графа потока управления.