ПРАВИТЕЛЬСТВО РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

Факультет компьютерных наук
Образовательная программа «Программная инженерия»

Подп. и дата	
Инв. № дубл.	
Взам. инв. №	
Подп. и дата	
Инв. № подл	

СОГЛАСОВАНО
Доцент базовой кафедры
«Системное программирование»
Института системного программирования
им В.П. Иванникова РАН (ИСП РАН)
факультета компьютерных наук,
кандидат физико-математических наук

_____ А. С. Камкин
«01» апреля 2023 г.

УТВЕРЖДАЮ
Академический руководитель
образовательной программы факультета
компьютерных наук
«Программная инженерия»
профессор департамента программной
инженерии, кандидат технических наук

_____ В. В. Шилов
«7» апреля 2023 г.

ПРОТОТИП ОНЛАЙН-ГЕНЕРАТОРА ТЕСТОВЫХ ПРОГРАММ ДЛЯ МИКРОПРОЦЕССОРОВ

Текст программы

ЛИСТ УТВЕРЖДЕНИЯ

RU.17701729.04.04-01 12 01-1-ЛУ

Исполнитель
студент группы БПИ201
_____ / М. Ю. Литвинов /
«01» апреля 2023 г.

Москва, 2023

УТВЕРЖДЕН

RU.17701729.04.04-01 12 01-1-ЛУ

Подп. и дата	
Инв. № дубл.	
Взам. инв. №	
Подп. и дата	
Инв. № подл	

ПРОТОТИП ОНЛАЙН-ГЕНЕРАТОРА ТЕСТОВЫХ ПРОГРАММ ДЛЯ МИКРОПРОЦЕССОРОВ

Текст программы

RU.17701729.04.04-01 12 01-1-ЛУ

Листов 61

Москва, 2023

СОДЕРЖАНИЕ

Москва, 2023

Москва, 2023

1. ВВЕДЕНИЕ

Наименование программы

Прототип онлайн-генератора тестовых программ для микропроцессоров.

Наименование программы на английском языке

Prototype of Online Test Program Generator for Microprocessors.

Расположение исходных кодов программы

Исходные коды программы расположены в удаленном репозитории ИСП РАН:

https://forge.ispras.ru/projects/otpg

https://forge.ispras.ru/git/otpg.git

2. ИСХОДНЫЕ КОДЫ УРОВНЯ ЯДРА ПРОГРАММЫ

Содержание файла «main.c»

```
#ifndef GEN_MAIN_C
#define GEN_MAIN_C

#include "decls/ins_decls.h"
#include "decls/tests_decls.h"
#include "execution/runner.h"

#ifdef GEN_USE_STDIO
#include "stdio.h"

// In order to avoid warnings, backslashes between parts of a format string are
dropped.
//void print_configuration() {
//    printf("Generator ver. %s starting...\n"
//           "Configured with ->\n"
//           "Compiler path: %s\n"
//           "Compiler params: %s\n"
//           //"Hardware architecture: %d bit\n"
//           "ISA directory path: %s\n"
//           "Tests directory path: %s\n"
//           "Mutation functionality directory path: %s\n", GEN_VERSION,
GEN_CC_PATH, GEN_CC_PARAMS, /*GEN_WORDSIZE,*/ GEN_ISA_DIR, GEN_TESTS_DIR,
GEN_MUT_DIR);
//
//}


#endif // GEN_USE_STDIO


int main(int argc, char **argv) {
    #ifdef GEN_USE_STDIO
    printf("Generator ver. %s starting...\n"
           "Configured with ->\n"
           "Hardware architecture: %d bit\n", GEN_VERSION, GEN_WORDSIZE);
    #endif // GEN_USE_STDIO

    // ISA-related initializations.
    init_reg_dumps();
    init_groups();
    init_instructions();
    // Tests-related initializations.
    init_stacks();
    init_tree();
```

```
    init_test_patterns();
    init_asm_tests();
    init_misc();
    // Running tests of both categories.
    run_tests();
    #ifdef GEN_USE_STDIO
    printf("Generator stopped.\n");
    #endif // GEN_USE_STDIO
    return 0;
}

#endif // GEN_MAIN_C
```

Содержание файла «common/types.h»

```
#ifndef GEN_TYPES_H
#define GEN_TYPES_H

#define NULL ((void *)0)

typedef char int8_t;
typedef unsigned char uint8_t;
typedef uint8_t flags_t;
typedef uint8_t byte_t;
typedef uint8_t bool_t;
#define true 1
#define false 0

typedef short int16_t;
typedef unsigned short uint16_t;

typedef int int32_t;
typedef unsigned int uint32_t;
typedef unsigned int size_t;

struct InstructionGroup;
struct Instruction;
struct PseudoInstruction;
struct ParametrizedInstruction;

#if GEN_WORDSIZE == 32

#define HEX_WORDSIZE "8"
typedef uint32_t ins_t;
typedef uint32_t reg_t;

#else

#define HEX_WORDSIZE "16"
```

```
typedef long long int64_t;

typedef unsigned long long uint64_t;
typedef uint64_t ins_t;
typedef uint64_t reg_t

#endif // GEN_WORDSIZE == 32

#endif // GEN_TYPES_H
```

Содержание файла «common/utility.h»

```
#ifndef GEN_UTILITY_H
#define GEN_UTILITY_H

#include "../common/types.h"

extern ins_t fill(ins_t instruction, uint8_t pos, uint8_t len, ins_t value);
extern ins_t fill_sh(ins_t instruction, uint8_t pos, uint8_t len, uint8_t sh,
ins_t value);

#endif // GEN_UTILITY_H
```

Содержание файла «common/utility.c»

```
#ifndef GEN_UTILITY_C
#define GEN_UTILITY_C

#include "utility.h"

ins_t fill(ins_t instruction, uint8_t pos, uint8_t len, ins_t value) {
    return fill_sh(instruction, pos, len, 0, value);
}

ins_t fill_sh(ins_t instruction, uint8_t pos, uint8_t len, uint8_t sh, ins_t
value) {
    ins_t mask = 1 << (len - 1);
    mask |= (mask - 1);
    return instruction | ((value & mask) << (len * sh + pos));
}

#endif // GEN_UTILITY_C
```

Содержание файла «decls/ins_decls.h»

```
#ifndef GEN_INS_DECLS_H
#define GEN_INS_DECLS_H

#include "../common/types.h"
#include "predef_ins_macros.h"
#include "../common/utility.h"
```

```
#define FIRST_REGS_NAME "GEN_FIRST_REG_DUMP"
#define SECOND_REGS_NAME "GEN_SECOND_REG_DUMP"
#define BUF_REGS_NAME "GEN_BUF_REG_DUMP"

#ifndef GEN_INSTRUCTION_C

extern struct InstructionGroup *GEN_INS_GROUPS;
extern struct Instruction *GEN_INS;
extern struct PseudoInstruction *GEN_PSEUDO_INS;
extern reg_t *GEN_FIRST_REG_DUMP;
extern reg_t *GEN_SECOND_REG_DUMP;
extern reg_t *GEN_BUF_REG_DUMP;
extern bool_t *GEN_REG_INCLUDE_IN_DUMP;

#else

struct InstructionGroup *GEN_INS_GROUPS = NULL;
struct Instruction *GEN_INS = NULL;
struct PseudoInstruction *GEN_PSEUDO_INS = NULL;
reg_t *GEN_FIRST_REG_DUMP = NULL;
reg_t *GEN_SECOND_REG_DUMP = NULL;
reg_t *GEN_BUF_REG_DUMP = NULL;
bool_t *GEN_REG_INCLUDE_IN_DUMP = NULL;

#endif

extern uint8_t get_ins_len(struct InstructionGroup *groups, struct Instruction
*instructions, struct PseudoInstruction *pseudos, struct ParametrizedInstruction
*instruction);
extern ins_t encode_ins(struct ParametrizedInstruction *instruction);
extern void include_regs(struct ParametrizedInstruction *instruction, bool_t
*to_include);
extern void init_reg_dumps();
extern void init_groups();
extern void init_instructions();
#endif // GEN_INS_DECLS_H
```

Содержание файла «decls/mut_decls.h»

```
#ifndef GEN_MUT_DECLS_H
#define GEN_MUT_DECLS_H

#include "../common/types.h"
#include "instruction.h"
#include "../random/rand.h"
```

```
extern uint16_t mutate(struct ParametrizedInstruction *source, uint16_t
ins_count, uint16_t add_ins_count, struct ParametrizedInstruction *destination);

#endif // GEN_MUT_DECLS_H
```

Содержание файла «decls/predef_ins_macros.h»

```
#ifndef GEN_PREDEF_INS_MACROS_H
#define GEN_PREDEF_INS_MACROS_H

#ifndef IS_USED
#define IS_USED (1)
#endif

#ifndef IS_REGISTER
#define IS_REGISTER (1 << 1)
#endif

#ifndef IS_IMMEDIATE
#define IS_IMMEDIATE (1 << 2)
#endif

#ifndef IS_MEMORY_ADDR
#define IS_MEMORY_ADDR (1 << 3)
#endif

#ifndef IS_NOT_ZERO
#define IS_NOT_ZERO (1 << 4)
#endif


#ifndef IS_FIXED
#define IS_FIXED (1)
#endif

#endif // GEN_PREDEF_INS_MACROS_H
```

Содержание файла «decls/tests_decls.h»

```
#ifndef GEN_TESTS_DECLS_H
#define GEN_TESTS_DECLS_H

#include "../template_block/template_block.h"
#include "../random/rand.h"
#include "instruction.h"

#ifndef GEN_TESTS_C

extern void (**GEN_TEST_TEMPLATES)(struct TemplateBlock *tree, uint8_t
```

```
max_tree_depth, struct ParametrizedInstruction *stack, uint16_t max_ins_count);
extern uint8_t *GEN_TEST_TEMPLATES_ITERS;
extern void (**GEN_TEST_TEMPLATES_FILL_INS)(struct ParametrizedInstruction
*instruction);
extern uint16_t (**GEN_ASM_TESTS)(struct ParametrizedInstruction *ins, uint16_t
max_ins_count);
extern bool_t (**GEN_ASM_TESTS_DUMP_EQ)(ins_t *first_dump, ins_t *second_dump,
uint8_t reg_count);
extern bool_t *GEN_USED_ELEMENTS;
extern byte_t *GEN_OUTPUT;

extern struct ParametrizedInstruction *GEN_TEST_INSTRUCTIONS;
extern struct ParametrizedInstruction *GEN_BLOCK_INSTRUCTIONS_STACK;
extern struct ParametrizedInstruction *GEN_INSTRUCTIONS_BUF;
extern uint8_t *GEN_BLOCK_ORDER_STACK;
extern struct TemplateBlock *GEN_TEMPLATE_TREE;

#else

void (**GEN_TEST_TEMPLATES)(struct TemplateBlock *tree, uint8_t max_tree_depth,
struct ParametrizedInstruction *stack, uint16_t max_ins_count) = NULL;
uint8_t *GEN_TEST_TEMPLATES_ITERS = NULL;
void (**GEN_TEST_TEMPLATES_FILL_INS)(struct ParametrizedInstruction *instruction)
= NULL;
uint16_t (**GEN_ASM_TESTS)(struct ParametrizedInstruction *ins, uint16_t
max_ins_count) = NULL;
bool_t (**GEN_ASM_TESTS_DUMP_EQ)(ins_t *first_dump, ins_t *second_dump, uint8_t
reg_count) = NULL;
bool_t *GEN_USED_ELEMENTS = NULL;
byte_t *GEN_OUTPUT = NULL;

struct ParametrizedInstruction *GEN_TEST_INSTRUCTIONS = NULL;
struct ParametrizedInstruction *GEN_BLOCK_INSTRUCTIONS_STACK = NULL;
struct ParametrizedInstruction *GEN_INSTRUCTIONS_BUF = NULL;
uint8_t *GEN_BLOCK_ORDER_STACK = NULL;
struct TemplateBlock *GEN_TEMPLATE_TREE = NULL;

#endif

extern void init_stacks();
extern void init_tree();
extern void init_test_patterns();
extern void init_asm_tests();
extern void init_misc();
extern void default_fill_ins(struct ParametrizedInstruction *instruction);

#endif // GEN_TESTS_DECLS_H
```

Содержание файла «execution/runner.h»

```
#ifndef GEN_RUNNER_H
#define GEN_RUNNER_H


extern void run_tests();

#endif // GEN_RUNNER_H
```

Содержание файла «execution/runner.c»

```
#ifndef GEN_RUNNER_C
#define GEN_RUNNER_C

#include "runner.h"
#include "instruction.h"
#include "tests.h"
#include "../decls/mut_decls.h"
#include "../random/rand.h"
#include "../memory/memutils.h"
#include "../generation/generator.h"

#ifdef GEN_USE_STDIO
#include "stdio.h"

#define REGS_IN_ROW 3

void dump_core(ins_t *correct_reg_dump, ins_t *wrong_reg_dump, bool_t
*to_include, uint8_t reg_count) {
    printf("Core dumped:\n");
    uint8_t rows = reg_count / REGS_IN_ROW;
    rows += (reg_count - rows * REGS_IN_ROW);
    uint8_t element_ind = 0;
    for (uint8_t row = 0; row < rows; ++row) {
        for (; element_ind < reg_count && element_ind < (row + 1) * REGS_IN_ROW;
++element_ind) {
            printf(("x%u%s: 0x%0"HEX_WORDSIZE"x(0x%0"HEX_WORDSIZE"x)\t"),
element_ind, (to_include && to_include[element_ind]) ? "(!)" : "",
wrong_reg_dump[element_ind], correct_reg_dump[element_ind]);
        }
        printf("\n");
    }
}

#endif // GEN_USE_STDIO

void core_include_regs(struct ParametrizedInstruction *instructions, uint16_t
ins_count, bool_t *to_include, uint8_t reg_count) {
```

```
    fill_arr_u8t(to_include, reg_count, false);
    struct ParametrizedInstruction *ins;
    struct PseudoInstruction *p_ins;
    uint8_t group_id;
    while(ins_count--) {
        include_regs(instructions + ins_count, to_include);
        ins = instructions + ins_count;
        if (ins->pseudo_ins) {
            p_ins = GEN_PSEUDO_INS + ins->instruction_id;
            group_id = GEN_INS[p_ins->instruction_id].group_id;
        } else {
            p_ins = NULL;
            group_id = GEN_INS[ins->instruction_id].group_id;
        }
        if (GEN_INS_GROUPS[group_id].output == IS_REGISTER) {
            to_include[(p_ins && p_ins->output_fixated) ? p_ins->output : ins-
>output] = true;
        }
    }
}

bool_t reg_dump_eq(ins_t *first_dump, ins_t *second_dump, bool_t *to_include,
uint8_t reg_count) {
    while (reg_count--) {
        if (to_include[reg_count] && first_dump[reg_count] !=
second_dump[reg_count]) {
            return false;
        }
    }
    return true;
}


 void encode_ins_dump(struct ParametrizedInstruction *src, uint16_t ins_count,
ins_t *dest) {
    void *dest_ptr = dest;
    uint8_t ins_len;
    ins_t ins_val;
    for (uint16_t ins_id = 0; ins_id < ins_count; ++ins_id) {
        ins_len = get_ins_len(GEN_INS_GROUPS, GEN_INS, GEN_PSEUDO_INS, src +
ins_id);
        ins_val = encode_ins(&src[ins_id]);
        memmove(dest_ptr, &ins_val, ins_len);
        dest_ptr += ins_len;
    }
}
```

```
uint16_t run_pattern_tests(ins_t *first_ins_dump, ins_t *second_ins_dump, byte_t
*output) {
#ifdef GEN_USE_STDIO
    printf("//////////Running template tests...\n");
#endif // GEN_USE_STDIO
    struct GeneratorContext context;
    uint16_t result_bytes_count = 0;
    for (uint8_t test_index = 0; test_index < GEN_TEST_TEMPLATES_COUNT;
++test_index) {
        GEN_TEST_TEMPLATES[test_index](GEN_TEMPLATE_TREE,
GEN_MAX_TREE_BLOCK_COUNT, GEN_TEST_INSTRUCTIONS, GEN_MAX_INS_COUNT);
        context = (struct GeneratorContext) {    .tree = GEN_TEMPLATE_TREE,
                                                 .max_tree_depth =
GEN_MAX_TREE_BLOCK_COUNT,
                                                 .test_data =
GEN_TEST_INSTRUCTIONS,
                                                 .order_stack =
GEN_BLOCK_ORDER_STACK,
                                                 .stack =
GEN_BLOCK_INSTRUCTIONS_STACK,
                                                 .buf = GEN_INSTRUCTIONS_BUF,
                                                 .used_elements =
GEN_USED_ELEMENTS,
                                                 .max_ins_count =
GEN_MAX_INS_COUNT};
        for (uint8_t iter_index = 0; iter_index <
GEN_TEST_TEMPLATES_ITERS[test_index]; ++iter_index) {
            uint16_t gen_len = generate_and_fill(&context,
(GEN_TEST_TEMPLATES_FILL_INS[test_index]) ?
GEN_TEST_TEMPLATES_FILL_INS[test_index] : default_fill_ins);

            encode_ins_dump(GEN_BLOCK_INSTRUCTIONS_STACK, gen_len,
first_ins_dump);
            STORE_REG_DUMP(BUF_REGS_NAME)
            // Execute a generated test-case binary instruction sequence.
            ((void (*)())first_ins_dump)();
            STORE_REG_DUMP(FIRST_REGS_NAME)
            LOAD_REG_DUMP(BUF_REGS_NAME)

#ifdef GEN_MAX_MUT_INS_COUNT
            uint16_t res_len = mutate(GEN_BLOCK_INSTRUCTIONS_STACK, gen_len,
linrand(GEN_MAX_MUT_INS_COUNT) + 1, GEN_INSTRUCTIONS_BUF);
#else
            uint16_t res_len = mutate(GEN_BLOCK_INSTRUCTIONS_STACK, gen_len,
linrand(GEN_MAX_INS_COUNT - res_len) + 1, GEN_INSTRUCTIONS_BUF);
#endif // GEN_MAX_MUT_INS_COUNT

            encode_ins_dump(GEN_INSTRUCTIONS_BUF, res_len, second_ins_dump);
```

```
            STORE_REG_DUMP(BUF_REGS_NAME)
            // Execute a mutated test-case binary instruction sequence.
            ((void (*)())second_ins_dump)();
            STORE_REG_DUMP(SECOND_REGS_NAME)
            LOAD_REG_DUMP(BUF_REGS_NAME)

            core_include_regs(GEN_BLOCK_INSTRUCTIONS_STACK, gen_len,
GEN_REG_INCLUDE_IN_DUMP, GEN_REG_COUNT);
            bool_t dump_eq_result = reg_dump_eq(GEN_FIRST_REG_DUMP,
GEN_SECOND_REG_DUMP, GEN_REG_INCLUDE_IN_DUMP, GEN_REG_COUNT);

            if (dump_eq_result) {
#ifdef GEN_USE_STDIO
                printf("Test #%u, iter. #%u: OK\n", test_index, iter_index); //
Log results.
#endif // GEN_USE_STDIO
            } else {
#ifdef GEN_USE_STDIO
                printf("Test #%u, iter. #%u: Error\n", test_index, iter_index);
// Log results.
                dump_core(GEN_FIRST_REG_DUMP, GEN_SECOND_REG_DUMP,
GEN_REG_INCLUDE_IN_DUMP, GEN_REG_COUNT);
#endif // GEN_USE_STDIO
                output[result_bytes_count++] = 1;
                output[result_bytes_count++] = test_index;
                output[result_bytes_count++] = iter_index;
                break;
            }
        }
        if (result_bytes_count) {
            break;
        }
    }
#ifdef GEN_USE_STDIO
    printf("//////////Finished running template tests.\n");
#endif // GEN_USE_STDIO
    return result_bytes_count;
}

uint16_t run_asm_tests(ins_t *first_ins_dump, ins_t *second_ins_dump, byte_t
*output) {
#ifdef GEN_USE_STDIO
    printf("//////////Running ASM-tests...\n");
#endif // GEN_USE_STDIO
    uint16_t result_bytes_count = 0;
    for (uint8_t test_index = 0; test_index < GEN_ASM_TESTS_COUNT; ++test_index)
{
        uint16_t test_len = GEN_ASM_TESTS[test_index](GEN_TEST_INSTRUCTIONS,
```

```
GEN_MAX_INS_COUNT);

        encode_ins_dump(GEN_TEST_INSTRUCTIONS, test_len, first_ins_dump);
        STORE_REG_DUMP(BUF_REGS_NAME)
        // Execute a generated test-case binary instruction sequence.
        ((void (*)())first_ins_dump)();
        STORE_REG_DUMP(FIRST_REGS_NAME)
        LOAD_REG_DUMP(BUF_REGS_NAME)

#ifdef GEN_MAX_MUT_INS_COUNT
        uint16_t res_len = mutate(GEN_TEST_INSTRUCTIONS, test_len,
linrand(GEN_MAX_MUT_INS_COUNT) + 1, GEN_INSTRUCTIONS_BUF);
#else
        uint16_t res_len = mutate(GEN_TEST_INSTRUCTIONS, test_len,
linrand(GEN_MAX_INS_COUNT - res_len) + 1, GEN_INSTRUCTIONS_BUF);
#endif // GEN_MAX_MUT_INS_COUNT

        encode_ins_dump(GEN_INSTRUCTIONS_BUF, res_len, second_ins_dump);
        STORE_REG_DUMP(BUF_REGS_NAME)
        // Execute a mutated test-case binary instruction sequence.
        ((void (*)())second_ins_dump)();
        STORE_REG_DUMP(SECOND_REGS_NAME)
        LOAD_REG_DUMP(BUF_REGS_NAME)

        // Check if there exists a specific register dumps equality checker or if
the default one is required.
        bool_t dump_eq_result;
        if (GEN_ASM_TESTS_DUMP_EQ[test_index]) {
            dump_eq_result =
GEN_ASM_TESTS_DUMP_EQ[test_index](GEN_FIRST_REG_DUMP, GEN_SECOND_REG_DUMP,
GEN_REG_COUNT);
        } else {
            core_include_regs(GEN_TEST_INSTRUCTIONS, test_len,
GEN_REG_INCLUDE_IN_DUMP, GEN_REG_COUNT);
            dump_eq_result = reg_dump_eq(GEN_FIRST_REG_DUMP, GEN_SECOND_REG_DUMP,
GEN_REG_INCLUDE_IN_DUMP, GEN_REG_COUNT);
        }

        if (dump_eq_result) {
#ifdef GEN_USE_STDIO
            printf("Test #%u: OK\n", test_index); // Log results.
#endif // GEN_USE_STDIO
        } else {
#ifdef GEN_USE_STDIO
            printf("Test #%u: Error\n", test_index); // Log results.
            dump_core(GEN_FIRST_REG_DUMP, GEN_SECOND_REG_DUMP,
(GEN_ASM_TESTS_DUMP_EQ[test_index]) ? NULL : GEN_REG_INCLUDE_IN_DUMP,
GEN_REG_COUNT);
```

```
#endif // GEN_USE_STDIO
            output[result_bytes_count++] = 0;
            output[result_bytes_count++] = test_index;
            break;
        }
    }
#ifdef GEN_USE_STDIO
    printf("//////////Finished running ASM-tests.\n");
#endif // GEN_USE_STDIO
    return result_bytes_count;
}

void run_tests() {
    ins_t first_ins_dump[GEN_MAX_INS_COUNT] = {0};
    ins_t second_ins_dump[GEN_MAX_INS_COUNT] = {0};
    uint16_t failures_count = run_pattern_tests(first_ins_dump, second_ins_dump,
GEN_OUTPUT + sizeof(uint16_t)) / 3;
    failures_count += run_asm_tests(first_ins_dump, second_ins_dump, GEN_OUTPUT +
sizeof(uint16_t) + failures_count * 3) / 2;
    *((uint16_t *)GEN_OUTPUT) = failures_count;
#ifdef GEN_USE_STDIO
    printf("Failures: %u.\n",  *((uint16_t *)GEN_OUTPUT));
    if (failures_count) {
        byte_t *verdict = GEN_OUTPUT + sizeof(uint16_t);
        for (uint16_t fail_id = 0; fail_id < failures_count; ++fail_id) {
            if (*verdict) {
                printf("Failure #%u: template #%u, iter. #%u.\n", fail_id,
*(verdict + 1), *(verdict + 2));
                verdict += 3;
            } else {
                printf("Failure #%u: ASM-test #%u.\n", fail_id, *(verdict + 1));
                verdict += 2;
            }
        }
    }
#endif // GEN_USE_STDIO
}

#endif // GEN_RUNNER_C
```

Содержание файла «generation/generator_context.h»

```
#ifndef GEN_GENERATOR_CONTEXT_H
#define GEN_GENERATOR_CONTEXT_H

#include "../common/types.h"
#include "../template_block/template_block.h"

struct GeneratorContext {
```

```
    struct TemplateBlock *tree;
    uint8_t max_tree_depth;
    struct ParametrizedInstruction *test_data;
    uint8_t *order_stack;
    struct ParametrizedInstruction *stack;
    struct ParametrizedInstruction *buf;
    bool_t *used_elements;
    uint16_t max_ins_count;
};

#endif // GEN_GENERATOR_CONTEXT_H
```

Содержание файла «generation/generator.h»

```
#ifndef GEN_GENERATOR_H
#define GEN_GENERATOR_H

#include "../common/types.h"
#include "generator_context.h"

extern uint16_t generate(struct GeneratorContext *context);
extern uint16_t generate_and_fill(struct GeneratorContext *context, void
(*func)(struct ParametrizedInstruction *));
#endif // GEN_GENERATOR_H
```

Содержание файла «generation/generator.c»

```
#ifndef GEN_GENERATOR_C
#define GEN_GENERATOR_C

#include "generator.h"
#include "instruction.h"
#include "../template_block/template_block.h"
#include "../template_block/block_process_result.h"
#include "../template_block/template_block_type.h"
#include "../memory/ins_memutils.h"


// Function must be invisible to other source files.
uint8_t find_parent_id(uint8_t *order_stack, uint8_t start_ind, uint8_t value) {
    while(start_ind && order_stack[start_ind] != value) {
        --start_ind;
    }
    return start_ind;
}

// Function must be invisible to other source files.
void fill_all_instructions(struct ParametrizedInstruction *instructions, uint16_t
ins_count, void (*func)(struct ParametrizedInstruction *)) {
    --ins_count;
```

```
    while (ins_count--) {
        func(instructions + ins_count);
    }
}

uint16_t generate(struct GeneratorContext *context) {
    context->order_stack[0] = 0;
    uint16_t order_stack_len = 1;
    uint16_t stack_len = 0;
    uint8_t curr_order_stack_ind = 0;
    struct TemplateBlock *curr;
    struct BlockProcessResult result;
    do {
        // Focus on the currently viewed block.
        curr = context->tree + context->order_stack[curr_order_stack_ind];

        // Checking if a block has no inner ("child") blocks and is viewed for
the first time.
        if (curr->child_id != curr->id && curr_order_stack_ind == order_stack_len
- 1) {
            // View the first inner block next.
            context->order_stack[(curr_order_stack_ind = (order_stack_len++))] =
curr->child_id;
        } else {
            // Upload instructions from the storage if the block is a "leaf" (has
no explicit inner blocks).
            if (curr->child_id == curr->id) {
                curr->stack_start_ptr = context->stack + stack_len;
                curr->stack_lines_count = curr->data_lines_count;
                ins_memmove(curr->stack_start_ptr, curr->data_start_ptr, curr-
>data_lines_count);
                stack_len += curr->data_lines_count;
            }

            // Process the block (with viewed inner blocks if there were any).
            result = process_block(context, curr_order_stack_ind,
order_stack_len, stack_len);
            // Remove processed inner blocks from stack.
            stack_len -= result.inner_ins_count;
            order_stack_len = curr_order_stack_ind + 1;


            // Put generated instructions on stack.
            curr->stack_start_ptr = context->stack + stack_len;
            curr->stack_lines_count = result.generated_ins_count;
            ins_memmove(curr->stack_start_ptr, context->buf, curr-
>stack_lines_count);
            stack_len += curr->stack_lines_count;
```

```
            // If there are more blocks to be processed at the same level.
            if (curr->sibling_id != curr->id) {
                context->order_stack[(curr_order_stack_ind =
(order_stack_len++))] = curr->sibling_id;
            } else {
                curr_order_stack_ind = find_parent_id(context->order_stack,
curr_order_stack_ind, curr->parent_id);
            }
        }
    } while (order_stack_len > 1);
    return stack_len;
}

uint16_t generate_and_fill(struct GeneratorContext *context, void (*func)(struct
ParametrizedInstruction *)) {
    uint16_t ins_count = generate(context);
    fill_all_instructions(context->stack, ins_count, func);
    return ins_count;
}

#endif // GEN_GENERATOR_C
```

Содержание файла «memory/memutils.h»

```
#ifndef GEN_MEMUTILS_H
#define GEN_MEMUTILS_H

#include "../common/types.h"

extern void *memcpy(void *dest, const void *src, unsigned long len);
extern void *memmove(void *dest, const void *src, unsigned long len);
extern void *memset(void *dest, int val, unsigned long len);
extern int memcmp(const void *first, const void *second, unsigned long len);

extern void fill_arr_u8t(uint8_t *array, uint16_t len, uint8_t value);
extern void fill_arr_u16t(uint16_t *array, uint16_t len, uint16_t value);

#endif // GEN_MEMUTILS_H
```

Содержание файла «memory/memutils.c»

```
#ifndef GEN_MEMUTILS_C
#define GEN_MEMUTILS_C

#include "memutils.h"

void *memcpy(void *dest, const void *src, unsigned long len) {
    byte_t *dest_v = dest;
    const byte_t *src_v = src;
```

```
    while (len--) {
        *(dest_v++) = *(src_v++);
    }
    return dest;
}

void *memmove(void *dest, const void *src, unsigned long len) {
    if (dest < src) {
        byte_t *dest_v = dest;
        const byte_t *src_v = src;
        while (len--) {
            *(dest_v++) = *(src_v++);
        }
    } else {
        byte_t *dest_v = dest + (len - 1);
        const byte_t *src_v = src + (len - 1);
        while (len--) {
            *(dest_v--) = *(src_v--);
        }
    }
    return dest;
}

void *memset(void *dest, int val, unsigned long len) {
    byte_t *dest_v = dest;
    while (len--) {
        *(dest_v++) = val & 0xFF; // A mask is applied to extract exactly one
byte.
    }
    return dest;
}

int memcmp(const void *first, const void *second, unsigned long len) {
    const byte_t *first_v = first;
    const byte_t *second_v = second;
    while (len--) {
        if (*(first_v++) != *(second_v++)) {
            return (first_v[-1] < second_v[-1]) ? -1 : 1;
        }
    }
    return 0;
}

void fill_arr_u8t(uint8_t *array, uint16_t len, uint8_t value) {
    while (len--) {
        array[len] = value;
    }
}
```

```
void fill_arr_u16t(uint16_t *array, uint16_t len, uint16_t value) {
    while (len--) {
        array[len] = value;
    }
}

#endif // GEN_MEMUTILS_C
```

Содержание файла «memory/ins_memutils.h»

```
#ifndef GEN_INS_MEMUTILS_H
#define GEN_INS_MEMUTILS_H

#include "../common/types.h"

extern struct ParametrizedInstruction *ins_memcpy(struct ParametrizedInstruction
*dest, const struct ParametrizedInstruction *src, uint16_t len);
extern struct ParametrizedInstruction *ins_memmove(struct ParametrizedInstruction
*dest, const struct ParametrizedInstruction *src, uint16_t len);

#endif // GEN_INS_MEMUTILS_H
```

Содержание файла «memory/ins_memutils.c»

```
#ifndef GEN_INS_MEMUTILS_C
#define GEN_INS_MEMUTILS_C

#include "../memory/ins_memutils.h"
#include "instruction.h"

struct ParametrizedInstruction *ins_memcpy(struct ParametrizedInstruction *dest,
const struct ParametrizedInstruction *src, uint16_t len) {
    struct ParametrizedInstruction *dest_v = dest;
    const struct ParametrizedInstruction *src_v = src;
    while (len--) {
        *(dest_v++) = *(src_v++);
    }
    return dest;
}

struct ParametrizedInstruction *ins_memmove(struct ParametrizedInstruction *dest,
const struct ParametrizedInstruction *src, uint16_t len) {
    if (dest < src) {
        struct ParametrizedInstruction *dest_v = dest;
        const struct ParametrizedInstruction *src_v = src;
        while (len--) {
            *(dest_v++) = *(src_v++);
        }
    } else {
        struct ParametrizedInstruction *dest_v = dest + (len - 1);
```

```
        const struct ParametrizedInstruction *src_v = src + (len - 1);
        while (len--) {
            *(dest_v--) = *(src_v--);
        }
    }
    return dest;
}

#endif // GEN_INS_MEMUTILS_C
```

Содержание файла «random/rand.h»

```
#ifndef GEN_RAND_H
#define GEN_RAND_H

#include "../common/types.h"

#define M1 10000
#define M  100000000

extern uint32_t linrand(uint32_t r);
extern uint32_t linrand_intr(uint32_t left, uint32_t right);
#endif // GEN_RAND_H
```

Содержание файла «random/rand.c»

```
#ifndef GEN_RAND_C
#define GEN_RAND_C

#include "rand.h"

 uint32_t mult(uint32_t p, uint32_t q) {
    uint32_t p1;
    uint32_t p0;
    uint32_t q1;
    uint32_t q0;

    p1 = p / M1; p0 = p % M1;
    q1 = q / M1; q0 = q % M1;

    return ((((p0 * q1 + p1 * q0) % M1) * M1 + p0 * q0) % M);
}

uint32_t linrand(uint32_t r) {
    static uint32_t a = 1234567;

    a = (mult(a,31415821) + 1) % M;

    return (((a / M1) * r) / M1);
}
```

```
uint32_t linrand_intr(uint32_t left, uint32_t right) {
    return left + linrand(right - left);
}

#endif // GEN_RAND_C
```

Содержание файла «template_block/template_block_decls.h»

```
#ifndef GEN_TEMPLATE_BLOCK_DECLS_H
#define GEN_TEMPLATE_BLOCK_DECLS_H

// ----------------------Template Tests-----------------------

#define SEQUENCE (1)
#define POOL (1 << 1)

// -----------------------------------------------------------
// ----------------------Internal flags-----------------------

#define EMPTY (0)

// Pool:
#define PICK_BLOCKS (1)
#define WITH_REPEATS (1 << 1)
#define NO_ORDER (1 << 2)

// -----------------------------------------------------------
// ----------------------External flags-----------------------

// -----------------------------------------------------------


#endif // GEN_TEMPLATE_BLOCK_DECLS_H
```

Содержание файла «template_block/block_process_info.h»

```
#ifndef GEN_BLOCK_PROCESS_INFO_H
#define GEN_BLOCK_PROCESS_INFO_H

#include "../common/types.h"
#include "../generation/generator_context.h"

struct BlockProcessInfo {
    struct GeneratorContext *context;
    uint8_t curr_ord_stack_ind;
    uint16_t ord_stack_len;
    flags_t flags;
    uint16_t param;
    uint16_t ins_count;
```

```
    uint16_t ins_stack_len;
};

#endif // GEN_BLOCK_PROCESS_INFO_H
```

Содержание файла «template_block/block_process_result.h»

```
#ifndef GEN_BLOCK_PROCESS_RESULT_H
#define GEN_BLOCK_PROCESS_RESULT_H

#include "../common/types.h"

struct BlockProcessResult {
    uint16_t generated_ins_count;
    uint16_t inner_ins_count;
};

#endif // GEN_BLOCK_PROCESS_RESULT_H
```

Содержание файла «template_block/template_block.h»

```
#ifndef GEN_TEMPLATE_BLOCK_H
#define GEN_TEMPLATE_BLOCK_H

#include "template_block_decls.h"
#include "../common/types.h"

struct TemplateBlock {
    uint8_t id;
    flags_t internal_flags;
    uint16_t param;
    flags_t type;
    uint8_t parent_id;
    uint8_t sibling_id;
    uint8_t child_id;
    struct ParametrizedInstruction *data_start_ptr;
    uint16_t data_lines_count;
    struct ParametrizedInstruction *stack_start_ptr;
    uint16_t stack_lines_count;
};

#endif // GEN_TEMPLATE_BLOCK_H
```

Содержание файла «template_block/template_block_type.h»

```
#ifndef GEN_TEMPLATE_BLOCK_TYPE_H
#define GEN_TEMPLATE_BLOCK_TYPE_H

#include "../generation/generator_context.h"
#include "block_process_result.h"
```

```
extern struct BlockProcessResult process_block(struct GeneratorContext *context,
uint8_t curr_ord_stack_ind, uint16_t ord_stack_len, uint16_t stack_len);

#endif // GEN_TEMPLATE_BLOCK_TYPE_H
```

Содержание файла «template_block/template_block_type.c»

```
#ifndef GEN_TEMPLATE_BLOCK_TYPE_C
#define GEN_TEMPLATE_BLOCK_TYPE_C

#include "template_block_type.h"
#include "instruction.h"
#include "tests.h"
#include "template_block.h"
#include "block_process_info.h"
#include "../memory/ins_memutils.h"
#include "../memory/memutils.h"
#include "../random/rand.h"



struct BlockProcessResult process_sequence_block(struct BlockProcessInfo *info) {
    struct TemplateBlock *curr;
    uint16_t generated_ins_count = 0;
    for (uint16_t block_id = info->curr_ord_stack_ind; block_id < info-
>ord_stack_len; ++block_id) {
        curr = info->context->tree + info->context->order_stack[block_id];
        ins_memmove(info->context->buf + generated_ins_count, curr-
>stack_start_ptr, curr->stack_lines_count);
        generated_ins_count += curr->stack_lines_count;
    }
    return (struct BlockProcessResult) {.generated_ins_count =
generated_ins_count, .inner_ins_count = generated_ins_count};
}

struct BlockProcessResult process_pool_block_without_rel_order(struct
BlockProcessInfo *info) {
    fill_arr_u8t(info->context->used_elements, info->context->max_ins_count,
false);
    uint16_t element_count = info->param;
    uint16_t left_index;
    uint16_t right_border;
    uint16_t rnd_id;
    uint16_t generated_ins_count = 0;
    if (info->flags & PICK_BLOCKS) {
        struct TemplateBlock *curr;
        left_index = info->curr_ord_stack_ind;
        right_border = info->ord_stack_len;
        for (uint16_t element_id = 0; element_id < element_count; ++element_id) {
```

```
            do {
                rnd_id = linrand_intr(left_index, right_border);
            } while (!(info->flags & WITH_REPEATS) && info->context-
>used_elements[rnd_id - left_index]);
            info->context->used_elements[rnd_id - left_index] = true;
            curr = info->context->tree + info->context->order_stack[rnd_id];
            ins_memmove(info->context->buf + generated_ins_count, curr-
>stack_start_ptr, curr->stack_lines_count);
            generated_ins_count += curr->stack_lines_count;
        }
    } else {
        left_index = info->ins_stack_len - info->ins_count;
        right_border = info->ins_stack_len;
        for (uint16_t element_id = 0; element_id < element_count; ++element_id) {
            do {
                rnd_id = linrand_intr(left_index, right_border);
            } while (!(info->flags & WITH_REPEATS) && info->context-
>used_elements[rnd_id - left_index]);
            info->context->used_elements[rnd_id - left_index] = true;
            info->context->buf[generated_ins_count++] = info->context-
>stack[rnd_id];
        }
    }
    return (struct BlockProcessResult) {.generated_ins_count =
generated_ins_count, .inner_ins_count = info->ins_count};
}

struct BlockProcessResult process_pool_block_with_rel_order(struct
BlockProcessInfo *info) {
    uint16_t element_count = info->param;
    uint16_t left_index;
    uint16_t right_border;
    uint16_t rnd_id;
    uint16_t generated_ins_count = 0;
    if (info->flags & PICK_BLOCKS) {
        struct TemplateBlock *curr;
        left_index = info->curr_ord_stack_ind;
        right_border = info->ord_stack_len + 1 - element_count;
        for (uint16_t element_id = 0; element_id < element_count; ++element_id) {
            rnd_id = linrand_intr(left_index, right_border);
            curr = info->context->tree + info->context->order_stack[rnd_id];
            ins_memmove(info->context->buf + generated_ins_count, curr-
>stack_start_ptr, curr->stack_lines_count);
            generated_ins_count += curr->stack_lines_count;
            left_index = rnd_id + !(info->flags & WITH_REPEATS);
            ++right_border;
        }
    } else {
```

```
        left_index = info->ins_stack_len - info->ins_count;
        right_border = info->ins_stack_len + 1 - element_count;
        for (uint16_t element_id = 0; element_id < element_count; ++element_id) {
            rnd_id = linrand_intr(left_index, right_border);
            info->context->buf[generated_ins_count++] = info->context-
>stack[rnd_id];
            left_index = rnd_id + !(info->flags & WITH_REPEATS);
            ++right_border;
        }
    }
    return (struct BlockProcessResult) {.generated_ins_count =
generated_ins_count, .inner_ins_count = info->ins_count};
}


struct BlockProcessResult process_pool_block(struct BlockProcessInfo *info) {
    if (info->flags & NO_ORDER) {
        return process_pool_block_without_rel_order(info);
    } else {
        return process_pool_block_with_rel_order(info);
    }
}

struct BlockProcessResult process_block(struct GeneratorContext *context, uint8_t
curr_ord_stack_ind, uint16_t ord_stack_len, uint16_t stack_len) {
    struct TemplateBlock *parent = context->tree + context-
>order_stack[curr_ord_stack_ind];
    flags_t flags = parent->internal_flags;
    uint16_t param = parent->param;
    flags_t type = parent->type;
    uint16_t ins_count;
    if (parent->child_id != parent->id) {
        // If there are inner blocks for the current parent block, the parent
block is excluded from processing.
        ++curr_ord_stack_ind;
        ins_count = (context->stack + stack_len) - context->tree[parent-
>child_id].stack_start_ptr;
    } else {
        // If there are no inner blocks for the current parent block, the block
itself is treated as an implicitly-inner block.
        ins_count = parent->data_lines_count;
    }
    struct BlockProcessInfo info = {.context = context, .flags = flags,
                                    .param = param, .curr_ord_stack_ind =
curr_ord_stack_ind,
                                    .ord_stack_len = ord_stack_len, .ins_count =
ins_count, .ins_stack_len = stack_len};
```

```
    if (type & SEQUENCE) {
        return process_sequence_block(&info);
    } else if (type & POOL) {
        return process_pool_block(&info);
    }
    return (struct BlockProcessResult) {.generated_ins_count = 0,
.inner_ins_count = 0};
}

#endif // GEN_TEMPLATE_BLOCK_TYPE_C
```

Содержание файла «CMakeLists.txt»

```
cmake_minimum_required(VERSION 3.16 FATAL_ERROR)
project(OTPGenerator VERSION 0.0.1 LANGUAGES C)
set(CMAKE_C_STANDARD 99)

## Input arguments:
## CC_PATH      ex. "/home/user/RISCV/compiler/bin/riscv32-unknown-linux-gnu-gcc"
## CC_PARAMS    ex. "-march=rv32i -mabi=ilp32 -Wl,-nostdlib"
## WSIZE        ex. "32" or "64"
## ISA_DIR
## TESTS_DIR
## MUT_DIR
## OUT_DIR
## PRINT

set(WORKING_DIR "")
execute_process(COMMAND "pwd" OUTPUT_VARIABLE WORKING_DIR
OUTPUT_STRIP_TRAILING_WHITESPACE)

message("----------------------------------------")

if(CC_PATH)
    if (NOT IS_ABSOLUTE ${CC_PATH})
        set(CC_PATH "${WORKING_DIR}/${CC_PATH}")
    endif()
    set(CMAKE_C_COMPILER ${CC_PATH} CACHE FILEPATH "" FORCE)
    message("Chosen C compiler: ${CMAKE_C_COMPILER}")
else()
    message(FATAL_ERROR "No C compiler was specified." )
endif()

if(CC_PARAMS)
    add_compile_options(${CC_PARAMS})
    message("Chosen C compiler parameters: ${CC_PARAMS}")
else()
    set(CC_PARAMS "NO")
endif()
```

```
if(NOT WSIZE OR ${WSIZE} EQUAL "32")
    set(WSIZE 32 CACHE UNINITIALIZED "" FORCE)
elseif(${WSIZE} EQUAL "64")
    set(WSIZE 64 CACHE UNINITIALIZED "" FORCE)
else()
    message(FATAL_ERROR "Incorrect (or no at all) machine word size was specified
(32/64)." )
endif()

message("Chosen target system machine word size: ${WSIZE}")

if(ISA_DIR)
    if (NOT IS_ABSOLUTE ${ISA_DIR})
        set(ISA_DIR "${WORKING_DIR}/${ISA_DIR}")
    endif()
    set(ISA_DIR ${ISA_DIR} CACHE FILEPATH "" FORCE)
    message("Chosen ISA directory: ${ISA_DIR}")
else()
    message(FATAL_ERROR "No ISA directory path was specified." )
endif()

if(TESTS_DIR)
    if (NOT IS_ABSOLUTE ${TESTS_DIR})
        set(TESTS_DIR "${WORKING_DIR}/${TESTS_DIR}")
    endif()
    set(TESTS_DIR ${TESTS_DIR} CACHE FILEPATH "" FORCE)
    message("Chosen tests directory: ${TESTS_DIR}")
else()
    message(FATAL_ERROR "No tests directory path was specified." )
endif()

if(MUT_DIR)
    if (NOT IS_ABSOLUTE ${MUT_DIR})
        set(MUT_DIR "${WORKING_DIR}/${MUT_DIR}")
    endif()
    set(MUT_DIR ${MUT_DIR} CACHE FILEPATH "" FORCE)
    message("Chosen mutation functionality directory: ${MUT_DIR}")
else()
    message(FATAL_ERROR "No mutation functionality directory path was specified."
)
endif()

if(NOT OUT_DIR)
    set(OUT_DIR ".")
endif()

if (NOT IS_ABSOLUTE ${OUT_DIR})
```

```
        set(OUT_DIR "${WORKING_DIR}/${OUT_DIR}")
    endif()
    set(CMAKE_RUNTIME_OUTPUT_DIRECTORY ${OUT_DIR} CACHE FILEPATH "" FORCE)
    message("Chosen binary executable output directory:
${CMAKE_RUNTIME_OUTPUT_DIRECTORY}")

message("----------------------------------------")

include_directories(BEFORE
        "${CMAKE_SOURCE_DIR}/core/common"
        "${CMAKE_SOURCE_DIR}/core/template_block"
        "${CMAKE_SOURCE_DIR}/core/decls"
        "${CMAKE_SOURCE_DIR}/core/random"
        "${CMAKE_SOURCE_DIR}/core/memory"
        ${ISA_DIR}
        ${TESTS_DIR}
        ${MUT_DIR}
        )

list(APPEND GENERATOR_MACROS
        GEN_VERSION="${CMAKE_PROJECT_VERSION}"
        #GEN_CC_PATH=${CC_PATH}
        #GEN_CC_PARAMS=${CC_PARAMS}
        GEN_WORDSIZE=${WSIZE}
        #GEN_ISA_DIR=${ISA_DIR}
        #GEN_TESTS_DIR=${TESTS_DIR}
        #GEN_MUT_DIR=${MUT_DIR}
        )

if(PRINT)
list(APPEND GENERATOR_MACROS
        GEN_USE_STDIO)
endif()


add_compile_definitions(${GENERATOR_MACROS})

set(CORE_SRC
        core/main.c
        core/common/utility.c
        core/generation/generator.c
        core/execution/runner.c
        core/random/rand.c
        core/template_block/template_block_type.c
        core/memory/memutils.c
        core/memory/ins_memutils.c
        )
```

```
add_executable(Generator
        ${CORE_SRC}
        ${ISA_DIR}/instruction.c
        ${TESTS_DIR}/tests.c
        ${MUT_DIR}/mutator.c
        )

target_link_libraries(Generator "-static")
```

Содержание файла «compile.sh»

```
#!/bin/bash

SCRIPT_DIR=$( cd -- "$( dirname -- "${BASH_SOURCE[0]}" )" &> /dev/null && pwd )

if [ -d $7/otpg_build ]; then :
  rm -rf $7/otpg_build;
fi

if [ $# -eq 8 ] && [ $8 = "-p" ]; then :

  cmake -DCC_PATH:FILEPATH=$1 -D CC_PARAMS=$2 -DWSIZE=$3 -DISA_DIR=$4 -
DTESTS_DIR=$5 -DMUT_DIR=$6 -DOUT_DIR=$7 -DPRINT=1 -S $SCRIPT_DIR -B
$7/otpg_build;

elif [ $# -eq 7 ]; then :

  cmake -DCC_PATH:FILEPATH=$1 -D CC_PARAMS=$2 -DWSIZE=$3 -DISA_DIR=$4 -
DTESTS_DIR=$5 -DMUT_DIR=$6 -DOUT_DIR=$7 -S $SCRIPT_DIR -B $7/otpg_build;

else :

  echo "OTPGenerator: wrong amount of arguments supplied (there can be either 7
or 8)."
  exit

fi


cmake --build $7/otpg_build;
#rm -rf $7/otpg_build;
```

3. ИСХОДНЫЕ КОДЫ УРОВНЯ ПОЛЬЗОВАТЕЛЯ (RISC-V RV32I)

Содержание файла «instruction.h»

```
#ifndef GEN_INSTRUCTION_H
#define GEN_INSTRUCTION_H
```

```
#include "ins_decls.h"

#define GEN_GPR_COUNT 32
#define GEN_FPR_COUNT 0
#define GEN_REG_COUNT (GEN_GPR_COUNT + GEN_FPR_COUNT)
#define GEN_INS_GROUP_COUNT 6
#define GEN_INS_COUNT 47
#define GEN_PSEUDO_INS_COUNT 7

#define SLLI_ID 24
#define SRLI_ID 25
#define SRAI_ID 26
#define FENCE_ID 37
#define FENCE_I_ID 38
#define ECALL_ID 39
#define EBREAK_ID 40

#define NOP (struct ParametrizedInstruction) {.instruction_id = 0, .pseudo_ins =
true}
#define RET (struct ParametrizedInstruction) {.instruction_id = 6, .pseudo_ins =
true}

#define STORE_REG_DUMP(dump) __asm__( \
    "sw t6, -4(sp)\n" \
    "\tlui t6, %hi(" dump ")\n" \
    "\tlw t6, %lo(" dump ")(t6)\n" \
    "\tsw zero, 0(t6)\n" \
    "\tsw ra, 4(t6)\n" \
    "\tsw sp, 8(t6)\n" \
    "\tsw gp, 12(t6)\n" \
    "\tsw tp, 16(t6)\n" \
    "\tsw t0, 20(t6)\n" \
    "\tsw t1, 24(t6)\n" \
    "\tsw t2, 28(t6)\n" \
    "\tsw s0, 32(t6)\n" \
    "\tsw s1, 36(t6)\n" \
    "\tsw a0, 40(t6)\n" \
    "\tsw a1, 44(t6)\n" \
    "\tsw a2, 48(t6)\n" \
    "\tsw a3, 52(t6)\n" \
    "\tsw a4, 56(t6)\n" \
    "\tsw a5, 60(t6)\n" \
    "\tsw a6, 64(t6)\n" \
    "\tsw a7, 68(t6)\n" \
    "\tsw s2, 72(t6)\n" \
    "\tsw s3, 76(t6)\n" \
    "\tsw x4, 80(t6)\n" \
```

```
    "\tsw s5, 84(t6)\n" \
    "\tsw s6, 88(t6)\n" \
    "\tsw x7, 92(t6)\n" \
    "\tsw s8, 96(t6)\n" \
    "\tsw s9, 100(t6)\n" \
    "\tsw s10, 104(t6)\n" \
    "\tsw s11, 108(t6)\n" \
    "\tsw t3, 112(t6)\n" \
    "\tsw t4, 116(t6)\n" \
    "\tsw t5, 120(t6)\n" \
    "\tsw t5, -8(sp)\n" \
    "\taddi t5, t6, 0\n" \
    "\tlw t6, -4(sp)\n" \
    "\tsw t6, 124(t5)\n" \
    "\tlw t5, -8(sp)\n");

#define LOAD_REG_DUMP(dump) __asm__( \
    "lui t6, %hi(" dump ")\n" \
    "\tlw t6, %lo(" dump ")(t6)\n" \
    "\tlw zero, 0(t6)\n" \
    "\tlw ra, 4(t6)\n" \
    "\tlw sp, 8(t6)\n" \
    "\tlw gp, 12(t6)\n" \
    "\tlw tp, 16(t6)\n" \
    "\tlw t0, 20(t6)\n" \
    "\tlw t1, 24(t6)\n" \
    "\tlw t2, 28(t6)\n" \
    "\tlw s0, 32(t6)\n" \
    "\tlw s1, 36(t6)\n" \
    "\tlw a0, 40(t6)\n" \
    "\tlw a1, 44(t6)\n" \
    "\tlw a2, 48(t6)\n" \
    "\tlw a3, 52(t6)\n" \
    "\tlw a4, 56(t6)\n" \
    "\tlw a5, 60(t6)\n" \
    "\tlw a6, 64(t6)\n" \
    "\tlw a7, 68(t6)\n" \
    "\tlw s2, 72(t6)\n" \
    "\tlw s3, 76(t6)\n" \
    "\tlw x4, 80(t6)\n" \
    "\tlw s5, 84(t6)\n" \
    "\tlw s6, 88(t6)\n" \
    "\tlw x7, 92(t6)\n" \
    "\tlw s8, 96(t6)\n" \
    "\tlw s9, 100(t6)\n" \
    "\tlw s10, 104(t6)\n" \
    "\tlw s11, 108(t6)\n" \
    "\tlw t3, 112(t6)\n" \
```

```
    "\tlw t4, 116(t6)\n" \
    "\tlw t5, 120(t6)\n" \
    "\tlw t6, 124(t6)\n");

#define GENERAL_INS 1
#define JUMP_INS 2
#define STORING_INS 4
#define BRANCH_INS 8

// todo: Check padding.
struct InstructionGroup {
    // uint8_t flags;
    // uint8_t id;
    flags_t type;
    uint8_t ins_len;
    flags_t output;
    flags_t first_op;
    flags_t second_op;
    flags_t first_funct;
    flags_t second_funct;
    flags_t first_add_bit;
    flags_t second_add_bit;
    flags_t first_add_immediate;
    flags_t second_add_immediate;
    uint8_t opcode_pos;
    uint8_t opcode_len;
    uint8_t output_pos;
    uint8_t output_len;
    uint8_t first_op_pos;
    uint8_t first_op_len;
    uint8_t second_op_pos;
    uint8_t second_op_len;
    uint8_t first_funct_pos;
    uint8_t first_funct_len;
    uint8_t second_funct_pos;
    uint8_t second_funct_len;
    uint8_t first_add_bit_pos;
    uint8_t second_add_bit_pos;
    uint8_t first_add_immediate_pos;
    uint8_t first_add_immediate_len;
    uint8_t second_add_immediate_pos;
    uint8_t second_add_immediate_len;
};

struct Instruction {
    uint8_t group_id;
    uint8_t opcode;
    uint32_t first_funct;
```

```
    uint32_t second_funct;
};

struct PseudoInstruction {
    uint8_t instruction_id;
    bool_t output_fixated;
    bool_t first_op_fixated;
    bool_t second_op_fixated;
    bool_t first_add_bit_fixated;
    bool_t second_add_bit_fixated;
    bool_t first_add_immediate_fixated;
    bool_t second_add_immediate_fixated;
    uint32_t output;
    uint32_t first_op;
    uint32_t second_op;
    uint8_t first_add_bit;
    uint8_t second_add_bit;
    uint32_t first_add_immediate;
    uint32_t second_add_immediate;
};

struct ParametrizedInstruction {
    bool_t pseudo_ins;
    uint8_t instruction_id;
    bool_t output_fixated;
    bool_t first_op_fixated;
    bool_t second_op_fixated;
    bool_t first_add_bit_fixated;
    bool_t second_add_bit_fixated;
    bool_t first_add_immediate_fixated;
    bool_t second_add_immediate_fixated;
    uint32_t output;
    uint32_t first_op;
    uint32_t second_op;
    uint8_t first_add_bit;
    uint8_t second_add_bit;
    uint32_t first_add_immediate;
    uint32_t second_add_immediate;
};

#ifndef GEN_INSTRUCTION_C
extern uint32_t C_GEN_FIRST_REG_DUMP[GEN_GPR_COUNT];
extern uint32_t C_GEN_SECOND_REG_DUMP[GEN_GPR_COUNT];
extern uint32_t C_GEN_BUF_REG_DUMP[GEN_GPR_COUNT];
extern bool_t C_GEN_REG_INCLUDE_IN_DUMP[GEN_GPR_COUNT];
extern struct InstructionGroup C_GEN_INS_GROUPS[GEN_INS_GROUP_COUNT];
extern struct Instruction C_GEN_INS[GEN_INS_COUNT];
extern struct PseudoInstruction C_GEN_PSEUDO_INS[GEN_PSEUDO_INS_COUNT];
```

```
#else
uint32_t C_GEN_FIRST_REG_DUMP[GEN_GPR_COUNT] = {0};
uint32_t C_GEN_SECOND_REG_DUMP[GEN_GPR_COUNT] = {0};
uint32_t C_GEN_BUF_REG_DUMP[GEN_GPR_COUNT] = {0};
bool_t C_GEN_REG_INCLUDE_IN_DUMP[GEN_GPR_COUNT] = {false};

struct InstructionGroup C_GEN_INS_GROUPS[GEN_INS_GROUP_COUNT] = {
        // ------------------------RV32I-------------------------
        // R
        {.type = GENERAL_INS, .ins_len = 4, .output = IS_REGISTER, .first_op =
IS_REGISTER, .second_op = IS_REGISTER, .first_funct = IS_USED, .second_funct =
IS_USED, .first_add_bit = !IS_USED, .second_add_bit = !IS_USED,
.first_add_immediate = !IS_USED, .second_add_immediate = !IS_USED, .opcode_pos =
0, .opcode_len = 7, .output_pos = 7, .output_len = 5, .first_funct_pos = 12,
.first_funct_len = 3, .first_op_pos = 15, .first_op_len = 5, .second_op_pos = 20,
.second_op_len = 5, .second_funct_pos = 25, .second_funct_len = 7},
        // I
        {.type = GENERAL_INS, .ins_len = 4, .output = IS_REGISTER, .first_op =
IS_REGISTER, .second_op = IS_IMMEDIATE, .first_funct = IS_USED, .second_funct =
!IS_USED, .first_add_bit = !IS_USED, .second_add_bit = !IS_USED,
.first_add_immediate = !IS_USED, .second_add_immediate = !IS_USED, .opcode_pos =
0, .opcode_len = 7, .output_pos = 7, .output_len = 5, .first_funct_pos = 12,
.first_funct_len = 3, .first_op_pos = 15, .first_op_len = 5, .second_op_pos = 20,
.second_op_len = 12},
        // S
        {.type = STORING_INS, .ins_len = 4, .output = !IS_USED, .first_op =
IS_REGISTER, .second_op = IS_REGISTER, .first_funct = IS_USED, .second_funct =
!IS_USED, .first_add_bit = !IS_USED, .second_add_bit = !IS_USED,
.first_add_immediate = IS_USED, .second_add_immediate = IS_USED, .opcode_pos = 0,
.opcode_len = 7, .first_add_immediate_pos = 7, .first_add_immediate_len = 5,
.first_funct_pos = 12, .first_funct_len = 3, .first_op_pos = 15, .first_op_len =
5, .second_op_pos = 20, .second_op_len = 5, .second_add_immediate_pos = 25,
.second_add_immediate_len = 7},
        // B
        {.type = BRANCH_INS, .ins_len = 4, .output = !IS_USED, .first_op =
IS_REGISTER, .second_op = IS_REGISTER, .first_funct = IS_USED, .second_funct =
!IS_USED, .first_add_bit = IS_USED, .second_add_bit = IS_USED,
.first_add_immediate = IS_USED, .second_add_immediate = IS_USED, .opcode_pos = 0,
.opcode_len = 7, .first_add_immediate_pos = 8, .first_add_immediate_len = 4,
.first_funct_pos = 12, .first_funct_len = 3, .first_op_pos = 15, .first_op_len =
5, .second_op_pos = 20, .second_op_len = 5, .second_add_immediate_pos = 25,
.second_add_immediate_len = 6, .first_add_bit_pos = 7, .second_add_bit_pos = 31},
        // U
        {.type = GENERAL_INS, .ins_len = 4, .output = IS_REGISTER, .first_op =
IS_IMMEDIATE, .second_op = !IS_USED, .first_funct = !IS_USED, .second_funct =
!IS_USED, .first_add_bit = !IS_USED, .second_add_bit = !IS_USED,
.first_add_immediate = !IS_USED, .second_add_immediate = !IS_USED, .opcode_pos =
0, .opcode_len = 7, .output_pos = 7, .output_len = 5, .first_op_pos = 12,
```

```
.first_op_len = 20},
        // J
        {.type = JUMP_INS, .ins_len = 4, .output = IS_REGISTER, .first_op =
IS_IMMEDIATE, .second_op = !IS_USED, .first_funct = !IS_USED, .second_funct =
!IS_USED, .first_add_bit = IS_USED, .second_add_bit = IS_USED,
.first_add_immediate = !IS_USED, .second_add_immediate = !IS_USED,  .opcode_pos =
0, .opcode_len = 7, .output_pos = 7, .output_len = 5, .first_op_pos = 12,
.first_op_len = 8, .second_op_pos = 21, .second_op_len = 10, .first_add_bit_pos =
20, .second_add_bit_pos = 31}};

struct Instruction C_GEN_INS[GEN_INS_COUNT] = {
        // ------------------------RV32I-------------------------
        // LUI                              AUIPC                              JAL
        {.group_id = 4, .opcode = 067}, {.group_id = 4, .opcode = 027},
{.group_id = 5, .opcode = 0157},
        // JALR                                                    BEQ
        {.group_id = 1, .opcode = 0147, .first_funct = 00}, {.group_id = 3,
.opcode = 0143, .first_funct = 00},
        // BNE                                                     BLT
        {.group_id = 3, .opcode = 0143, .first_funct = 01}, {.group_id = 3,
.opcode = 0143, .first_funct = 04},
        // BGE                                                     BLTU
        {.group_id = 3, .opcode = 0143, .first_funct = 05}, {.group_id = 3,
.opcode = 0143, .first_funct = 06},
        // BGEU                                                    LB
        {.group_id = 3, .opcode = 0143, .first_funct = 07}, {.group_id = 1,
.opcode = 03, .first_funct = 00},
        // LH                                                      LW
        {.group_id = 1, .opcode = 03, .first_funct = 01}, {.group_id = 1, .opcode
= 03, .first_funct = 02},
        // LBU                                                     LHU
        {.group_id = 1, .opcode = 03, .first_funct = 04}, {.group_id = 1, .opcode
= 03, .first_funct = 05},
        // SB                                                      SH
        {.group_id = 2, .opcode = 043, .first_funct = 00}, {.group_id = 2,
.opcode = 043, .first_funct = 01},
        // SW                                                      ADDI
        {.group_id = 2, .opcode = 043, .first_funct = 02}, {.group_id = 1,
.opcode = 023, .first_funct = 00},
        // SLTI                                                    SLTI
        {.group_id = 1, .opcode = 023, .first_funct = 02}, {.group_id = 1,
.opcode = 023, .first_funct = 03},
        // XORI                                                    ORI
        {.group_id = 1, .opcode = 023, .first_funct = 04}, {.group_id = 1,
.opcode = 023, .first_funct = 06},
        // ANDI                                                    SLLI
        {.group_id = 1, .opcode = 023, .first_funct = 07}, {.group_id = 1,
.opcode = 023, .first_funct = 01},
```

```
        // Next two instructions have to be encoded as almost identical I-
instructions, yet with different binary masks on immediate values.
        // SRLI                                                  SRAI
        {.group_id = 1, .opcode = 023, .first_funct = 05}, {.group_id = 1,
.opcode = 023, .first_funct = 05},
        // ADD
        {.group_id = 0, .opcode = 063, .first_funct = 00, .second_funct = 00},
        // SUB
        {.group_id = 0, .opcode = 063, .first_funct = 00, .second_funct = 040},
        // SLL
        {.group_id = 0, .opcode = 063, .first_funct = 01, .second_funct = 00},
        // SLT
        {.group_id = 0, .opcode = 063, .first_funct = 02, .second_funct = 00},
        // SLTU
        {.group_id = 0, .opcode = 063, .first_funct = 03, .second_funct = 00},
        // XOR
        {.group_id = 0, .opcode = 063, .first_funct = 04, .second_funct = 00},
        // SRL
        {.group_id = 0, .opcode = 063, .first_funct = 05, .second_funct = 00},
        // SRA
        {.group_id = 0, .opcode = 063, .first_funct = 05, .second_funct = 040},
        // OR
        {.group_id = 0, .opcode = 063, .first_funct = 06, .second_funct = 00},
        // AND
        {.group_id = 0, .opcode = 063, .first_funct = 07, .second_funct = 00},
        // FENCE                                                 FENCE.I
        {.group_id = 1, .opcode = 017, .first_funct = 00}, {.group_id = 1,
.opcode = 017, .first_funct = 01},
        // Next two instructions have to be encoded as almost identical I-
instructions, yet with different binary masks on immediate values.
        // ECALL                                                 EBREAK
        {.group_id = 1, .opcode = 0163, .first_funct = 00}, {.group_id = 1,
.opcode = 0163, .first_funct = 00},
        // CSRRW                                                 CSRRS
        {.group_id = 1, .opcode = 0163, .first_funct = 01}, {.group_id = 1,
.opcode = 0163, .first_funct = 02},
        // CSRRC                                                 CSRRWI
        {.group_id = 1, .opcode = 0163, .first_funct = 03}, {.group_id = 1,
.opcode = 0163, .first_funct = 05},
        // CSRRSI                                                CSRRCI
        {.group_id = 1, .opcode = 0163, .first_funct = 06}, {.group_id = 1,
.opcode = 0163, .first_funct = 07}};

struct PseudoInstruction C_GEN_PSEUDO_INS[GEN_PSEUDO_INS_COUNT] = {
        // -------------------------PSEUD-------------------------
        // NOP
        {.instruction_id = 18, .output_fixated = true, .first_op_fixated = true,
.second_op_fixated = true, .first_add_bit_fixated = false,
```

```
.second_add_bit_fixated = false, .first_add_immediate_fixated = false,
.second_add_immediate_fixated = false, .output = 0, .first_op = 0, .second_op =
0},
        // MV
        {.instruction_id = 18, .output_fixated = false, .first_op_fixated =
false, .second_op_fixated = true, .first_add_bit_fixated = false,
.second_add_bit_fixated = false, .first_add_immediate_fixated = false,
.second_add_immediate_fixated = false, .second_op = 0},
        // J
        {.instruction_id = 2, .output_fixated = true, .first_op_fixated = false,
.second_op_fixated = false, .first_add_bit_fixated = false,
.second_add_bit_fixated = false, .first_add_immediate_fixated = false,
.second_add_immediate_fixated = false, .output = 0},
        // JAL
        {.instruction_id = 2, .output_fixated = true, .first_op_fixated = false,
.second_op_fixated = false, .first_add_bit_fixated = false,
.second_add_bit_fixated = false, .first_add_immediate_fixated = false,
.second_add_immediate_fixated = false, .output = 1},
        // JR
        {.instruction_id = 3, .output_fixated = true, .first_op_fixated = false,
.second_op_fixated = true, .first_add_bit_fixated = false,
.second_add_bit_fixated = false, .first_add_immediate_fixated = false,
.second_add_immediate_fixated = false, .output = 0, .second_op = 0},
        // JALR
        {.instruction_id = 3, .output_fixated = true, .first_op_fixated = false,
.second_op_fixated = true, .first_add_bit_fixated = false,
.second_add_bit_fixated = false, .first_add_immediate_fixated = false,
.second_add_immediate_fixated = false, .output = 1, .second_op = 0},
        // RET
        {.instruction_id = 3, .output_fixated = true, .first_op_fixated = true,
.second_op_fixated = true, .first_add_bit_fixated = false,
.second_add_bit_fixated = false, .first_add_immediate_fixated = false,
.second_add_immediate_fixated = false, .output = 0, .first_op = 1, .second_op =
0}};
#endif



#endif // GEN_INSTRUCTION_H
```

Содержание файла «instruction.c»

```
#ifndef GEN_INSTRUCTION_C
#define GEN_INSTRUCTION_C

#include "instruction.h"


uint8_t get_ins_len(struct InstructionGroup *groups, struct Instruction
```

```
*instructions, struct PseudoInstruction *pseudos, struct ParametrizedInstruction
*instruction) {
    return groups[
                instructions[
                        (instruction->pseudo_ins) ? pseudos[instruction-
>instruction_id].instruction_id : instruction->instruction_id
                        ].group_id
                ].ins_len;
}

ins_t encode_ins(struct ParametrizedInstruction *instruction) {
    uint32_t result = 0;
    struct PseudoInstruction *pseudo_ins;
    struct Instruction *ins;
    // No loop for a situation (ParametrizedInstruction -> PseudoInstruction ->
PseudoInstruction -> ... -> Instruction).
    if (instruction->pseudo_ins) {
        pseudo_ins = GEN_PSEUDO_INS + instruction->instruction_id;
        ins = GEN_INS + pseudo_ins->instruction_id;
    } else {
        pseudo_ins = NULL;
        ins = GEN_INS + instruction->instruction_id;
    }
    struct InstructionGroup *grp = GEN_INS_GROUPS + ins->group_id;
    result = fill(result, grp->opcode_pos, grp->opcode_len, ins->opcode);
    if (grp->output) {
        switch (ins - GEN_INS) {
            case FENCE_ID:
            case FENCE_I_ID:
            case ECALL_ID:
            case EBREAK_ID: result = fill(result, grp->output_pos, grp-
>output_len, 0); break;
            default: result = fill(result, grp->output_pos, grp->output_len,
                                    (pseudo_ins != NULL && pseudo_ins-
>output_fixated) ? pseudo_ins->output : instruction->output);
            break;
        }

    }
    if (grp->first_op) {
        switch (ins - GEN_INS) {
            case FENCE_ID:
            case FENCE_I_ID:
            case ECALL_ID:
            case EBREAK_ID: result = fill(result, grp->first_op_pos, grp-
>first_op_len, 0); break;
            default: result = fill(result, grp->first_op_pos, grp->first_op_len,
                                    (pseudo_ins != NULL && pseudo_ins-
```

```
>first_op_fixated) ? pseudo_ins->first_op : instruction->first_op);
            break;
        }
    }
    // todo: Check mask once again.
    if (grp->second_op) {
        if (pseudo_ins != NULL && pseudo_ins->second_op_fixated) {
            result = fill(result, grp->second_op_pos, grp->second_op_len,
pseudo_ins->second_op);
        } else {
            switch (ins - GEN_INS) {
                case SLLI_ID:
                case SRLI_ID: result = fill(result, grp->second_op_pos, grp-
>second_op_len, instruction->second_op & 0x1F); break;
                case SRAI_ID: result = fill(result, grp->second_op_pos, grp-
>second_op_len, instruction->second_op & 0x41F); break;
                case FENCE_ID: result = fill(result, grp->second_op_pos, grp-
>second_op_len, instruction->second_op & 0xFF); break;
                case FENCE_I_ID:
                case ECALL_ID: result = fill(result, grp->second_op_pos, grp-
>second_op_len, 0); break;
                case EBREAK_ID: result = fill(result, grp->second_op_pos, grp-
>second_op_len, 1); break;
                default: result = fill(result, grp->second_op_pos, grp-
>second_op_len, instruction->second_op); break;
            }
        }
    }
    if (grp->first_funct) {
        result = fill(result, grp->first_funct_pos, grp->first_funct_len, ins-
>first_funct);
    }
    // todo: Check mask once again.
    if (grp->second_funct) {
        // A mask is applied as in RV32I only one bit in second funct-field may
not be null.
        result = fill(result, grp->second_funct_pos, grp->second_funct_len, ins-
>second_funct & 0x20);
    }
    if (grp->first_add_bit) {
        result = fill(result, grp->first_add_bit_pos, 1,
                      (pseudo_ins != NULL && pseudo_ins->first_add_bit_fixated) ?
pseudo_ins->first_add_bit : instruction->first_add_bit);
    }
    if (grp->second_add_bit) {
        result = fill(result, grp->second_add_bit_pos, 1,
                      (pseudo_ins != NULL && pseudo_ins->second_add_bit_fixated)
? pseudo_ins->second_add_bit : instruction->second_add_bit);
```

```
    }
    if (grp->first_add_immediate) {
        result = fill(result, grp->first_add_immediate_pos, grp-
>first_add_immediate_len,
                      (pseudo_ins != NULL && pseudo_ins-
>first_add_immediate_fixated) ? pseudo_ins->first_add_immediate : instruction-
>first_add_immediate);
    }
    if (grp->second_add_immediate) {
        result = fill(result, grp->second_add_immediate_pos, grp-
>second_add_immediate_len,
                      (pseudo_ins != NULL && pseudo_ins-
>second_add_immediate_fixated) ? pseudo_ins->second_add_immediate : instruction-
>second_add_immediate);
    }
    return result;
}

void include_regs(struct ParametrizedInstruction *instruction, bool_t
*to_include) {
    struct PseudoInstruction *p_ins;
    uint8_t group_id;
    if (instruction->pseudo_ins) {
        p_ins = GEN_PSEUDO_INS + instruction->instruction_id;
        group_id = GEN_INS[p_ins->instruction_id].group_id;
    } else {
        p_ins = NULL;
        group_id = GEN_INS[instruction->instruction_id].group_id;
    }
    if (GEN_INS_GROUPS[group_id].output == IS_REGISTER) {
        to_include[(p_ins && p_ins->output_fixated) ? p_ins->output :
instruction->output] = true;
    }
}

void init_reg_dumps() {
    GEN_FIRST_REG_DUMP = C_GEN_FIRST_REG_DUMP;
    GEN_SECOND_REG_DUMP = C_GEN_SECOND_REG_DUMP;
    GEN_BUF_REG_DUMP = C_GEN_BUF_REG_DUMP;
    GEN_REG_INCLUDE_IN_DUMP = C_GEN_REG_INCLUDE_IN_DUMP;
}

void init_groups() {
    GEN_INS_GROUPS = C_GEN_INS_GROUPS;
}


void init_instructions() {
```

```
    GEN_INS = C_GEN_INS;
    GEN_PSEUDO_INS = C_GEN_PSEUDO_INS;
}



//bool_t dump_eq(ins_t *first_dump, ins_t *second_dump) {
//    // zero(x0) is hardwired to 0.
//    if (first_dump[0] != 0 || second_dump[0] != 0) {
//        return false;
//    }
//    // ra(x1) and sp(x2) registers do not have to be the same.
//    // gp(x3) and tp(x4) registers have to be the same.
//    for (uint8_t ind = 3; ind < GEN_GPR_COUNT; ++ind) {
//        if (first_dump[ind] != second_dump[ind]) {
//            return false;
//        }
//    }
//    return true;
//}


#endif // GEN_INSTRUCTION_C
```

Содержание файла «tests.h»

```
#ifndef GEN_TESTS_H
#define GEN_TESTS_H

#include "tests_decls.h"

#define GEN_TEST_TEMPLATES_COUNT 11
#define GEN_ASM_TESTS_COUNT 2
#define GEN_MAX_INS_COUNT 1000
#define GEN_MAX_MUT_INS_COUNT 7
#define GEN_MAX_TREE_BLOCK_COUNT 200

// Test templates forward-declaration.
void single_sequence(struct TemplateBlock *tree, uint8_t max_tree_depth, struct
ParametrizedInstruction *data, uint16_t max_ins_count);

void single_pool_ins(struct TemplateBlock *tree, uint8_t max_tree_depth, struct
ParametrizedInstruction *data, uint16_t max_ins_count);
void single_pool_ins_without_order(struct TemplateBlock *tree, uint8_t
max_tree_depth, struct ParametrizedInstruction *data, uint16_t max_ins_count);
void single_pool_ins_with_repeats(struct TemplateBlock *tree, uint8_t
max_tree_depth, struct ParametrizedInstruction *data, uint16_t max_ins_count);
void single_pool_ins_with_repeats_without_order(struct TemplateBlock *tree,
uint8_t max_tree_depth, struct ParametrizedInstruction *data, uint16_t
```

```
max_ins_count);
void single_pool_blocks(struct TemplateBlock *tree, uint8_t max_tree_depth,
struct ParametrizedInstruction *data, uint16_t max_ins_count);
void single_pool_blocks_without_order(struct TemplateBlock *tree, uint8_t
max_tree_depth, struct ParametrizedInstruction *data, uint16_t max_ins_count);
void single_pool_blocks_with_repeats(struct TemplateBlock *tree, uint8_t
max_tree_depth, struct ParametrizedInstruction *data, uint16_t max_ins_count);
void single_pool_blocks_with_repeats_without_order(struct TemplateBlock *tree,
uint8_t max_tree_depth, struct ParametrizedInstruction *data, uint16_t
max_ins_count);

void basic_template_3blocks(struct TemplateBlock *tree, uint8_t max_tree_depth,
struct ParametrizedInstruction *data, uint16_t max_ins_count);
void template_test_to_fail(struct TemplateBlock *tree, uint8_t max_tree_depth,
struct ParametrizedInstruction *data, uint16_t max_ins_count);


// ASM-tests forward-declaration.
uint16_t basic_addi(struct ParametrizedInstruction *ins, uint16_t max_ins_count);
uint16_t asm_test_to_fail(struct ParametrizedInstruction *ins, uint16_t
max_ins_count);
bool_t asm_test_to_fail_eq(ins_t *first_dump, ins_t *second_dump, uint8_t
reg_count);


#ifndef GEN_TESTS_C
extern void (*C_GEN_TEST_TEMPLATES[GEN_TEST_TEMPLATES_COUNT])(struct
TemplateBlock *tree, uint8_t max_tree_depth, struct ParametrizedInstruction
*stack, uint16_t max_ins_count);
extern uint8_t C_GEN_TEST_TEMPLATES_ITERS[GEN_TEST_TEMPLATES_COUNT];
extern void (*C_GEN_TEST_TEMPLATES_FILL_INS[GEN_TEST_TEMPLATES_COUNT])(struct
ParametrizedInstruction *instruction);
extern uint16_t (*C_GEN_ASM_TESTS[GEN_ASM_TESTS_COUNT])(struct
ParametrizedInstruction *ins, uint16_t max_ins_count);
extern bool_t (*C_GEN_ASM_TESTS_DUMP_EQ[GEN_ASM_TESTS_COUNT])(ins_t *first_dump,
ins_t *second_dump, uint8_t reg_count);
extern bool_t C_GEN_USED_ELEMENTS[GEN_MAX_INS_COUNT];
extern byte_t C_GEN_OUTPUT[30];

extern struct ParametrizedInstruction C_GEN_TEST_INSTRUCTIONS[GEN_MAX_INS_COUNT];
extern struct ParametrizedInstruction
C_GEN_BLOCK_INSTRUCTIONS_STACK[GEN_MAX_INS_COUNT];
extern struct ParametrizedInstruction C_GEN_INSTRUCTIONS_BUF[GEN_MAX_INS_COUNT];
extern uint8_t C_GEN_BLOCK_ORDER_STACK[GEN_MAX_TREE_BLOCK_COUNT];
extern struct TemplateBlock C_GEN_TEMPLATE_TREE[GEN_MAX_TREE_BLOCK_COUNT];
#else
void (*C_GEN_TEST_TEMPLATES[GEN_TEST_TEMPLATES_COUNT])(struct TemplateBlock
*tree, uint8_t max_tree_depth, struct ParametrizedInstruction *stack, uint16_t
```

```
max_ins_count) =
        {single_sequence, single_pool_ins, single_pool_ins_without_order,
single_pool_ins_with_repeats,
         single_pool_ins_with_repeats_without_order, single_pool_blocks,
single_pool_blocks_without_order,
         single_pool_blocks_with_repeats,
single_pool_blocks_with_repeats_without_order,
         basic_template_3blocks, template_test_to_fail};
uint8_t C_GEN_TEST_TEMPLATES_ITERS[GEN_TEST_TEMPLATES_COUNT] = {1, 1, 1, 1, 1, 1,
1, 1, 1, 2, 1};
void (*C_GEN_TEST_TEMPLATES_FILL_INS[GEN_TEST_TEMPLATES_COUNT])(struct
ParametrizedInstruction *instruction) =
        {NULL, NULL , NULL, NULL,
         NULL, NULL, NULL, NULL,
         NULL, NULL, NULL};
uint16_t (*C_GEN_ASM_TESTS[GEN_ASM_TESTS_COUNT])(struct ParametrizedInstruction
*ins, uint16_t max_ins_count) = {basic_addi, asm_test_to_fail};
bool_t (*C_GEN_ASM_TESTS_DUMP_EQ[GEN_ASM_TESTS_COUNT])(ins_t *first_dump, ins_t
*second_dump, uint8_t reg_count) = {NULL, asm_test_to_fail_eq};
bool_t C_GEN_USED_ELEMENTS[GEN_MAX_INS_COUNT] = {false};
byte_t C_GEN_OUTPUT[30] = {0};

struct ParametrizedInstruction C_GEN_TEST_INSTRUCTIONS[GEN_MAX_INS_COUNT];
struct ParametrizedInstruction C_GEN_BLOCK_INSTRUCTIONS_STACK[GEN_MAX_INS_COUNT];
struct ParametrizedInstruction C_GEN_INSTRUCTIONS_BUF[GEN_MAX_INS_COUNT];
uint8_t C_GEN_BLOCK_ORDER_STACK[GEN_MAX_TREE_BLOCK_COUNT] = {0};
struct TemplateBlock C_GEN_TEMPLATE_TREE[GEN_MAX_TREE_BLOCK_COUNT];
#endif

#endif // GEN_TESTS_H
```

Содержание файла «tests.c»

```
#ifndef GEN_TESTS_C
#define GEN_TESTS_C

#include "tests.h"

void init_stacks() {
    GEN_TEST_INSTRUCTIONS = C_GEN_TEST_INSTRUCTIONS;
    GEN_BLOCK_INSTRUCTIONS_STACK = C_GEN_BLOCK_INSTRUCTIONS_STACK;
    GEN_INSTRUCTIONS_BUF = C_GEN_INSTRUCTIONS_BUF;
    GEN_BLOCK_ORDER_STACK = C_GEN_BLOCK_ORDER_STACK;
}

void init_tree() {
    GEN_TEMPLATE_TREE = C_GEN_TEMPLATE_TREE;
}
```

```
void init_test_patterns() {
    GEN_TEST_TEMPLATES = C_GEN_TEST_TEMPLATES;
    GEN_TEST_TEMPLATES_ITERS = C_GEN_TEST_TEMPLATES_ITERS;
    GEN_TEST_TEMPLATES_FILL_INS = C_GEN_TEST_TEMPLATES_FILL_INS;
}

void init_asm_tests() {
    GEN_ASM_TESTS = C_GEN_ASM_TESTS;
    GEN_ASM_TESTS_DUMP_EQ = C_GEN_ASM_TESTS_DUMP_EQ;
}

void init_misc() {
    GEN_USED_ELEMENTS = C_GEN_USED_ELEMENTS;
    GEN_OUTPUT = C_GEN_OUTPUT;
}

// ----------------Test Templates Definitions-----------------

void single_sequence(struct TemplateBlock *tree, uint8_t max_tree_depth, struct
ParametrizedInstruction *data, uint16_t max_ins_count) {
    tree[0] = (struct TemplateBlock) {.id = 0, .child_id = 0, .parent_id = 0,
.sibling_id = 0, .data_start_ptr = data, .data_lines_count = 4, .type =
SEQUENCE};
    // ADDI t1, zero, ... .
    data[0] = (struct ParametrizedInstruction) {.instruction_id = 18, .pseudo_ins
= false, .output = 6, .output_fixated = true, .first_op = 0, .first_op_fixated =
true, .second_op = 7};
    // ADDI t2, zero, ... .
    data[1] = (struct ParametrizedInstruction) {.instruction_id = 18, .pseudo_ins
= false, .output = 7, .output_fixated = true, .first_op = 0, .first_op_fixated =
true, .second_op = 3};
    // ADD t2, t1, t2.
    data[2] = (struct ParametrizedInstruction) {.instruction_id = 27, .pseudo_ins
= false, .output = 7, .output_fixated = true, .first_op = 6, .first_op_fixated =
true, .second_op = 7, .second_op_fixated = true};
    // Return.
    data[3] = RET;
}

void single_pool_ins(struct TemplateBlock *tree, uint8_t max_tree_depth, struct
ParametrizedInstruction *data, uint16_t max_ins_count) {
    tree[0] = (struct TemplateBlock) {.id = 0, .child_id = 1, .parent_id = 0,
.sibling_id = 0, .type = SEQUENCE};

    tree[1] = (struct TemplateBlock) {.id = 1, .child_id = 1, .parent_id = 0,
.sibling_id = 2, .data_start_ptr = data, .data_lines_count = 2, .type =
SEQUENCE};
    // ADDI t1, zero, ... .
```

```
    data[0] = (struct ParametrizedInstruction) {.instruction_id = 18, .pseudo_ins
= false, .output = 6, .output_fixated = true, .first_op = 0, .first_op_fixated =
true};
    // ADDI t2, zero, ... .
    data[1] = (struct ParametrizedInstruction) {.instruction_id = 18, .pseudo_ins
= false, .output = 7, .output_fixated = true, .first_op = 0, .first_op_fixated =
true};

    tree[2] = (struct TemplateBlock) {.id = 2, .child_id = 2, .parent_id = 0,
.sibling_id = 3, .data_start_ptr = data + 2, .data_lines_count = 5, .type = POOL,
.param = 3, .internal_flags = EMPTY};
    // ADDI t1, t1, ... .
    data[2] = (struct ParametrizedInstruction) {.instruction_id = 18, .pseudo_ins
= false, .output = 6, .output_fixated = true, .first_op = 6, .first_op_fixated =
true};
    // ADDI t2, t2, ... .
    data[3] = (struct ParametrizedInstruction) {.instruction_id = 18, .pseudo_ins
= false, .output = 7, .output_fixated = true, .first_op = 7, .first_op_fixated =
true};
    // ADD t2, t1, t2.
    data[4] = (struct ParametrizedInstruction) {.instruction_id = 27, .pseudo_ins
= false, .output = 7, .output_fixated = true, .first_op = 6, .first_op_fixated =
true, .second_op = 7, .second_op_fixated = true};
    // SUB t1, t2, t1.
    data[5] = (struct ParametrizedInstruction) {.instruction_id = 28, .pseudo_ins
= false, .output = 6, .output_fixated = true, .first_op = 7, .first_op_fixated =
true, .second_op = 6, .second_op_fixated = true};
    // OR t2, t1, t2.
    data[6] = (struct ParametrizedInstruction) {.instruction_id = 35, .pseudo_ins
= false, .output = 7, .output_fixated = true, .first_op = 6, .first_op_fixated =
true, .second_op = 7, .second_op_fixated = true};

    tree[3] = (struct TemplateBlock) {.id = 3, .child_id = 3, .parent_id = 0,
.sibling_id = 3, .data_start_ptr = data + 7, .data_lines_count = 1, .type =
SEQUENCE};
    // Return.
    data[7] = RET;
}

void single_pool_ins_without_order(struct TemplateBlock *tree, uint8_t
max_tree_depth, struct ParametrizedInstruction *data, uint16_t max_ins_count) {
    tree[0] = (struct TemplateBlock) {.id = 0, .child_id = 1, .parent_id = 0,
.sibling_id = 0, .type = SEQUENCE};

    tree[1] = (struct TemplateBlock) {.id = 1, .child_id = 1, .parent_id = 0,
.sibling_id = 2, .data_start_ptr = data, .data_lines_count = 2, .type =
SEQUENCE};
    // ADDI t1, zero, ... .
```

```
    data[0] = (struct ParametrizedInstruction) {.instruction_id = 18, .pseudo_ins
= false, .output = 6, .output_fixated = true, .first_op = 0, .first_op_fixated =
true};
    // ADDI t2, zero, ... .
    data[1] = (struct ParametrizedInstruction) {.instruction_id = 18, .pseudo_ins
= false, .output = 7, .output_fixated = true, .first_op = 0, .first_op_fixated =
true};

    tree[2] = (struct TemplateBlock) {.id = 2, .child_id = 2, .parent_id = 0,
.sibling_id = 3, .data_start_ptr = data + 2, .data_lines_count = 5, .type = POOL,
.param = 3, .internal_flags = NO_ORDER};
    // ADDI t1, t1, ... .
    data[2] = (struct ParametrizedInstruction) {.instruction_id = 18, .pseudo_ins
= false, .output = 6, .output_fixated = true, .first_op = 6, .first_op_fixated =
true};
    // ADDI t2, t2, ... .
    data[3] = (struct ParametrizedInstruction) {.instruction_id = 18, .pseudo_ins
= false, .output = 7, .output_fixated = true, .first_op = 7, .first_op_fixated =
true};
    // ADD t2, t1, t2.
    data[4] = (struct ParametrizedInstruction) {.instruction_id = 27, .pseudo_ins
= false, .output = 7, .output_fixated = true, .first_op = 6, .first_op_fixated =
true, .second_op = 7, .second_op_fixated = true};
    // SUB t1, t2, t1.
    data[5] = (struct ParametrizedInstruction) {.instruction_id = 28, .pseudo_ins
= false, .output = 6, .output_fixated = true, .first_op = 7, .first_op_fixated =
true, .second_op = 6, .second_op_fixated = true};
    // OR t2, t1, t2.
    data[6] = (struct ParametrizedInstruction) {.instruction_id = 35, .pseudo_ins
= false, .output = 7, .output_fixated = true, .first_op = 6, .first_op_fixated =
true, .second_op = 7, .second_op_fixated = true};

    tree[3] = (struct TemplateBlock) {.id = 3, .child_id = 3, .parent_id = 0,
.sibling_id = 3, .data_start_ptr = data + 7, .data_lines_count = 1, .type =
SEQUENCE};
    // Return.
    data[7] = RET;
}
void single_pool_ins_with_repeats(struct TemplateBlock *tree, uint8_t
max_tree_depth, struct ParametrizedInstruction *data, uint16_t max_ins_count) {
    tree[0] = (struct TemplateBlock) {.id = 0, .child_id = 1, .parent_id = 0,
.sibling_id = 0, .type = SEQUENCE};

    tree[1] = (struct TemplateBlock) {.id = 1, .child_id = 1, .parent_id = 0,
.sibling_id = 2, .data_start_ptr = data, .data_lines_count = 2, .type =
SEQUENCE};
    // ADDI t1, zero, ... .
    data[0] = (struct ParametrizedInstruction) {.instruction_id = 18, .pseudo_ins
```

```
= false, .output = 6, .output_fixated = true, .first_op = 0, .first_op_fixated =
true};
    // ADDI t2, zero, ... .
    data[1] = (struct ParametrizedInstruction) {.instruction_id = 18, .pseudo_ins
= false, .output = 7, .output_fixated = true, .first_op = 0, .first_op_fixated =
true};

    tree[2] = (struct TemplateBlock) {.id = 2, .child_id = 2, .parent_id = 0,
.sibling_id = 3, .data_start_ptr = data + 2, .data_lines_count = 5, .type = POOL,
.param = 3, .internal_flags = WITH_REPEATS};
    // ADDI t1, t1, ... .
    data[2] = (struct ParametrizedInstruction) {.instruction_id = 18, .pseudo_ins
= false, .output = 6, .output_fixated = true, .first_op = 6, .first_op_fixated =
true};
    // ADDI t2, t2, ... .
    data[3] = (struct ParametrizedInstruction) {.instruction_id = 18, .pseudo_ins
= false, .output = 7, .output_fixated = true, .first_op = 7, .first_op_fixated =
true};
    // ADD t2, t1, t2.
    data[4] = (struct ParametrizedInstruction) {.instruction_id = 27, .pseudo_ins
= false, .output = 7, .output_fixated = true, .first_op = 6, .first_op_fixated =
true, .second_op = 7, .second_op_fixated = true};
    // SUB t1, t2, t1.
    data[5] = (struct ParametrizedInstruction) {.instruction_id = 28, .pseudo_ins
= false, .output = 6, .output_fixated = true, .first_op = 7, .first_op_fixated =
true, .second_op = 6, .second_op_fixated = true};
    // OR t2, t1, t2.
    data[6] = (struct ParametrizedInstruction) {.instruction_id = 35, .pseudo_ins
= false, .output = 7, .output_fixated = true, .first_op = 6, .first_op_fixated =
true, .second_op = 7, .second_op_fixated = true};

    tree[3] = (struct TemplateBlock) {.id = 3, .child_id = 3, .parent_id = 0,
.sibling_id = 3, .data_start_ptr = data + 7, .data_lines_count = 1, .type =
SEQUENCE};
    // Return.
    data[7] = RET;
}

void single_pool_ins_with_repeats_without_order(struct TemplateBlock *tree,
uint8_t max_tree_depth, struct ParametrizedInstruction *data, uint16_t
max_ins_count) {
    tree[0] = (struct TemplateBlock) {.id = 0, .child_id = 1, .parent_id = 0,
.sibling_id = 0, .type = SEQUENCE};

    tree[1] = (struct TemplateBlock) {.id = 1, .child_id = 1, .parent_id = 0,
.sibling_id = 2, .data_start_ptr = data, .data_lines_count = 2, .type =
SEQUENCE};
    // ADDI t1, zero, ... .
```

```
    data[0] = (struct ParametrizedInstruction) {.instruction_id = 18, .pseudo_ins
= false, .output = 6, .output_fixated = true, .first_op = 0, .first_op_fixated =
true};
    // ADDI t2, zero, ... .
    data[1] = (struct ParametrizedInstruction) {.instruction_id = 18, .pseudo_ins
= false, .output = 7, .output_fixated = true, .first_op = 0, .first_op_fixated =
true};

    tree[2] = (struct TemplateBlock) {.id = 2, .child_id = 2, .parent_id = 0,
.sibling_id = 3, .data_start_ptr = data + 2, .data_lines_count = 5, .type = POOL,
.param = 3, .internal_flags = WITH_REPEATS | NO_ORDER};
    // ADDI t1, t1, ... .
    data[2] = (struct ParametrizedInstruction) {.instruction_id = 18, .pseudo_ins
= false, .output = 6, .output_fixated = true, .first_op = 6, .first_op_fixated =
true};
    // ADDI t2, t2, ... .
    data[3] = (struct ParametrizedInstruction) {.instruction_id = 18, .pseudo_ins
= false, .output = 7, .output_fixated = true, .first_op = 7, .first_op_fixated =
true};
    // ADD t2, t1, t2.
    data[4] = (struct ParametrizedInstruction) {.instruction_id = 27, .pseudo_ins
= false, .output = 7, .output_fixated = true, .first_op = 6, .first_op_fixated =
true, .second_op = 7, .second_op_fixated = true};
    // SUB t1, t2, t1.
    data[5] = (struct ParametrizedInstruction) {.instruction_id = 28, .pseudo_ins
= false, .output = 6, .output_fixated = true, .first_op = 7, .first_op_fixated =
true, .second_op = 6, .second_op_fixated = true};
    // OR t2, t1, t2.
    data[6] = (struct ParametrizedInstruction) {.instruction_id = 35, .pseudo_ins
= false, .output = 7, .output_fixated = true, .first_op = 6, .first_op_fixated =
true, .second_op = 7, .second_op_fixated = true};

    tree[3] = (struct TemplateBlock) {.id = 3, .child_id = 3, .parent_id = 0,
.sibling_id = 3, .data_start_ptr = data + 7, .data_lines_count = 1, .type =
SEQUENCE};
    // Return.
    data[7] = RET;
}


void single_pool_blocks(struct TemplateBlock *tree, uint8_t max_tree_depth,
struct ParametrizedInstruction *data, uint16_t max_ins_count) {
    tree[0] = (struct TemplateBlock) {.id = 0, .child_id = 1, .parent_id = 0,
.sibling_id = 0, .type = SEQUENCE};

    tree[1] = (struct TemplateBlock) {.id = 1, .child_id = 1, .parent_id = 0,
.sibling_id = 2, .data_start_ptr = data, .data_lines_count = 2, .type =
SEQUENCE};
```

```
    // ADDI t1, zero, ... .
    data[0] = (struct ParametrizedInstruction) {.instruction_id = 18, .pseudo_ins
= false, .output = 6, .output_fixated = true, .first_op = 0, .first_op_fixated =
true};
    // ADDI t2, zero, ... .
    data[1] = (struct ParametrizedInstruction) {.instruction_id = 18, .pseudo_ins
= false, .output = 7, .output_fixated = true, .first_op = 0, .first_op_fixated =
true};

    tree[2] = (struct TemplateBlock) {.id = 2, .child_id = 3, .parent_id = 0,
.sibling_id = 8, .type = POOL, .param = 3, .internal_flags = PICK_BLOCKS};

    tree[3] = (struct TemplateBlock) {.id = 3, .child_id = 3, .parent_id = 2,
.sibling_id = 4, .data_start_ptr = data + 2, .data_lines_count = 2, .type =
SEQUENCE};
    // ADDI t1, t1, ... .
    data[2] = (struct ParametrizedInstruction) {.instruction_id = 18, .pseudo_ins
= false, .output = 6, .output_fixated = true, .first_op = 6, .first_op_fixated =
true, .second_op = 7};
    // ADDI t2, t2, ... .
    data[3] = (struct ParametrizedInstruction) {.instruction_id = 18, .pseudo_ins
= false, .output = 7, .output_fixated = true, .first_op = 7, .first_op_fixated =
true, .second_op = 3};

    tree[4] = (struct TemplateBlock) {.id = 4, .child_id = 4, .parent_id = 2,
.sibling_id = 5, .data_start_ptr = data + 4, .data_lines_count = 2, .type =
SEQUENCE};
    // ADD t1, t2, t1.
    data[4] = (struct ParametrizedInstruction) {.instruction_id = 27, .pseudo_ins
= false, .output = 6, .output_fixated = true, .first_op = 7, .first_op_fixated =
true, .second_op = 6, .second_op_fixated = true};
    // ADD t2, t1, t2.
    data[5] = (struct ParametrizedInstruction) {.instruction_id = 27, .pseudo_ins
= false, .output = 7, .output_fixated = true, .first_op = 6, .first_op_fixated =
true, .second_op = 7, .second_op_fixated = true};

    tree[5] = (struct TemplateBlock) {.id = 5, .child_id = 5, .parent_id = 2,
.sibling_id = 6, .data_start_ptr = data + 6, .data_lines_count = 2, .type =
SEQUENCE};
    // SUB t1, t2, t1.
    data[6] = (struct ParametrizedInstruction) {.instruction_id = 28, .pseudo_ins
= false, .output = 6, .output_fixated = true, .first_op = 7, .first_op_fixated =
true, .second_op = 6, .second_op_fixated = true};
    // SUB t2, t1, t2.
    data[7] = (struct ParametrizedInstruction) {.instruction_id = 28, .pseudo_ins
= false, .output = 7, .output_fixated = true, .first_op = 6, .first_op_fixated =
true, .second_op = 7, .second_op_fixated = true};
```

```
    tree[6] = (struct TemplateBlock) {.id = 6, .child_id = 6, .parent_id = 2,
.sibling_id = 7, .data_start_ptr = data + 8, .data_lines_count = 2, .type =
SEQUENCE};
    // AND t1, t2, t1.
    data[8] = (struct ParametrizedInstruction) {.instruction_id = 36, .pseudo_ins
= false, .output = 6, .output_fixated = true, .first_op = 7, .first_op_fixated =
true, .second_op = 6, .second_op_fixated = true};
    // OR t2, t1, t2.
    data[9] = (struct ParametrizedInstruction) {.instruction_id = 35, .pseudo_ins
= false, .output = 7, .output_fixated = true, .first_op = 6, .first_op_fixated =
true, .second_op = 7, .second_op_fixated = true};

    tree[7] = (struct TemplateBlock) {.id = 7, .child_id = 7, .parent_id = 2,
.sibling_id = 7, .data_start_ptr = data + 10, .data_lines_count = 2, .type =
SEQUENCE};
    // XORI t1, t1, ... .
    data[10] = (struct ParametrizedInstruction) {.instruction_id = 21,
.pseudo_ins = false, .output = 6, .output_fixated = true, .first_op = 6,
.first_op_fixated = true};
    // ORI t2, t1, ... .
    data[11] = (struct ParametrizedInstruction) {.instruction_id = 21,
.pseudo_ins = false, .output = 7, .output_fixated = true, .first_op = 6,
.first_op_fixated = true};

    tree[8] = (struct TemplateBlock) {.id = 8, .child_id = 8, .parent_id = 0,
.sibling_id = 8, .data_start_ptr = data + 12, .data_lines_count = 1, .type =
SEQUENCE};
    // Return.
    data[12] = RET;
}
void single_pool_blocks_without_order(struct TemplateBlock *tree, uint8_t
max_tree_depth, struct ParametrizedInstruction *data, uint16_t max_ins_count) {
    tree[0] = (struct TemplateBlock) {.id = 0, .child_id = 1, .parent_id = 0,
.sibling_id = 0, .type = SEQUENCE};

    tree[1] = (struct TemplateBlock) {.id = 1, .child_id = 1, .parent_id = 0,
.sibling_id = 2, .data_start_ptr = data, .data_lines_count = 2, .type =
SEQUENCE};
    // ADDI t1, zero, ... .
    data[0] = (struct ParametrizedInstruction) {.instruction_id = 18, .pseudo_ins
= false, .output = 6, .output_fixated = true, .first_op = 0, .first_op_fixated =
true};
    // ADDI t2, zero, ... .
    data[1] = (struct ParametrizedInstruction) {.instruction_id = 18, .pseudo_ins
= false, .output = 7, .output_fixated = true, .first_op = 0, .first_op_fixated =
true};

    tree[2] = (struct TemplateBlock) {.id = 2, .child_id = 3, .parent_id = 0,
```

```
.sibling_id = 8, .type = POOL, .param = 3, .internal_flags = PICK_BLOCKS |
NO_ORDER};

    tree[3] = (struct TemplateBlock) {.id = 3, .child_id = 3, .parent_id = 2,
.sibling_id = 4, .data_start_ptr = data + 2, .data_lines_count = 2, .type =
SEQUENCE};
    // ADDI t1, t1, ... .
    data[2] = (struct ParametrizedInstruction) {.instruction_id = 18, .pseudo_ins
= false, .output = 6, .output_fixated = true, .first_op = 6, .first_op_fixated =
true, .second_op = 7};
    // ADDI t2, t2, ... .
    data[3] = (struct ParametrizedInstruction) {.instruction_id = 18, .pseudo_ins
= false, .output = 7, .output_fixated = true, .first_op = 7, .first_op_fixated =
true, .second_op = 3};

    tree[4] = (struct TemplateBlock) {.id = 4, .child_id = 4, .parent_id = 2,
.sibling_id = 5, .data_start_ptr = data + 4, .data_lines_count = 2, .type =
SEQUENCE};
    // ADD t1, t2, t1.
    data[4] = (struct ParametrizedInstruction) {.instruction_id = 27, .pseudo_ins
= false, .output = 6, .output_fixated = true, .first_op = 7, .first_op_fixated =
true, .second_op = 6, .second_op_fixated = true};
    // ADD t2, t1, t2.
    data[5] = (struct ParametrizedInstruction) {.instruction_id = 27, .pseudo_ins
= false, .output = 7, .output_fixated = true, .first_op = 6, .first_op_fixated =
true, .second_op = 7, .second_op_fixated = true};

    tree[5] = (struct TemplateBlock) {.id = 5, .child_id = 5, .parent_id = 2,
.sibling_id = 6, .data_start_ptr = data + 6, .data_lines_count = 2, .type =
SEQUENCE};
    // SUB t1, t2, t1.
    data[6] = (struct ParametrizedInstruction) {.instruction_id = 28, .pseudo_ins
= false, .output = 6, .output_fixated = true, .first_op = 7, .first_op_fixated =
true, .second_op = 6, .second_op_fixated = true};
    // SUB t2, t1, t2.
    data[7] = (struct ParametrizedInstruction) {.instruction_id = 28, .pseudo_ins
= false, .output = 7, .output_fixated = true, .first_op = 6, .first_op_fixated =
true, .second_op = 7, .second_op_fixated = true};

    tree[6] = (struct TemplateBlock) {.id = 6, .child_id = 6, .parent_id = 2,
.sibling_id = 7, .data_start_ptr = data + 8, .data_lines_count = 2, .type =
SEQUENCE};
    // AND t1, t2, t1.
    data[8] = (struct ParametrizedInstruction) {.instruction_id = 36, .pseudo_ins
= false, .output = 6, .output_fixated = true, .first_op = 7, .first_op_fixated =
true, .second_op = 6, .second_op_fixated = true};
    // OR t2, t1, t2.
    data[9] = (struct ParametrizedInstruction) {.instruction_id = 35, .pseudo_ins
```

```
= false, .output = 7, .output_fixated = true, .first_op = 6, .first_op_fixated =
true, .second_op = 7, .second_op_fixated = true};

    tree[7] = (struct TemplateBlock) {.id = 7, .child_id = 7, .parent_id = 2,
.sibling_id = 7, .data_start_ptr = data + 10, .data_lines_count = 2, .type =
SEQUENCE};
    // XORI t1, t1, ... .
    data[10] = (struct ParametrizedInstruction) {.instruction_id = 21,
.pseudo_ins = false, .output = 6, .output_fixated = true, .first_op = 6,
.first_op_fixated = true};
    // ORI t2, t1, ... .
    data[11] = (struct ParametrizedInstruction) {.instruction_id = 21,
.pseudo_ins = false, .output = 7, .output_fixated = true, .first_op = 6,
.first_op_fixated = true};

    tree[8] = (struct TemplateBlock) {.id = 8, .child_id = 8, .parent_id = 0,
.sibling_id = 8, .data_start_ptr = data + 12, .data_lines_count = 1, .type =
SEQUENCE};
    // Return.
    data[12] = RET;
}
void single_pool_blocks_with_repeats(struct TemplateBlock *tree, uint8_t
max_tree_depth, struct ParametrizedInstruction *data, uint16_t max_ins_count) {
    tree[0] = (struct TemplateBlock) {.id = 0, .child_id = 1, .parent_id = 0,
.sibling_id = 0, .type = SEQUENCE};

    tree[1] = (struct TemplateBlock) {.id = 1, .child_id = 1, .parent_id = 0,
.sibling_id = 2, .data_start_ptr = data, .data_lines_count = 2, .type =
SEQUENCE};
    // ADDI t1, zero, ... .
    data[0] = (struct ParametrizedInstruction) {.instruction_id = 18, .pseudo_ins
= false, .output = 6, .output_fixated = true, .first_op = 0, .first_op_fixated =
true};
    // ADDI t2, zero, ... .
    data[1] = (struct ParametrizedInstruction) {.instruction_id = 18, .pseudo_ins
= false, .output = 7, .output_fixated = true, .first_op = 0, .first_op_fixated =
true};

    tree[2] = (struct TemplateBlock) {.id = 2, .child_id = 3, .parent_id = 0,
.sibling_id = 8, .type = POOL, .param = 3, .internal_flags = PICK_BLOCKS |
WITH_REPEATS};

    tree[3] = (struct TemplateBlock) {.id = 3, .child_id = 3, .parent_id = 2,
.sibling_id = 4, .data_start_ptr = data + 2, .data_lines_count = 2, .type =
SEQUENCE};
    // ADDI t1, t1, ... .
    data[2] = (struct ParametrizedInstruction) {.instruction_id = 18, .pseudo_ins
= false, .output = 6, .output_fixated = true, .first_op = 6, .first_op_fixated =
```

```
true, .second_op = 7};
    // ADDI t2, t2, ... .
    data[3] = (struct ParametrizedInstruction) {.instruction_id = 18, .pseudo_ins
= false, .output = 7, .output_fixated = true, .first_op = 7, .first_op_fixated =
true, .second_op = 3};

    tree[4] = (struct TemplateBlock) {.id = 4, .child_id = 4, .parent_id = 2,
.sibling_id = 5, .data_start_ptr = data + 4, .data_lines_count = 2, .type =
SEQUENCE};
    // ADD t1, t2, t1.
    data[4] = (struct ParametrizedInstruction) {.instruction_id = 27, .pseudo_ins
= false, .output = 6, .output_fixated = true, .first_op = 7, .first_op_fixated =
true, .second_op = 6, .second_op_fixated = true};
    // ADD t2, t1, t2.
    data[5] = (struct ParametrizedInstruction) {.instruction_id = 27, .pseudo_ins
= false, .output = 7, .output_fixated = true, .first_op = 6, .first_op_fixated =
true, .second_op = 7, .second_op_fixated = true};

    tree[5] = (struct TemplateBlock) {.id = 5, .child_id = 5, .parent_id = 2,
.sibling_id = 6, .data_start_ptr = data + 6, .data_lines_count = 2, .type =
SEQUENCE};
    // SUB t1, t2, t1.
    data[6] = (struct ParametrizedInstruction) {.instruction_id = 28, .pseudo_ins
= false, .output = 6, .output_fixated = true, .first_op = 7, .first_op_fixated =
true, .second_op = 6, .second_op_fixated = true};
    // SUB t2, t1, t2.
    data[7] = (struct ParametrizedInstruction) {.instruction_id = 28, .pseudo_ins
= false, .output = 7, .output_fixated = true, .first_op = 6, .first_op_fixated =
true, .second_op = 7, .second_op_fixated = true};

    tree[6] = (struct TemplateBlock) {.id = 6, .child_id = 6, .parent_id = 2,
.sibling_id = 7, .data_start_ptr = data + 8, .data_lines_count = 2, .type =
SEQUENCE};
    // AND t1, t2, t1.
    data[8] = (struct ParametrizedInstruction) {.instruction_id = 36, .pseudo_ins
= false, .output = 6, .output_fixated = true, .first_op = 7, .first_op_fixated =
true, .second_op = 6, .second_op_fixated = true};
    // OR t2, t1, t2.
    data[9] = (struct ParametrizedInstruction) {.instruction_id = 35, .pseudo_ins
= false, .output = 7, .output_fixated = true, .first_op = 6, .first_op_fixated =
true, .second_op = 7, .second_op_fixated = true};

    tree[7] = (struct TemplateBlock) {.id = 7, .child_id = 7, .parent_id = 2,
.sibling_id = 7, .data_start_ptr = data + 10, .data_lines_count = 2, .type =
SEQUENCE};
    // XORI t1, t1, ... .
    data[10] = (struct ParametrizedInstruction) {.instruction_id = 21,
.pseudo_ins = false, .output = 6, .output_fixated = true, .first_op = 6,
```

```
.first_op_fixated = true};
    // ORI t2, t1, ... .
    data[11] = (struct ParametrizedInstruction) {.instruction_id = 21,
.pseudo_ins = false, .output = 7, .output_fixated = true, .first_op = 6,
.first_op_fixated = true};

    tree[8] = (struct TemplateBlock) {.id = 8, .child_id = 8, .parent_id = 0,
.sibling_id = 8, .data_start_ptr = data + 12, .data_lines_count = 1, .type =
SEQUENCE};
    // Return.
    data[12] = RET;
}
void single_pool_blocks_with_repeats_without_order(struct TemplateBlock *tree,
uint8_t max_tree_depth, struct ParametrizedInstruction *data, uint16_t
max_ins_count) {
    tree[0] = (struct TemplateBlock) {.id = 0, .child_id = 1, .parent_id = 0,
.sibling_id = 0, .type = SEQUENCE};

    tree[1] = (struct TemplateBlock) {.id = 1, .child_id = 1, .parent_id = 0,
.sibling_id = 2, .data_start_ptr = data, .data_lines_count = 2, .type =
SEQUENCE};
    // ADDI t1, zero, ... .
    data[0] = (struct ParametrizedInstruction) {.instruction_id = 18, .pseudo_ins
= false, .output = 6, .output_fixated = true, .first_op = 0, .first_op_fixated =
true};
    // ADDI t2, zero, ... .
    data[1] = (struct ParametrizedInstruction) {.instruction_id = 18, .pseudo_ins
= false, .output = 7, .output_fixated = true, .first_op = 0, .first_op_fixated =
true};

    tree[2] = (struct TemplateBlock) {.id = 2, .child_id = 3, .parent_id = 0,
.sibling_id = 8, .type = POOL, .param = 3, .internal_flags = PICK_BLOCKS |
WITH_REPEATS | NO_ORDER};

    tree[3] = (struct TemplateBlock) {.id = 3, .child_id = 3, .parent_id = 2,
.sibling_id = 4, .data_start_ptr = data + 2, .data_lines_count = 2, .type =
SEQUENCE};
    // ADDI t1, t1, ... .
    data[2] = (struct ParametrizedInstruction) {.instruction_id = 18, .pseudo_ins
= false, .output = 6, .output_fixated = true, .first_op = 6, .first_op_fixated =
true, .second_op = 7};
    // ADDI t2, t2, ... .
    data[3] = (struct ParametrizedInstruction) {.instruction_id = 18, .pseudo_ins
= false, .output = 7, .output_fixated = true, .first_op = 7, .first_op_fixated =
true, .second_op = 3};

    tree[4] = (struct TemplateBlock) {.id = 4, .child_id = 4, .parent_id = 2,
.sibling_id = 5, .data_start_ptr = data + 4, .data_lines_count = 2, .type =
```

```
SEQUENCE};
    // ADD t1, t2, t1.
    data[4] = (struct ParametrizedInstruction) {.instruction_id = 27, .pseudo_ins
= false, .output = 6, .output_fixated = true, .first_op = 7, .first_op_fixated =
true, .second_op = 6, .second_op_fixated = true};
    // ADD t2, t1, t2.
    data[5] = (struct ParametrizedInstruction) {.instruction_id = 27, .pseudo_ins
= false, .output = 7, .output_fixated = true, .first_op = 6, .first_op_fixated =
true, .second_op = 7, .second_op_fixated = true};

    tree[5] = (struct TemplateBlock) {.id = 5, .child_id = 5, .parent_id = 2,
.sibling_id = 6, .data_start_ptr = data + 6, .data_lines_count = 2, .type =
SEQUENCE};
    // SUB t1, t2, t1.
    data[6] = (struct ParametrizedInstruction) {.instruction_id = 28, .pseudo_ins
= false, .output = 6, .output_fixated = true, .first_op = 7, .first_op_fixated =
true, .second_op = 6, .second_op_fixated = true};
    // SUB t2, t1, t2.
    data[7] = (struct ParametrizedInstruction) {.instruction_id = 28, .pseudo_ins
= false, .output = 7, .output_fixated = true, .first_op = 6, .first_op_fixated =
true, .second_op = 7, .second_op_fixated = true};

    tree[6] = (struct TemplateBlock) {.id = 6, .child_id = 6, .parent_id = 2,
.sibling_id = 7, .data_start_ptr = data + 8, .data_lines_count = 2, .type =
SEQUENCE};
    // AND t1, t2, t1.
    data[8] = (struct ParametrizedInstruction) {.instruction_id = 36, .pseudo_ins
= false, .output = 6, .output_fixated = true, .first_op = 7, .first_op_fixated =
true, .second_op = 6, .second_op_fixated = true};
    // OR t2, t1, t2.
    data[9] = (struct ParametrizedInstruction) {.instruction_id = 35, .pseudo_ins
= false, .output = 7, .output_fixated = true, .first_op = 6, .first_op_fixated =
true, .second_op = 7, .second_op_fixated = true};

    tree[7] = (struct TemplateBlock) {.id = 7, .child_id = 7, .parent_id = 2,
.sibling_id = 7, .data_start_ptr = data + 10, .data_lines_count = 2, .type =
SEQUENCE};
    // XORI t1, t1, ... .
    data[10] = (struct ParametrizedInstruction) {.instruction_id = 21,
.pseudo_ins = false, .output = 6, .output_fixated = true, .first_op = 6,
.first_op_fixated = true};
    // ORI t2, t1, ... .
    data[11] = (struct ParametrizedInstruction) {.instruction_id = 21,
.pseudo_ins = false, .output = 7, .output_fixated = true, .first_op = 6,
.first_op_fixated = true};

    tree[8] = (struct TemplateBlock) {.id = 8, .child_id = 8, .parent_id = 0,
.sibling_id = 8, .data_start_ptr = data + 12, .data_lines_count = 1, .type =
```

```
SEQUENCE};
    // Return.
    data[12] = RET;
}

void basic_template_3blocks(struct TemplateBlock *tree, uint8_t max_tree_depth,
struct ParametrizedInstruction *data, uint16_t max_ins_count) {
    tree[0] = (struct TemplateBlock) {.id = 0, .child_id = 1, .parent_id = 0,
.sibling_id = 0, .type = SEQUENCE};


    tree[1] = (struct TemplateBlock) {.id = 1, .child_id = 1, .parent_id = 0,
.sibling_id = 2, .data_start_ptr = data, .data_lines_count = 3, .type =
SEQUENCE};
    // ADDI t1, zero, ... .
    data[0] = (struct ParametrizedInstruction) {.instruction_id = 18, .pseudo_ins
= false, .output = 6, .output_fixated = true, .first_op = 0, .first_op_fixated =
true};
    // ADDI t2, zero, ... .
    data[1] = (struct ParametrizedInstruction) {.instruction_id = 18, .pseudo_ins
= false, .output = 7, .output_fixated = true, .first_op = 0, .first_op_fixated =
true};
    // ADD t2, t1, t2.
    data[2] = (struct ParametrizedInstruction) {.instruction_id = 27, .pseudo_ins
= false, .output = 7, .output_fixated = true, .first_op = 6, .first_op_fixated =
true, .second_op = 7, .second_op_fixated = true};


    tree[2] = (struct TemplateBlock) {.id = 2, .child_id = 2, .parent_id = 0,
.sibling_id = 3, .data_start_ptr = data + 3, .data_lines_count = 3, .type = POOL,
.param = 2, .internal_flags = WITH_REPEATS | NO_ORDER};
    // ADDI t1, t1, ... .
    data[3] = (struct ParametrizedInstruction) {.instruction_id = 18, .pseudo_ins
= false, .output = 6, .output_fixated = true, .first_op = 6, .first_op_fixated =
true};
    // ADDI t2, t2, ... .
    data[4] = (struct ParametrizedInstruction) {.instruction_id = 18, .pseudo_ins
= false, .output = 7, .output_fixated = true, .first_op = 7, .first_op_fixated =
true};
    // ADD t2, t1, t2.
    data[5] = (struct ParametrizedInstruction) {.instruction_id = 27, .pseudo_ins
= false, .output = 7, .output_fixated = true, .first_op = 6, .first_op_fixated =
true, .second_op = 7, .second_op_fixated = true};


    tree[3] = (struct TemplateBlock) {.id = 3, .child_id = 3, .parent_id = 0,
.sibling_id = 3, .data_start_ptr = data + 6, .data_lines_count = 1, .type =
SEQUENCE};
```

```
    // Return.
    data[6] = RET;
}

void template_test_to_fail(struct TemplateBlock *tree, uint8_t max_tree_depth,
struct ParametrizedInstruction *data, uint16_t max_ins_count) {

    tree[0] = (struct TemplateBlock) {.id = 0, .child_id = 0, .parent_id = 0,
.sibling_id = 0, .data_start_ptr = data, .data_lines_count = 27, .type =
SEQUENCE};

    for (uint8_t index = 6; index < GEN_REG_COUNT; ++index) {
        // ADDI %i, %i, ... .
        data[index - 6] = (struct ParametrizedInstruction) {.instruction_id = 18,
.pseudo_ins = false, .output = index, .output_fixated = true, .first_op = index,
.first_op_fixated = true};
    }

    // Return.
    data[26] = RET;
}

// ------------------------------------------------------------
// ------------------------ASM Tests-------------------------

uint16_t basic_addi(struct ParametrizedInstruction *ins, uint16_t max_ins_count)
{
    // ADDI t1, zero, 7.
    ins[0] = (struct ParametrizedInstruction) {.instruction_id = 18, .pseudo_ins
= false, .output = 6, .output_fixated = true, .first_op = 0, .first_op_fixated =
true, .second_op = 7, .second_op_fixated = true};
    // ADDI t2, zero, 3.
    ins[1] = (struct ParametrizedInstruction) {.instruction_id = 18, .pseudo_ins
= false, .output = 7, .output_fixated = true, .first_op = 0, .first_op_fixated =
true, .second_op = 3, .second_op_fixated = true};
    // ADD t2, t1, t2.
    ins[2] = (struct ParametrizedInstruction) {.instruction_id = 27, .pseudo_ins
= false, .output = 7, .output_fixated = true, .first_op = 6, .first_op_fixated =
true, .second_op = 7, .second_op_fixated = true};
    // Return.
    ins[3] = RET;
    return 4;
}

uint16_t asm_test_to_fail(struct ParametrizedInstruction *ins, uint16_t
max_ins_count) {
    // ADDI t1, zero, 25.
    ins[0] = (struct ParametrizedInstruction) {.instruction_id = 18, .pseudo_ins
```

```
= false, .output = 6, .output_fixated = true, .first_op = 0, .first_op_fixated =
true, .second_op = 25, .second_op_fixated = true};
    return 1;
}

bool_t asm_test_to_fail_eq(ins_t *first_dump, ins_t *second_dump, uint8_t
reg_count) {
    return first_dump[6] == 0 && second_dump[6] == 0;
}

#define RND_LOWER_BORDER 10
#define RND_UPPER_BORDER 51

void default_fill_ins(struct ParametrizedInstruction *instruction) {
    struct PseudoInstruction *pseudo_ins;
    struct Instruction *ins;
    // No loop for a situation (ParametrizedInstruction -> PseudoInstruction ->
PseudoInstruction -> ... -> Instruction).
    if (instruction->pseudo_ins) {
        pseudo_ins = GEN_PSEUDO_INS + instruction->instruction_id;
        ins = GEN_INS + pseudo_ins->instruction_id;
    } else {
        pseudo_ins = NULL;
        ins = GEN_INS + instruction->instruction_id;
    }
    struct InstructionGroup *grp = GEN_INS_GROUPS + ins->group_id;
    if (grp->output && !instruction->output_fixated) {
        instruction->output = (grp->output & IS_REGISTER) ?
linrand(GEN_REG_COUNT) : linrand_intr(RND_LOWER_BORDER, RND_UPPER_BORDER);
    }
    if (grp->first_op && !instruction->first_op_fixated) {
        instruction->first_op = (grp->first_op & IS_REGISTER) ?
linrand(GEN_REG_COUNT) : linrand_intr(RND_LOWER_BORDER, RND_UPPER_BORDER);
    }
    if (grp->second_op && !instruction->second_op_fixated) {
        instruction->second_op = (grp->second_op & IS_REGISTER) ?
linrand(GEN_REG_COUNT) : linrand_intr(RND_LOWER_BORDER, RND_UPPER_BORDER);
    }
}

#endif // GEN_TESTS_C
```

Содержание файла «mutator.c»

```
#ifndef GEN_MUTATOR_C
#define GEN_MUTATOR_C

#include "mut_decls.h"
```

```
uint16_t mutate(struct ParametrizedInstruction *source, uint16_t ins_count,
uint16_t add_ins_count, struct ParametrizedInstruction *destination) {
    uint16_t new_count = ins_count + add_ins_count;
    for (uint16_t index = 0, src_index = 0; index < new_count; ++index) {
        destination[index] = (index - src_index < add_ins_count && (src_index ==
(ins_count - 1) || linrand(2))) ? NOP : source[src_index++];
    }
    return new_count;
}

#endif // GEN_MUTATOR_C
```

Содержание файла «riscv.sh»

```
#!/bin/bash

SCRIPT_DIR=$( cd -- "$( dirname -- "${BASH_SOURCE[0]}" )" &> /dev/null && pwd )

$SCRIPT_DIR/../../../compile.sh $1/"riscv32-unknown-linux-gnu-gcc" "-
march=rv32i;-mabi=ilp32;-Wl,-nostdlib" 32 $SCRIPT_DIR $SCRIPT_DIR/tests
$SCRIPT_DIR/mutators $2 -p
```