Федеральное государственное автономное
образовательное учреждение высшего образования

Университет ИТМО

Отчет
кпрактическомузаданию№3
подисциплине“Системноепрограммноеобеспечение”

Выполнил студент группы P4114: Манна Рани

Преподаватель

: Кореньков

Юрий

Дмитриевич.

г. Санкт-Петербург

2024

Цели

1) *Реализация формирования линейного кода в терминах некоторого набора инструкций посредством анализа графа потока управления для набора подпрограмм.*

2) *Вывод полученного линейного кода в мнемонической форме в выходной текстовый файл.*

Задачи

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*

Описаниеработы

На вход подается текст тестовой программы, который преобразуется в ассемблерный листинг. Полученный ассемблерный листинг передается в виртуальную машину вместе с архитектурой разработанной по варианту, где исполняется. В качестве примера была написана программа-калькулятор, принимающая на вход строку, например “2+3”, и возвращающая результат подсчета, например “5”.

Листинг программы-калькулятора:

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

void printZeroError(){ write(101);
        write(114); write(114);
        write(111); write(114);
        write(10);
}

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

void main() { int firstNumber = 0; int
        secondNumber = 0; int
        operation = 0; int state = 0; int
        res = 0; bool error = false;
        while(true){ int i = read();
        if ((i >= 0x30) && (i <= 0x39)){ int num = i - 0x30;
```

```
        if (state == 0){ firstNumber = firstNumber * 10 + num;
        } else if (state == 2) { secondNumber = secondNumber * 10 + num;
        } else { state = -1; write(i);
                write(10); break;
        }
} else if (state == 0){ if (i == 0x2b){ // +
        state = 2; operation = 1;
        }else if (i == 0x2d){ // state = 2; operation =
                2;
        }else if (i == 0x2a){ // * state = 2; operation
                = 3;
        }else if (i == 0x2f){ // / state = 2; operation =
        4; }
} else if (state == 2) { state = 3; break;
} else {
        write(i); write(10); state = -1;
        break;
}
}
if (state == 3) { if (operation == 1){ res = firstNumber +
secondNumber;
} else if (operation == 2) { res = firstNumber - secondNumber;
} else if (operation == 3) { res = firstNumber * secondNumber;
} else if (operation == 4) { if(secondNumber == 0){
        error = true;
                printZeroError();
        } else { res = firstNumber / secondNumber; }
}
if(error == false){
```

```
                printNumber(firstNumber, secondNumber, operation, res);
            }
            }
}
```

Пример ассемблерного кода, сгенерированного в результате работы программы:

```
[section data_ram]
label_write_val: dw 0x0
printNumber_0: dw 0x0
printNumber_1: dw 0x0 label_6:
dw 0x0 label_7: dw 0x0 label_17:
dw 0x0 label_18: dw 0x0 label_19:
dw 0x0
[section code_ram] jump start
write_5: load label_write_val, outReg ret

printNumber_2: mov 10, r0
        push r0 pop r0 store r0,
        label_6 mov 0, r0 push
        r0 pop r0
        store r0, label_7
            ;while label_8:
            ;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_9 mov
        1, r0 jump label_10
label_9:
        mov 0, r0
label_10: push r0
```

```
        pop r0
        jumpeq r0, label_11
            ;while body load
        label_7, 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_7 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_8
label_11:
            ;end while
            ;while label_12:
            ;NOTEQUAL(revertedNum, 0) load
        label_7, r0 push r0 mov 0, r0 push r0
        pop r1 pop r0 sub r0, r1, r0 jumpeq r0,
        label_13 mov 1, r0
```

2024

```
        jump label_14
label_13:
         mov 0, r0 label_14: push r0
pop r0 jumpeq r0, label_15
            ;while body load label_7, 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_5 push r0
        load label_7, r0 push r0 mov 10, r0
        push r0 pop r1 pop r0 div r0, r1, r0
        push r0 pop r0 store r0, label_7
        jump label_12
label_15:
            ;end while
            ;if
            ;n load printNumber_1, r0
        push r0 pop r0 jumpeq r0,
        label_16
            ;then mov 10, r0 push r0 pop r0
        store r0, label_write_val
```

```
        call write_5 push r0
            ;endif
label_16: ret
start:
main_3:
        mov 6, r0 push r0 pop r0
        store r0, label_17 mov 15,
        r0 push r0 pop r0 store r0,
        label_18 load label_17, r0
        push r0 load label_18, r0
        push r0 pop r1 pop r0 add
        r0, r1, r0 push r0 pop r0
        store r0, label_19 load
        label_19, r0 push r0 pop
        r0
        store r0, printNumber_0
        mov 0, r0 push r0 pop r0 store r0,
        printNumber_1 call
        printNumber_2
        push r0 ret
jump halt halt: hlt
```

Аспектыреализации

Разработанная архитектура:

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

        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 0000 } { to = op1
            << 1;
            ip = ip + 4;
        };

        instruction asr = { 0000 0101, reg as op1, reg as to, 0000
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 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 0000 } { to =
          !op1; ip = ip + 4;
          };
          instruction negative = { 0000 1001, reg as op1, reg as to,
0000 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 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 }
{
      };
```

```
/* instruction readIn = {} {
        r0 = inp;
      };

      instruction writeOut = {} {
        outp = r0;
      };
*/ 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; }
```

Результаты

Результатом работы является программа, способная транслировать текст тестовой программы на ассемблер для запуска на виртуальной машине.

Выводы

В ходе практической работы №3 я изучил работу с ассемблером на варианте, написал описание архитектуры эмулятора и реализовал транслятор в ассемблер. Это было довольно сложно, но в результате мой кругозор успешно расширился.