МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«Санкт Петербургский национальный-исследовательский университет Информационных технологий, механики и оптики»

Факультет программной инженерии и компьютерной техники

«Системное программное обеспечение»

Выполнил студент группы №P4116
Веденин Вадим Витальевич

Проверил
Кореньков Юрий Дмитриевич

САНКТ -ПЕТЕРБУРГ
2024

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

Архитектура эмулятора

```
architecture spo {

registers:
      storage r0st [16];
      storage r1st [16];
      storage r2st [16];
      storage r3st [16];
      storage ip [16];
      storage spst [16];
      storage fpst [16];

      storage inp [8];
      storage outp [8];

    view inReg = inp;
    view outReg = outp;
      view r0 = r0st;
      view r1 = r1st;
      view r2 = r2st;
      view r3 = r3st;
      view ipv = ip;
      view sp = spst;
      view fp = fpst;

memory:

      range code_ram [0x0000 .. 0xffff] {
            cell = 8;
            endianess = little-endian;
            granularity = 2;
      }

      range data_ram [0x0000 .. 0xffff] {
            cell = 8;
            endianess = little-endian;
            granularity = 2;
      }

instructions:

      encode imm16 field = immediate [16];

      encode reg field = register {
            r0 = 0000,
            r1 = 0001,
            r2 = 0010,
            r3 = 0011,
            ipv = 0100,
            sp = 0101,
            fp = 0110,
            inReg = 0111,
            outReg = 1000
      };
```

```
      instruction add-reg2reg = { 0000 0000, reg as op1, reg as op2, reg as to,
0000 0000 0000} {
            to = op1 + op2;
            ip = ip + 4;
      };

      instruction add-imm2reg = { 0000 0001, reg as op1, imm16 as value, reg as
to} {
            to = op1 + value;
            ip = ip + 4;
      };
      instruction sub-reg2reg = { 0000 0010, reg as op1, reg as op2, reg as to,
0000 0000 0000} {
            to = op1 - op2;
            ip = ip + 4;
      };

      instruction sub-imm2reg = { 0000 0011, reg as op1, imm16 as value, reg as
to} {
            to = op1 - value;
            ip = ip + 4;
      };

      instruction asl = { 0000 0100, reg as op1, reg as to, 0000 0000 0000 0000
} {
            to = op1 << 1;
            ip = ip + 4;
      };

      instruction asr = { 0000 0101, reg as op1, reg as to, 0000 0000 0000 0000
} {
            to = op1 << 1;
            ip = ip + 4;
      };

    instruction mov-reg2reg = { 0000 0110, reg as op1, reg as to, 0000 0000 0000
0000 } {
        to = op1;
        ip = ip + 4;
    };

    instruction mov-imm2reg = { 0000 0111, imm16 as value, reg as to, 0000 } {
        to = value;
        ip = ip + 4;
    };

    instruction invert = { 0000 1000, reg as op1, reg as to, 0000 0000 0000 0000
} {
        to = !op1;
        ip = ip + 4;
    };

    instruction negative = { 0000 1001, reg as op1, reg as to, 0000 0000 0000
0000 } {
        to = -op1;
        ip = ip + 4;
    };

    instruction and-reg2reg = { 0000 1010, reg as op1, reg as op2, reg as to,
0000 0000 0000} {
            to = op1 && op2;
            ip = ip + 4;
      };
```

```
    instruction jumpgt = { 0001 0110, reg as op1, imm16 as to, 0000} {
       if (op1 >> 15 == 0x0) && (op1 != 0x0) then
       {
           ip = to;
       }
       else
       {
           ip = ip + 4;
       }
    };
    instruction jumpge = { 0001 0111, reg as op1, imm16 as to, 0000} {
        if (op1 >> 15 == 0x0) then
        {
           ip = to;
        }
        else
        {
           ip = ip + 4;
        }
    };
    instruction jumplt = { 0001 1000, reg as op1, imm16 as to, 0000} {
        if op1 >> 15 == 0x1 then // op1 < 0
            ip = to;
        else
            ip = ip + 4;
    };

    instruction jumple = { 0001 1001, reg as op1, imm16 as to, 0000} {
       if (op1 >> 15 == 0x1) || (op1 == 0x0) then // op1 <= 0
       {
           ip = to;
       }
       else
       {
           ip = ip + 4;
       }
    };

      encode bank sequence = alternatives {
            d = {0000},
            c = {0001},
            t = {0011}
      };

      instruction st = { 0001 1010 0000, reg as from, imm16 as ptr } {
            data_ram:1[ptr] = from;
            data_ram:1[ptr+1] = from>>8;

            ip = ip + 4;
      };

      instruction ld = { 0001 1011 0000, reg as to, imm16 as ptr } {
            to = data_ram:1[ptr] + (data_ram:1[ptr+1] << 8);

            ip = ip + 4;
      };

       instruction push = { 0001 1100, reg as from, 0000 0000 0000 0000 0000 }
{
        sp = sp - 2;
        data_ram:1[sp] = from;
        data_ram:1[sp+1] = from >> 8;
            ip = ip + 4;
    };
       instruction pop = { 0001 1101, reg as to, 0000 0000 0000 0000 0000 } {
        to = data_ram:1[sp] + (data_ram:1[sp+1] << 8);
        sp = sp + 2;
         ip = ip + 4;
    };
```

```
    instruction and-imm2reg = { 0000 1011, reg as op1, imm16 as value, reg as
to} {
             to = op1 & value;
             ip = ip + 4;
      };
    instruction or-reg2reg = { 0000 1100, reg as op1, reg as op2, reg as to,
0000 0000 0000} {
        to = op1 | op2;
        ip = ip + 4;
    };

    instruction or-imm2reg = { 0000 1101, reg as op1, imm16 as value, reg as
to} {
        to = op1 | value;
        ip = ip + 4;
    };
    instruction div-reg2reg = { 0000 1110, reg as op1, reg as op2, reg as to,
0000 0000 0000} {
        to = op1 / op2;
        ip = ip + 4;
    };

    instruction div-imm2reg = { 0000 1111, reg as op1, imm16 as value, reg as
to} {
        to = op1 / value;
        ip = ip + 4;
    };
    instruction mul-reg2reg = { 0001 0000, reg as op1, reg as op2, reg as to,
0000 0000 0000} {
        to = op1 * op2;
        ip = ip + 4;
    };

    instruction mul-imm2reg = { 0001 0001, reg as op1, imm16 as value, reg as
to} {
        to = op1 * value;
        ip = ip + 4;
    };
    instruction rem-reg2reg = { 0001 0010, reg as op1, reg as op2, reg as to,
0000 0000 0000} {
        to = op1 % op2;
        ip = ip + 4;
    };

    instruction rem-imm2reg = { 0001 0011, reg as op1, imm16 as value, reg as
to} {
        to = op1 % value;
        ip = ip + 4;
    };

    instruction jump = { 0001 0100, imm16 as to, 0000 0000} {
        ip = to;
    };

    instruction jumpeq = { 0001 0101, reg as op1, imm16 as to, 0000} {
        if op1 == 0x0 then
        {
            ip = to;
        }
        else
        {
            ip = ip + 4;
        }
    };
```

```
    instruction jumpgt = { 0001 0110, reg as op1, imm16 as to, 0000} {
       if (op1 >> 15 == 0x0) && (op1 != 0x0) then
       {
           ip = to;
       }
       else
       {
           ip = ip + 4;
       }
    };
    instruction jumpge = { 0001 0111, reg as op1, imm16 as to, 0000} {
        if (op1 >> 15 == 0x0) then
        {
           ip = to;
        }
        else
        {
           ip = ip + 4;
        }
    };
    instruction jumplt = { 0001 1000, reg as op1, imm16 as to, 0000} {
        if op1 >> 15 == 0x1 then // op1 < 0
            ip = to;
        else
            ip = ip + 4;
    };
    instruction jumple = { 0001 1001, reg as op1, imm16 as to, 0000} {
       if (op1 >> 15 == 0x1) || (op1 == 0x0) then // op1 <= 0
       {
           ip = to;
       }
       else
       {
           ip = ip + 4;
       }
    };

      encode bank sequence = alternatives {
            d = {0000},
            c = {0001},
            t = {0011}
      };

      instruction st = { 0001 1010 0000, reg as from, imm16 as ptr } {
            data_ram:1[ptr] = from;
            data_ram:1[ptr+1] = from>>8;

            ip = ip + 4;
      };

      instruction ld = { 0001 1011 0000, reg as to, imm16 as ptr } {
            to = data_ram:1[ptr] + (data_ram:1[ptr+1] << 8);

            ip = ip + 4;
      };
    instruction push = { 0001 1100, reg as from, 0000 0000 0000 0000 0000 } {
        sp = sp - 2;
        data_ram:1[sp] = from;
        data_ram:1[sp+1] = from >> 8;
            ip = ip + 4;
    };
```

```
    instruction pop = { 0001 1101, reg as to, 0000 0000 0000 0000 0000 } {
        to = data_ram:1[sp] + (data_ram:1[sp+1] << 8);
        sp = sp + 2;
             ip = ip + 4;
    };

    instruction call = { 0001 1110, imm16 as ptr, 0000 0000 } {
        sp = sp - 2;
        data_ram:1[sp] = ip;
        data_ram:1[sp+1] = ip >> 8;
        sp = sp - 2;
        data_ram:1[sp] = fp;
        data_ram:1[sp+1] = fp >> 8;
        fp = sp;
        ip = ptr;
    };
    instruction ret = { 0001 1111 1111 1111 1111 1111 0000 0000 } {
        if fp != 0 then {
            sp = fp;
            fp = data_ram:1[sp] + (data_ram:1[sp+1] << 8);
            sp = sp + 2;
            ip = data_ram:1[sp] + (data_ram:1[sp+1] << 8);
            sp = sp + 2;
        }

        ip = ip + 4;
    };
      instruction hlt = { 1111 1111 1111 1111 1111 1111 1111 1111 } {
      };

mnemonics:

    mnemonic hlt();
    mnemonic ret();

    format plain1 is "{1}";
    format plain2 is "{1}, {2}";
    format plain3 is "{1}, {2}, {3}";

    mnemonic store for st(from, ptr) plain2;
    mnemonic load for ld(ptr, to) plain2;

    mnemonic call for call(ptr) plain1;

    mnemonic neg for negative (op1, to) plain2;
    mnemonic _not for invert (op1, to) plain2;

    mnemonic add for add-reg2reg (op1, op2, to) plain3,
                         for add-imm2reg (op1, value, to) plain3;

    mnemonic sub for sub-reg2reg (op1, op2, to) plain3,
                         for sub-imm2reg (op1, value, to) plain3;
    mnemonic mov for mov-reg2reg (op1, to) plain2,
                         for mov-imm2reg ( value, to) plain2;
    mnemonic _and for and-reg2reg (op1, op2, to) plain3,
                         for and-imm2reg (op1, value, to) plain3;
    mnemonic _or for or-reg2reg (op1, op2, to) plain3,
                         for or-imm2reg (op1, value, to) plain3;
    mnemonic mul for mul-reg2reg (op1, op2, to) plain3,
                         for mul-imm2reg (op1, value, to) plain3;
    mnemonic div for div-reg2reg (op1, op2, to) plain3,
                         for div-imm2reg (op1, value, to) plain3;
    mnemonic rem for rem-reg2reg (op1, op2, to) plain3,
                         for rem-imm2reg (op1, value, to) plain3;
```

```
mnemonic push for push(from) plain1;
    mnemonic pop for pop(to) plain1;

    mnemonic jump for jump(to) plain1;
    mnemonic jumpeq for jumpeq(op1, to) plain2;
    mnemonic jumpgt for jumpgt(op1, to) plain2;
    mnemonic jumpge for jumpge(op1, to) plain2;
    mnemonic jumplt for jumplt(op1, to) plain2;
    mnemonic jumple for jumple(op1, to) plain2;
```

Тестовая программа:

```
int read();
void write(int r);

void printNumber(int num, bool n){
    int nextLine = 10;
    int revertedNum = 0;

    if (num == 0) {
        write(48);
    } else {

        while (num != 0) {
            revertedNum = (revertedNum * 10) + (num % 10);
            num = num / 10;
        }
        while (revertedNum != 0) {
            write((revertedNum % 10) + 0x30);
            revertedNum = revertedNum / 10;
        }
        if(n){
            write(10);
        }
    }
}
int readNumber() {
    int c = read();
    int res = 0;
    while(true) {
        if ((c >= 0x30) && (c <= 0x39)){
            res = res * 10 + (c - 0x30);
        } else {
            break;
        }
        c = read();
    }
    return res;
}

void main() {
    int x = 123;
    int r = readNumber();
    int result = x + r;
    int i = 0;
    while(i < 10) {
        result = result + 1;
        i = i + 1;
    }
```

```
    printNumber(result, true);
}
```

Сгенерированный ассемблер

```
[section data_ram]
label_write_val: dw 0x0
printNumber_0: dw 0x0
printNumber_1: dw 0x0
label_8: dw 0x0
label_9: dw 0x0
label_23: dw 0x0
label_24: dw 0x0
label_33: dw 0x0
label_34: dw 0x0
label_35: dw 0x0
label_36: dw 0x0
[section code_ram]
      jump start
read_5:
      mov inReg, r0
      ret

write_7:
      load label_write_val, outReg
      ret

printNumber_2:
      mov 10, r0
      push r0
      pop r0
      store r0, label_8
      mov 0, r0
      push r0
      pop r0
      store r0, label_9
            ;if
            ;EQUALITY(num, 0)
      load printNumber_0, r0
      push r0
      mov 0, r0
      push r0
      pop r1
      pop r0
      sub r0, r1, r0
      jumpeq r0, label_10
      mov 0, r0
      jump label_11
label_10:
      mov 1, r0
label_11:
      push r0
      pop r0
      jumpeq r0, label_13
            ;then
      mov 48, r0
      push r0
      pop r0
      store r0, label_write_val
      call write_7
```

```
push r0
	jump label_12
label_13:
		;else
		;while
label_14:
		;NOTEQUAL(num, 0)
	load printNumber_0, r0
	push r0
	mov 0, r0
	push r0
	pop r1
	pop r0
	sub r0, r1, r0
	jumpeq r0, label_15
	mov 1, r0
	jump label_16
label_15:
	mov 0, r0
label_16:
	push r0
	pop r0
	jumpeq r0, label_17
		;while body
	load label_9, r0
	push r0
	mov 10, r0
	push r0
	pop r1
	pop r0
	mul r0, r1, r0
	push r0
	load printNumber_0, r0
	push r0
	mov 10, r0
	push r0
	pop r1
	pop r0
	rem r0, r1, r0
	push r0
	pop r1
	pop r0
	add r0, r1, r0
	push r0
	pop r0
	store r0, label_9
	load printNumber_0, r0
	push r0
	mov 10, r0
	push r0
	pop r1
	pop r0
	div r0, r1, r0
	push r0
	pop r0
	store r0, printNumber_0
	jump label_14
label_17:
		;end while
		;while
label_18:
		;NOTEQUAL(revertedNum, 0)
	load label_9, r0
	push r0
```

```
mov 0, r0
	push r0
	pop r1
	pop r0
	sub r0, r1, r0
	jumpeq r0, label_19
	mov 1, r0
	jump label_20
label_19:
	mov 0, r0
label_20:
	push r0
	pop r0
	jumpeq r0, label_21
		;while body
	load label_9, r0
	push r0
	mov 10, r0
	push r0
	pop r1
	pop r0
	rem r0, r1, r0
	push r0
	mov 48, r0
	push r0
	pop r1
	pop r0
	add r0, r1, r0
	push r0
	pop r0
	store r0, label_write_val
	call write_7
	push r0
	load label_9, r0
	push r0
	mov 10, r0
	push r0
	pop r1
	pop r0
	div r0, r1, r0
	push r0
	pop r0
	store r0, label_9
	jump label_18
label_21:
		;end while
		;if
		;n
	load printNumber_1, r0
	push r0
	pop r0
	jumpeq r0, label_22
		;then
	mov 10, r0
	push r0
	pop r0
	store r0, label_write_val
	call write_7
	push r0
		;endif
label_22:
		;endif
label_12:
	ret
```

```
readNumber_3:
	call read_5
	push r0
	pop r0
	store r0, label_23
	mov 0, r0
	push r0
	pop r0
	store r0, label_24
		;while
label_25:
		;TRUE(NULL, NULL)
	mov 1, r0
	push r0
	pop r0
	jumpeq r0, label_26
		;while body
		;if
		;AND(braces(GREATERTHANEQ(c, 48), NULL), braces(LESSTHANEQ(c, 57), NULL))
	load label_23, r0
	push r0
	mov 48, r0
	push r0
	pop r1
	pop r0
	sub r0, r1, r0
	jumpge r0, label_27
	mov 0, r0
	jump label_28
label_27:
	mov -1, r0
label_28:
	push r0
	load label_23, r0
	push r0
	mov 57, r0
	push r0
	pop r1
	pop r0
	sub r0, r1, r0
	jumple r0, label_29
	mov 0, r0
	jump label_30
label_29:
	mov -1, r0
label_30:
	push r0
	pop r1
	pop r0
	_and r0, r1, r0
	push r0
	pop r0
	jumpeq r0, label_32
		;then
	load label_24, r0
	push r0
	mov 10, r0
	push r0
	pop r1
	pop r0
	mul r0, r1, r0
	push r0
	load label_23, r0
	push r0
```

```
push r0
	pop r1
	pop r0
	sub r0, r1, r0
	push r0
	pop r1
	pop r0
	add r0, r1, r0
	push r0
	pop r0
	store r0, label_24
	jump label_31
label_32:
		;else
		;break
	jump label_26
		;endif
label_31:
	call read_5
	push r0
	pop r0
	store r0, label_23
	jump label_25
label_26:
		;end while
	load label_24, r0
	push r0
	pop r0
	ret
	ret
start:
main_4:
	mov 123, r0
	push r0
	pop r0
	store r0, label_33
	call readNumber_3
	push r0
	pop r0
	store r0, label_34
	load label_33, r0
	push r0
	load label_34, r0
	push r0
	pop r1
	pop r0
	add r0, r1, r0
	push r0
	pop r0
	store r0, label_35
	mov 0, r0
	push r0
	pop r0
	store r0, label_36
		;while
label_37:
		;LESSTHAN(i, 10)
	load label_36, r0
	push r0
	mov 10, r0
	push r0
	pop r1
	pop r0
	sub r0, r1, r0
	jumplt r0, label_38
```

```
mov 0, r0
      jump label_39
label_38:
      mov -1, r0
label_39:
      push r0
      pop r0
      jumpeq r0, label_40
            ;while body
      load label_35, r0
      push r0
      mov 1, r0
      push r0
      pop r1
      pop r0
      add r0, r1, r0
      push r0
      pop r0
      store r0, label_35
      load label_36, r0
      push r0
      mov 1, r0
      push r0
      pop r1
      pop r0
      add r0, r1, r0
      push r0
      pop r0
      store r0, label_36
      jump label_37
label_40:
            ;end while
      load label_35, r0
      push r0
      pop r0
      store r0, printNumber_0
      mov 1, r0
      push r0
      pop r0
      store r0, printNumber_1
      call printNumber_2
      push r0
      ret
      jump halt
halt:
      hlt
```

Входные данные из файла

```
100
```

Результат запуска

```
./vm/Portable.RemoteTasks.Manager.exe -ul $(cat .env | grep RemoteTasksLogin | c
ut -f2 -d"=" | tr -d '\r') -up $(cat .env | grep RemoteTasksPassword | cut -f2 -
d"=") -s ExecuteBinaryWithInput -w \
                definitionFile arch/spo.target.pdsl \
                archName spo \
                codeRamBankName code_ram \
                ipRegStorageName ip \
                stdinRegStName inp \
        stdoutRegStName outp \
                finishMnemonicName hlt \
                inputFile input/stdin.txt \
                binaryFileToRun out/out.ptptb
Task ExecuteBinaryWithInput started with id 5c5aa3aa-eef8-4f31-bab4-d04db948634
Waiting for completion...
Here is task output interpreted with UTF8:
233
```

Вывод

В результате выполнения данной лабораторной работы было написано описание архитектуры для эмулятора, изучено и реализовано формирование линейного кода в терминах некоторого набора инструкций.