from Binary to Binary: How Qemu Works

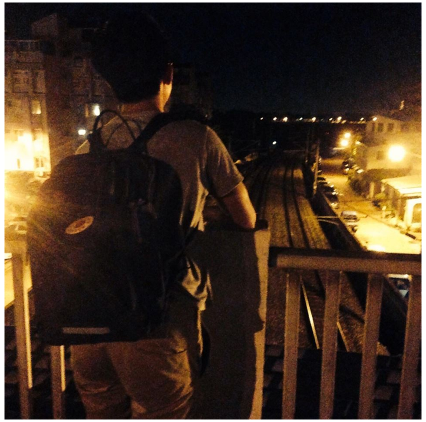
魏禎 (@_zhenwei_) <zhenwei.tw@gmail.com>
林致民 (Doraemon) <r06944005@csie.ntu.edu.tw>

August 11, 2018 / COSCUP 2018

Who are we?

- 林致民 (Doraemon)
    - Master Student @ NTU Compiler Optimization and Virtualization Lab
    - Insterested in compiler optimization and system performance

- 魏禎 (@_zhenwei_)
    - From Tainan, Taiwan
    - Master student @ NTU
    - Interested in Computer Architecture, Virtual Machine and Compiler stuff



Outline

- Introduction of Qemu
- Guest binary to TCG-IR translation
- **Block Chain**ing !
- TCG-IR to x86_64 translation
- Do not cover ...
  - Full system emulation
  - Interrupt handling
  - Multi-thread implementation
  - Optimization ...

What is Qemu

- Created by Fabrice Bellard in 2003

- Features
  - Just-in-time (JIT) compilation support ot achieve high performance
  - Cross-platform (most UNIX-like system and MS-Windows)
  - Lots of target hosts and targets support (full system emulation)
    - x86, aarch32, aarch64, mips, sparc, risc-v
  - User mode emulation: Qemu can run applications compiled for another CPU (same OS)

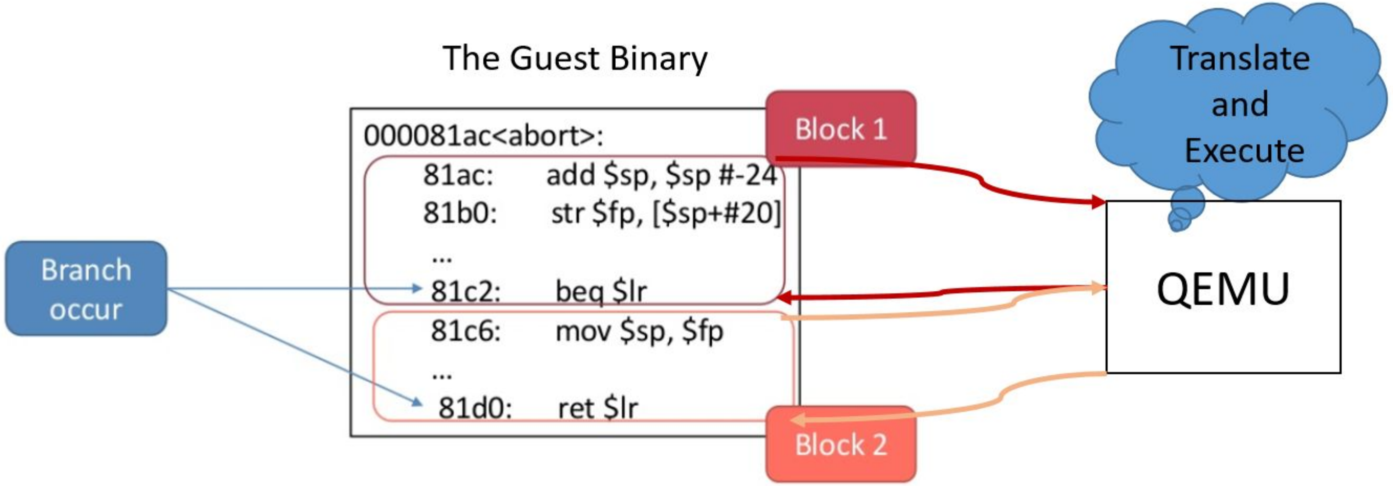
- More excellent slides !
  - Qemu JIT Code Generator and System Emulation
  - QEMU - Binary Translation

Environment

- Guest (target) machine: RISC-V
- Host machine: Intel x86_64
- Tools
    - Qemu 1.12.0 https://www.qemu.org
    - RISC-V GNU Toolchain https://github.com/riscv/riscv-gnu-toolchain.git

Translation Block (tb)

- Definition of translation block (tb)
  - Encounter the branch (modify PC)
  - Encounter the system call
  - Reach the page boundary



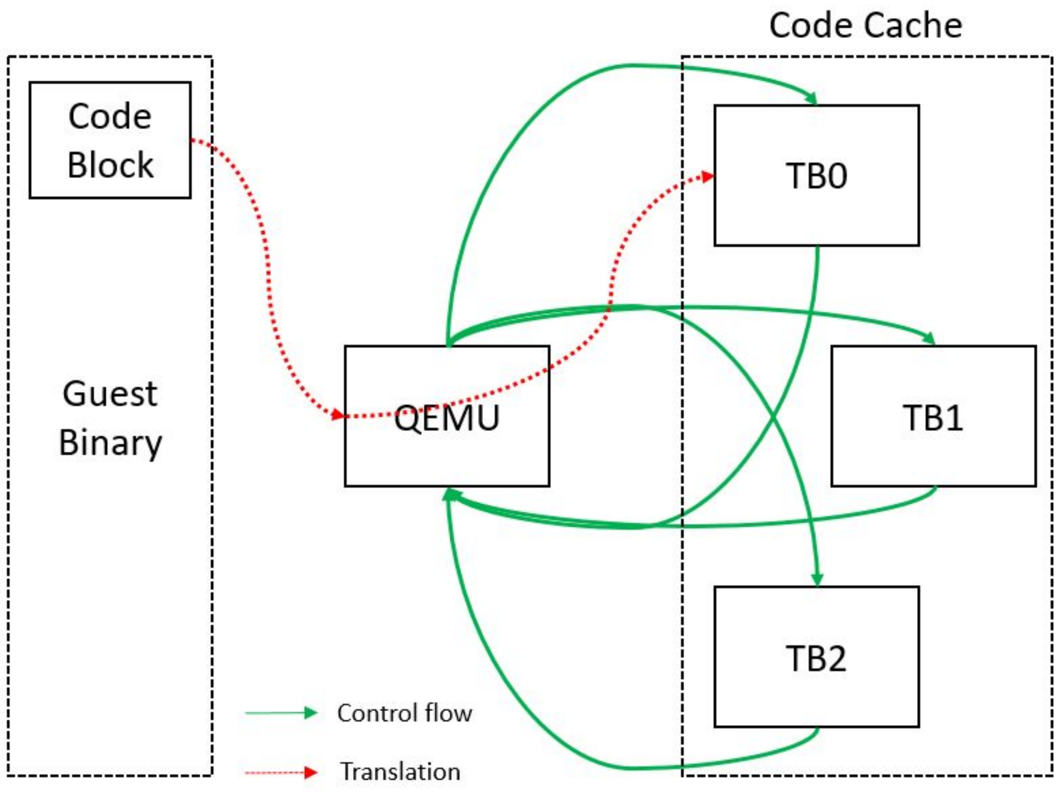
```
000081ac<abort>:
    81ac:    add $sp, $sp #-24
    81b0:    str $fp, [$sp+#20]
    ...
    81c2:    beq $lr
    81c6:    mov $sp, $fp
    ...
    81d0:    ret $lr
```

The picture is referenced from “QEMU - Binary Translation” by Jiann-Fuh Liaw

Dynamic Binary Translation

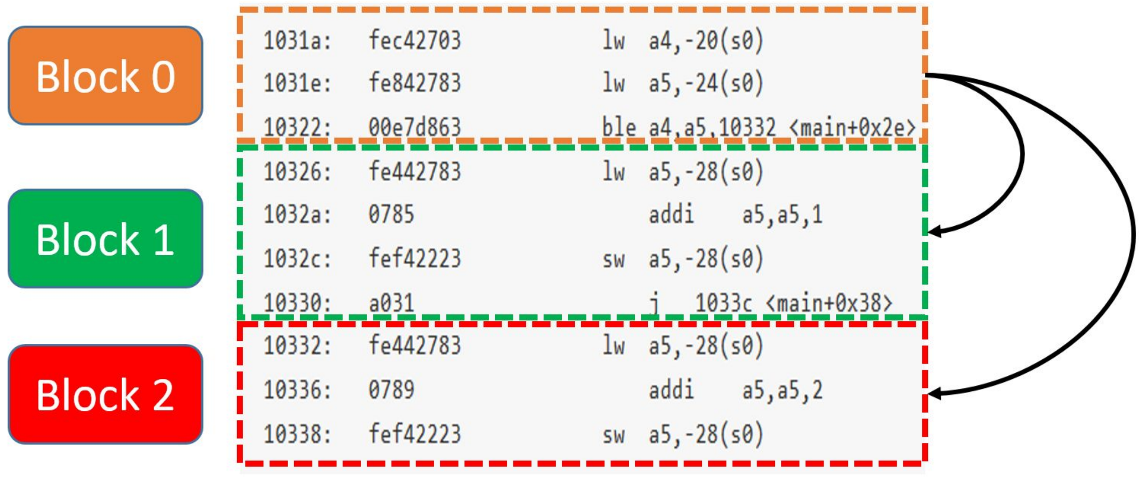
- Translate guest ISA instuction to Host ISA instruction (runtime)

- After the translation block is executed, the control come back to the Qemu

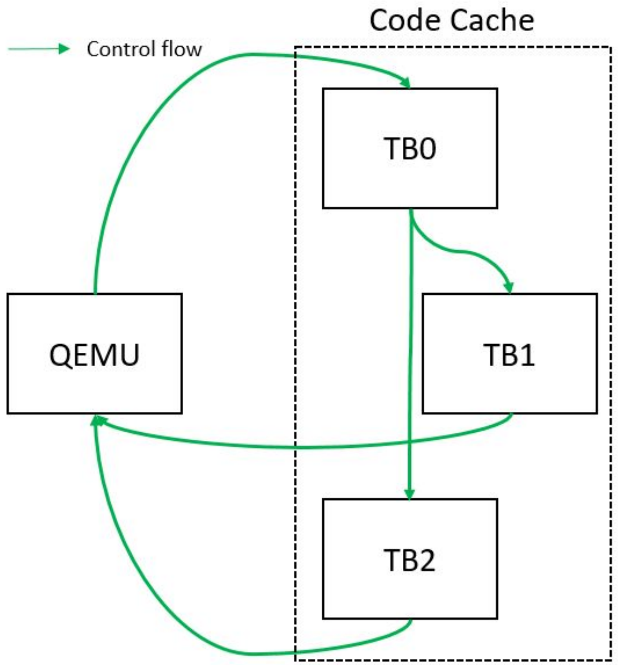


Dynamic Binary Translation

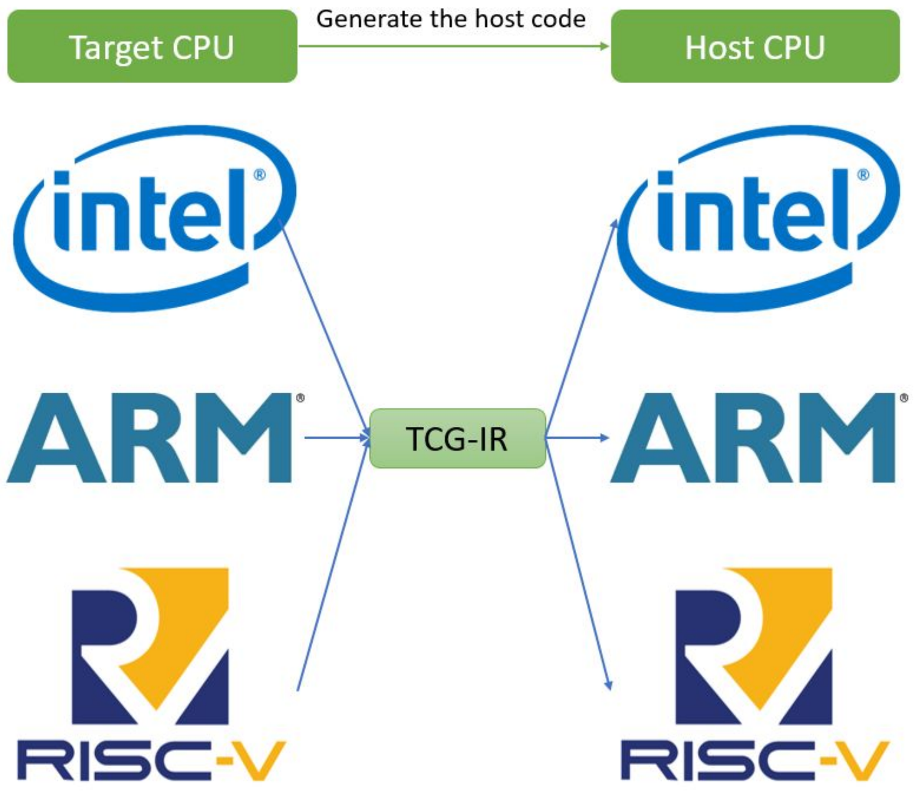
- Block Chaining - avoid the "context switching" overhead

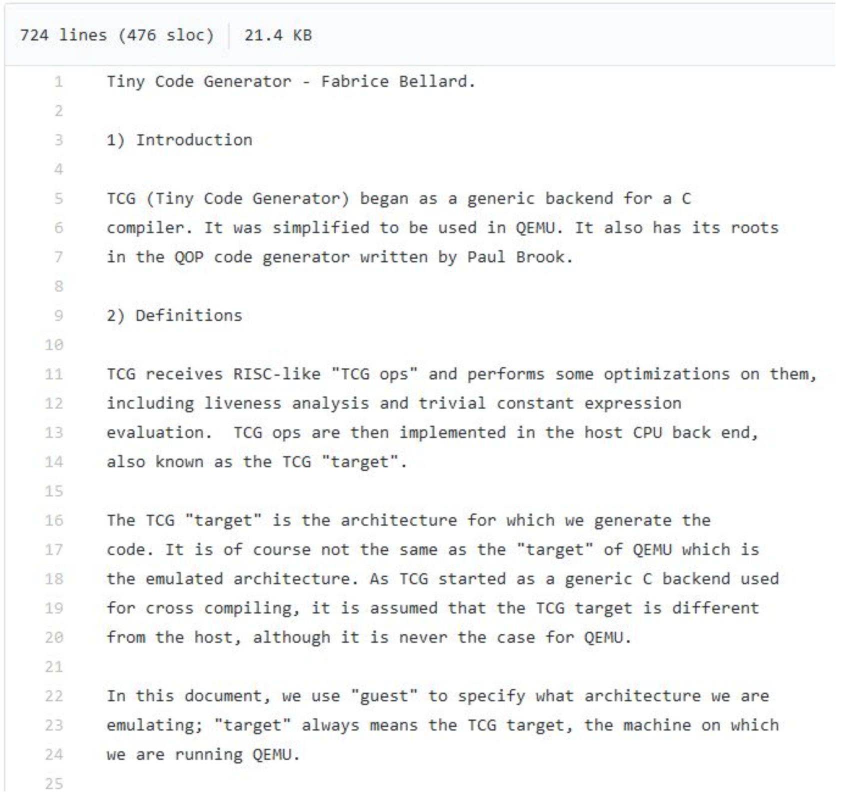


```
1031a:   fec42703          lw  a4,-20(s0)
1031e:   fe842783          lw  a5,-24(s0)
10322:   00e7d863          ble a4,a5,10332 <main+0x2e>
10326:   fe442783          lw  a5,-28(s0)
1032a:   0785                  addi    a5,a5,1
1032c:   fef42223          sw  a5,-28(s0)
10330:   a031                    j   1033c <main+0x38>
10332:   fe442783          lw  a5,-28(s0)
10336:   0789                  addi    a5,a5,2
10338:   fef42223          sw  a5,-28(s0)
```



Tiny Code Generator (TCG)





```
724 lines (476 sloc) | 21.4 KB

Tiny Code Generator - Fabrice Bellard.

1) Introduction

TCG (Tiny Code Generator) began as a generic backend for a C
compiler. It was simplified to be used in QEMU. It also has its roots
in the QOP code generator written by Paul Brook.

2) Definitions

TCG receives RISC-like "TCG ops" and performs some optimizations on them,
including liveness analysis and trivial constant expression
evaluation.  TCG ops are then implemented in the host CPU back end,
also known as the TCG "target".

The TCG "target" is the architecture for which we generate the
code. It is of course not the same as the "target" of QEMU which is
the emulated architecture. As TCG started as a generic C backend used
for cross compiling, it is assumed that the TCG target is different
from the host, although it is never the case for QEMU.

In this document, we use "guest" to specify what architecture we are
emulating; "target" always means the TCG target, the machine on which
we are running QEMU.
```

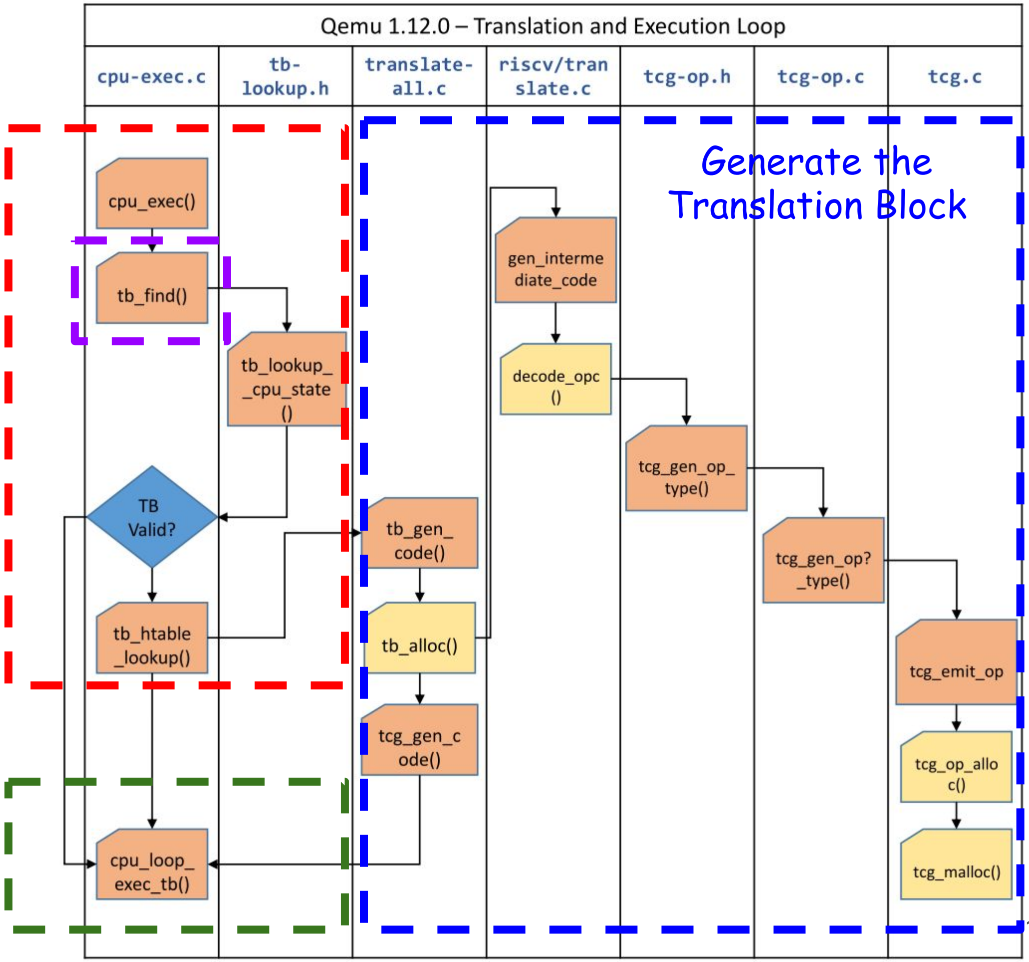
It is referenced from tcg/README

Qemu Execution Flow

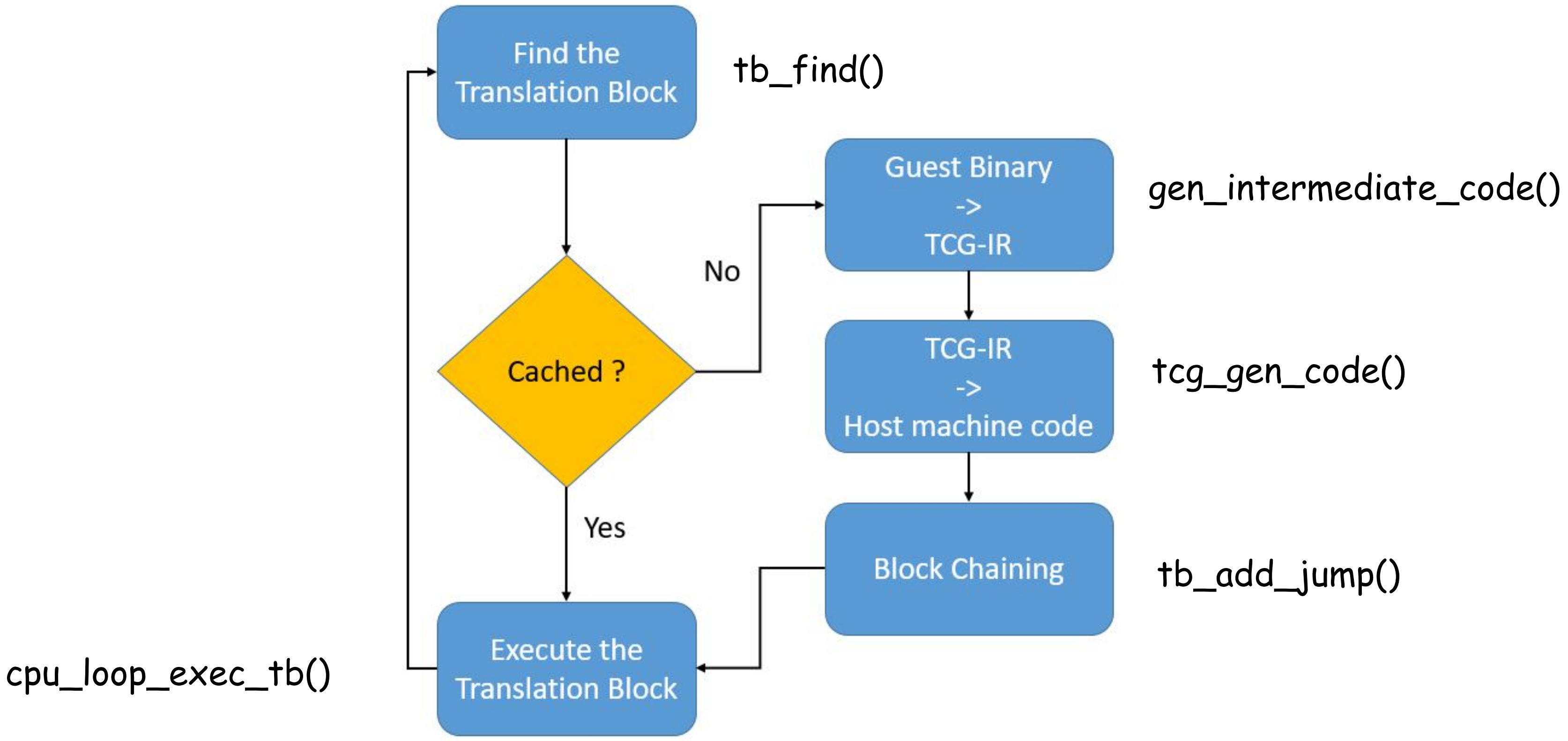
Chain the generated Translation Block to existed Block

Find the Translation Block

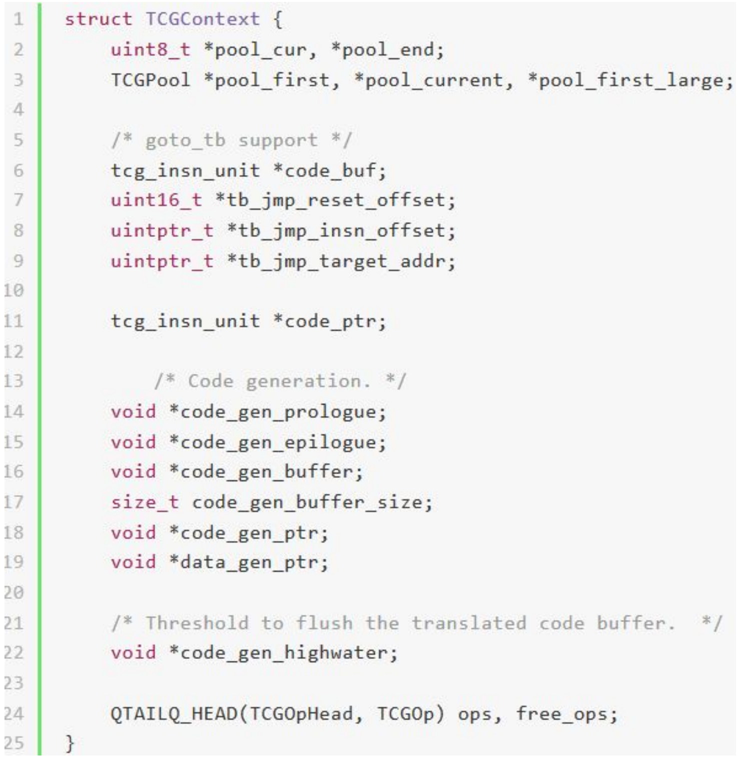
Execute the Translation Block



Qemu Execution Flow

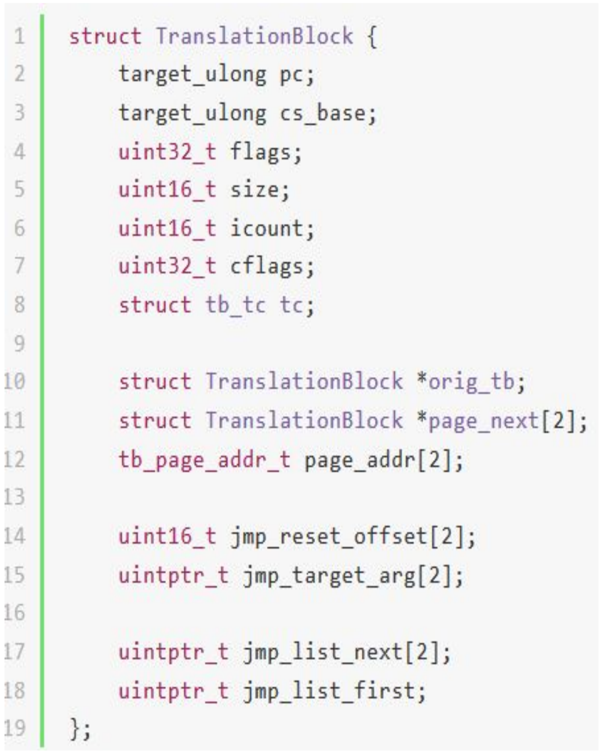


Data structures would be used later ...



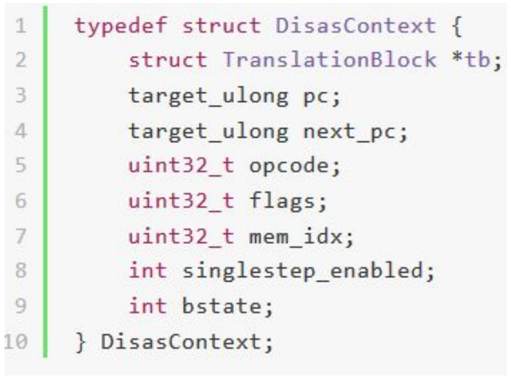
```c
struct TCGContext {
    uint8_t *pool_cur, *pool_end;
    TCGPool *pool_first, *pool_current, *pool_first_large;

    /* goto_tb support */
    tcg_insn_unit *code_buf;
    uint16_t *tb_jmp_reset_offset;
    uintptr_t *tb_jmp_insn_offset;
    uintptr_t *tb_jmp_target_addr;

    tcg_insn_unit *code_ptr;

        /* Code generation. */
    void *code_gen_prologue;
    void *code_gen_epilogue;
    void *code_gen_buffer;
    size_t code_gen_buffer_size;
    void *code_gen_ptr;
    void *data_gen_ptr;

    /* Threshold to flush the translated code buffer.  */
    void *code_gen_highwater;

    QTAILQ_HEAD(TCGOpHead, TCGOp) ops, free_ops;
}
```

TCGContext



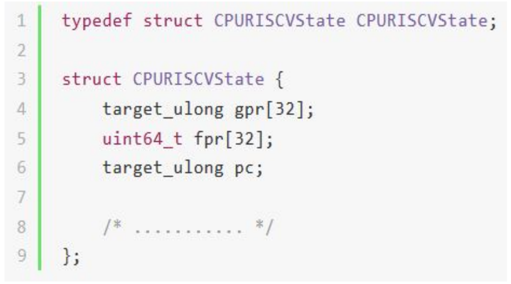
```c
struct TranslationBlock {
    target_ulong pc;
    target_ulong cs_base;
    uint32_t flags;
    uint16_t size;
    uint16_t icount;
    uint32_t cflags;
    struct tb_tc tc;

    struct TranslationBlock *orig_tb;
    struct TranslationBlock *page_next[2];
    tb_page_addr_t page_addr[2];

    uint16_t jmp_reset_offset[2];
    uintptr_t jmp_target_arg[2];

    uintptr_t jmp_list_next[2];
    uintptr_t jmp_list_first;
};
```

TranslationBlock



```c
typedef struct DisasContext {
    struct TranslationBlock *tb;
    target_ulong pc;
    target_ulong next_pc;
    uint32_t opcode;
    uint32_t flags;
    uint32_t mem_idx;
    int singlestep_enabled;
    int bstate;
} DisasContext;
```

DisasContext



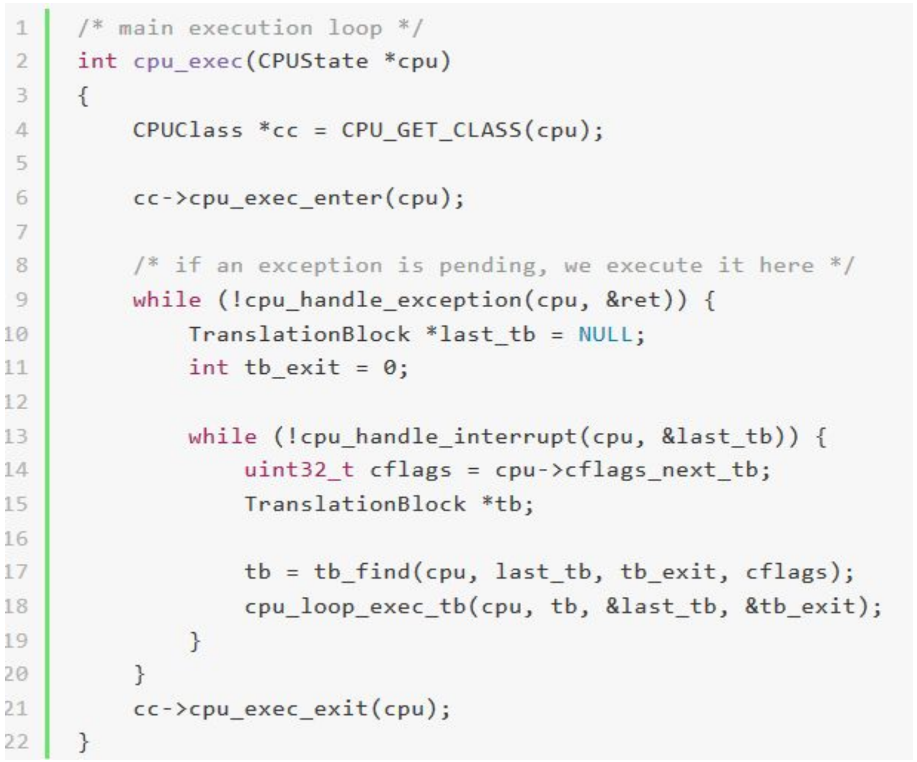
```c
typedef struct CPURISCVState CPURISCVState;

struct CPURISCVState {
    target_ulong gpr[32];
    uint64_t fpr[32];
    target_ulong pc;

    /* .......... */
};
```

CPURISCVState

The main loop

cpu_exec() @ accel/tcg/cpu-exec.c

- *tb_find()*
  - Find the desired translation block by pc value
- *cpu_loop_exec_tb()*
  - Execute the native code in the translation block

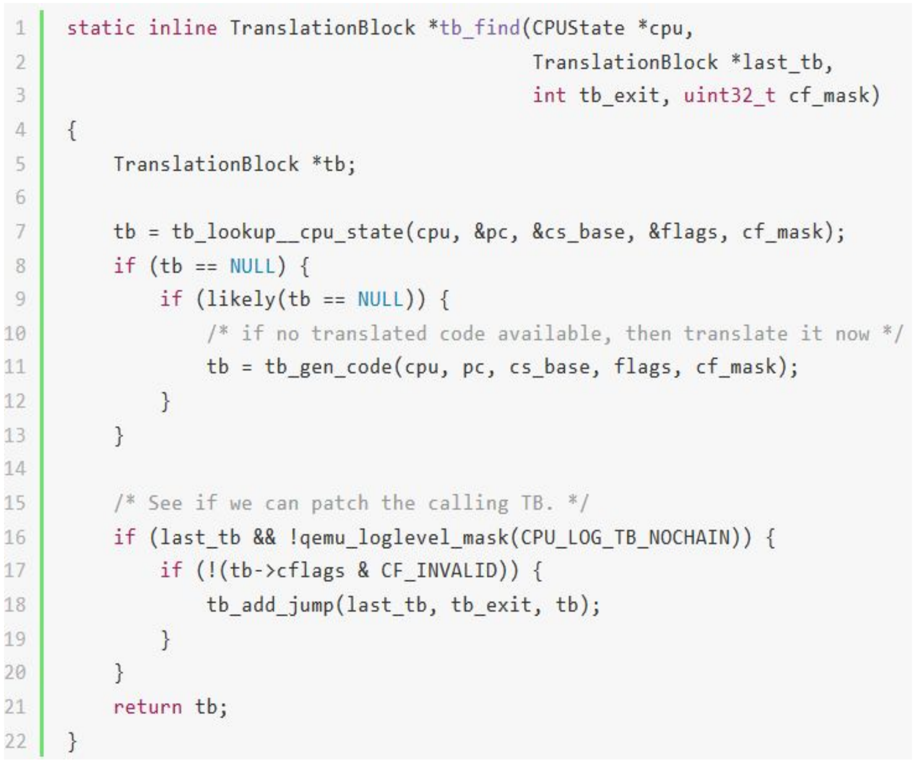


```c
/* main execution loop */
int cpu_exec(CPUState *cpu)
{
    CPUClass *cc = CPU_GET_CLASS(cpu);

    cc->cpu_exec_enter(cpu);

    /* if an exception is pending, we execute it here */
    while (!cpu_handle_exception(cpu, &ret)) {
        TranslationBlock *last_tb = NULL;
        int tb_exit = 0;

        while (!cpu_handle_interrupt(cpu, &last_tb)) {
            uint32_t cflags = cpu->cflags_next_tb;
            TranslationBlock *tb;

            tb = tb_find(cpu, last_tb, tb_exit, cflags);
            cpu_loop_exec_tb(cpu, tb, &last_tb, &tb_exit);
        }
    }
    cc->cpu_exec_exit(cpu);
}
```

Find the desired Translation Block or create one

tb_find() @ accel/tcg/cpu-exec.c

- *tb_lookup__cpu_state()*
  - Find the specific tb (Translation Block) by pc value
- *tb_gen_code()*
  - If the desired tb hasn't been generated yet, we just create one
- *tb_add_jump()*
  - The block chaining patch point!
  - We will talk about it later ...

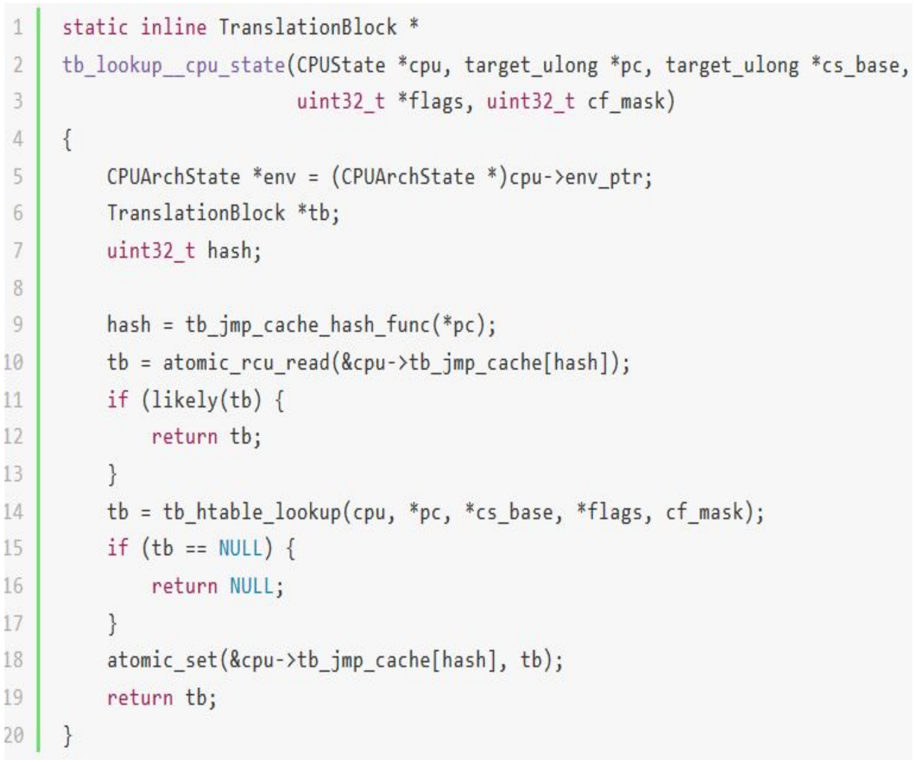


```c
static inline TranslationBlock *tb_find(CPUState *cpu,
                                        TranslationBlock *last_tb,
                                        int tb_exit, uint32_t cf_mask)
{
    TranslationBlock *tb;

    tb = tb_lookup__cpu_state(cpu, &pc, &cs_base, &flags, cf_mask);
    if (tb == NULL) {
        if (likely(tb == NULL)) {
            /* if no translated code available, then translate it now */
            tb = tb_gen_code(cpu, pc, cs_base, flags, cf_mask);
        }
    }

    /* See if we can patch the calling TB. */
    if (last_tb && !qemu_loglevel_mask(CPU_LOG_TB_NOCHAIN)) {
        if (!(tb->cflags & CF_INVALID)) {
            tb_add_jump(last_tb, tb_exit, tb);
        }
    }
    return tb;
}
```

The Translation Block Finding Algorithm

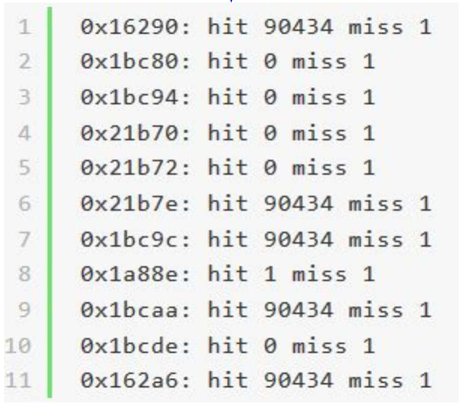
tb_lookup__cpu_state() @ include/exec/tb-lookup.h

- *tb_jmp_cache_hash_func()*
  - A level-1 lookup cache is implemented (fast path)
- *tb_htable_lookup()*
  - A traditional hash table is used to find the specific tb by hash value (slow path)
  - Update the level-1 lookup cache if found it



```
static inline TranslationBlock *
tb_lookup__cpu_state(CPUState *cpu, target_ulong *pc, target_ulong *cs_base,
                     uint32_t *flags, uint32_t cf_mask)
{
    CPUArchState *env = (CPUArchState *)cpu->env_ptr;
    TranslationBlock *tb;
    uint32_t hash;

    hash = tb_jmp_cache_hash_func(*pc);
    tb = atomic_rcu_read(&cpu->tb_jmp_cache[hash]);
    if (likely(tb) {
        return tb;
    }
    tb = tb_htable_lookup(cpu, *pc, *cs_base, *flags, cf_mask);
    if (tb == NULL) {
        return NULL;
    }
    atomic_set(&cpu->tb_jmp_cache[hash], tb);
    return tb;
}
```

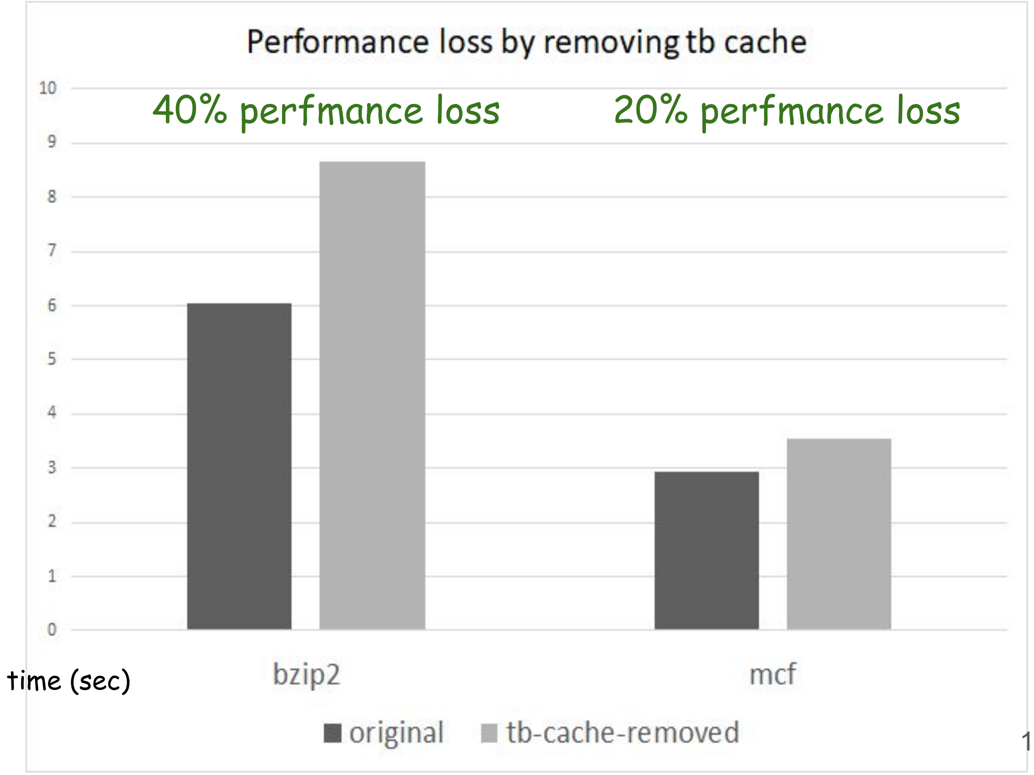
The Translation Block Finding Algorithm



```
0x16290: hit 90434 miss 1
0x1bc80: hit 0 miss 1
0x1bc94: hit 0 miss 1
0x21b70: hit 0 miss 1
0x21b72: hit 0 miss 1
0x21b7e: hit 90434 miss 1
0x1bc9c: hit 90434 miss 1
0x1a88e: hit 1 miss 1
0x1bcaa: hit 90434 miss 1
0x1bcde: hit 0 miss 1
0x162a6: hit 90434 miss 1
```

Some tbs would be executed many times throughout the program

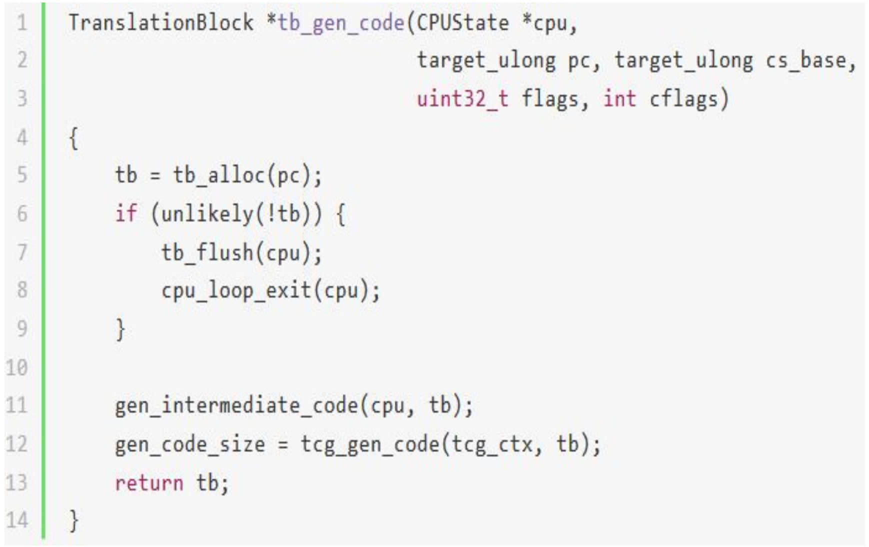
	bzip2	mcf
# of tb executed	32877233	8125325
tb-cache miss ratio	0.01 %	2.10 %



Start to generate the Translation Block

tb_gen_code() @ accel/tcg/translate-all.c

- *tb_alloc()*
  - This function would allocate the space from the Code Cache in TCGContext
  - If the Code Cache is full, just flush it !
- *gen_intermediate_code()*
  - We will get the TCG-IR produced in this function, which is stored in the TCGContext
- *tcg_gen_code()*
  - Generate the host machine code according to the TCG-IR, and it would be stored in the tb



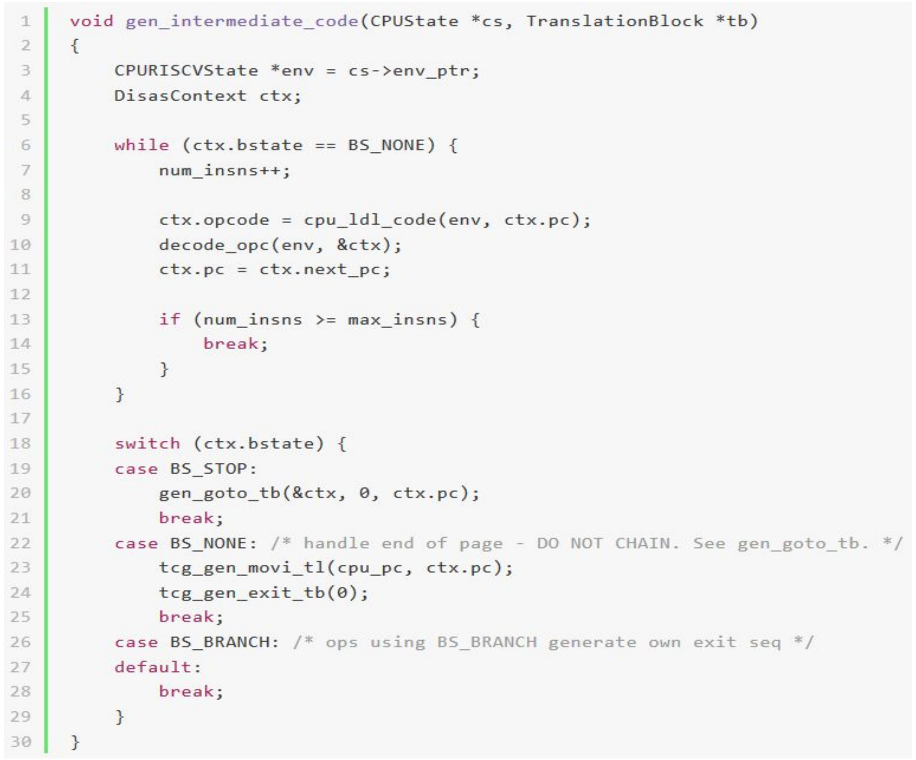
```
TranslationBlock *tb_gen_code(CPUState *cpu,
                              target_ulong pc, target_ulong cs_base,
                              uint32_t flags, int cflags)
{
    tb = tb_alloc(pc);
    if (unlikely(!tb)) {
        tb_flush(cpu);
        cpu_loop_exit(cpu);
    }

    gen_intermediate_code(cpu, tb);
    gen_code_size = tcg_gen_code(tcg_ctx, tb);
    return tb;
}
```

Generate the TCG-IR first !

gen_intermediate_code() @ target/riscv/translate.c

- The while loop would decode each instruction in the guest binary until encounter the branch or reach the tb size

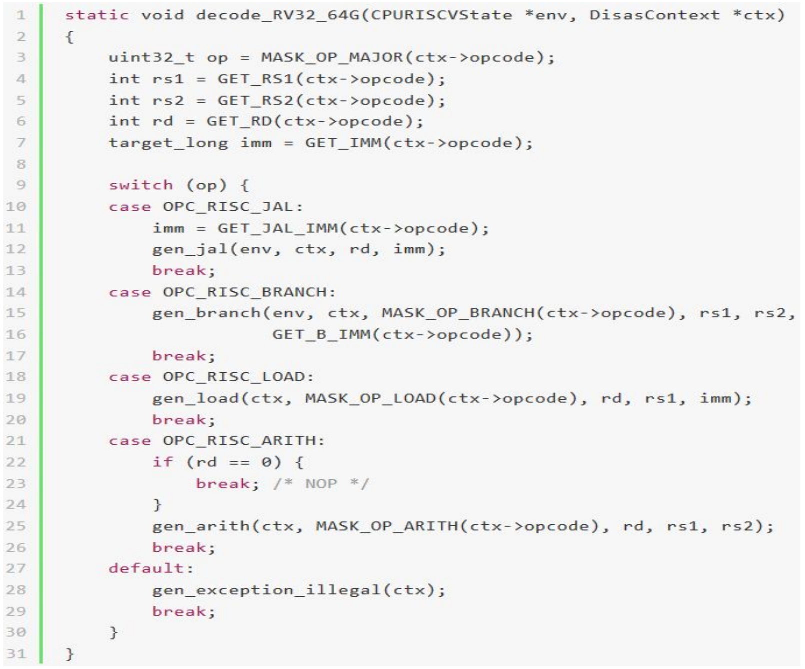
- *decode_opc()*
  - Decode the instruction in binary form



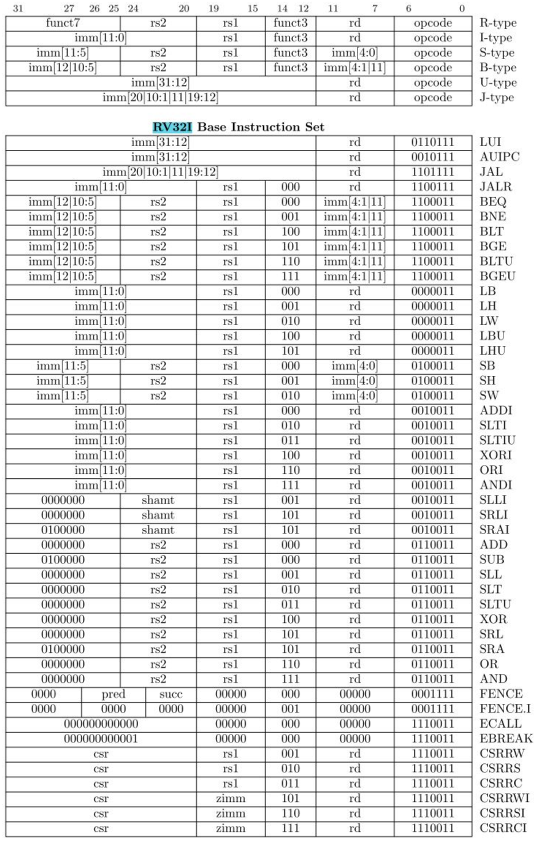
```
void gen_intermediate_code(CPUState *cs, TranslationBlock *tb)
{
    CPURISCVState *env = cs->env_ptr;
    DisasContext ctx;

    while (ctx.bstate == BS_NONE) {
        num_insns++;

        ctx.opcode = cpu_ldl_code(env, ctx.pc);
        decode_opc(env, &ctx);
        ctx.pc = ctx.next_pc;

        if (num_insns >= max_insns) {
            break;
        }
    }

    switch (ctx.bstate) {
    case BS_STOP:
        gen_goto_tb(&ctx, 0, ctx.pc);
        break;
    case BS_NONE: /* handle end of page - DO NOT CHAIN. See gen_goto_tb. */
        tcg_gen_movi_tl(cpu_pc, ctx.pc);
        tcg_gen_exit_tb(0);
        break;
    case BS_BRANCH: /* ops using BS_BRANCH generate own exit seq */
    default:
        break;
    }
}
```

Decode the instruction in guest binary

decode_RV32_64G() @
target/riscv/translate.c



```c
static void decode_RV32_64G(CPURISCVState *env, DisasContext *ctx)
{
    uint32_t op = MASK_OP_MAJOR(ctx->opcode);
    int rs1 = GET_RS1(ctx->opcode);
    int rs2 = GET_RS2(ctx->opcode);
    int rd = GET_RD(ctx->opcode);
    target_long imm = GET_IMM(ctx->opcode);

    switch (op) {
    case OPC_RISC_JAL:
        imm = GET_JAL_IMM(ctx->opcode);
        gen_jal(env, ctx, rd, imm);
        break;
    case OPC_RISC_BRANCH:
        gen_branch(env, ctx, MASK_OP_BRANCH(ctx->opcode), rs1, rs2,
                   GET_B_IMM(ctx->opcode));
        break;
    case OPC_RISC_LOAD:
        gen_load(ctx, MASK_OP_LOAD(ctx->opcode), rd, rs1, imm);
        break;
    case OPC_RISC_ARITH:
        if (rd == 0) {
            break; /* NOP */
        }
        gen_arith(ctx, MASK_OP_ARITH(ctx->opcode), rd, rs1, rs2);
        break;
    default:
        gen_exception_illegal(ctx);
        break;
    }
}
```



31 27 26 25	24 20	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]				rd	opcode	U-type
imm[20|10:1|11|19:12]				rd	opcode	J-type

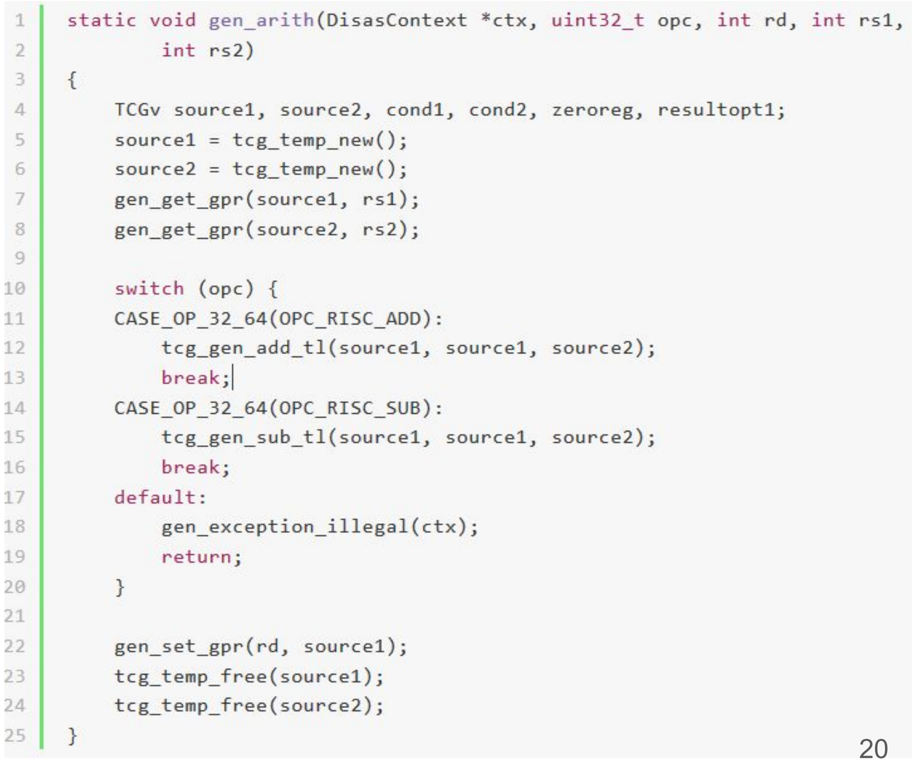
RV32I Base Instruction Set

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND
0000 pred succ		00000	000	00000	0001111	FENCE
0000 0000 0000		00000	001	00000	0001111	FENCE.I
000000000000		00000	000	00000	1110011	ECALL
000000000001		00000	000	00000	1110011	EBREAK
csr		rs1	001	rd	1110011	CSRRW
csr		rs1	010	rd	1110011	CSRRS
csr		rs1	011	rd	1110011	CSRRC
csr		zimm	101	rd	1110011	CSRRWI
csr		zimm	110	rd	1110011	CSRRSI
csr		zimm	111	rd	1110011	CSRRCI

Generate the TCG-IR first ! (E.g. arithmetic instr.)

gen_arith() @ target/riscv/translate.c

- The guest instruction need to be implemented in the TCG variable system
- The TCG variables are declared and can be assigned the value from the architecture state or constants
- The TCG frontend ops would operate on these TCG variables.

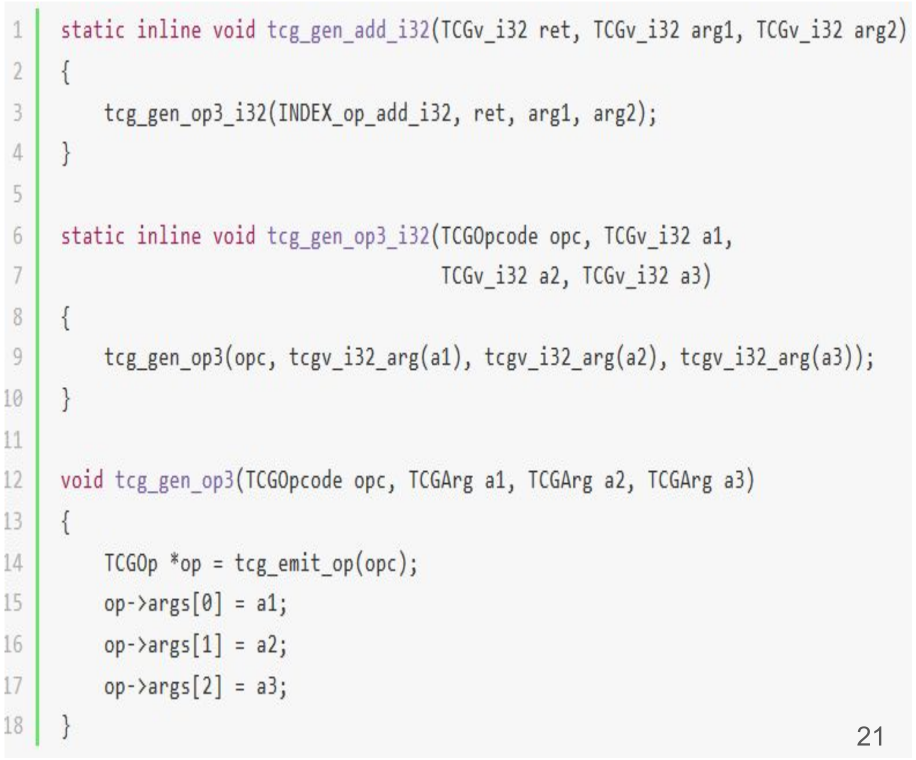


```c
static void gen_arith(DisasContext *ctx, uint32_t opc, int rd, int rs1,
        int rs2)
{
    TCGv source1, source2, cond1, cond2, zeroreg, resultopt1;
    source1 = tcg_temp_new();
    source2 = tcg_temp_new();
    gen_get_gpr(source1, rs1);
    gen_get_gpr(source2, rs2);

    switch (opc) {
    CASE_OP_32_64(OPC_RISC_ADD):
        tcg_gen_add_tl(source1, source1, source2);
        break;
    CASE_OP_32_64(OPC_RISC_SUB):
        tcg_gen_sub_tl(source1, source1, source2);
        break;
    default:
        gen_exception_illegal(ctx);
        return;
    }

    gen_set_gpr(rd, source1);
    tcg_temp_free(source1);
    tcg_temp_free(source2);
}
```

Generate the TCG-IR first ! (E.g. arithmetic instr.)

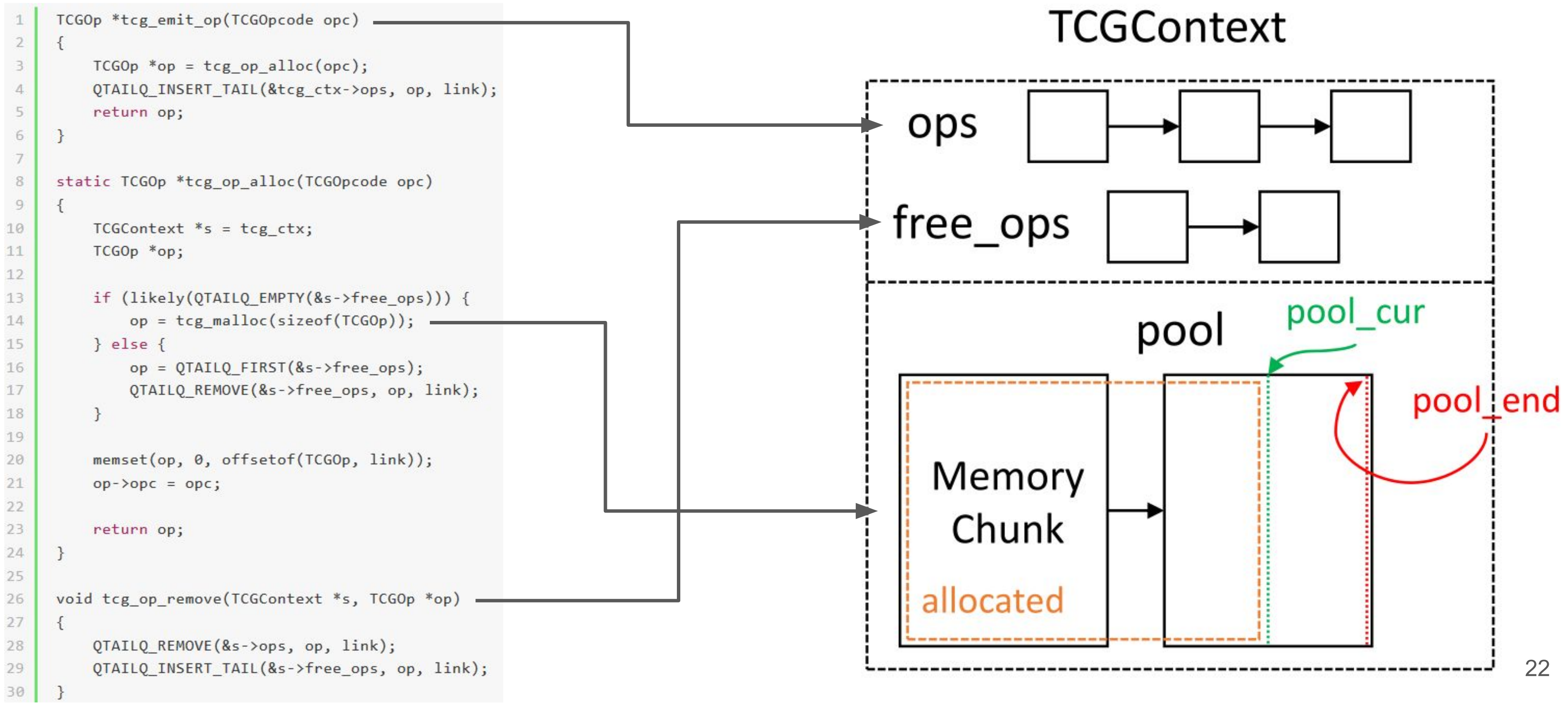
tcg_gen_? @ tcg/tcg-op.h & tcg/tcg-op.c

- *tcg_emit_op()*
  - This function will allocate the space from TCGContext and insert it into the ops linked-list
- After getting the allocated space, the tcg opcode and arguments are filled into it.
- The TCG instrution generated would be showed later ...



```
static inline void tcg_gen_add_i32(TCGv_i32 ret, TCGv_i32 arg1, TCGv_i32 arg2)
{
    tcg_gen_op3_i32(INDEX_op_add_i32, ret, arg1, arg2);
}

static inline void tcg_gen_op3_i32(TCGOpcode opc, TCGv_i32 a1,
                                   TCGv_i32 a2, TCGv_i32 a3)
{
    tcg_gen_op3(opc, tcgv_i32_arg(a1), tcgv_i32_arg(a2), tcgv_i32_arg(a3));
}

void tcg_gen_op3(TCGOpcode opc, TCGArg a1, TCGArg a2, TCGArg a3)
{
    TCGOp *op = tcg_emit_op(opc);
    op->args[0] = a1;
    op->args[1] = a2;
    op->args[2] = a3;
}
```

More about *tcg_emit_op()*

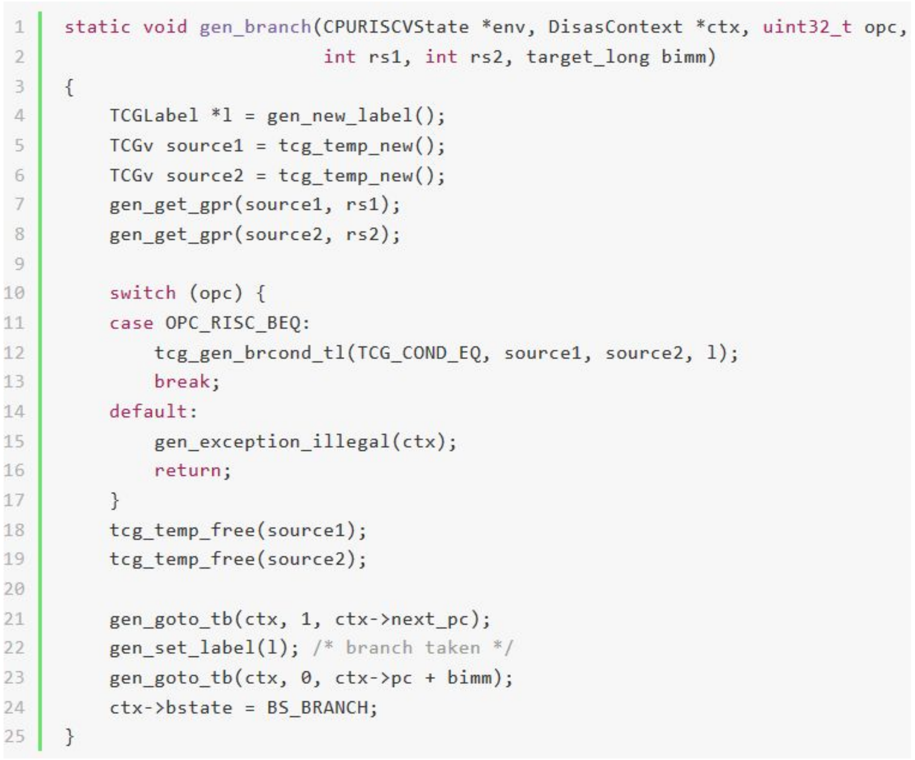


```
TCGOp *tcg_emit_op(TCGOpcode opc)
{
    TCGOp *op = tcg_op_alloc(opc);
    QTAILQ_INSERT_TAIL(&tcg_ctx->ops, op, link);
    return op;
}

static TCGOp *tcg_op_alloc(TCGOpcode opc)
{
    TCGContext *s = tcg_ctx;
    TCGOp *op;

    if (likely(QTAILQ_EMPTY(&s->free_ops))) {
        op = tcg_malloc(sizeof(TCGOp));
    } else {
        op = QTAILQ_FIRST(&s->free_ops);
        QTAILQ_REMOVE(&s->free_ops, op, link);
    }

    memset(op, 0, offsetof(TCGOp, link));
    op->opc = opc;

    return op;
}

void tcg_op_remove(TCGContext *s, TCGOp *op)
{
    QTAILQ_REMOVE(&s->ops, op, link);
    QTAILQ_INSERT_TAIL(&s->free_ops, op, link);
}
```

Generate the TCG-IR first ! (E.g. branch instr.)

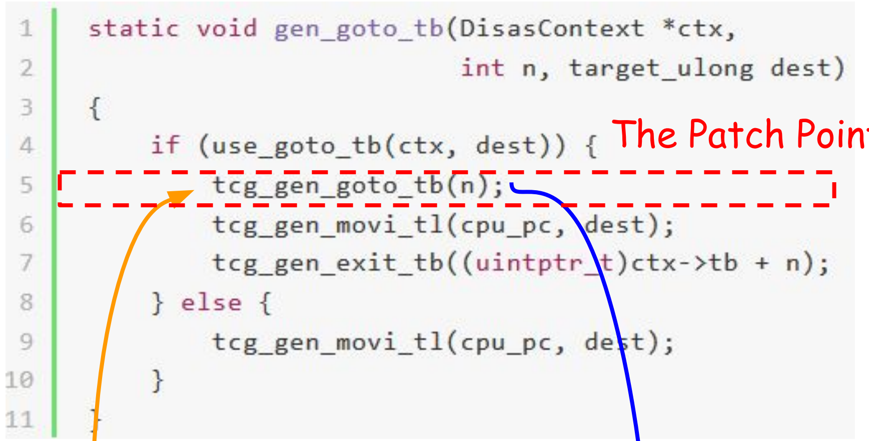
gen_branch() @ target/riscv/translate.c

- The Label also needed to generated via TCG-IR form
- *gen_goto_tb()*
  - Jump into the specific translation block !
- When encountered the branch, which means it the end of the tb.
  - The *ctx->bstate* is set to break the outer while loop in *gen_intermediate_code()*



```c
static void gen_branch(CPURISCVState *env, DisasContext *ctx, uint32_t opc,
                       int rs1, int rs2, target_long bimm)
{
    TCGLabel *l = gen_new_label();
    TCGv source1 = tcg_temp_new();
    TCGv source2 = tcg_temp_new();
    gen_get_gpr(source1, rs1);
    gen_get_gpr(source2, rs2);

    switch (opc) {
    case OPC_RISC_BEQ:
        tcg_gen_brcond_tl(TCG_COND_EQ, source1, source2, l);
        break;
    default:
        gen_exception_illegal(ctx);
        return;
    }
    tcg_temp_free(source1);
    tcg_temp_free(source2);

    gen_goto_tb(ctx, 1, ctx->next_pc);
    gen_set_label(l); /* branch taken */
    gen_goto_tb(ctx, 0, ctx->pc + bimm);
    ctx->bstate = BS_BRANCH;
}
```

How **Block Chain**ing works?

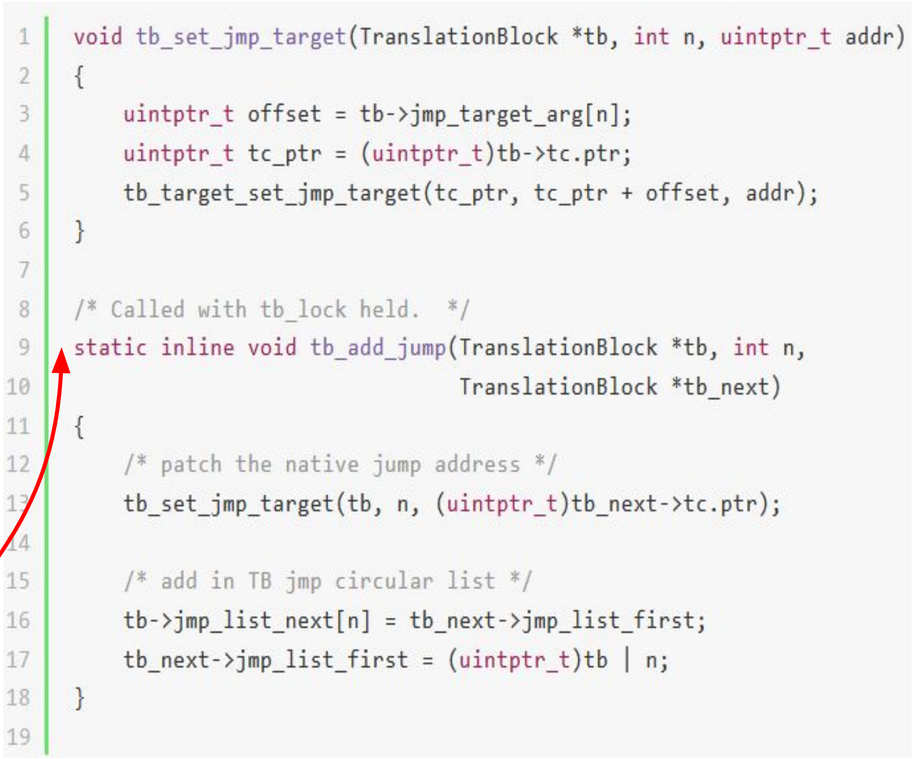


```c
static void gen_goto_tb(DisasContext *ctx,
                        int n, target_ulong dest)
{
    if (use_goto_tb(ctx, dest)) {
        tcg_gen_goto_tb(n);
        tcg_gen_movi_tl(cpu_pc, dest);
        tcg_gen_exit_tb((uintptr_t)ctx->tb + n);
    } else {
        tcg_gen_movi_tl(cpu_pc, dest);
    }
}
```

The Patch Point

A slot waits for patching.
If not patched yet, it would just jump to the next instruction

The location of this slot would be recorded in the *tb->jump_target_arg* when generating host machine code



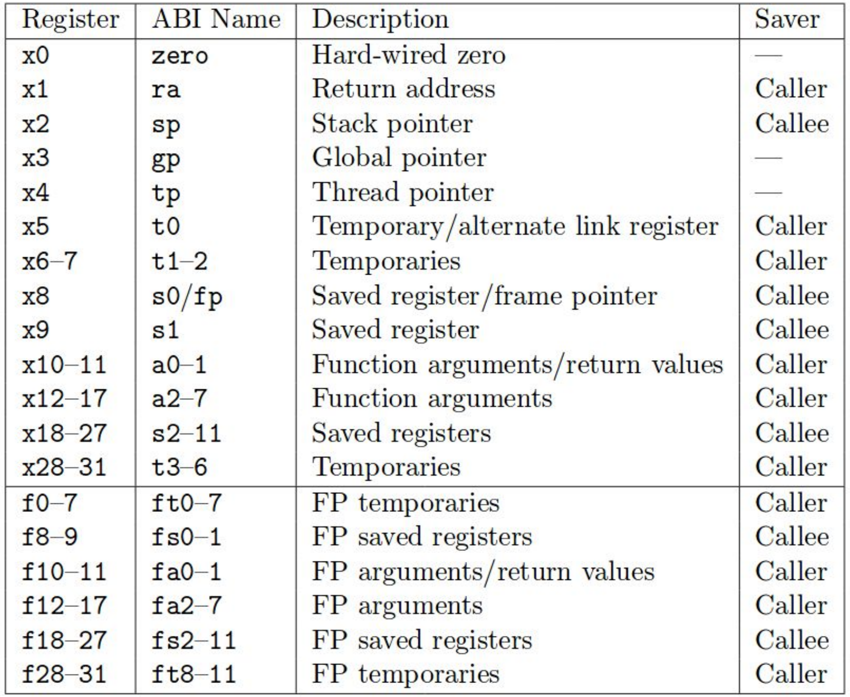
```c
void tb_set_jmp_target(TranslationBlock *tb, int n, uintptr_t addr)
{
    uintptr_t offset = tb->jmp_target_arg[n];
    uintptr_t tc_ptr = (uintptr_t)tb->tc.ptr;
    tb_target_set_jmp_target(tc_ptr, tc_ptr + offset, addr);
}

/* Called with tb_lock held.  */
static inline void tb_add_jump(TranslationBlock *tb, int n,
                               TranslationBlock *tb_next)
{
    /* patch the native jump address */
    tb_set_jmp_target(tb, n, (uintptr_t)tb_next->tc.ptr);

    /* add in TB jmp circular list */
    tb->jmp_list_next[n] = tb_next->jmp_list_first;
    tb_next->jmp_list_first = (uintptr_t)tb | n;
}
```

The generating tb will patch the last executed tb

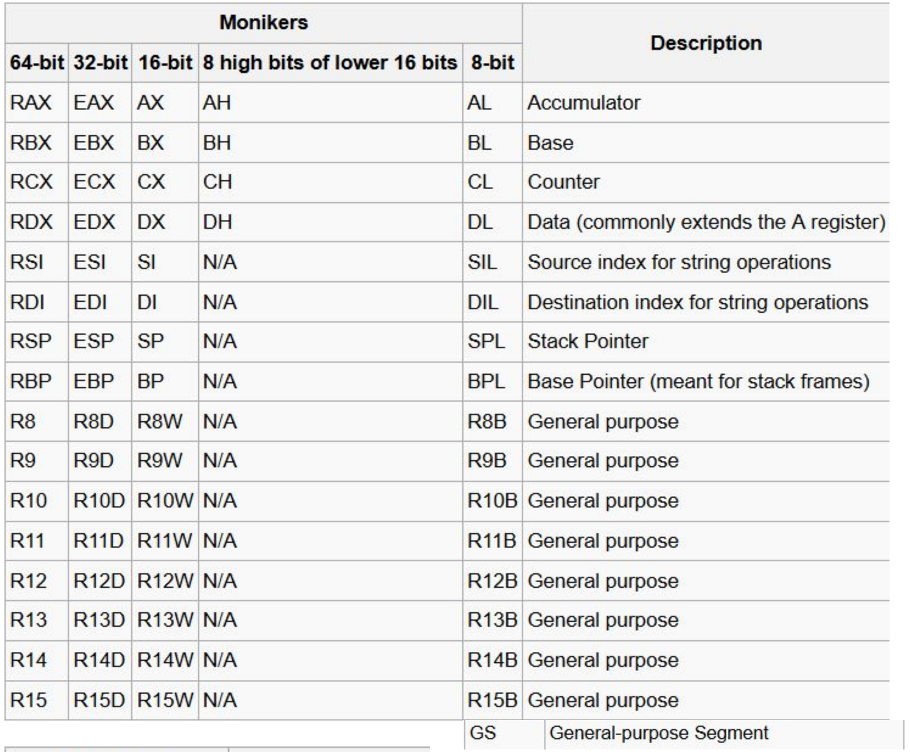
Translate TCG-IR to x86_64 binary code

TCG-IR to x86_64 translation

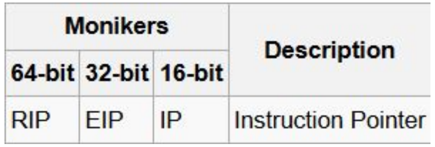
- Before entering the backend …



Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6–7	t1–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller



Monikers					Description
64-bit	32-bit	16-bit	8 high bits of lower 16 bits	8-bit	
RAX	EAX	AX	AH	AL	Accumulator
RBX	EBX	BX	BH	BL	Base
RCX	ECX	CX	CH	CL	Counter
RDX	EDX	DX	DH	DL	Data (commonly extends the A register)
RSI	ESI	SI	N/A	SIL	Source index for string operations
RDI	EDI	DI	N/A	DIL	Destination index for string operations
RSP	ESP	SP	N/A	SPL	Stack Pointer
RBP	EBP	BP	N/A	BPL	Base Pointer (meant for stack frames)
R8	R8D	R8W	N/A	R8B	General purpose
R9	R9D	R9W	N/A	R9B	General purpose
R10	R10D	R10W	N/A	R10B	General purpose
R11	R11D	R11W	N/A	R11B	General purpose
R12	R12D	R12W	N/A	R12B	General purpose
R13	R13D	R13W	N/A	R13B	General purpose
R14	R14D	R14W	N/A	R14B	General purpose
R15	R15D	R15W	N/A	R15B	General purpose
				GS	General-purpose Segment



Monikers			Description
64-bit	32-bit	16-bit	
RIP	EIP	IP	Instruction Pointer

TCG-IR to x86_64 translation

- Let’s take a look at some examples: **Store**

RISC-V 32bit

```
IN: sysmalloc
0x0001c7fe:  sw                s3,1156(s2)
```

TCG-IR

Load s2 to tmp0

tmp0 = tmp0 + 1156(0x484)

Load s3 to tmp1

Store tmp1 back to address tmp0

```
 ---- 0001c7fe
 mov_i32 tmp0,s2
 mov_i32 tmp2,$0x484
 add_i32 tmp0,tmp0,tmp2
 mov_i32 tmp1,s3
 qemu_st_i32 tmp1,tmp0,leul,0
```

x86_64

```
movl     0x48(%r14)/*s2*/, %ebp /*tmp0*/
addl     $0x484, %ebp/*tmp0*/
movl     0x4c(%r14)/*s3*/, %ebx/*tmp1*/
movl     %ebx/*tmp1*/, %gs:0(%ebp/*tmp0*/)
```

How does QEMU generate binary code?

TCG-IR to x86_64 translation

- Let’s take a look at some examples: **add**

RISC-V 32bit

```
IN: __libc_setup_tls
0x0001079e:   addi            a5,a5,32
```

TCG-IR

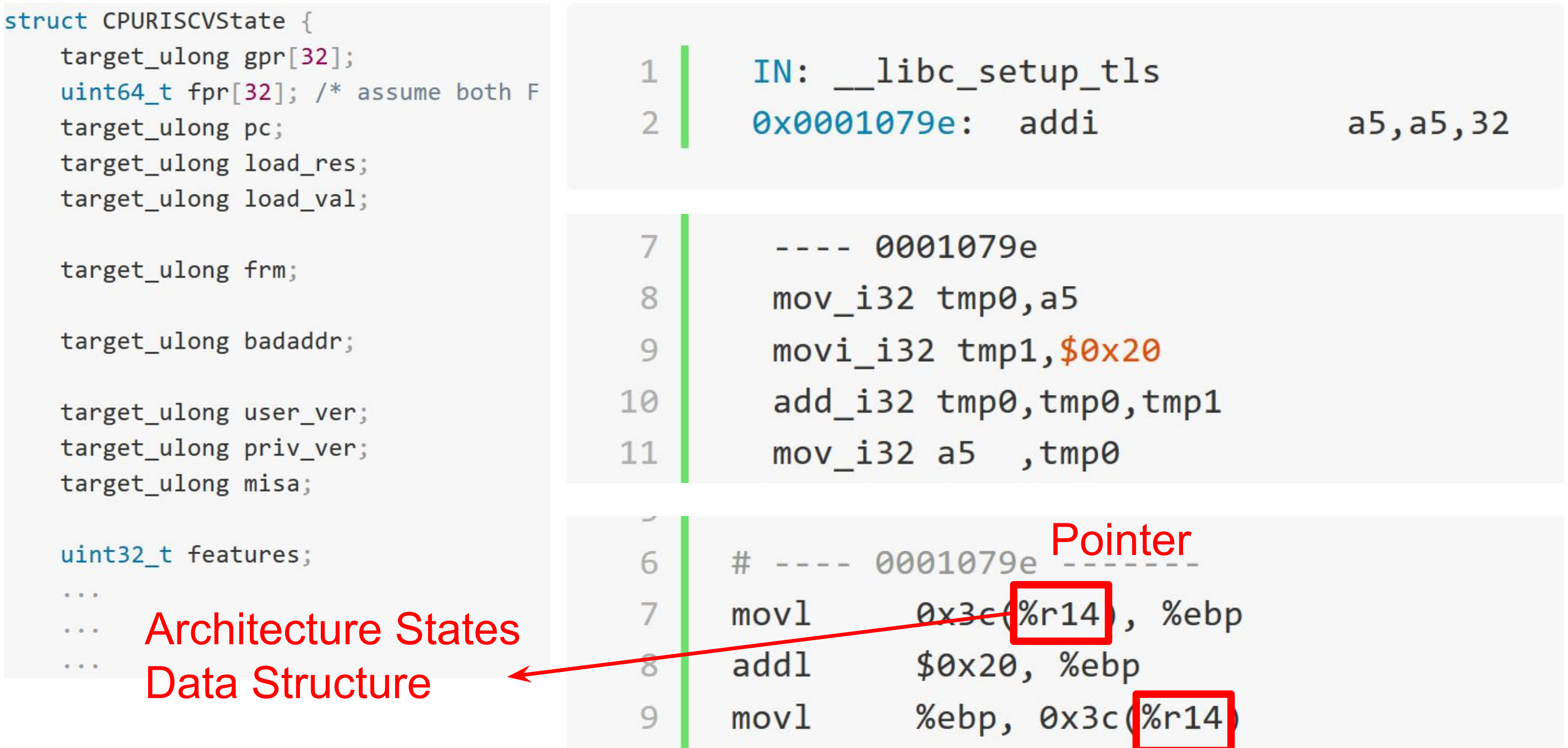
```
 ---- 0001079e
 mov_i32 tmp0,a5
 movi_i32 tmp1,$0x20
 add_i32 tmp0,tmp0,tmp1
 mov_i32 a5   ,tmp0
```

x86_64

```
# ---- 0001079e -------
movl     0x3c(%r14), %ebp
addl     $0x20, %ebp
movl     %ebp, 0x3c(%r14)
```

TCG-IR to x86_64 translation

- Let’s take a look at some examples: **add**



```
struct CPURISCVState {
    target_ulong gpr[32];
    uint64_t fpr[32]; /* assume both F
    target_ulong pc;
    target_ulong load_res;
    target_ulong load_val;

    target_ulong frm;

    target_ulong badaddr;

    target_ulong user_ver;
    target_ulong priv_ver;
    target_ulong misa;

    uint32_t features;
    ...
    ...
    ...
```

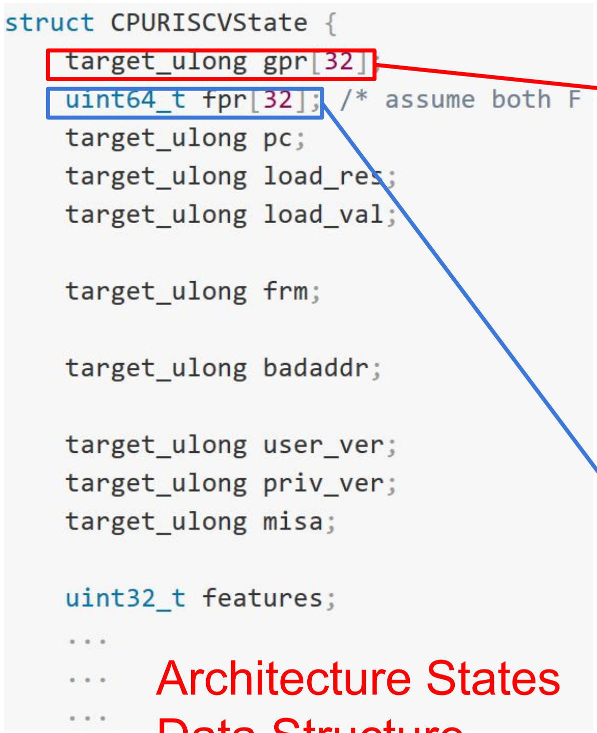
```
IN: __libc_setup_tls
0x0001079e:   addi            a5,a5,32
```

```
 ---- 0001079e
 mov_i32 tmp0,a5
 movi_i32 tmp1,$0x20
 add_i32 tmp0,tmp0,tmp1
 mov_i32 a5  ,tmp0
```

```
# ---- 0001079e --------
movl      0x3c(%r14), %ebp
addl      $0x20, %ebp
movl      %ebp, 0x3c(%r14)
```

TCG-IR to x86_64 translation

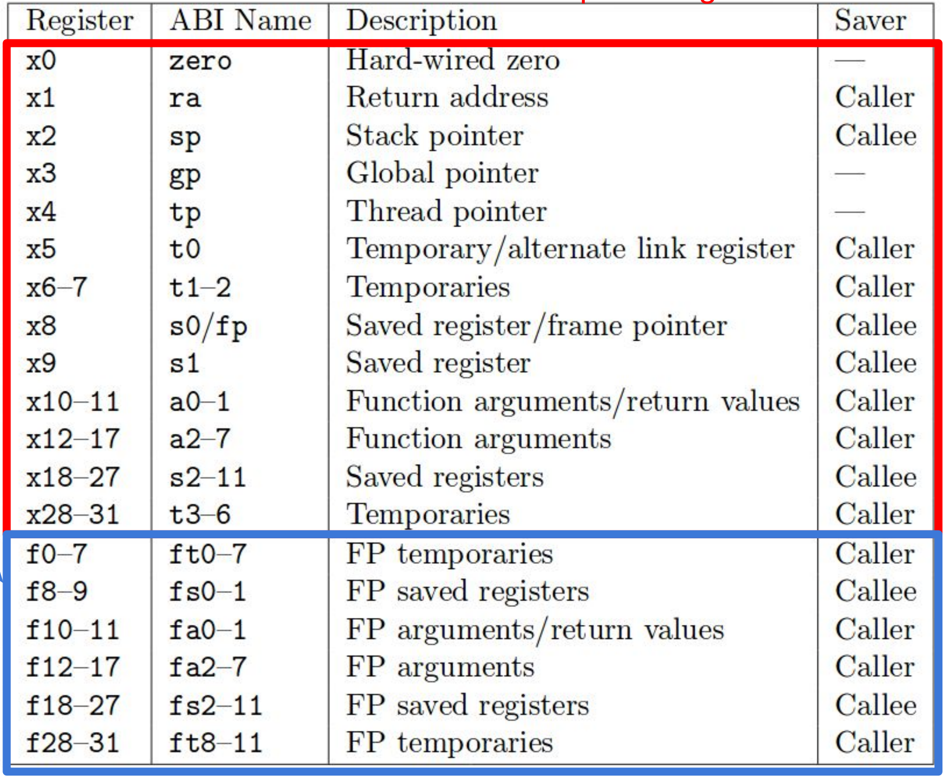
- Let’s take a look at some examples: **add**



```
struct CPURISCVState {
    target_ulong gpr[32];
    uint64_t fpr[32]; /* assume both F
    target_ulong pc;
    target_ulong load_res;
    target_ulong load_val;

    target_ulong frm;

    target_ulong badaddr;

    target_ulong user_ver;
    target_ulong priv_ver;
    target_ulong misa;

    uint32_t features;
    ...
    ...
    ...
```

Architecture States
Data Structure

General Purpose Registers



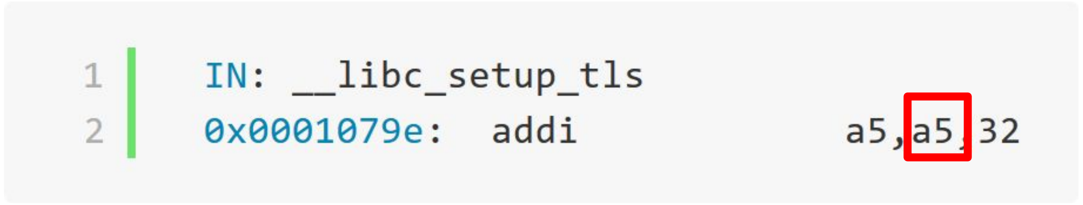
Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6–7	t1–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller

Floating Point Registers

TCG-IR to x86_64 translation

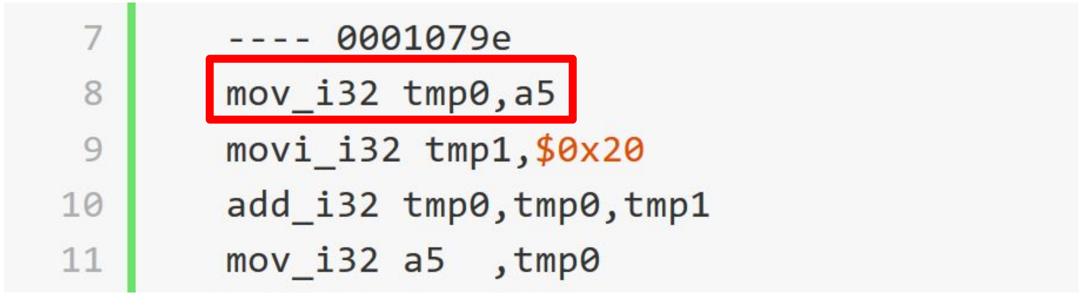
- Let’s take a look at some examples: **add**

RISC-V 32bit



```
IN: __libc_setup_tls
0x0001079e:   addi                  a5,a5,32
```

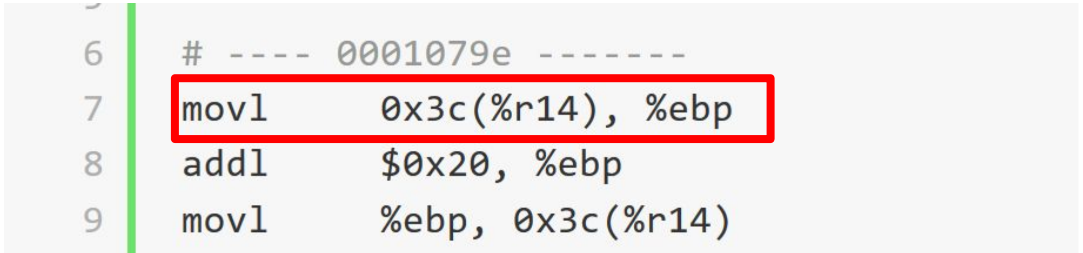
TCG-IR



```
 ---- 0001079e
 mov_i32 tmp0,a5
 movi_i32 tmp1,$0x20
 add_i32 tmp0,tmp0,tmp1
 mov_i32 a5  ,tmp0
```

Load data from architecture state **a5 reg.**

x86_64

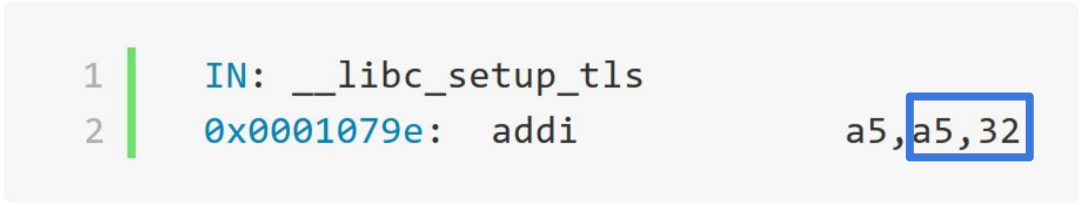


```
# ---- 0001079e -------
movl      0x3c(%r14), %ebp
addl      $0x20, %ebp
movl      %ebp, 0x3c(%r14)
```

TCG-IR to x86_64 translation

- Let's take a look at some examples: **add**

RISC-V 32bit

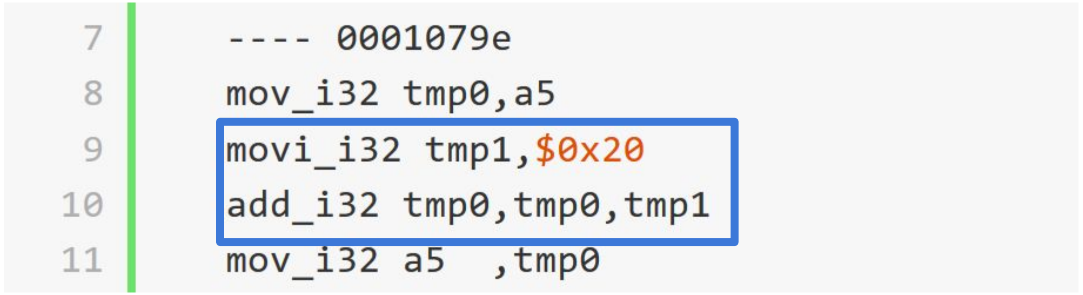


```
IN: __libc_setup_tls
0x0001079e:   addi                a5,a5,32
```

Load data from architecture state **a5**

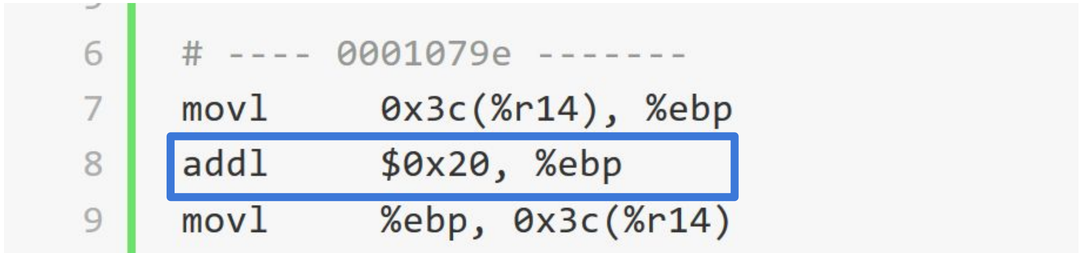
TCG-IR

tmp0 = tmp0 + tmp1



```
 ---- 0001079e
mov_i32 tmp0,a5
movi_i32 tmp1,$0x20
add_i32 tmp0,tmp0,tmp1
mov_i32 a5   ,tmp0
```

x86_64



```
# ---- 0001079e -------
movl     0x3c(%r14), %ebp
addl     $0x20, %ebp
movl     %ebp, 0x3c(%r14)
```

TCG-IR to x86_64 translation

- Let’s take a look at some examples: **add**

RISC-V 32bit

```
IN: __libc_setup_tls
0x0001079e:   addi                     a5,a5,32
```

Load data from architecture state **a5**

tmp0 = tmp0 + tmp1

TCG-IR

```
 ---- 0001079e
 mov_i32 tmp0,a5
 movi_i32 tmp1,$0x20
 add_i32 tmp0,tmp0,tmp1
 mov_i32 a5   ,tmp0
```

Store tmp0 back to architecture state **a5 reg**

x86_64

```
# ---- 0001079e -------
movl      0x3c(%r14), %ebp
addl      $0x20, %ebp
movl      %ebp, 0x3c(%r14)
```

TCG-IR to x86_64 translation

- Let’s take a look at some examples: **load**

RISC-V 32bit

```
IN: __libc_setup_tls
0x00010798:   lw                 a4,0(a5)
```

TCG-IR

```
---- 00010798
mov_i32 tmp0,a5
qemu_ld_i32 tmp1,tmp0,leul,0
mov_i32 a4   ,tmp1
```

Load data from address ‘tmp0’ to tmp1

x86_64

```
// ---- 00010798
movl      0x3c(%r14) /*a5*/, %ebp /*tmp0*/
movl      %gs:0(%ebp /*tmp0*/), %ebp /*tmp1*/
movl      %ebp /*tmp1*/, 0x38(%r14) /*a4*/
```

TCG-IR to x86_64 translation

- Let’s take a look at some examples: **load**

RISC-V 32bit

```
IN: __libc_setup_tls
0x00010798:  lw                a4,0(a5)
```

Load data from address ‘tmp0’ to tmp1

TCG-IR

Store tmp1 back to archi. state **a4 reg**

```
---- 00010798
mov_i32 tmp0,a5
qemu_ld_i32 tmp1,tmp0,leul,0
mov_i32 a4   ,tmp1
```

x86_64

```
// ---- 00010798
movl      0x3c(%r14) /*a5*/, %ebp /*tmp0*/
movl      %gs:0(%ebp /*tmp0*/), %ebp /*tmp1*/
movl      %ebp /*tmp1*/, 0x38(%r14) /*a4*/
```

TCG-IR to x86_64 translation

- Let’s take a look at some examples: **Store**

RISC-V 32bit

```
IN: sysmalloc
0x0001c7fe:   sw                      s3,1156(s2)
```

TCG-IR

```
---- 0001c7fe
mov_i32 tmp0,s2
movi_i32 tmp2,$0x484
add_i32 tmp0,tmp0,tmp2
mov_i32 tmp1,s3
qemu_st_i32 tmp1,tmp0,leul,0
```

Load s2 to tmp0

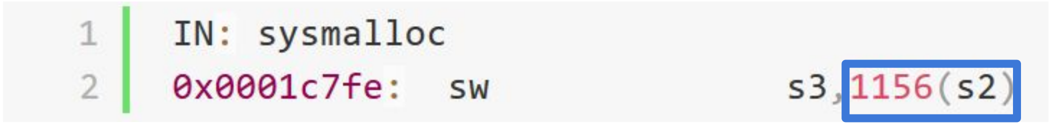
x86_64

```
movl     0x48(%r14)/*s2*/, %ebp /*tmp0*/
addl     $0x484, %ebp/*tmp0*/
movl     0x4c(%r14)/*s3*/, %ebx/*tmp1*/
movl     %ebx/*tmp1*/, %gs:0(%ebp/*tmp0*/)
```

TCG-IR to x86_64 translation

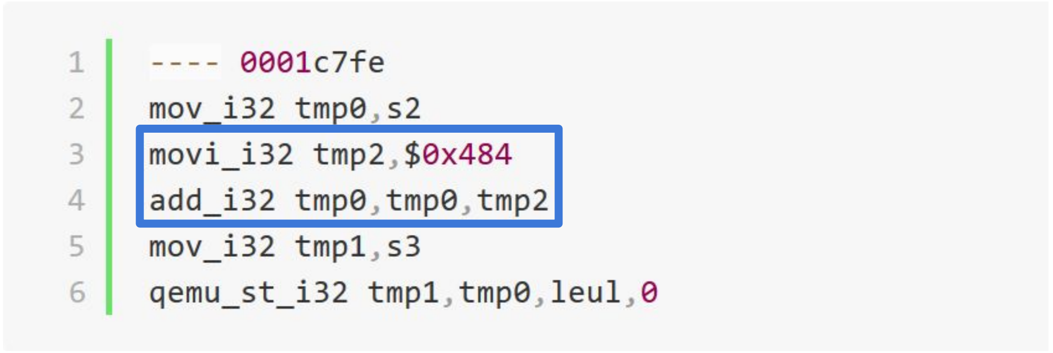
- Let’s take a look at some examples: **Store**

RISC-V 32bit



```
IN: sysmalloc
0x0001c7fe:   sw            s3,1156(s2)
```

TCG-IR

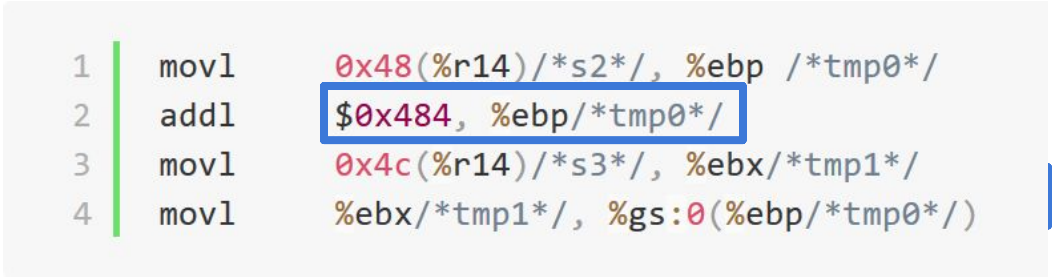


```
---- 0001c7fe
mov_i32 tmp0,s2
movi_i32 tmp2,$0x484
add_i32 tmp0,tmp0,tmp2
mov_i32 tmp1,s3
qemu_st_i32 tmp1,tmp0,leul,0
```

Load s2 to tmp0

tmp0 = tmp0 + 1156(0x484)

x86_64



```
movl      0x48(%r14)/*s2*/, %ebp /*tmp0*/
addl      $0x484, %ebp/*tmp0*/
movl      0x4c(%r14)/*s3*/, %ebx/*tmp1*/
movl      %ebx/*tmp1*/, %gs:0(%ebp/*tmp0*/)
```

TCG-IR to x86_64 translation

- Let's take a look at some examples: **Store**

RISC-V 32bit

```
IN: sysmalloc
0x0001c7fe:   sw            s3,1156(s2)
```

TCG-IR

Load s2 to tmp0

tmp0 = tmp0 + 1156(0x484)

Load from s3 to tmp1

```
---- 0001c7fe
mov_i32 tmp0,s2
movi_i32 tmp2,$0x484
add_i32 tmp0,tmp0,tmp2
mov_i32 tmp1,s3
qemu_st_i32 tmp1,tmp0,leul,0
```

x86_64

```
movl     0x48(%r14)/*s2*/, %ebp /*tmp0*/
addl     $0x484, %ebp/*tmp0*/
movl     0x4c(%r14)/*s3*/, %ebx/*tmp1*/
movl     %ebx/*tmp1*/, %gs:0(%ebp/*tmp0*/)
```

TCG-IR to x86_64 translation

- Let's take a look at some examples: **Store**

RISC-V 32bit

```
IN: sysmalloc
0x0001c7fe:  sw                  s3,1156(s2)
```

TCG-IR

Load s2 to tmp0

tmp0 = tmp0 + 1156(0x484)

Load s3 to tmp1

Store tmp1 to address tmp 0

```
---- 0001c7fe
mov_i32 tmp0,s2
movi_i32 tmp2,$0x484
add_i32 tmp0,tmp0,tmp2
mov_i32 tmp1,s3
qemu_st_i32 tmp1,tmp0,leul,0
```

x86_64

```
movl      0x48(%r14)/*s2*/, %ebp /*tmp0*/
addl      $0x484, %ebp/*tmp0*/
movl      0x4c(%r14)/*s3*/, %ebx/*tmp1*/
movl      %ebx/*tmp1*/, %gs:0(%ebp/*tmp0*/)
```

TCG-IR to x86_64 translation

- Let’s take a look at some examples: **Store**

RISC-V 32bit

```
IN: sysmalloc
0x0001c7fe:  sw              s3,1156(s2)
```

TCG-IR

```
---- 0001c7fe
mov_i32 tmp0,s2
movi_i32 tmp2,$0x484
add_i32 tmp0,tmp0,tmp2
mov_i32 tmp1,s3
qemu_st_i32 tmp1,tmp0,leul,0
```

Load s2 to tmp0

tmp0 = tmp0 + 1156(0x484)

Load s3 to tmp1

Store tmp1 back to address tmp0

x86_64

```
movl     0x48(%r14)/*s2*/, %ebp /*tmp0*/
addl     $0x484, %ebp/*tmp0*/
movl     0x4c(%r14)/*s3*/, %ebx/*tmp1*/
movl     %ebx/*tmp1*/, %gs:0(%ebp/*tmp0*/)
```

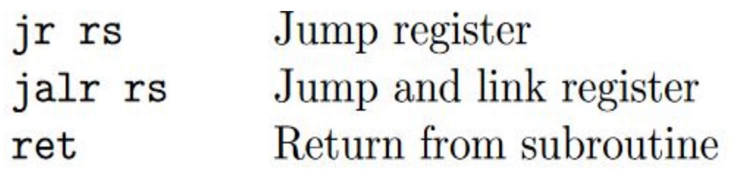
How does QEMU handle branch instructions?

TCG-IR to x86_64 translation

- Let’s take a look at some examples: **Branch**
  - **Direct Branch**
    - Conditional Branch

      beqz rs, offset / bgt rs, rt, offset / …..
    - Unconditional Branch

      j offset / jal offset / call offset / ...
  - **Indirect Branch**
    - Switch/Case → Branch table
    - Indirect function call
    - Return Instructions (ret)



`jr rs`	Jump register
`jalr rs`	Jump and link register
`ret`	Return from subroutine

TCG-IR to x86_64 translation

- Let’s take a look at some examples: **Direct Branch (Unconditional)**

RISC-V 32bit

```
 IN: _dl_aux_init
0x000229c6:  lw              s9,4(a0)
0x000229ca:  addi            s6,zero,1
0x000229cc:  j               -316            # 0x22890
```

TCG-IR

```
---- 000229cc
goto_tb $0x0
movi_i32 pc,$0x22890
exit_tb $0x55c9819eff40
```

x86_64

```
0x55c9819effe7:  jmp      0x55c9819effec
0x55c9819effec:  movl     $0x22890, 0x180(%r14)
0x55c9819efff4:
0x55c9819efff7:  leaq     -0xbe(%rip), %rax
0x55c9819efffe:  jmp      0x55c9819ef018
```

TCG-IR to x86_64 translation

- Let’s take a look at some examples: **Direct Branch (Unconditional)**

RISC-V 32bit

```
 IN: _dl_aux_init
0x000229c6:  lw              s9,4(a0)
0x000229ca:  addi            s6,zero,1
0x000229cc:  j               -316              # 0x22890
```

Remind:
Patch point for block chaining

TCG-IR

```
---- 000229cc
goto_tb $0x0
movi_i32 pc,$0x22890
exit_tb $0x55c9819eff40
```

x86_64

```
0x55c9819effe7:  jmp      0x55c9819effec
0x55c9819effec:  movl     $0x22890, 0x180(%r14)
0x55c9819efff4:
0x55c9819efff7:  leaq     -0xbe(%rip), %rax
0x55c9819efffe:  jmp      0x55c9819ef018
```

TCG-IR to x86_64 translation

- Let’s take a look at some examples: **Direct Branch (Unconditional)**

RISC-V 32bit

```
 IN: _dl_aux_init
0x000229c6:  lw           s9,4(a0)
0x000229ca:  addi         s6,zero,1
0x000229cc:  j            -316          # 0x22890
```

TCG-IR

Synchronize program counter to architecture states

```
---- 000229cc
goto_tb $0x0
movi_i32 pc,$0x22890
exit_tb $0x55c9819eff40
```

x86_64

```
0x55c9819effe7:  jmp     0x55c9819effec
0x55c9819effec:  movl    $0x22890, 0x180(%r14)
0x55c9819efff4:
0x55c9819efff7:  leaq    -0xbe(%rip), %rax
0x55c9819efffe:  jmp     0x55c9819ef018
```

TCG-IR to x86_64 translation

- Let’s take a look at some examples: **Direct Branch (Unconditional)**

RISC-V 32bit

```
 IN: _dl_aux_init
0x000229c6:  lw              s9,4(a0)
0x000229ca:  addi            s6,zero,1
0x000229cc:  j               -316            # 0x22890
```

TCG-IR

Synchronize program counter

Prepare return value

```
---- 000229cc
goto_tb $0x0
movi_i32 pc,$0x22890
exit_tb $0x55c9819eff40
```

x86_64

```
0x55c9819effe7:  jmp      0x55c9819effec
0x55c9819effec:  movl     $0x22890, 0x180(%r14)
0x55c9819efff4:
0x55c9819efff7:  leaq     -0xbe(%rip), %rax
0x55c9819efffe:  jmp      0x55c9819ef018
```

TCG-IR to x86_64 translation

- Let’s take a look at some examples: **Direct Branch (Unconditional)**

RISC-V 32bit

```
 IN: _dl_aux_init
0x000229c6:  lw              s9,4(a0)
0x000229ca:  addi            s6,zero,1
0x000229cc:  j               -316            # 0x22890
```

TCG-IR

Synchronize program counter

Prepare return value

Go back to QEMU to find next Translation Block

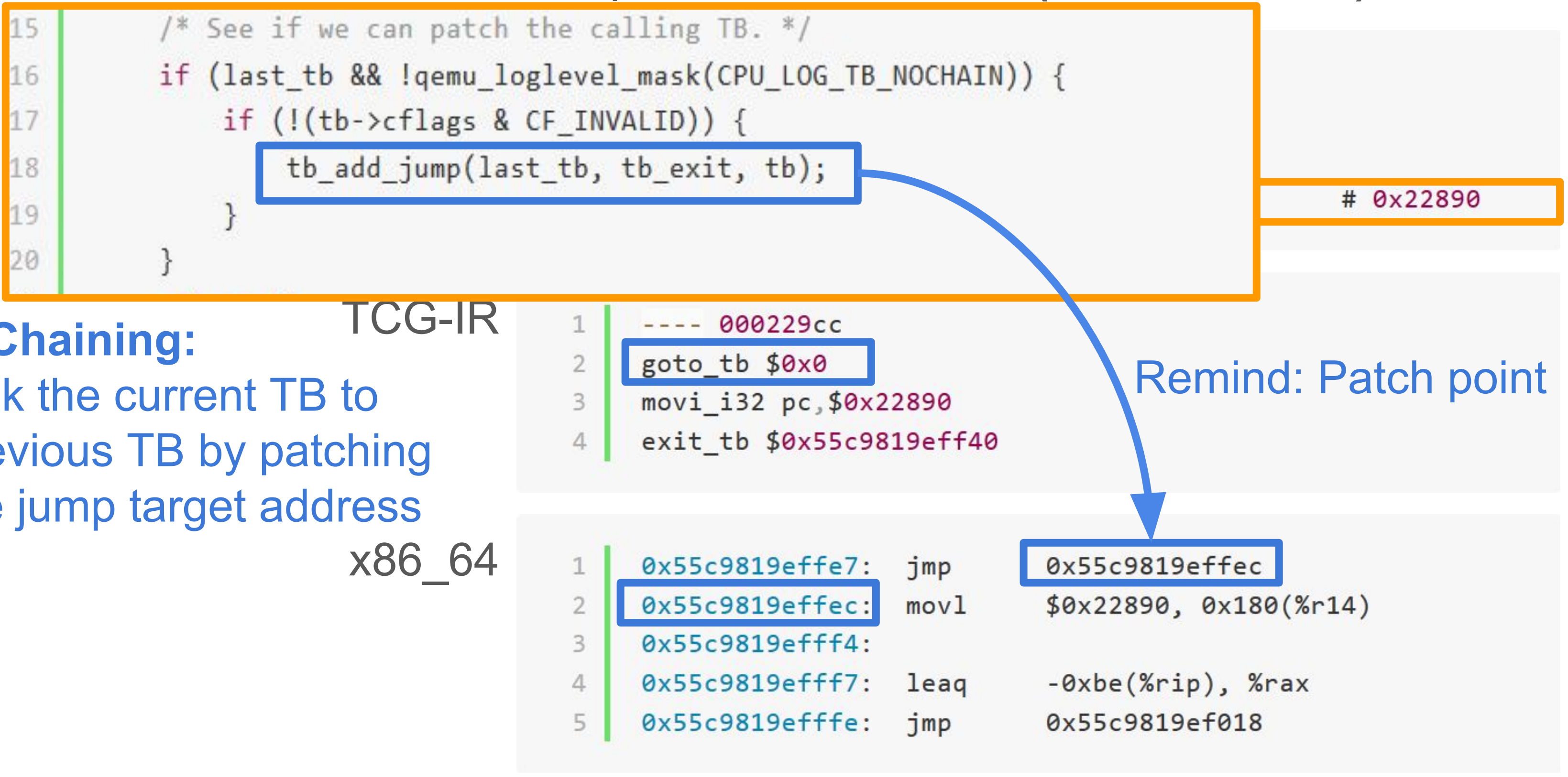
```
---- 000229cc
goto_tb $0x0
movi_i32 pc,$0x22890
exit_tb $0x55c9819eff40
```

x86_64

```
0x55c9819effe7:  jmp      0x55c9819effec
0x55c9819effec:  movl     $0x22890, 0x180(%r14)
0x55c9819efff4:
0x55c9819efff7:  leaq     -0xbe(%rip), %rax
0x55c9819efffe:  jmp      0x55c9819ef018
```

TCG-IR to x86_64 translation

- Let's take a look at some examples: **Direct Branch (Unconditional)**



```
    /* See if we can patch the calling TB. */
    if (last_tb && !qemu_loglevel_mask(CPU_LOG_TB_NOCHAIN)) {
        if (!(tb->cflags & CF_INVALID)) {
            tb_add_jump(last_tb, tb_exit, tb);
        }
    }
```

```
# 0x22890
```

TCG-IR

```
---- 000229cc
goto_tb $0x0
movi_i32 pc,$0x22890
exit_tb $0x55c9819eff40
```

x86_64

```
0x55c9819effe7:  jmp     0x55c9819effec
0x55c9819effec:  movl    $0x22890, 0x180(%r14)
0x55c9819efff4:
0x55c9819efff7:  leaq    -0xbe(%rip), %rax
0x55c9819efffe:  jmp     0x55c9819ef018
```

Remind: Patch point

Block Chaining:

Link the current TB to previous TB by patching the jump target address

TCG-IR to x86_64 translation

- Let’s take a look at some examples: **Direct Branch (Conditional)**

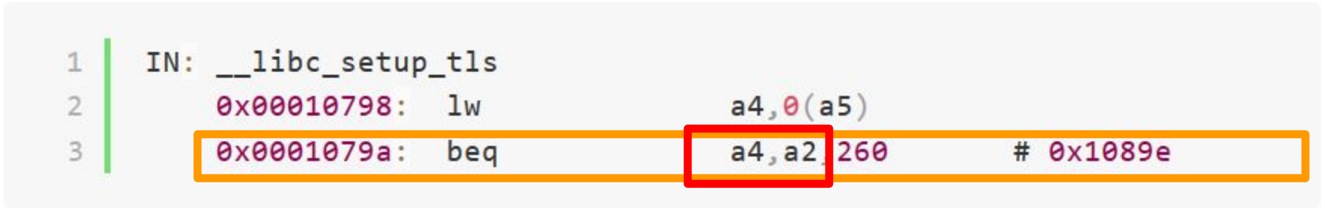
```
IN: __libc_setup_tls
    0x00010798:  lw                  a4,0(a5)
    0x0001079a:  beq                 a4,a2,260          # 0x1089e
```

```
---- 0001079a
mov_i32 tmp0,a4
mov_i32 tmp1,a2
brcond_i32 tmp0,tmp1,eq,$L1 // If equal, goto L1
    goto_tb $0x1 // Patch point
    movi_i32 pc,$0x1079e // Save PC back to CPUState
    exit_tb $0x5578dbc76741 // Return to QEMU to find NON-TAKEN

set_label $L1
    goto_tb $0x0 // Patch point
    movi_i32 pc,$0x1089e // Save PC back to CPUState
    exit_tb $0x5578dbc76740 // Return to qemu to find TAKEN
```

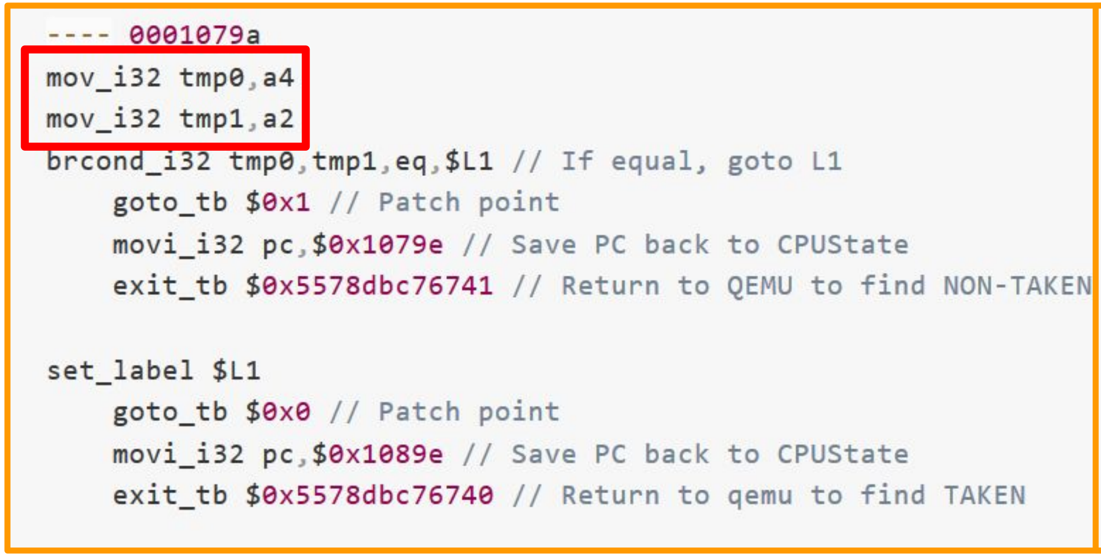
```
---- 0001079a
    movl      0x38(%r14), %ebp
    movl      0x30(%r14), %ebx
    cmpl      %ebx, %ebp
    je        L1
    nop
    jmp       0x5578dbc767ec     // Patch Point
    0x5578dbc767ec: movl     $0x1079e, 0x180(%r14)
                    leaq     -0xbd(%rip), %rax
                    jmp      0x5578dbc71018

L1:
    jmp       0x5578dbc76808 // Patch Point
    0x5578dbc76808:  movl     $0x1089e, 0x180(%r14)
                     leaq     -0xda(%rip), %rax
                     jmp      0x5578dbc71018 // QEUM
```

TCG-IR to x86_64 translation

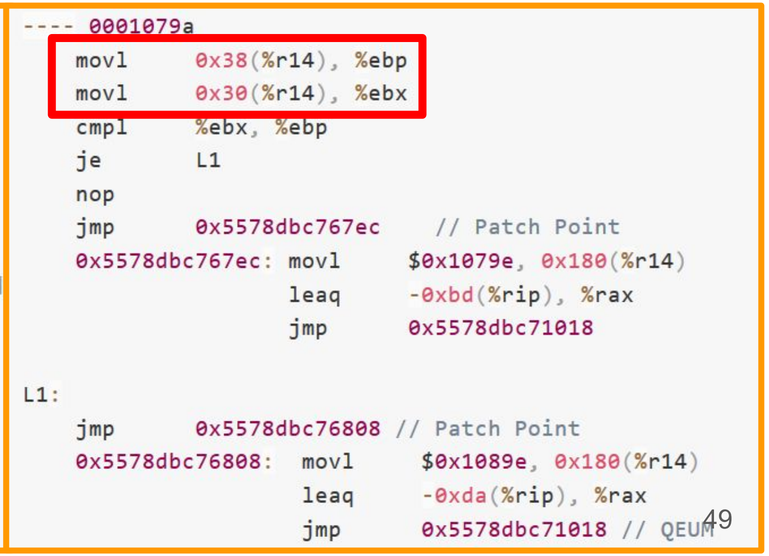
- Let’s take a look at some examples: **Direct Branch (Conditional)**



```
IN: __libc_setup_tls
    0x00010798:  lw              a4,0(a5)
    0x0001079a:  beq             a4,a2,260         # 0x1089e
```



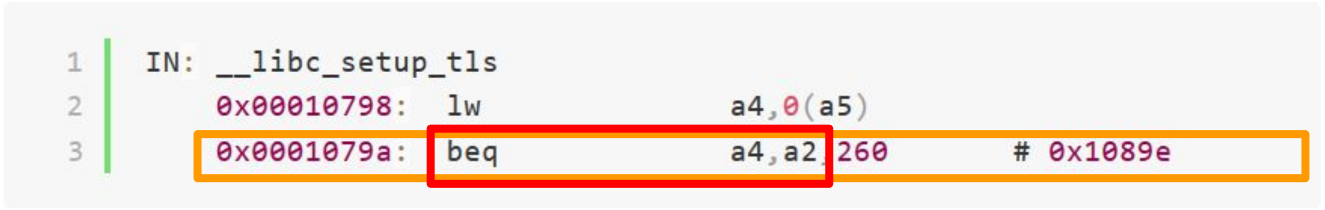
```
---- 0001079a
mov_i32 tmp0,a4
mov_i32 tmp1,a2
brcond_i32 tmp0,tmp1,eq,$L1 // If equal, goto L1
    goto_tb $0x1 // Patch point
    movi_i32 pc,$0x1079e // Save PC back to CPUState
    exit_tb $0x5578dbc76741 // Return to QEMU to find NON-TAKEN

set_label $L1
    goto_tb $0x0 // Patch point
    movi_i32 pc,$0x1089e // Save PC back to CPUState
    exit_tb $0x5578dbc76740 // Return to qemu to find TAKEN
```



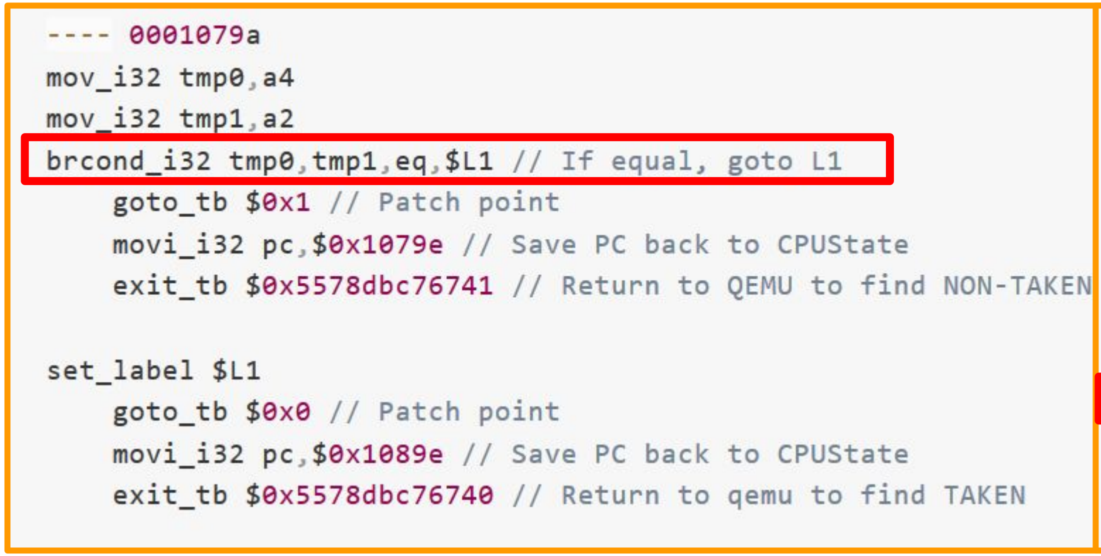
```
---- 0001079a
    movl      0x38(%r14), %ebp
    movl      0x30(%r14), %ebx
    cmpl      %ebx, %ebp
    je        L1
    nop
    jmp       0x5578dbc767ec     // Patch Point
    0x5578dbc767ec: movl     $0x1079e, 0x180(%r14)
                    leaq     -0xbd(%rip), %rax
                    jmp      0x5578dbc71018

L1:
    jmp       0x5578dbc76808 // Patch Point
    0x5578dbc76808:  movl     $0x1089e, 0x180(%r14)
                     leaq     -0xda(%rip), %rax
                     jmp      0x5578dbc71018 // QEUM
```

TCG-IR to x86_64 translation

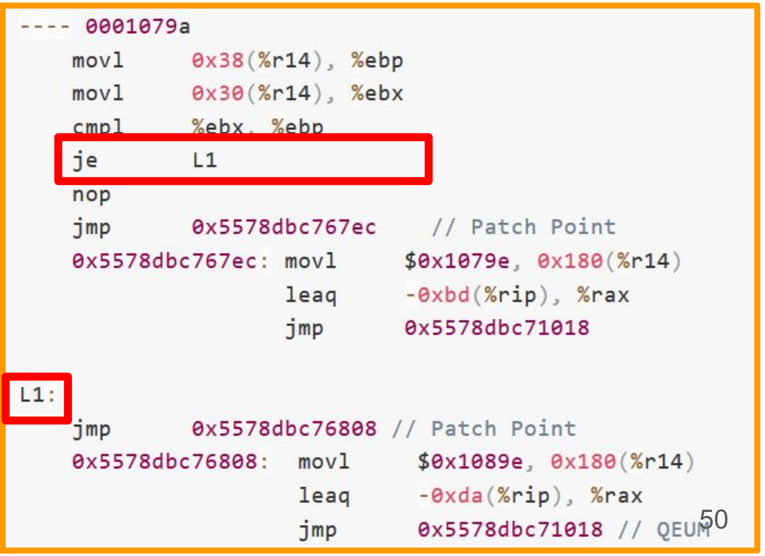
- Let’s take a look at some examples: **Direct Branch (Conditional)**



```
IN: __libc_setup_tls
    0x00010798:  lw              a4,0(a5)
    0x0001079a:  beq             a4,a2,260        # 0x1089e
```



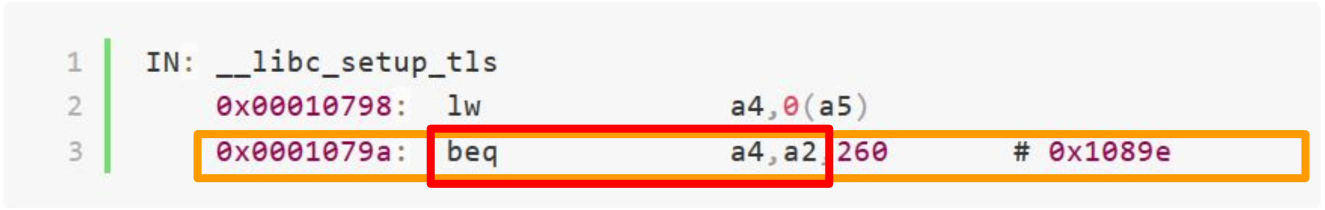
```
---- 0001079a
mov_i32 tmp0,a4
mov_i32 tmp1,a2
brcond_i32 tmp0,tmp1,eq,$L1 // If equal, goto L1
    goto_tb $0x1 // Patch point
    movi_i32 pc,$0x1079e // Save PC back to CPUState
    exit_tb $0x5578dbc76741 // Return to QEMU to find NON-TAKEN

set_label $L1
    goto_tb $0x0 // Patch point
    movi_i32 pc,$0x1089e // Save PC back to CPUState
    exit_tb $0x5578dbc76740 // Return to qemu to find TAKEN
```



```
---- 0001079a
    movl      0x38(%r14), %ebp
    movl      0x30(%r14), %ebx
    cmpl      %ebx, %ebp
    je        L1
    nop
    jmp       0x5578dbc767ec     // Patch Point
    0x5578dbc767ec: movl     $0x1079e, 0x180(%r14)
                    leaq     -0xbd(%rip), %rax
                    jmp      0x5578dbc71018


L1:
    jmp       0x5578dbc76808 // Patch Point
    0x5578dbc76808:  movl     $0x1089e, 0x180(%r14)
                     leaq     -0xda(%rip), %rax
                     jmp      0x5578dbc71018 // QEMU
```

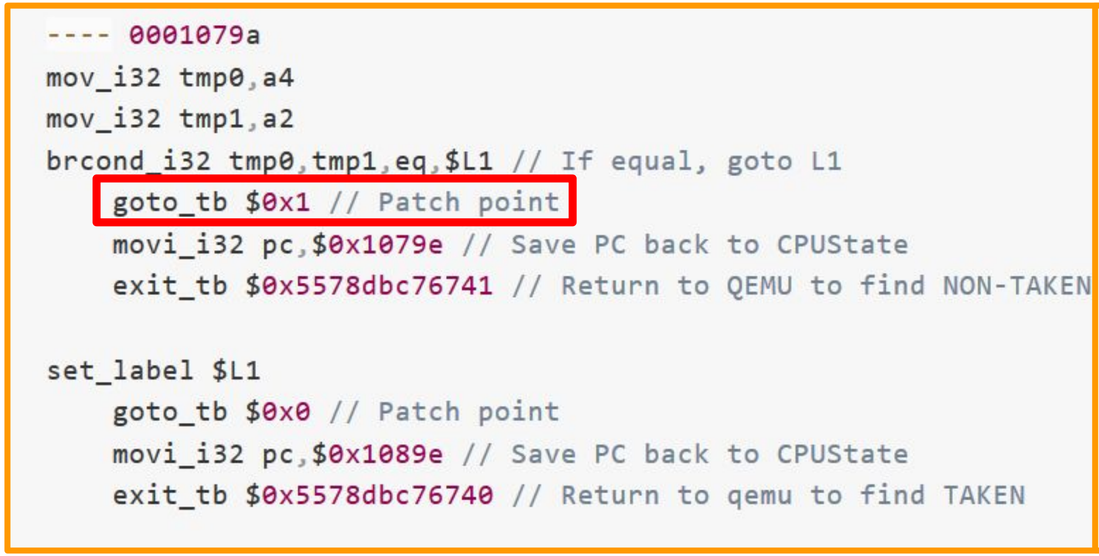
TCG-IR to x86_64 translation

- Let’s take a look at some examples: **Direct Branch (Conditional)**

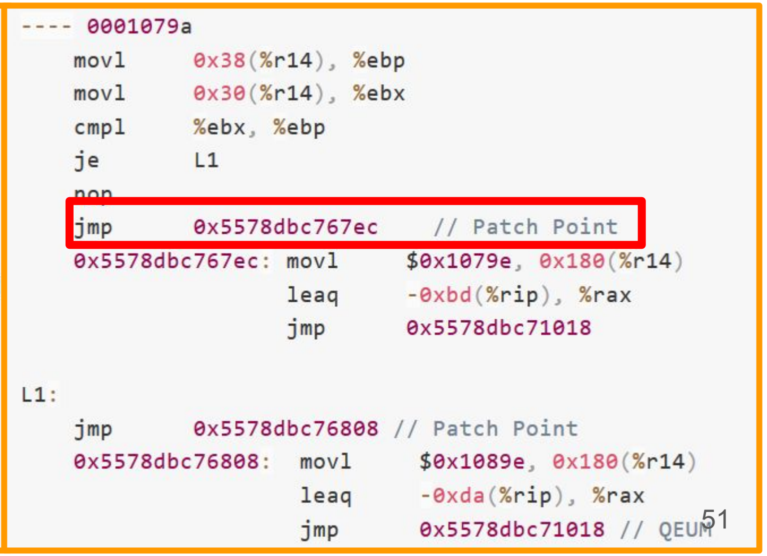


```
IN: __libc_setup_tls
    0x00010798:  lw                  a4,0(a5)
    0x0001079a:  beq                 a4,a2,260          # 0x1089e
```

If the block is chained by QEMU, jump to target translation block



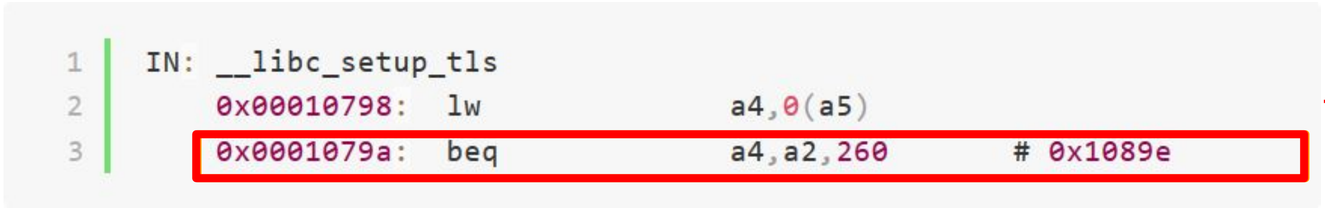
```
---- 0001079a
mov_i32 tmp0,a4
mov_i32 tmp1,a2
brcond_i32 tmp0,tmp1,eq,$L1 // If equal, goto L1
    goto_tb $0x1 // Patch point
    movi_i32 pc,$0x1079e // Save PC back to CPUState
    exit_tb $0x5578dbc76741 // Return to QEMU to find NON-TAKEN

set_label $L1
    goto_tb $0x0 // Patch point
    movi_i32 pc,$0x1089e // Save PC back to CPUState
    exit_tb $0x5578dbc76740 // Return to qemu to find TAKEN
```



```
---- 0001079a
    movl      0x38(%r14), %ebp
    movl      0x30(%r14), %ebx
    cmpl      %ebx, %ebp
    je        L1
    nop
    jmp       0x5578dbc767ec     // Patch Point
    0x5578dbc767ec: movl      $0x1079e, 0x180(%r14)
                    leaq      -0xbd(%rip), %rax
                    jmp       0x5578dbc71018


L1:
    jmp       0x5578dbc76808 // Patch Point
    0x5578dbc76808:  movl      $0x1089e, 0x180(%r14)
                     leaq      -0xda(%rip), %rax
                     jmp       0x5578dbc71018 // QEUM
```

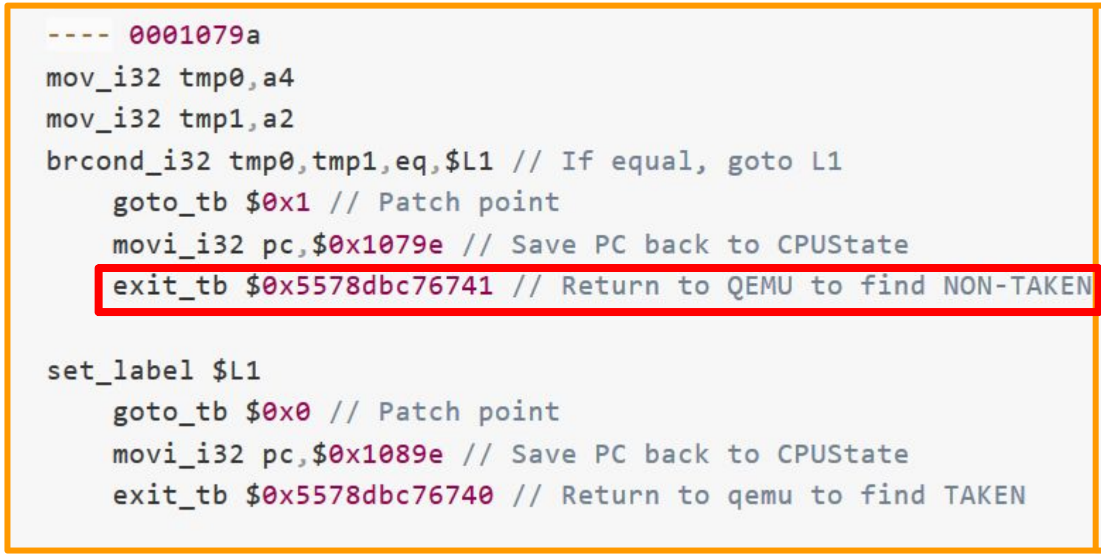
TCG-IR to x86_64 translation

- Let’s take a look at some examples: **Direct Branch (Conditional)**

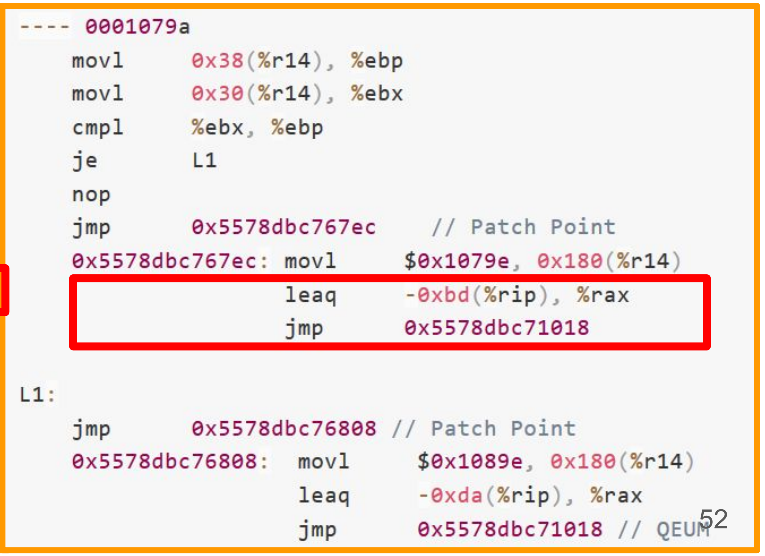


```
IN: __libc_setup_tls
    0x00010798:  lw          a4,0(a5)
    0x0001079a:  beq         a4,a2,260        # 0x1089e
```

Go back to QEMU to find next Translation Block



```
---- 0001079a
mov_i32 tmp0,a4
mov_i32 tmp1,a2
brcond_i32 tmp0,tmp1,eq,$L1 // If equal, goto L1
    goto_tb $0x1 // Patch point
    movi_i32 pc,$0x1079e // Save PC back to CPUState
    exit_tb $0x5578dbc76741 // Return to QEMU to find NON-TAKEN

set_label $L1
    goto_tb $0x0 // Patch point
    movi_i32 pc,$0x1089e // Save PC back to CPUState
    exit_tb $0x5578dbc76740 // Return to qemu to find TAKEN
```



```
---- 0001079a
    movl      0x38(%r14), %ebp
    movl      0x30(%r14), %ebx
    cmpl      %ebx, %ebp
    je        L1
    nop
    jmp       0x5578dbc767ec     // Patch Point
    0x5578dbc767ec: movl       $0x1079e, 0x180(%r14)
                    leaq       -0xbd(%rip), %rax
                    jmp        0x5578dbc71018

L1:
    jmp       0x5578dbc76808 // Patch Point
    0x5578dbc76808:  movl      $0x1089e, 0x180(%r14)
                     leaq      -0xda(%rip), %rax
                     jmp       0x5578dbc71018 // QEUM
```

TCG-IR to x86_64 translation

- Let’s take a look at some examples: **Indirect Branch - return instruction**

RISC-V 32bit

```
0x000103b4:   lw                  s0,28(sp)
0x000103b6:   addi                sp,sp,32
0x000103b8:   ret
```

TCG-IR

```
---- 000103b8
mov_i32 pc,ra   // Move ra to program counter
movi_i32 tmp1,$0xfffffffffffffffe   // create mask
and_i32 pc,pc,tmp1   // Filter
exit_tb $0x0   // Go back to QEMU
```

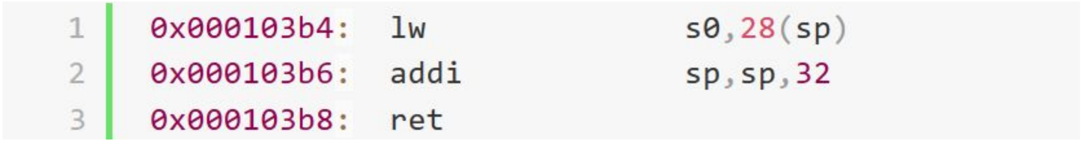
x86_64

```
---- 000103b8
movl     4(%r14)/*ra*/, %ebp
andl     $0xfffffffffffffffe, %ebp /*tmp1*/
movl     %ebp, 0x180(%r14)/*pc*/
jmp      0x555555b8d016
// Jump back to qemu, and clear %eax register (ret=0)
```

TCG-IR to x86_64 translation

- Let’s take a look at some examples: **Indirect Branch - return instruction**

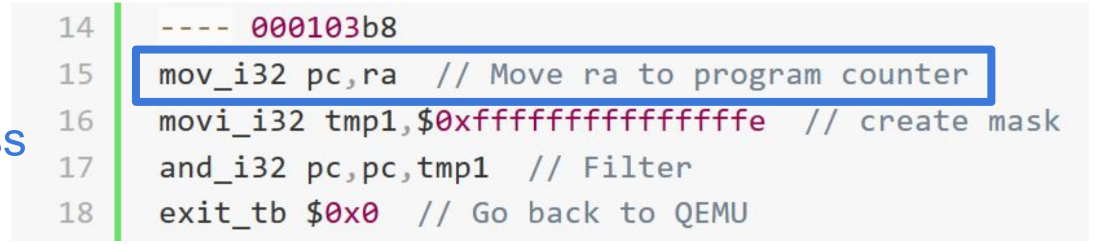
RISC-V 32bit



```
0x000103b4:   lw                s0,28(sp)
0x000103b6:   addi              sp,sp,32
0x000103b8:   ret
```

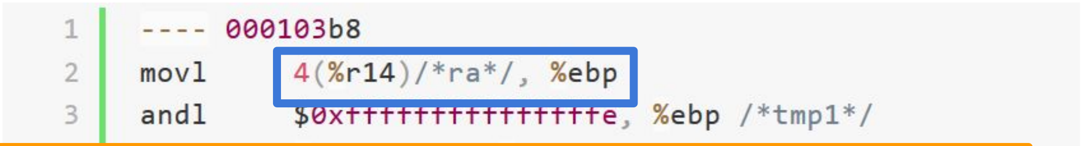
TCG-IR

Store the value from return address register to program counter

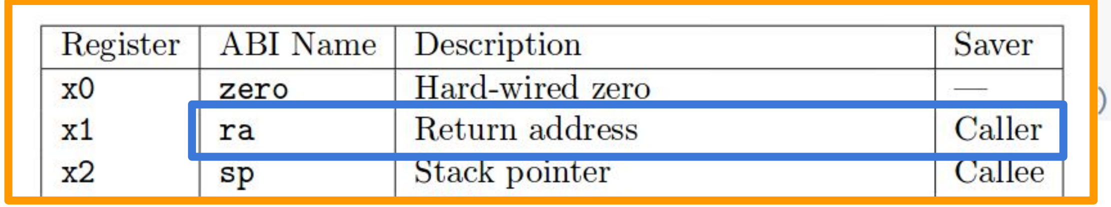


```
---- 000103b8
mov_i32 pc,ra   // Move ra to program counter
movi_i32 tmp1,$0xfffffffffffffffe   // create mask
and_i32 pc,pc,tmp1   // Filter
exit_tb $0x0   // Go back to QEMU
```

x86_64



```
---- 000103b8
movl     4(%r14)/*ra*/, %ebp
andl     $0xfffffffffffffffe, %ebp /*tmp1*/
```

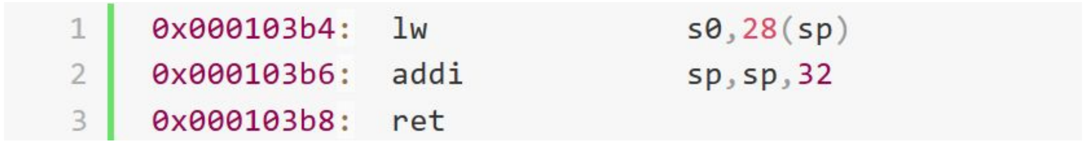


Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee

TCG-IR to x86_64 translation

- Let’s take a look at some examples: **Indirect Branch - return instruction**

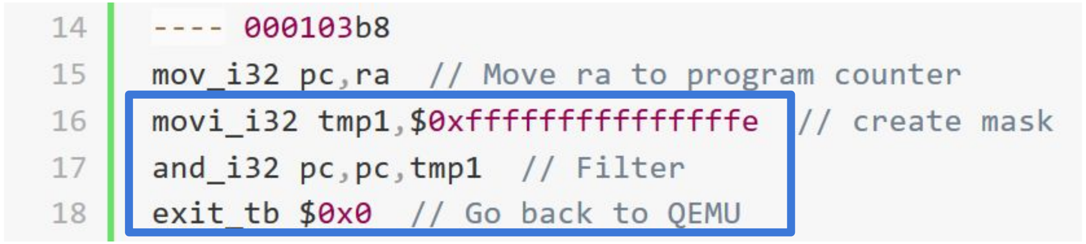
RISC-V 32bit



```
0x000103b4:   lw                s0,28(sp)
0x000103b6:   addi              sp,sp,32
0x000103b8:   ret
```

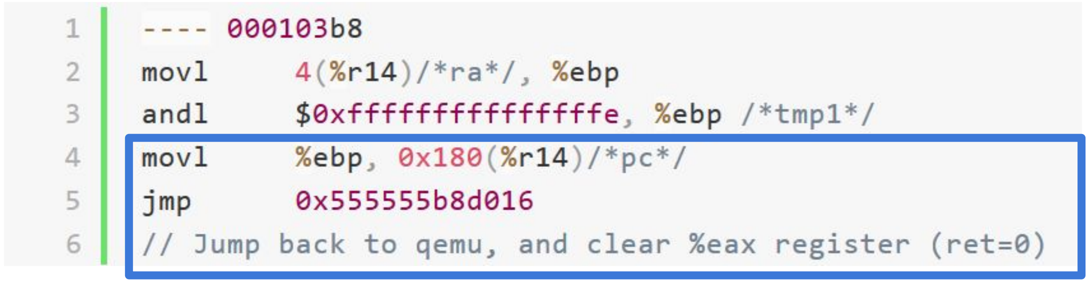
TCG-IR

Go back to QEMU to find next Translation Block in Program Counter



```
---- 000103b8
mov_i32 pc,ra   // Move ra to program counter
movi_i32 tmp1,$0xfffffffffffffffe  // create mask
and_i32 pc,pc,tmp1   // Filter
exit_tb $0x0   // Go back to QEMU
```

x86_64



```
---- 000103b8
movl      4(%r14)/*ra*/, %ebp
andl      $0xfffffffffffffffe, %ebp /*tmp1*/
movl      %ebp, 0x180(%r14)/*pc*/
jmp       0x555555b8d016
// Jump back to qemu, and clear %eax register (ret=0)
```

TCG-IR to x86_64 translation

- **helper function call**
  - QEMU provides a ‘hook’ for developers to write emulation behavior in C language. Commonly used in:
    - Emulate hardware not supported in host machine
      - e.g. Hardware FP, SIMD, AES, etc.
    - Dynamic Instrumentation
      - Collect runtime information from source program to analyze program’s behavior
        (e.g. Dynamic call graph / control flow graph)

TCG-IR to x86_64 translation

- **helper function call - example**
  - RISC-V source binary

```
0x00010408:   fadd.d          dyn,fa5,fa4,fa5
```

  - TCG-IR

```
---- 00010408
movi_i32 tmp0,$0x7
call set_rounding_mode,$0x20,$0,env,tmp0
call fadd_d,$0x10,$1,fa5 env,fa4 ,fa5
```

  - x86 binary

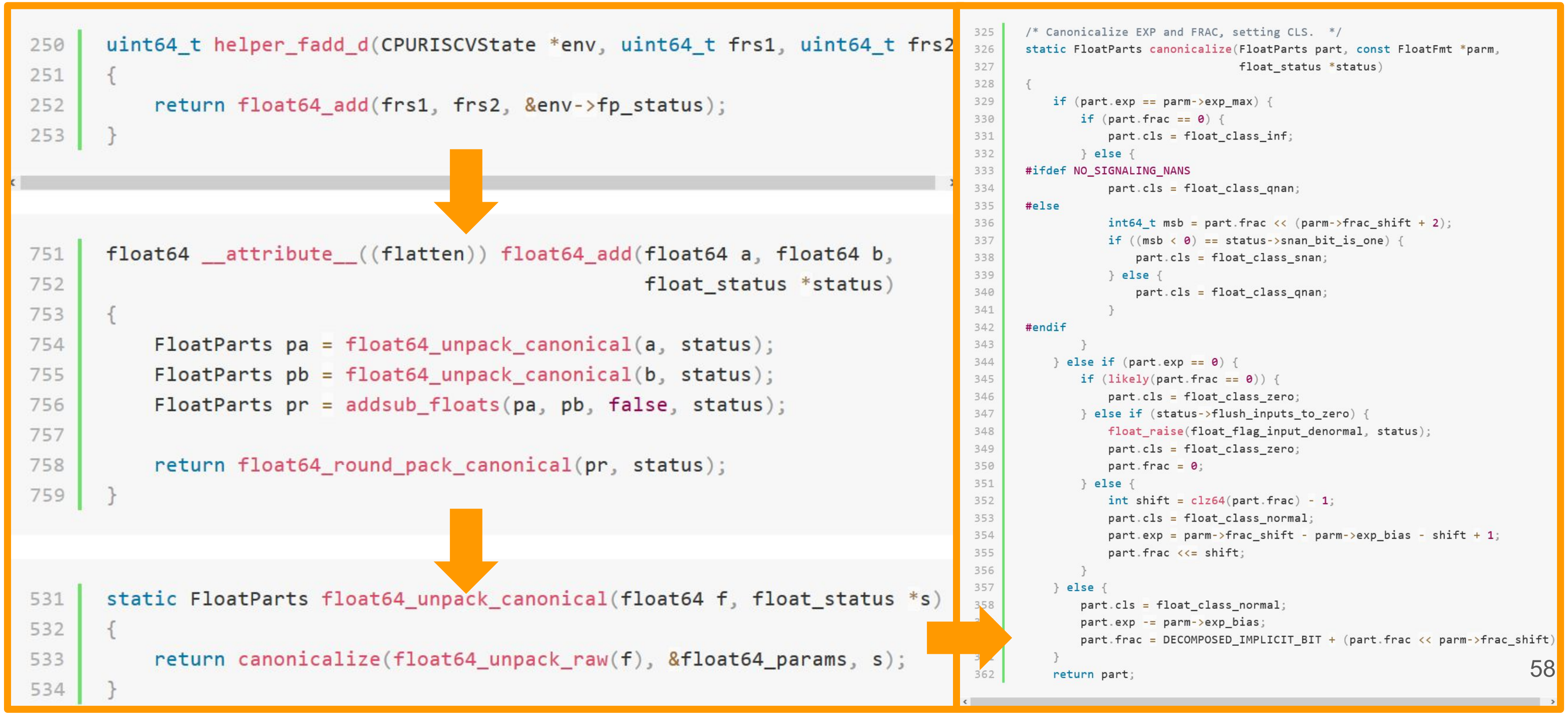
```
 movq      %r14, %rdi
 movq      %rbx, %rsi
 movq      %r12, %rdx
 callq     0x55e86662b7fd
 movq      %rax, 0xf8(%r14)
```

call helper_function

return result (%rax) to ‘**fa5**’

TCG-IR to x86_64 translation

- **helper - Emulate IEEE 754 floating point add with double precision**



```c
uint64_t helper_fadd_d(CPURISCVState *env, uint64_t frs1, uint64_t frs2
{
    return float64_add(frs1, frs2, &env->fp_status);
}
```

```c
float64 __attribute__((flatten)) float64_add(float64 a, float64 b,
                                             float_status *status)
{
    FloatParts pa = float64_unpack_canonical(a, status);
    FloatParts pb = float64_unpack_canonical(b, status);
    FloatParts pr = addsub_floats(pa, pb, false, status);

    return float64_round_pack_canonical(pr, status);
}
```

```c
static FloatParts float64_unpack_canonical(float64 f, float_status *s)
{
    return canonicalize(float64_unpack_raw(f), &float64_params, s);
}
```

```c
/* Canonicalize EXP and FRAC, setting CLS.  */
static FloatParts canonicalize(FloatParts part, const FloatFmt *parm,
                               float_status *status)
{
    if (part.exp == parm->exp_max) {
        if (part.frac == 0) {
            part.cls = float_class_inf;
        } else {
#ifdef NO_SIGNALING_NANS
            part.cls = float_class_qnan;
#else
            int64_t msb = part.frac << (parm->frac_shift + 2);
            if ((msb < 0) == status->snan_bit_is_one) {
                part.cls = float_class_snan;
            } else {
                part.cls = float_class_qnan;
            }
#endif
        }
    } else if (part.exp == 0) {
        if (likely(part.frac == 0)) {
            part.cls = float_class_zero;
        } else if (status->flush_inputs_to_zero) {
            float_raise(float_flag_input_denormal, status);
            part.cls = float_class_zero;
            part.frac = 0;
        } else {
            int shift = clz64(part.frac) - 1;
            part.cls = float_class_normal;
            part.exp = parm->frac_shift - parm->exp_bias - shift + 1;
            part.frac <<= shift;
        }
    } else {
        part.cls = float_class_normal;
        part.exp -= parm->exp_bias;
        part.frac = DECOMPOSED_IMPLICIT_BIT + (part.frac << parm->frac_shift)
    }
    return part;
```

TCG-IR to x86_64 translation

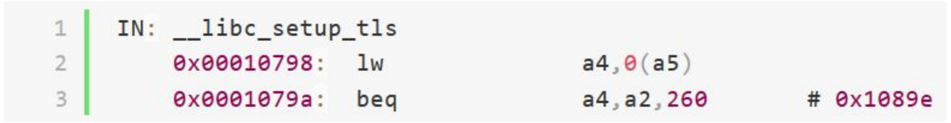
- **Prologue, epilogue - The entry point for each Translation Block**

```
IN: __libc_setup_tls
    0x00010798:  lw              a4,0(a5)
    0x0001079a:  beq             a4,a2,260        # 0x1089e
```

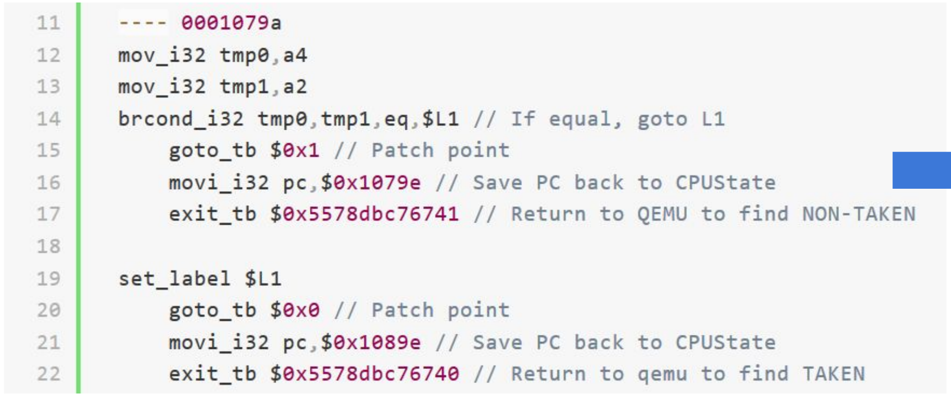
```
---- 0001079a
mov_i32 tmp0,a4
mov_i32 tmp1,a2
brcond_i32 tmp0,tmp1,eq,$L1 // If equal, goto L1
    goto_tb $0x1 // Patch point
    movi_i32 pc,$0x1079e // Save PC back to CPUState
    exit_tb $0x5578dbc76741 // Return to QEMU to find NON-TAKEN

set_label $L1
    goto_tb $0x0 // Patch point
    movi_i32 pc,$0x1089e // Save PC back to CPUState
    exit_tb $0x5578dbc76740 // Return to qemu to find TAKEN
```

TCG-IR to x86_64 translation

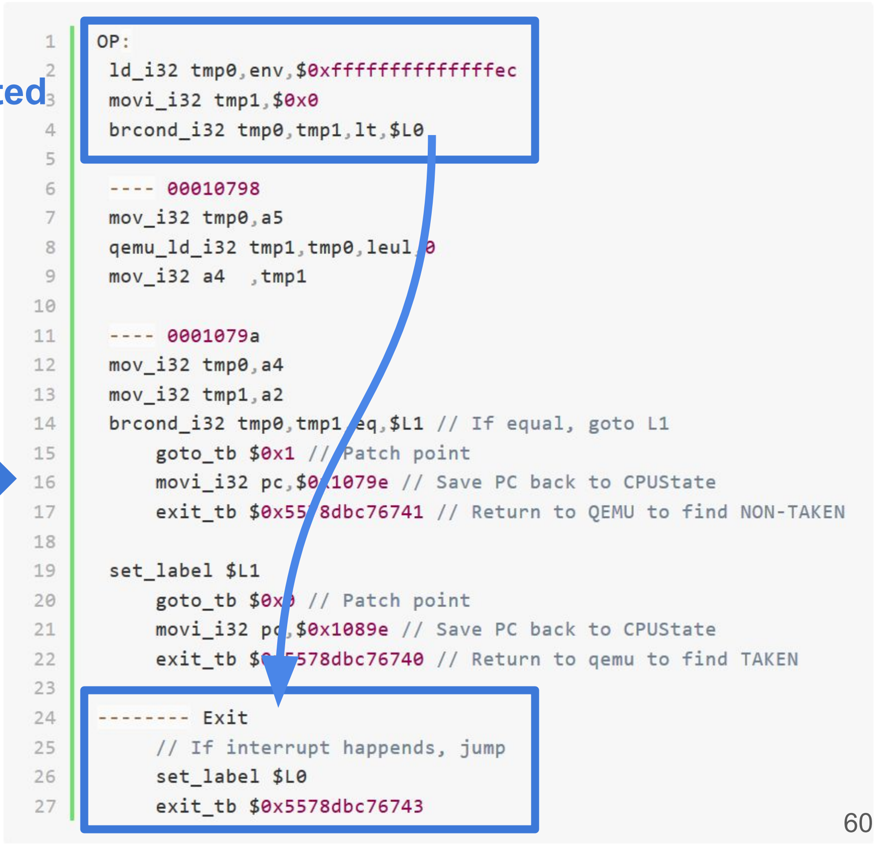
- **Prologue, epilogue**
  - **Decide whether current TB can be executed**



```
IN: __libc_setup_tls
    0x00010798:  lw          a4,0(a5)
    0x0001079a:  beq         a4,a2,260         # 0x1089e
```



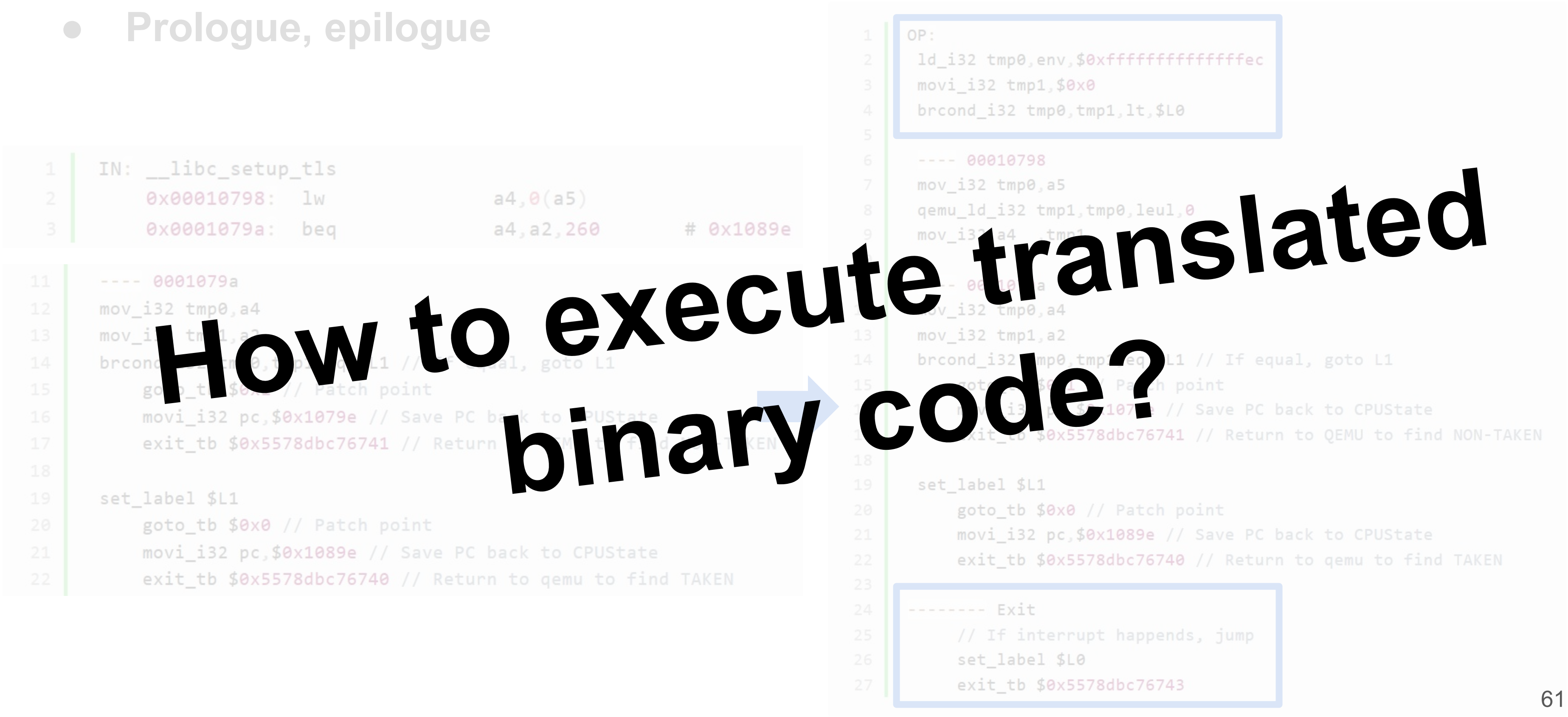
```
---- 0001079a
mov_i32 tmp0,a4
mov_i32 tmp1,a2
brcond_i32 tmp0,tmp1,eq,$L1 // If equal, goto L1
    goto_tb $0x1 // Patch point
    movi_i32 pc,$0x1079e // Save PC back to CPUState
    exit_tb $0x5578dbc76741 // Return to QEMU to find NON-TAKEN

set_label $L1
    goto_tb $0x0 // Patch point
    movi_i32 pc,$0x1089e // Save PC back to CPUState
    exit_tb $0x5578dbc76740 // Return to qemu to find TAKEN
```



```
OP:
 ld_i32 tmp0,env,$0xfffffffffffffec
 movi_i32 tmp1,$0x0
 brcond_i32 tmp0,tmp1,lt,$L0

 ---- 00010798
 mov_i32 tmp0,a5
 qemu_ld_i32 tmp1,tmp0,leul,0
 mov_i32 a4  ,tmp1

 ---- 0001079a
 mov_i32 tmp0,a4
 mov_i32 tmp1,a2
 brcond_i32 tmp0,tmp1,eq,$L1 // If equal, goto L1
     goto_tb $0x1 // Patch point
     movi_i32 pc,$0x1079e // Save PC back to CPUState
     exit_tb $0x5578dbc76741 // Return to QEMU to find NON-TAKEN

 set_label $L1
     goto_tb $0x0 // Patch point
     movi_i32 pc,$0x1089e // Save PC back to CPUState
     exit_tb $0x5578dbc76740 // Return to qemu to find TAKEN

-------- Exit
     // If interrupt happends, jump
     set_label $L0
     exit_tb $0x5578dbc76743
```

TCG-IR to x86_64 translation

- **Prologue, epilogue**



```
IN: __libc_setup_tls
    0x00010798:  lw               a4,0(a5)
    0x0001079a:  beq              a4,a2,260        # 0x1089e
```

```
---- 0001079a
mov_i32 tmp0,a4
mov_i32 tmp1,a2
brcond_i32 tmp0,tmp1,eq,$L1 // If equal, goto L1
    goto_tb $0x1 // Patch point
    movi_i32 pc,$0x1079e // Save PC back to CPUState
    exit_tb $0x5578dbc76741 // Return to QEMU to find NON-TAKEN

set_label $L1
    goto_tb $0x0 // Patch point
    movi_i32 pc,$0x1089e // Save PC back to CPUState
    exit_tb $0x5578dbc76740 // Return to qemu to find TAKEN
```

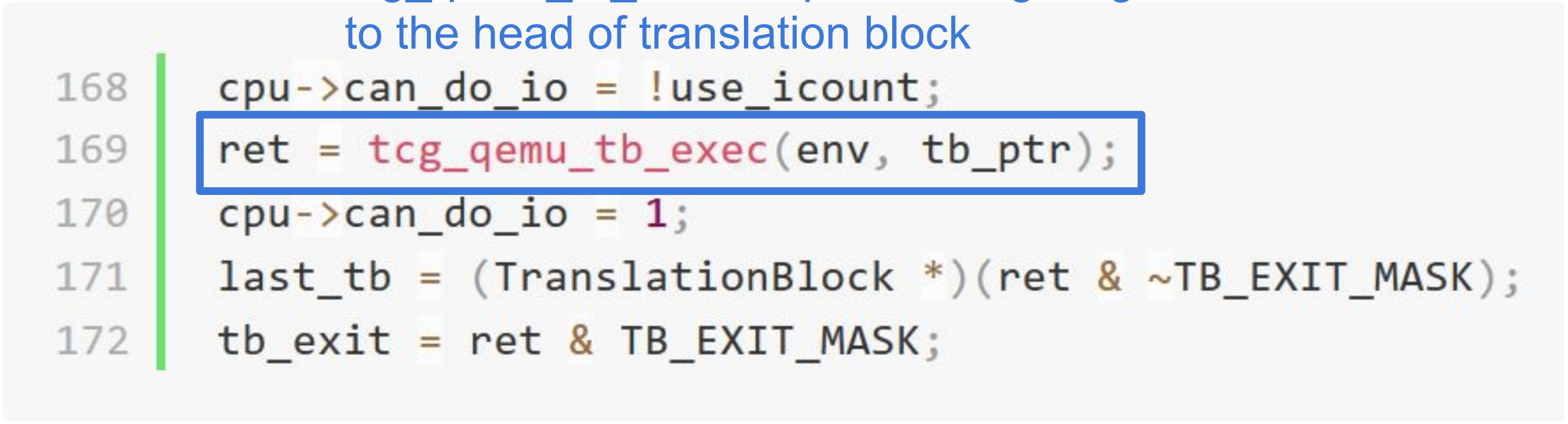
```
OP:
 ld_i32 tmp0,env,$0xffffffffffffffec
 movi_i32 tmp1,$0x0
 brcond_i32 tmp0,tmp1,lt,$L0

 ---- 00010798
 mov_i32 tmp0,a5
 qemu_ld_i32 tmp1,tmp0,leul,0
 mov_i32 a4,tmp1

 ---- 0001079a
 mov_i32 tmp0,a4
 mov_i32 tmp1,a2
 brcond_i32 tmp0,tmp1,eq,$L1 // If equal, goto L1
     goto_tb $0x1 // Patch point
     movi_i32 pc,$0x1079e // Save PC back to CPUState
     exit_tb $0x5578dbc76741 // Return to QEMU to find NON-TAKEN

 set_label $L1
     goto_tb $0x0 // Patch point
     movi_i32 pc,$0x1089e // Save PC back to CPUState
     exit_tb $0x5578dbc76740 // Return to qemu to find TAKEN

-------- Exit
     // If interrupt happends, jump
     set_label $L0
     exit_tb $0x5578dbc76743
```

How to execute translated binary code?

x86_64 code execution

- **Entry point -** qemu-riscv/accel/tcg/cpu-exec.c

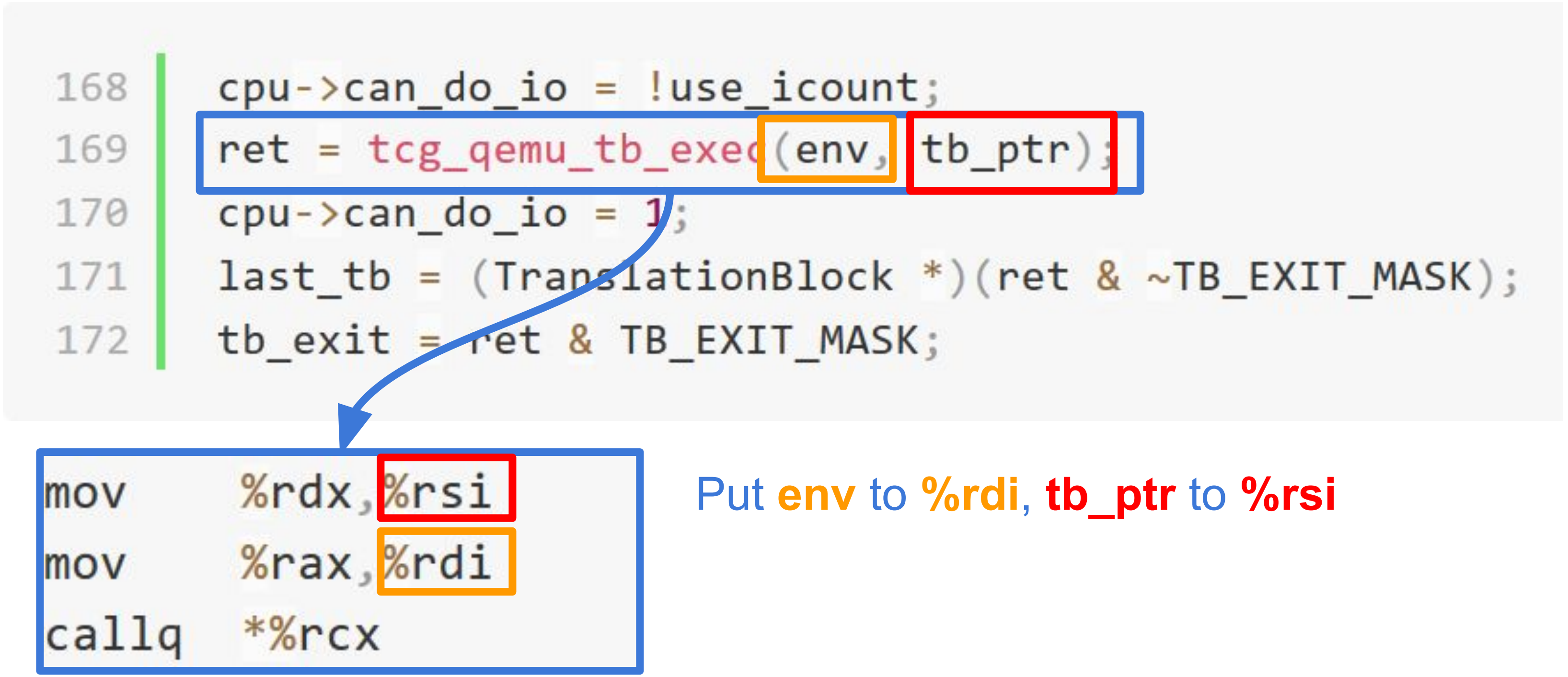
tcg_qemu_tb_exec: A pointer targeting to the head of translation block



```
cpu->can_do_io = !use_icount;
ret = tcg_qemu_tb_exec(env, tb_ptr);
cpu->can_do_io = 1;
last_tb = (TranslationBlock *)(ret & ~TB_EXIT_MASK);
tb_exit = ret & TB_EXIT_MASK;
```

x86_64 code execution

- **Entry point -** qemu-riscv/accel/tcg/cpu-exec.c



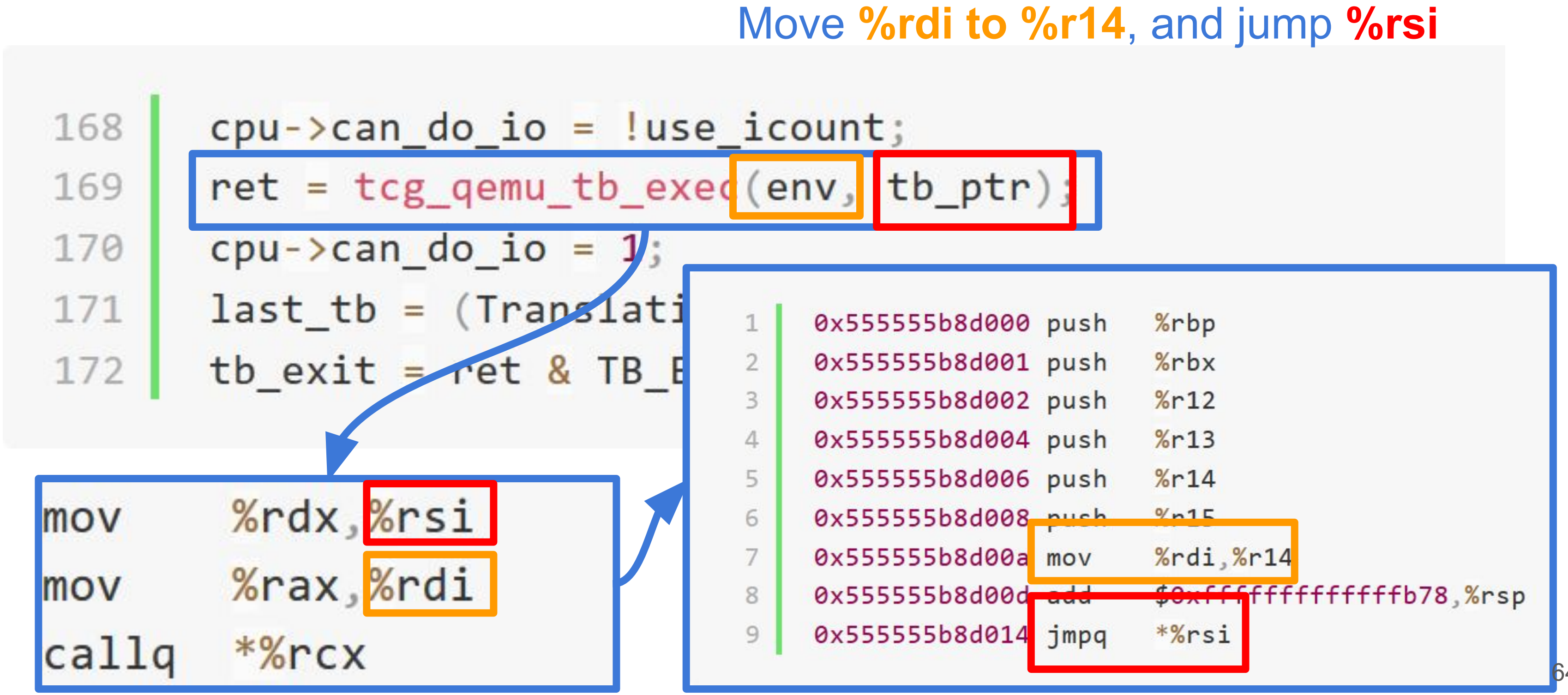
```
cpu->can_do_io = !use_icount;
ret = tcg_qemu_tb_exec(env, tb_ptr);
cpu->can_do_io = 1;
last_tb = (TranslationBlock *)(ret & ~TB_EXIT_MASK);
tb_exit = ret & TB_EXIT_MASK;
```

```
mov     %rdx,%rsi
mov     %rax,%rdi
callq   *%rcx
```

Put **env** to **%rdi**, **tb_ptr** to **%rsi**

x86_64 code execution

- **Entry point -** qemu-riscv/accel/tcg/cpu-exec.c

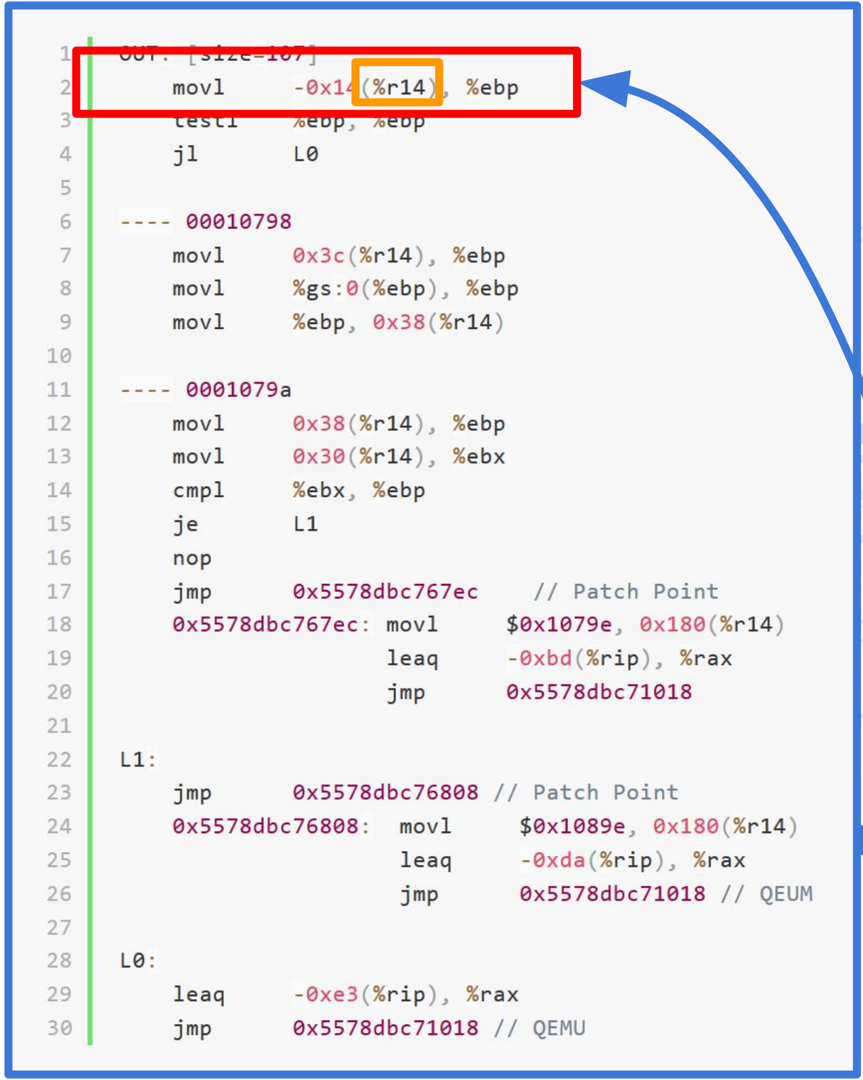


Move %rdi to %r14, and jump %rsi

```
cpu->can_do_io = !use_icount;
ret = tcg_qemu_tb_exec(env, tb_ptr);
cpu->can_do_io = 1;
last_tb = (Translati
tb_exit = ret & TB_E
```

```
mov     %rdx,%rsi
mov     %rax,%rdi
callq   *%rcx
```

```
0x555555b8d000 push   %rbp
0x555555b8d001 push   %rbx
0x555555b8d002 push   %r12
0x555555b8d004 push   %r13
0x555555b8d006 push   %r14
0x555555b8d008 push   %r15
0x555555b8d00a mov    %rdi,%r14
0x555555b8d00d add    $0xfffffffffffffb78,%rsp
0x555555b8d014 jmpq   *%rsi
```



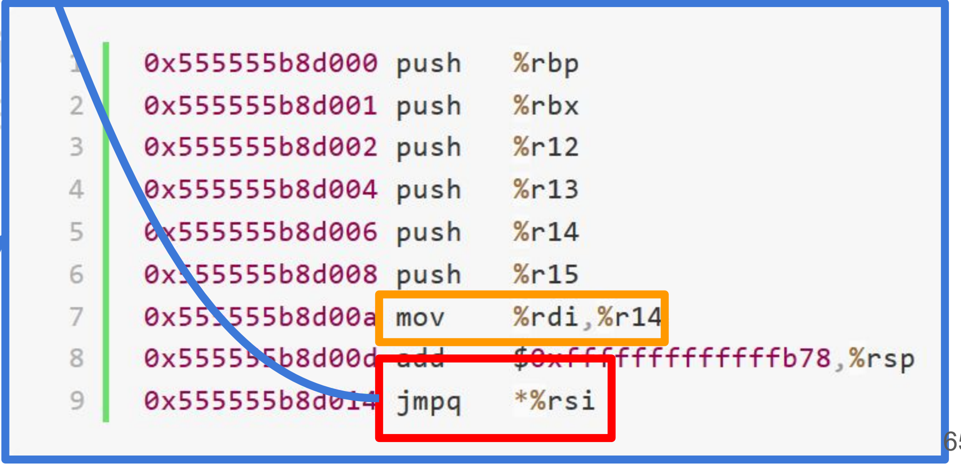
```
OUT: [size=107]
     movl     -0x14(%r14), %ebp
     testl    %ebp, %ebp
     jl       L0

---- 00010798
     movl     0x3c(%r14), %ebp
     movl     %gs:0(%ebp), %ebp
     movl     %ebp, 0x38(%r14)

---- 0001079a
     movl     0x38(%r14), %ebp
     movl     0x30(%r14), %ebx
     cmpl     %ebx, %ebp
     je       L1
     nop
     jmp      0x5578dbc767ec    // Patch Point
     0x5578dbc767ec: movl     $0x1079e, 0x180(%r14)
                     leaq     -0xbd(%rip), %rax
                     jmp      0x5578dbc71018

L1:
     jmp      0x5578dbc76808 // Patch Point
     0x5578dbc76808:  movl     $0x1089e, 0x180(%r14)
                      leaq     -0xda(%rip), %rax
                      jmp      0x5578dbc71018 // QEUM

L0:
     leaq     -0xe3(%rip), %rax
     jmp      0x5578dbc71018 // QEMU
```

g/cpu-exec.c

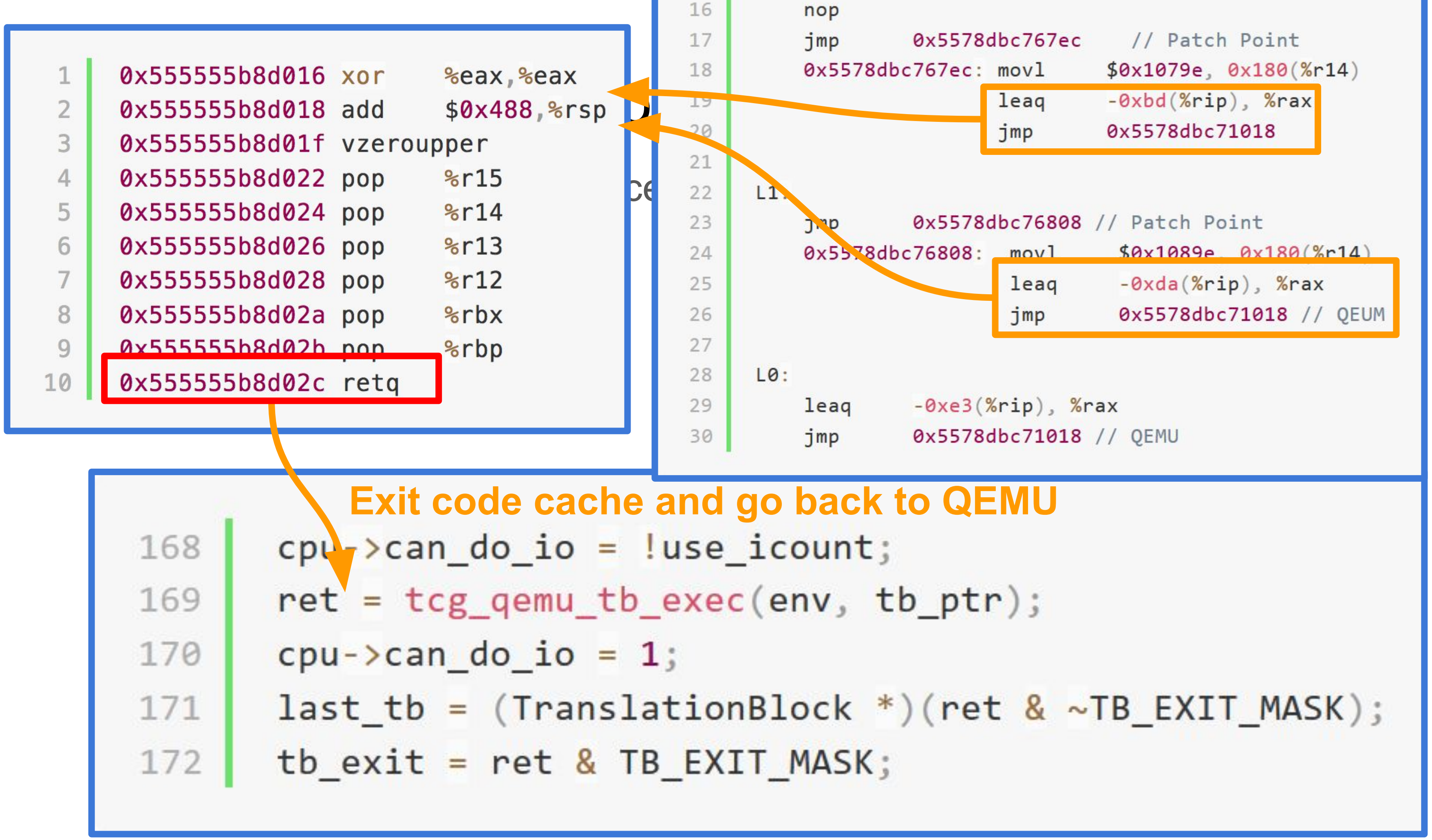
Move %rdi to %r14, and jump %rsi

se_icount;

exec(env, tb_ptr);



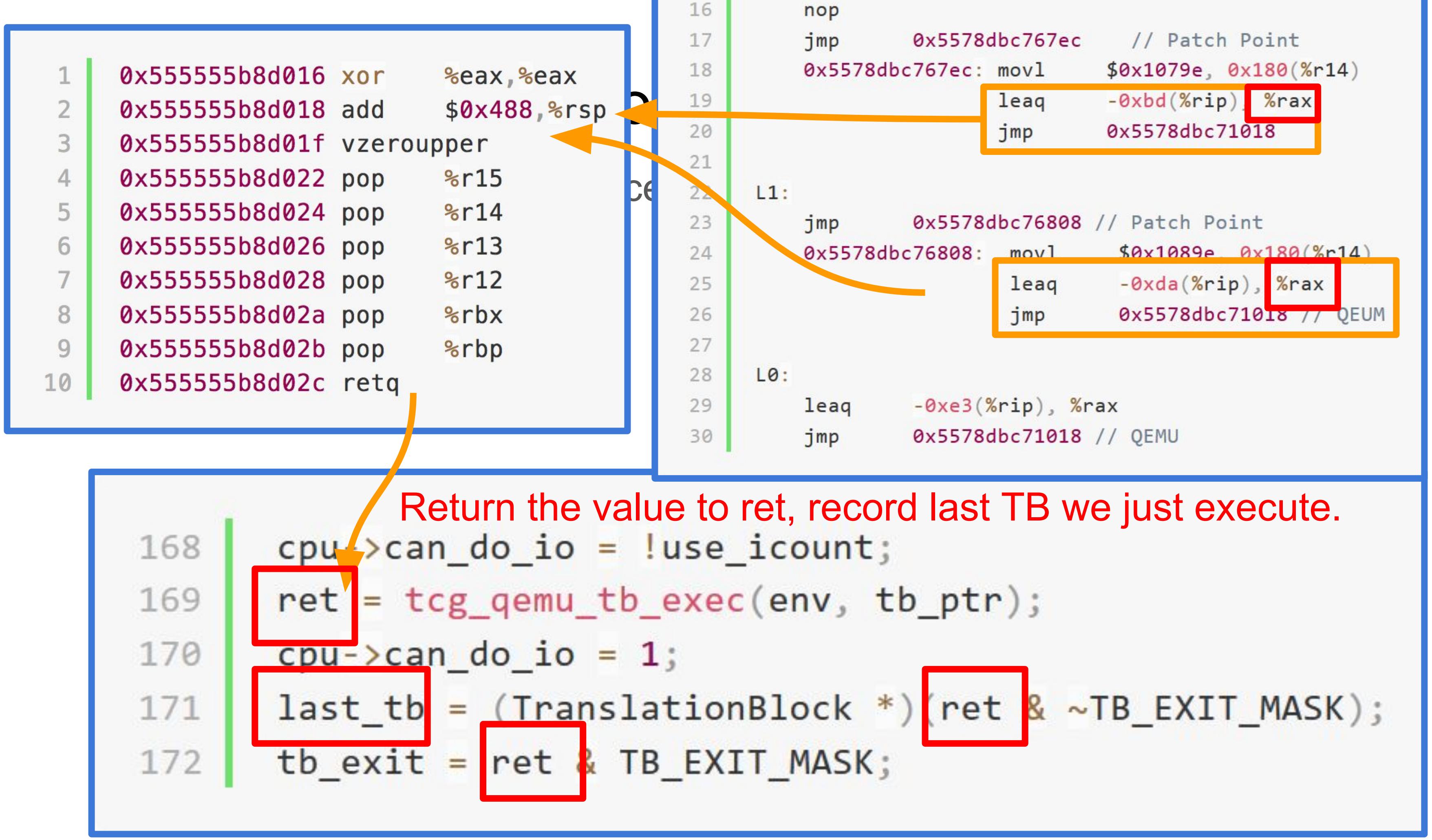
```
0x555555b8d000 push   %rbp
0x555555b8d001 push   %rbx
0x555555b8d002 push   %r12
0x555555b8d004 push   %r13
0x555555b8d006 push   %r14
0x555555b8d008 push   %r15
0x555555b8d00a mov    %rdi,%r14
0x555555b8d00d add    $0xfffffffffffffb78,%rsp
0x555555b8d014 jmpq   *%rsi
```



```
0x555555b8d016 xor    %eax,%eax
0x555555b8d018 add    $0x488,%rsp
0x555555b8d01f vzeroupper
0x555555b8d022 pop    %r15
0x555555b8d024 pop    %r14
0x555555b8d026 pop    %r13
0x555555b8d028 pop    %r12
0x555555b8d02a pop    %rbx
0x555555b8d02b pop    %rbp
0x555555b8d02c retq
```

```
        nop
        jmp        0x5578dbc767ec     // Patch Point
        0x5578dbc767ec:  movl      $0x1079e, 0x180(%r14)
                         leaq      -0xbd(%rip), %rax
                         jmp       0x5578dbc71018


    L1:
        jmp        0x5578dbc76808 // Patch Point
        0x5578dbc76808:  movl      $0x1089e, 0x180(%r14)
                          leaq      -0xda(%rip), %rax
                          jmp       0x5578dbc71018 // QEUM


    L0:
        leaq       -0xe3(%rip), %rax
        jmp        0x5578dbc71018 // QEMU
```

```
cpu->can_do_io = !use_icount;
ret = tcg_qemu_tb_exec(env, tb_ptr);
cpu->can_do_io = 1;
last_tb = (TranslationBlock *)(ret & ~TB_EXIT_MASK);
tb_exit = ret & TB_EXIT_MASK;
```



```
0x555555b8d016 xor     %eax,%eax
0x555555b8d018 add     $0x488,%rsp
0x555555b8d01f vzeroupper
0x555555b8d022 pop     %r15
0x555555b8d024 pop     %r14
0x555555b8d026 pop     %r13
0x555555b8d028 pop     %r12
0x555555b8d02a pop     %rbx
0x555555b8d02b pop     %rbp
0x555555b8d02c retq
```

```
        nop
        jmp       0x5578dbc767ec     // Patch Point
        0x5578dbc767ec:  movl      $0x1079e, 0x180(%r14)
                         leaq      -0xbd(%rip), %rax
                         jmp       0x5578dbc71018

    L1:
        jmp       0x5578dbc76808 // Patch Point
        0x5578dbc76808:  movl      $0x1089e, 0x180(%r14)
                         leaq      -0xda(%rip), %rax
                         jmp       0x5578dbc71018 // QEUM

    L0:
        leaq      -0xe3(%rip), %rax
        jmp       0x5578dbc71018 // QEMU
```

Return the value to ret, record last TB we just execute.

```
cpu->can_do_io = !use_icount;
ret = tcg_qemu_tb_exec(env, tb_ptr);
cpu->can_do_io = 1;
last_tb = (TranslationBlock *)(ret & ~TB_EXIT_MASK);
tb_exit = ret & TB_EXIT_MASK;
```

Reference

- Qemu JIT Code Generator and System Emulation
- QEMU - Binary Translation
- QEMU TCG Frontend Ops
- RISC-V Insturction Set Manual
- Doraemon’s Notes: QEMU Backend
    - Written in Mandarin Chinese