УНИВЕРСИТЕТ ИТМО

факультет программной инженерии и компьютерной техники

Направление подготовки 09.04.04 Системное и прикладное программное обеспечение

Системное програмное обеспечение
Семестр 1

Лабораторная работа No3

Студент
Хуссейн, Авин
P4114

Преподаватель
Кореньков Ю.Д

Санкт-Петербург, 2024 г.

Table of Contents

Санкт-Петербург, 2024 г.

1. Goals

Implement linear code generation using certain instructions set by CFG analysis for a number of subprograms. Print mnemonic representation of the generated linear code to the output file as an assembly listing.

2. Subtasks

1 Describe a virtual machine having instruction set and memory model according to your variant number

- a. Learn the notation for the target architecture definitions
- b. Describe VM according to your variant number
  - i. Describe a number of registers and memory banks
  - ii. Describe an instructions set: come up with an opcode encoding for each instruction, its
    operands and a VM state changing operation description
    - 1 Describe data movement and constant loading instructions
    - 2 Describe arithmetic and logical instructions
    - 3 Describe conditional and unconditional branching instructions
    - 4 Describe IO-instructions using hidden registry store and an internal IO-port
  - iii. Describe a number of mnemonics for the instructions set
- c. Prepare a script for assembled listing execution using target VM definition
  - i. Create a test listing using prepared mnemonics
  - ii. Use listing translator to generate binary module containing native code according to the target VM definition
  - iii. Execute binary module and retrieve its resulting output
  - iv. Ensure that all the prepared VM instructions work correctly

2 Choose and learn existing hardware VM ISA a. For a given VM

- i. There should be an existing emulator (for example: qemu)
- ii. There should be an existing toolchain (developer tools set): C compiler, assembler, disassembler, linker and debugger

- b. Approve your VM choice with the teacher
- c. Learn memory model and instruction set of the VM
- d. Learn how to use machine toolchain (building and launching programs in form of the assembly listing)
- e. Prepare a script for assembled listing execution using emulator
  - i. Create a test listing using VM instructions' mnemonics
  - ii. Use assembler and linker from the existing toolchain
  - iii. Execute binary module and get its resulting output

3. Solution description

modules this stage of the solution is included in 2 files, “asm.c”.”asm.h” ,“localVariabl.c” and “localVariable.h”

In the “localVariable.h” header file, we define different structures to represent a linked list of variables. The variables have the structure varDeclaration which was defined in lab2.

More detail will be mentioned on that structure, and for full reference please refer to СПО2 report. The basic information about variables declaration was saved previously in the control flow graph as Instruction structures. Those variable declaration structures keep information about the name and the type of the variable, this type was divided into 2 categories : simple and array. The job of this module is to depth first traverse the CFG and create a linked list of all the local variables.

After this list of variables is compiled, the module asm.c starts its job of translating the variables and also in traversing the control flow graph to create the linear code.

3.1 Modules Overview

The codebase is organized into several interrelated modules, each responsible for specific functionality. Below is an explanation of the primary modules and their purposes:

3.1.1. Local variable list

This module defines and manages local variables used in the program. It consists of:

- varDeclaration: holds information about the variable such as its type and id. The type can be a simple type or array. Simple types are ones such as int, bool and long. And array types have 2 important fields which are the type of the elements (which could also be an array) and the number of those elements.
- Node*: a pointer to the next variable.
- filename: name of the file in which the variable was declared.

In the module asm.c, the main function is translate(). This function invokes a number of different methods and is responsible for translating the cfg into a linear code.

3.1.2 dynamic typing and variable translation

- local variables declarations are being translated first. a structure was chosen to represent a variable in the memory in order to retrieve and change meta data about its type at runtime.
- This structure for meta data follows this example

```
i_input2:
  .type: dd 0x0 ; Offset for `type`
  .value: dd 0x10 ; Offset for `value
```

- For each variable we keep a label with its name and in this label we reserve a place in the memory to attach information about its type and value. The type changes at runtime depending on the operation and the operands.
- The types and values are pushed to the stack and inside the arithmetic functions in assembly a check is done at runtime to make sure that the operation is done in a type appropriate way. The following snippet shows an example.

```
            // Check if both operands are of type "number" (type == 2)
              if (type1 == 2) && (type2 == 2) then {
                let x = ram:2[ptr1 + 4];    // Fetch the value of the first operand
                let y = ram:2[ptr2 + 4];    // Fetch the value of the second operand

                let r = x % y;              // Perform addition
```

1.1.2 Translation implementation details

- **translate() Function**: this function distributes the traversal of the control flow graph into smaller functions that are responsible for translating:
    - Operation trees
    - While loops
    - If conditions
    - Literals and their types
- **translateOT() Function**: This is an important function. Its mission is to check the kinds of operations and operands and distribute them to the correct assembly operations. Since we are using a stack, the operands are pushed to the stack first and then they are translated into assembly functions such as wide_add and wide_store. These functions in assembly are responsible to type check the variables and literals in real time and also to store the values and types after the execution of the operations.

Code Generation Process

- **Assembly Instructions**: the assembly listing is written after the execution of this project
- **RemoteTasks**: is then used to assemble the listing, create a binary file and execute and debug this file.

5. Results

As a result of the work done in this lab I was able to implement linear code translation unit with dynamic type checking. The memory banks used is a common ram and a stack. The result of this lab allowed us to create a fully functioning program in our own language and execute it. First of all we can see the architecture created in the course of this lab work.

```
architecture spo {
/*
    case                    18
    data word length        2
    code model              m_stack
    spaces                  common ram
*/
registers:

  storage sp [16];      // Stack pointer
  storage ip [16];      // Program counter (Instruction pointer)
```

```
  storage inp [16];      // Input register (for input operations)
  storage outp [16];     // Output register (for output operations)

memory:

  range ram [0x0000..0xffff] {   // Common RAM space for code/data
    cell = 16;      // 8-bit cells
    endianess = little-endian;  // Little endian
    granularity = 2;  // 2-byte chunks
  }

  range m_stack [0x0000..0xffff] {   // Stack space
    cell = 16;
    endianess = little-endian;
    granularity = 2;
  }

instructions:
    encode imm16 field = immediate [16];
 // Push immediate value onto the stack
  instruction push = { 0000 0000, imm16 as value } {
    m_stack:2[sp] = value;
    sp = sp - 2;   // Decrement stack pointer
    ip = ip + 3;   // Increment instruction pointer
  };
  // Pop value from the stack into outp
  instruction pop = { 0000 0001 } {
    sp = sp + 2;        // Increment stack pointer
    ip = ip + 1;        // Increment instruction pointer
  };

  instruction popOut = { 0000 0010 } {
    sp = sp + 2;        // Increment stack pointer
    outp = m_stack:2[sp]; // Retrieve value from stack to standard
output
    ip = ip + 1;        // Increment instruction pointer
  };



// Add the top two integers on the stack
  instruction iadd = {0000 0011 } {
    sp = sp + 2;                // Increment stack pointer
    let x = m_stack:2[sp];          // Pop first value into inp (top
value of the stack)

    sp = sp + 2;                // Increment stack pointer
    let y = m_stack:2[sp];         // Pop second value into outp (next
top value)

    let r = x + y;          // Perform addition

    m_stack:2[sp] = r;         // Push result back onto the stack
    sp = sp - 2;                // Decrement stack pointer
    ip = ip + 1;                // Increment instruction pointer
};
// Add the top two longs on the stack
```

```
 instruction ladd = { 0000 0100 } {
    sp = sp + 2;                    // Increment stack pointer
    let x = m_stack:2[sp];              // Pop first value into inp (top
value of the stack)

    sp = sp + 2;                    // Increment stack pointer
    let y = m_stack:2[sp];            // Pop second value into outp (next
top value)

    let r = x + y;           // Perform addition

    m_stack:2[sp] = r;           // Push result back onto the stack
    sp = sp - 2;                    // Decrement stack pointer
    ip = ip + 1;                    // Increment instruction pointer
};
// store the type and the value , maybe I should push the type and
value on the top of the stack
//must test store
instruction st = { 0000 0101, imm16 as ptr }
{
    sp=sp + 2;
    ram:2[ptr] = m_stack:2[sp];
    //ram:1[ptr+1] = from>>8;

    ip = ip + 3;
};
// I will use this to store the size of the array for array type
instruction a_st = { 0000 0110, imm16 as ptr }
{
    sp=sp + 2;
    ram:2[ptr+4] = m_stack:2[sp];// we save 8 bytes for each variable ,
store the type in the first 4 bytes (32 bits from the stack) and the
value in the next
    ip = ip + 3;
};
// to create load I have to have different loads (iload, lload..) I
also need a general funciton to dispatch to them, but I also need a to
recieve the type
// and a way to extract it, and a way to handle the labels, in short I
need to know how I figuered out how to store it in metadata section
// were I keep all my variables and their types, or simple just how I
am storing so that I can load
  instruction jmp = { 0000 0111, imm16 as target } {
    ip = target;
  };


  instruction jz = { 0000 1000, imm16 as target } {
    sp = sp + 2;                    // Increment stack pointer
    let x = m_stack:2[sp];              // Pop first value into inp (top
value of the stack)

    if x == 0x0 then
      ip = target;
    else
      ip = ip + 3;
  };
```

```
instruction wide_add ={0000 1001} {
    // Increment stack pointer to "pop" the first operand's pointer
    sp = sp + 2;
    let ptr1 = m_stack:2[sp];          // First operand pointer (address
of operand)
    let type1 = ram:2[ptr1];          // Type of first operand (e.g., 2
for number)

    // Increment stack pointer to "pop" the second operand's pointer
    sp = sp + 2;
    let ptr2 = m_stack:2[sp];          // Second operand pointer
    let type2 = ram:2[ptr2];          // Type of second operand

    // Check if both operands are of type "number" (type == 2)
    if (type1 == 2) && (type2 == 2) then {
        let x = ram:2[ptr1 + 4];      // Fetch the value of the first
operand
        let y = ram:2[ptr2 + 4];      // Fetch the value of the second
operand

        let r = x + y;                 // Perform addition

        // Push the result back onto the stack
        m_stack:2[sp] = r;
        sp = sp - 2;                   // Decrement stack pointer
        m_stack:2[sp]= 2; // push the type
        sp= sp -2;
    } else {
        // Push error code (-1) onto the stack if types don't match
        m_stack:2[sp] = -1;
        sp = sp - 2;                   // Decrement stack pointer
        m_stack:2[sp] = -1;
        sp=sp -2; // push type error
    }

    // Increment instruction pointer to the next instruction
    ip = ip + 1;
};
instruction wide_store = { 0000 1010 }
{   sp=sp + 2;
    let ptr = m_stack:2[sp];// store type

    sp=sp + 2;
    let ptr2=m_stack:2[sp];
    ram:2[ptr] = ram:2[ptr2];// store type
    ram:2[ptr+4] = ram:2[ptr2+4];// store value
    ip = ip + 1;
};
instruction wide_sub ={0000 1011} {
    // Increment stack pointer to "pop" the first operand's pointer
    sp = sp + 2;
    let ptr1 = m_stack:2[sp];          // First operand pointer (address
of operand)
    let type1 = ram:2[ptr1];          // Type of first operand (e.g., 2
for number)

    // Increment stack pointer to "pop" the second operand's pointer
```

```
    sp = sp + 2;
    let ptr2 = m_stack:2[sp];          // Second operand pointer
    let type2 = ram:2[ptr2];          // Type of second operand

    // Check if both operands are of type "number" (type == 2)
    if (type1 == 2) && (type2 == 2) then {
        let x = ram:2[ptr1 + 4];      // Fetch the value of the first
operand
        let y = ram:2[ptr2 + 4];      // Fetch the value of the second
operand

        let r = x - y;                 // Perform addition

        // Push the result back onto the stack
        m_stack:2[sp] = r;
        sp = sp - 2;                   // Decrement stack pointer
        m_stack:2[sp]= 2; // push the type
        sp= sp -2;
    } else {
        // Push error code (-1) onto the stack if types don't match
        m_stack:2[sp] = -1;
        sp = sp - 2;                   // Decrement stack pointer
        m_stack:2[sp] = -1;
        sp=sp -2; // push type error
    }

    // Increment instruction pointer to the next instruction
    ip = ip + 1;
};
instruction wide_mult ={0000 1100} {
    // Increment stack pointer to "pop" the first operand's pointer
    sp = sp + 2;
    let ptr1 = m_stack:2[sp];          // First operand pointer (address
of operand)
    let type1 = ram:2[ptr1];          // Type of first operand (e.g., 2
for number)

    // Increment stack pointer to "pop" the second operand's pointer
    sp = sp + 2;
    let ptr2 = m_stack:2[sp];          // Second operand pointer
    let type2 = ram:2[ptr2];          // Type of second operand

    // Check if both operands are of type "number" (type == 2)
    if (type1 == 2) && (type2 == 2) then {
        let x = ram:2[ptr1 + 4];      // Fetch the value of the first
operand
        let y = ram:2[ptr2 + 4];      // Fetch the value of the second
operand

        let r = x * y;                 // Perform addition

        // Push the result back onto the stack
        m_stack:2[sp] = r;
        sp = sp - 2;                   // Decrement stack pointer
        m_stack:2[sp]= 2; // push the type
        sp= sp -2;
    } else {
```

```
        // Push error code (-1) onto the stack if types don't match
        m_stack:2[sp] = -1;
        sp = sp - 2;                   // Decrement stack pointer
        m_stack:2[sp] = -1;
        sp=sp -2; // push type error
    }

    // Increment instruction pointer to the next instruction
    ip = ip + 1;
};
instruction wide_div ={0000 1101} {
    // Increment stack pointer to "pop" the first operand's pointer
    sp = sp + 2;
    let ptr1 = m_stack:2[sp];          // First operand pointer (address
of operand)
    let type1 = ram:2[ptr1];          // Type of first operand (e.g., 2
for number)

    // Increment stack pointer to "pop" the second operand's pointer
    sp = sp + 2;
    let ptr2 = m_stack:2[sp];          // Second operand pointer
    let type2 = ram:2[ptr2];          // Type of second operand

    // Check if both operands are of type "number" (type == 2)
    if (type1 == 2) && (type2 == 2) then {
        let x = ram:2[ptr1 + 4];     // Fetch the value of the first
operand
        let y = ram:2[ptr2 + 4];     // Fetch the value of the second
operand

        let r = x / y;                 // Perform addition

        // Push the result back onto the stack
        m_stack:2[sp] = r;
        sp = sp - 2;                   // Decrement stack pointer
        m_stack:2[sp]= 2; // push the type
        sp= sp -2;
    } else {
        // Push error code (-1) onto the stack if types don't match
        m_stack:2[sp] = -1;
        sp = sp - 2;                   // Decrement stack pointer
        m_stack:2[sp] = -1;
        sp=sp -2; // push type error
    }

    // Increment instruction pointer to the next instruction
    ip = ip + 1;
};
  instruction jgt = { 0000 1111, imm16 as target } {
    sp = sp + 2;                // Increment stack pointer
    let x = m_stack:2[sp];          // Pop first value into inp (top
value of the stack)

    if (x >> 31 == 0x0) && (x != 0x0) then
      ip = target;
    else
      ip = ip + 3;
```

```
  };
    instruction jge = { 0001 0000, imm16 as target } {
    sp = sp + 2;                // Increment stack pointer
    let x = m_stack:2[sp];            // Pop first value into inp (top
value of the stack)

    if (x >> 31 == 0x0)  then
      ip = target;
    else
      ip = ip + 3;
  };

      instruction jlt = { 0001 0001, imm16 as target } {
    sp = sp + 2;                // Increment stack pointer
    let x = m_stack:2[sp];            // Pop first value into inp (top
value of the stack)

    if (x >> 31 == 0x1)  then
      ip = target;
    else
      ip = ip + 3;
  };

        instruction jle = { 0001 0010, imm16 as target } {
    sp = sp + 2;                // Increment stack pointer
    let x = m_stack:2[sp];            // Pop first value into inp (top
value of the stack)

    if (x >> 31 == 0x1) || (x == 0x0) then
      ip = target;
    else
      ip = ip + 3;
  };
  // Halt execution
  instruction hlt = { 1111 1111 } {
    // Execution halts
  };

    instruction write = { 0001 0011 } {
    let x = m_stack:2[sp];            // Pop first value into inp (top
value of the stack)

    outp = ram:2[x + 4]; // Retrieve value from stack to standard
output
    ip = ip + 1;        // Increment instruction pointer
  };
    instruction read = { 0001 0100  } {
    sp = sp + 2 ;
    let ptr= m_stack:2[sp];

    ram:2[ptr+4] = inp;
    ip = ip + 1;   // Increment instruction pointer
  };
    instruction jmp_emp = { 0001 0101, imm16 as ptr } {
    sp = sp + 2;                // Increment stack pointer
    let target = ram:2[ptr];
    ip = target;
```

```
  };
  instruction wide_mod ={0001 0110} {
    // Increment stack pointer to "pop" the first operand's pointer
    sp = sp + 2;
    let ptr1 = m_stack:2[sp];          // First operand pointer (address
of operand)
    let type1 = ram:2[ptr1];          // Type of first operand (e.g., 2
for number)

    // Increment stack pointer to "pop" the second operand's pointer
    sp = sp + 2;
    let ptr2 = m_stack:2[sp];          // Second operand pointer
    let type2 = ram:2[ptr2];          // Type of second operand

    // Check if both operands are of type "number" (type == 2)
    if (type1 == 2) && (type2 == 2) then {
        let x = ram:2[ptr1 + 4];      // Fetch the value of the first
operand
        let y = ram:2[ptr2 + 4];      // Fetch the value of the second
operand

        let r = x % y;                 // Perform addition

        // Push the result back onto the stack
        m_stack:2[sp] = r;
        sp = sp - 2;                   // Decrement stack pointer
        m_stack:2[sp]= 2; // push the type
        sp= sp -2;
    } else {
        // Push error code (-1) onto the stack if types don't match
        m_stack:2[sp] = -1;
        sp = sp - 2;                   // Decrement stack pointer
        m_stack:2[sp] = -1;
        sp=sp -2; // push type error
    }

    // Increment instruction pointer to the next instruction
    ip = ip + 1;
};





mnemonics:

  mnemonic push (value) "{1}";
  mnemonic pop() ;
  mnemonic popOut();
  mnemonic iadd();
  mnemonic ladd() ;
  mnemonic jump for jmp(target) "{1}";
  mnemonic jz(target) "{1}";
  mnemonic jgt(target) "{1}";
  mnemonic jge(target) "{1}";
  mnemonic jlt(target) "{1}";
```

```
  mnemonic jle(target) "{1}";

  mnemonic store for st(ptr) "{1}";
  mnemonic astore for a_st(ptr) "{1}";
  mnemonic wide_add();
  mnemonic wide_store();
  mnemonic wide_sub();
  mnemonic wide_mult();
  mnemonic wide_div();
  mnemonic wide_mod();
  mnemonic read();
  mnemonic write();
  mnemonic jmp_emp(ptr) "{1}";
  mnemonic hlt();
}
```

Architecture.pdsl

And we used our language to create a simple calculator. This calculator expects 2 numbers as its input and an operation + - / *. The calculator uses loops to enter the input from the console one byte by the other. An if statement is used to direct the output to a different result depending on what operation the user want to apply to the numbers. A separate function is created to output the result from ASCII encoding, that is continuously dividing and taking the modulo of the number with 10. The output is then printed in the console. Here we can see this code example according to variant 3.

```
function calculator()
  dim i,  sum, tmp1, tmp2 as int
  tmp1 = 10;
  tmp2= 0;

  scan(i);
  sum =  i - 48;
  scan(i);
  while i != 32
    tmp1= i - 48;
    tmp2 = tmp1 * 10;
    sum = sum + tmp2;
    scan(i);
  wend
  dim i1,  sum1, tmp11, tmp21, res, x as int
  tmp11 = 10;
  tmp21= 0;

  scan(i1);
  sum1 =  i1 - 48;
  scan(i1);
  while i1 != 32
    tmp11= i1 - 48;
    tmp21 = tmp11 * 10;
    sum1 = sum1 + tmp21;
```

```
        scan(i1);
    wend

    scan(x);
    if x ==43 then
        res= sum + sum1;
    else
         if x ==45 then
            res= sum - sum1;
         else
            if x ==47 then
               res = sum / sum1;
            else
               if x ==42 then
                   res = sum * sum1;
               end if

            end if
        end if
    end if

    print_value(res);
end function

function print_value(res)
    dim nextLine, revertedNum, tmp as int
    nextLine = 10;
    revertedNum = 0;
    while res != 0
    revertedNum = revertedNum * 10  ;
    tmp = res % 10;
    tmp=tmp;
    revertedNum= revertedNum + tmp;
    res = res / 10;
    wend
    while revertedNum != 0
    tmp = revertedNum % 10;
    tmp=tmp + 48;
    print(tmp);
    revertedNum = revertedNum / 10;
    wend



end function
```

Input.txt

As a result of executing the translation module we get the following listing.

```
[section ram, code]
        store ret.value
        push 2
        store i_input2
        push 2
        store sum_input2
        push 2
        store tmp1_input2
        push 2
        store tmp2_input2
        push 2
        store i1_input2
        push 2
        store sum1_input2
        push 2
        store tmp11_input2
        push 2
        store tmp21_input2
        push 2
        store res_input2
        push 2
        store x_input2
        push 2
        store nextLine_input2
        push 2
        store revertedNum_input2
        push 2
        store tmp_input2
calculator:
        push label_0
        push tmp1_input2
        wide_store
        push label_1
        push tmp2_input2
        wide_store
        push i_input2
        read
        push label_2
        push i_input2
        wide_sub
        store label_3.type
        store label_3.value
        push label_3
        push sum_input2
        wide_store
        push i_input2
        read
```

```
                    ;while
          label_4:
                push label_5
                push i_input2
                wide_sub
                pop
                jz label_6
                push 1
                    ;false branch
                jump label_7
          label_6:
                push 0
                    ;true branch
          label_7:
                jz label_8
                    ;while body
                push label_9
                push i_input2
                wide_sub
                store label_10.type
                store label_10.value
                push label_10
                push tmp1_input2
                wide_store
                push label_11
                push tmp1_input2
                wide_mult
                store label_12.type
                store label_12.value
                push label_12
                push tmp2_input2
                wide_store
                push tmp2_input2
                push sum_input2
                wide_add
                store label_13.type
                store label_13.value
                push label_13
                push sum_input2
                wide_store
                push i_input2
                read
                jump label_4
          label_8:
                    ;end while
                push label_14
                push tmp11_input2
                wide_store
```

```
            push label_15
            push tmp21_input2
            wide_store
            push i1_input2
            read
            push label_16
            push i1_input2
            wide_sub
            store label_17.type
            store label_17.value
            push label_17
            push sum1_input2
            wide_store
            push i1_input2
            read
                        ;while
label_18:
            push label_19
            push i1_input2
            wide_sub
            pop
            jz label_20
            push 1
                        ;false branch
            jump label_21
label_20:
            push 0
                        ;true branch
label_21:
            jz label_22
                        ;while body
            push label_23
            push i1_input2
            wide_sub
            store label_24.type
            store label_24.value
            push label_24
            push tmp11_input2
            wide_store
            push label_25
            push tmp11_input2
            wide_mult
            store label_26.type
            store label_26.value
            push label_26
            push tmp21_input2
            wide_store
            push tmp21_input2
```

```
        push sum1_input2
        wide_add
        store label_27.type
        store label_27.value
        push label_27
        push sum1_input2
        wide_store
        push i1_input2
        read
        jump label_18
label_22:
                ;end while
        push x_input2
        read
        push label_28
        push x_input2
        wide_sub
        pop
        jz label_29
        push 0
                ;false branch
        jump label_30
label_29:
        push 1
                ;true branch
label_30:
                ;if
        jz label_32
                ;then
        push sum1_input2
        push sum_input2
        wide_add
        store label_33.type
        store label_33.value
        push label_33
        push res_input2
        wide_store
        push res_input2
        push label_34
        store ret.value
        jump print_value
label_34:
        push halt
        store ret.value
        jump label_31
label_32:
                ;else
        push label_35
```

```
            push x_input2
            wide_sub
            pop
            jz label_36
            push 0
                        ;false branch
            jump label_37
label_36:
            push 1
                        ;true branch
label_37:
                        ;if
            jz label_39
                        ;then
            push sum1_input2
            push sum_input2
            wide_sub
            store label_40.type
            store label_40.value
            push label_40
            push res_input2
            wide_store
            push res_input2
            push label_41
            store ret.value
            jump print_value
label_41:
            push halt
            store ret.value
            jump label_38
label_39:
                        ;else
            push label_42
            push x_input2
            wide_sub
            pop
            jz label_43
            push 0
                        ;false branch
            jump label_44
label_43:
            push 1
                        ;true branch
label_44:
                        ;if
            jz label_46
                        ;then
            push sum1_input2
```

```
        push sum_input2
        wide_div
        store label_47.type
        store label_47.value
        push label_47
        push res_input2
        wide_store
        push res_input2
        push label_48
        store ret.value
        jump print_value
label_48:
        push halt
        store ret.value
        jump label_45
label_46:
                ;else
        push label_49
        push x_input2
        wide_sub
        pop
        jz label_50
        push 0
                ;false branch
        jump label_51
label_50:
        push 1
                ;true branch
label_51:
                ;if
        jz label_52
                ;then
        push sum1_input2
        push sum_input2
        wide_mult
        store label_53.type
        store label_53.value
        push label_53
        push res_input2
        wide_store
        push res_input2
        push label_54
        store ret.value
        jump print_value
label_54:
        push halt
        store ret.value
                ;endif
```

```
label_52:
        jump label_45
                ;endif
label_45:
        jump label_38
                ;endif
label_38:
        jump label_31
                ;endif
label_31:
        jmp_emp ret.value
print_value:
        push label_55
        push nextLine_input2
        wide_store
        push label_56
        push revertedNum_input2
        wide_store
                ;while
label_57:
        push label_58
        push res_input2
        wide_sub
        pop
        jz label_59
        push 1
                ;false branch
        jump label_60
label_59:
        push 0
                ;true branch
label_60:
        jz label_61
                ;while body
        push label_62
        push revertedNum_input2
        wide_mult
        store label_63.type
        store label_63.value
        push label_63
        push revertedNum_input2
        wide_store
        push label_64
        push res_input2
        wide_mod
        store label_65.type
        store label_65.value
        push label_65
```

```
		push tmp_input2
		wide_store
		push tmp_input2
		push tmp_input2
		wide_store
		push tmp_input2
		push revertedNum_input2
		wide_add
		store label_66.type
		store label_66.value
		push label_66
		push revertedNum_input2
		wide_store
		push label_67
		push res_input2
		wide_div
		store label_68.type
		store label_68.value
		push label_68
		push res_input2
		wide_store
		jump label_57
	label_61:
			;end while
			;while
	label_69:
		push label_70
		push revertedNum_input2
		wide_sub
		pop
		jz label_71
		push 1
			;false branch
		jump label_72
	label_71:
		push 0
			;true branch
	label_72:
		jz label_73
			;while body
		push label_74
		push revertedNum_input2
		wide_mod
		store label_75.type
		store label_75.value
		push label_75
		push tmp_input2
		wide_store
```

```
        push label_76
        push tmp_input2
        wide_add
        store label_77.type
        store label_77.value
        push label_77
        push tmp_input2
        wide_store
        push tmp_input2
        write
        push label_78
        push revertedNum_input2
        wide_div
        store label_79.type
        store label_79.value
        push label_79
        push revertedNum_input2
        wide_store
        jump label_69
label_73:
                ;end while
        jmp_emp ret.value
        jump halt
ret:
   .type: dd 0x0 ; Offset for `type`
   .value: dd '' ; Offset for `value`
i_input2:
   .type: dd 0x0 ; Offset for `type`
   .value: dd 0x10 ; Offset for `value`
sum_input2:
   .type: dd 0x0 ; Offset for `type`
   .value: dd 0x10 ; Offset for `value`
tmp1_input2:
   .type: dd 0x0 ; Offset for `type`
   .value: dd 0x10 ; Offset for `value`
tmp2_input2:
   .type: dd 0x0 ; Offset for `type`
   .value: dd 0x10 ; Offset for `value`
i1_input2:
   .type: dd 0x0 ; Offset for `type`
   .value: dd 0x10 ; Offset for `value`
sum1_input2:
   .type: dd 0x0 ; Offset for `type`
   .value: dd 0x10 ; Offset for `value`
tmp11_input2:
   .type: dd 0x0 ; Offset for `type`
   .value: dd 0x10 ; Offset for `value`
tmp21_input2:
```

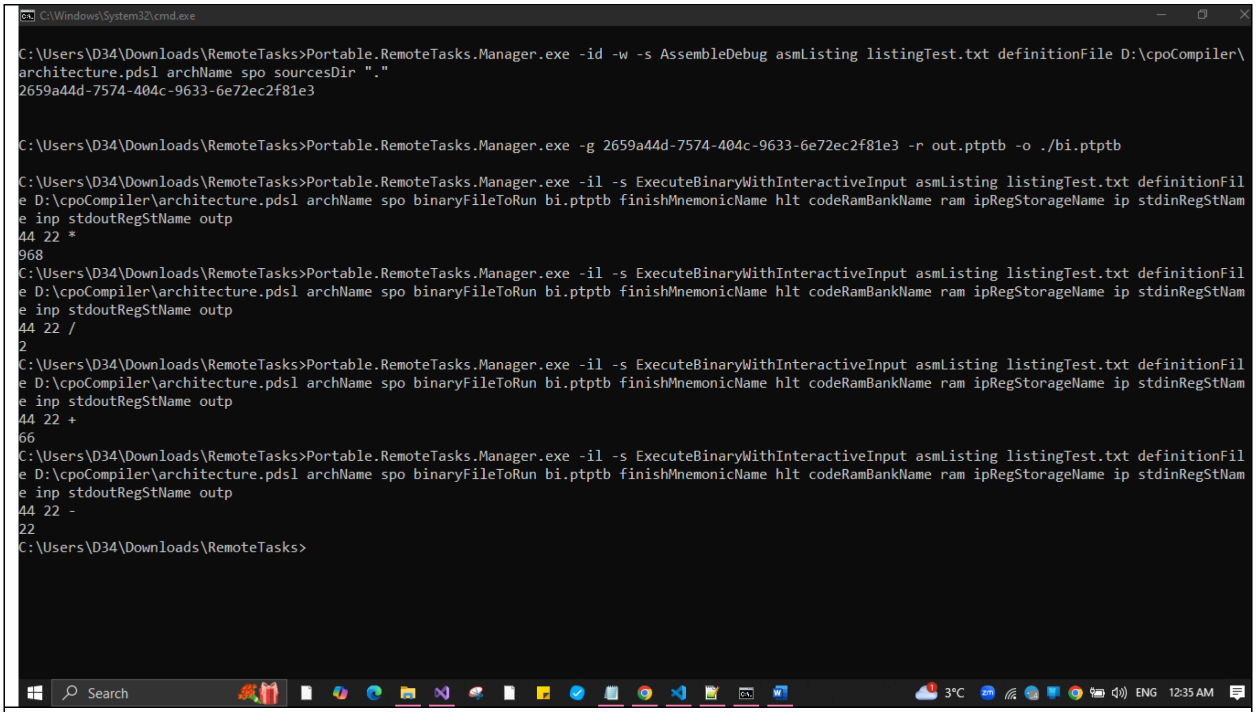
```
    .type: dd 0x0 ; Offset for `type`
    .value: dd 0x10 ; Offset for `value`
res_input2:
    .type: dd 0x0 ; Offset for `type`
    .value: dd 0x10 ; Offset for `value`
x_input2:
    .type: dd 0x0 ; Offset for `type`
    .value: dd 0x10 ; Offset for `value`
nextLine_input2:
    .type: dd 0x0 ; Offset for `type`
    .value: dd 0x10 ; Offset for `value`
revertedNum_input2:
    .type: dd 0x0 ; Offset for `type`
    .value: dd 0x10 ; Offset for `value`
tmp_input2:
    .type: dd 0x0 ; Offset for `type`
    .value: dd 0x10 ; Offset for `value`
halt:
        hlt
label_0:
    .type: dd 0x2 ; Offset for `type`
    .value: dd 10 ; Offset for `value`
label_1:
    .type: dd 0x2 ; Offset for `type`
    .value: dd 0 ; Offset for `value`
label_2:
    .type: dd 0x2 ; Offset for `type`
    .value: dd 48 ; Offset for `value`
label_3:
    .type: dd 0x0 ; Offset for `type`
    .value: dd 0x0 ; Offset for `value`
label_5:
    .type: dd 0x2 ; Offset for `type`
    .value: dd 32 ; Offset for `value`
label_9:
    .type: dd 0x2 ; Offset for `type`
    .value: dd 48 ; Offset for `value`
label_10:
    .type: dd 0x0 ; Offset for `type`
    .value: dd 0x0 ; Offset for `value`
label_11:
    .type: dd 0x2 ; Offset for `type`
    .value: dd 10 ; Offset for `value`
label_12:
    .type: dd 0x0 ; Offset for `type`
    .value: dd 0x0 ; Offset for `value`
label_13:
    .type: dd 0x0 ; Offset for `type`
```

```
    .value: dd 0x0 ; Offset for `value`
label_14:
    .type: dd 0x2 ; Offset for `type`
    .value: dd 10 ; Offset for `value`
label_15:
    .type: dd 0x2 ; Offset for `type`
    .value: dd 0 ; Offset for `value`
label_16:
    .type: dd 0x2 ; Offset for `type`
    .value: dd 48 ; Offset for `value`
label_17:
    .type: dd 0x0 ; Offset for `type`
    .value: dd 0x0 ; Offset for `value`
label_19:
    .type: dd 0x2 ; Offset for `type`
    .value: dd 32 ; Offset for `value`
label_23:
    .type: dd 0x2 ; Offset for `type`
    .value: dd 48 ; Offset for `value`
label_24:
    .type: dd 0x0 ; Offset for `type`
    .value: dd 0x0 ; Offset for `value`
label_25:
    .type: dd 0x2 ; Offset for `type`
    .value: dd 10 ; Offset for `value`
label_26:
    .type: dd 0x0 ; Offset for `type`
    .value: dd 0x0 ; Offset for `value`
label_27:
    .type: dd 0x0 ; Offset for `type`
    .value: dd 0x0 ; Offset for `value`
label_28:
    .type: dd 0x2 ; Offset for `type`
    .value: dd 43 ; Offset for `value`
label_33:
    .type: dd 0x0 ; Offset for `type`
    .value: dd 0x0 ; Offset for `value`
label_35:
    .type: dd 0x2 ; Offset for `type`
    .value: dd 45 ; Offset for `value`
label_40:
    .type: dd 0x0 ; Offset for `type`
    .value: dd 0x0 ; Offset for `value`
label_42:
    .type: dd 0x2 ; Offset for `type`
    .value: dd 47 ; Offset for `value`
label_47:
    .type: dd 0x0 ; Offset for `type`
```

```
    .value: dd 0x0 ; Offset for `value`
label_49:
    .type: dd 0x2 ; Offset for `type`
    .value: dd 42 ; Offset for `value`
label_53:
    .type: dd 0x0 ; Offset for `type`
    .value: dd 0x0 ; Offset for `value`
label_55:
    .type: dd 0x2 ; Offset for `type`
    .value: dd 10 ; Offset for `value`
label_56:
    .type: dd 0x2 ; Offset for `type`
    .value: dd 0 ; Offset for `value`
label_58:
    .type: dd 0x2 ; Offset for `type`
    .value: dd 0 ; Offset for `value`
label_62:
    .type: dd 0x2 ; Offset for `type`
    .value: dd 10 ; Offset for `value`
label_63:
    .type: dd 0x0 ; Offset for `type`
    .value: dd 0x0 ; Offset for `value`
label_64:
    .type: dd 0x2 ; Offset for `type`
    .value: dd 10 ; Offset for `value`
label_65:
    .type: dd 0x0 ; Offset for `type`
    .value: dd 0x0 ; Offset for `value`
label_66:
    .type: dd 0x0 ; Offset for `type`
    .value: dd 0x0 ; Offset for `value`
label_67:
    .type: dd 0x2 ; Offset for `type`
    .value: dd 10 ; Offset for `value`
label_68:
    .type: dd 0x0 ; Offset for `type`
    .value: dd 0x0 ; Offset for `value`
label_70:
    .type: dd 0x2 ; Offset for `type`
    .value: dd 0 ; Offset for `value`
label_74:
    .type: dd 0x2 ; Offset for `type`
    .value: dd 10 ; Offset for `value`
label_75:
    .type: dd 0x0 ; Offset for `type`
    .value: dd 0x0 ; Offset for `value`
label_76:
    .type: dd 0x2 ; Offset for `type`
```

.value: dd 48 ; Offset for `value` label_77: .type: dd 0x0 ; Offset for `type` .value: dd 0x0 ; Offset for `value` label_78: .type: dd 0x2 ; Offset for `type` .value: dd 10 ; Offset for `value` label_79: .type: dd 0x0 ; Offset for `type` .value: dd 0x0 ; Offset for `value`
listingTest.txt

And as a result of assembling and executing this listing according to our architecture we get the following results



Executing Calculator function with different operations with input and output in console

6. Conclusions

I believe the results of this lab work align with the goals. The architecture description of the instruction set of our virtual machine was created successfully in alignment with our variant to include: arithmetic functions, branching, data movement and IO handeling. The module responsible for translating the control flow graphs' information to linear code was created successfully. And the result of this work was executed successfully to solve a meaningful task.

During the process of completing this lab assignment I learned what a virtual machine instruction set is, how the instructions are performed at a low level, what stages the compiler takes to reach an executable file, how to translate a code into a linear code, what dynamic and static typing is and learned a lot about real life architectures in the process of learning how to implement this task.

Санкт-Петербург, 2024 г.

7. Link to the project on gitlab

https://gitlab.se.ifmo.ru/Aveena/cpoCompiler

Санкт-Петербург, 2024 г.