

QEMU

20190822

Loongson Lab - Binary Translation

主要内容

- 相关概念 & 主要目录结构
- 代码块 **TranslationBlock** 组织结构
- 主循环：查找、翻译、执行、上下文切换
- 细节问题探索
- 已有的优化措施

概念: target & host

- target: 被 QEMU 模拟的硬件平台, 也叫 guest
- host: 运行 QEMU 的硬件平台
- 例: 在 Linux MIPS 上运行 Linux x86 程序
 - target: x86
 - host: MIPS

概念：TCG

- 前端翻译：从 target 平台到 QEMU 的中间代码
- 后端翻译：从 QEMU 中间代码到 host 平台
- Tiny Code Generator
 - tcg_gen_xxx 前端翻译 上下文数据结构 DisasContext
 - tcg_out_xxx 后端翻译 上下文数据结构 TCGContext
- 例：在 Linux MIPS 平台上由 QEMU 运行 Linux x86 程序



概念： 地址

- 代码作为数据保存在内存中，由 QEMU 管理
 - 翻译后地址不同 / 不同架构对内存空间理解不同 / 主要是数据段和代码段
- SPC: 源地址，前端翻译前代码在 target 上的虚拟地址 (GVA)
- TPC: 目标地址，后端翻译后代码在 host 上的虚拟地址 (HVA)
- QEMU以代码块为基本单位，由 TranslationBlock 数据结构管理
 - SPC保存在 TranslationBlock.pc 中
 - TPC所在的物理页地址保存在属性 TranslationBlock.page_addr 中
 - 完整过程: GVA - GPA - HVA - HPA
 - QEMU负责 GVA - GPA - HVA, host 系统负责 HVA - HPA

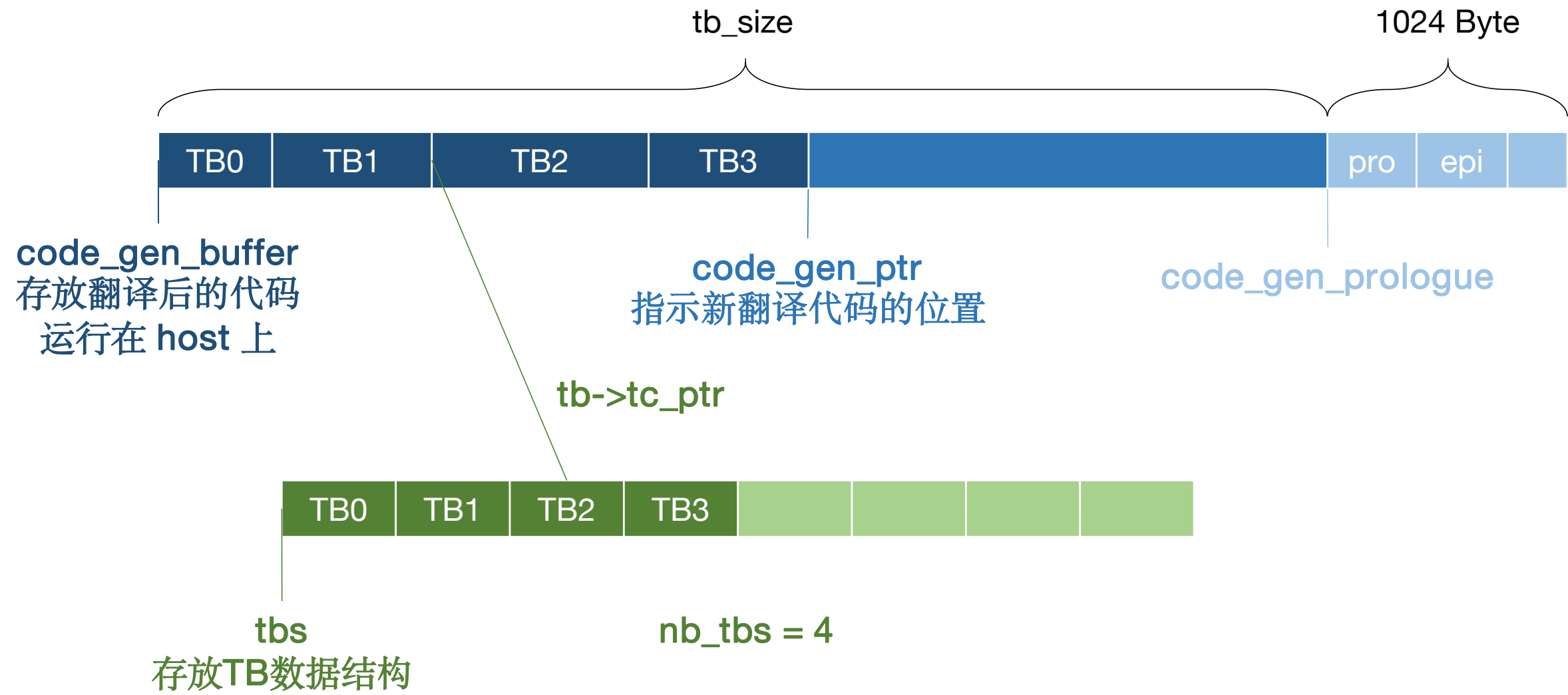
主要目录结构

- `linux-usr/` `main`函数，负责解析加载二进制文件等初始化工作
- `cpu-exec.c` 主循环：基本块的查找、翻译、执行
- `exec.c` 基本块的管理、虚拟页的管理
- `translate-all.c` QEMU 的翻译框架，调用两次翻译过程，生成 `host` 代码
- `tcg/` 即 TCG，`tcg.c` 和 `tcg.h` 定义如何生成中间代码
文件夹如 `mips` 定义如何从中间代码生成特定架构指令 (`mips`)
- `target-i386/` 对模拟 `i386` 的支持，如 `i386` 的虚拟CPU定义 (`CPUX86State`)
`i386` 指令到中间代码的翻译 (解析 `i386` 指令并调用 TCG)

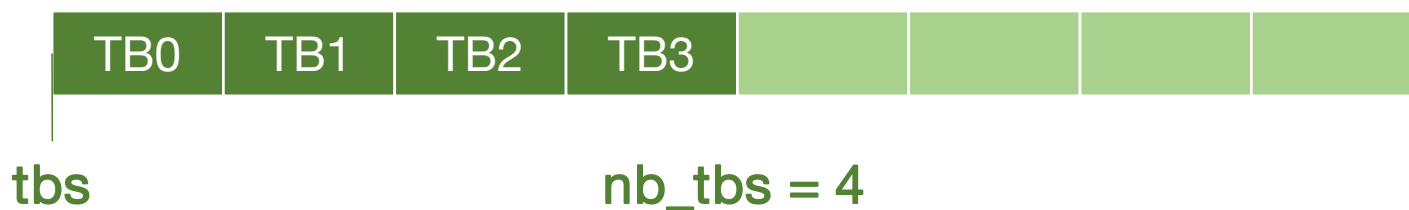
TranslationBlock 数据结构

target_ulong	pc	}	• PC
target_ulong	cs_base		• target 代码
uint16_t	size		• 代码段基址 cs_base
uint8_t	*tc_ptr	}	• 大小 size
tb_page_addr_t	page_addr[2]		➤ 指针 tc_ptr
struct TranslationBlock *phys_hash_next		➤ host 代码	➤ 所在 host 页的虚地址
uint16_t	tb_next_offset[2]	}	➔ 用于哈希表链接的指针
uint16_t	tb_jmp_offset[2]		运行时动态绑定 TB 间跳转关系
unsigned long	tb_next[2]		
struct TranslationBlock *tb_loopup_cache[N]		➔	运行时快速查找下一个 TB

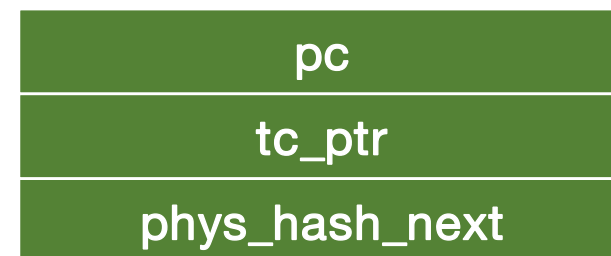
TB数据内存分配



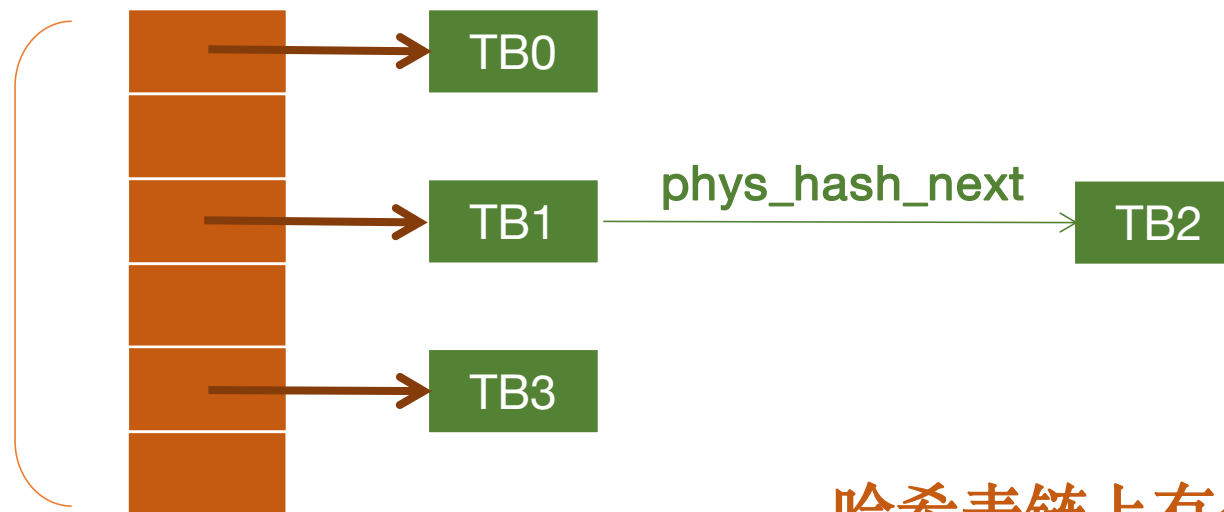
TB数据结构组织



TranslationBlock



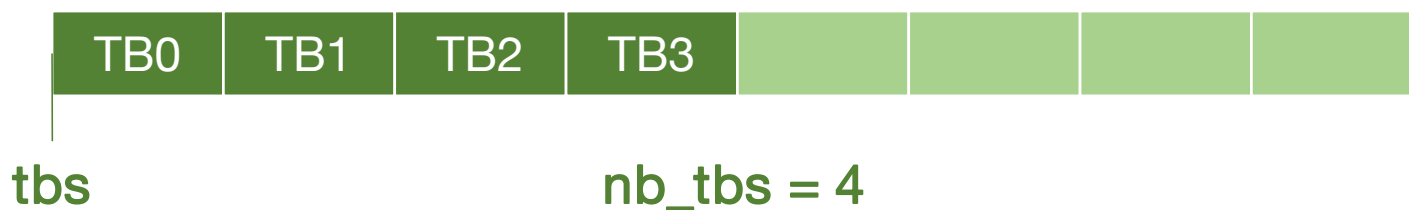
索引
`hash_func(phys_pc)`
通过物理地址计算



tb_phys_hash

哈希表链上有全部的TB

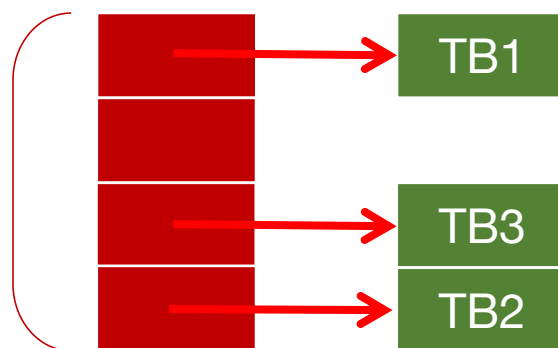
TB数据结构组织



TranslationBlock

pc
tc_ptr
phys_hash_next

索引
hash_func(pc)
通过 tb->pc 计算



env->tb_jmp_cache

cache存有部分或全部的TB
相当于每个node只有一个TB的哈希表

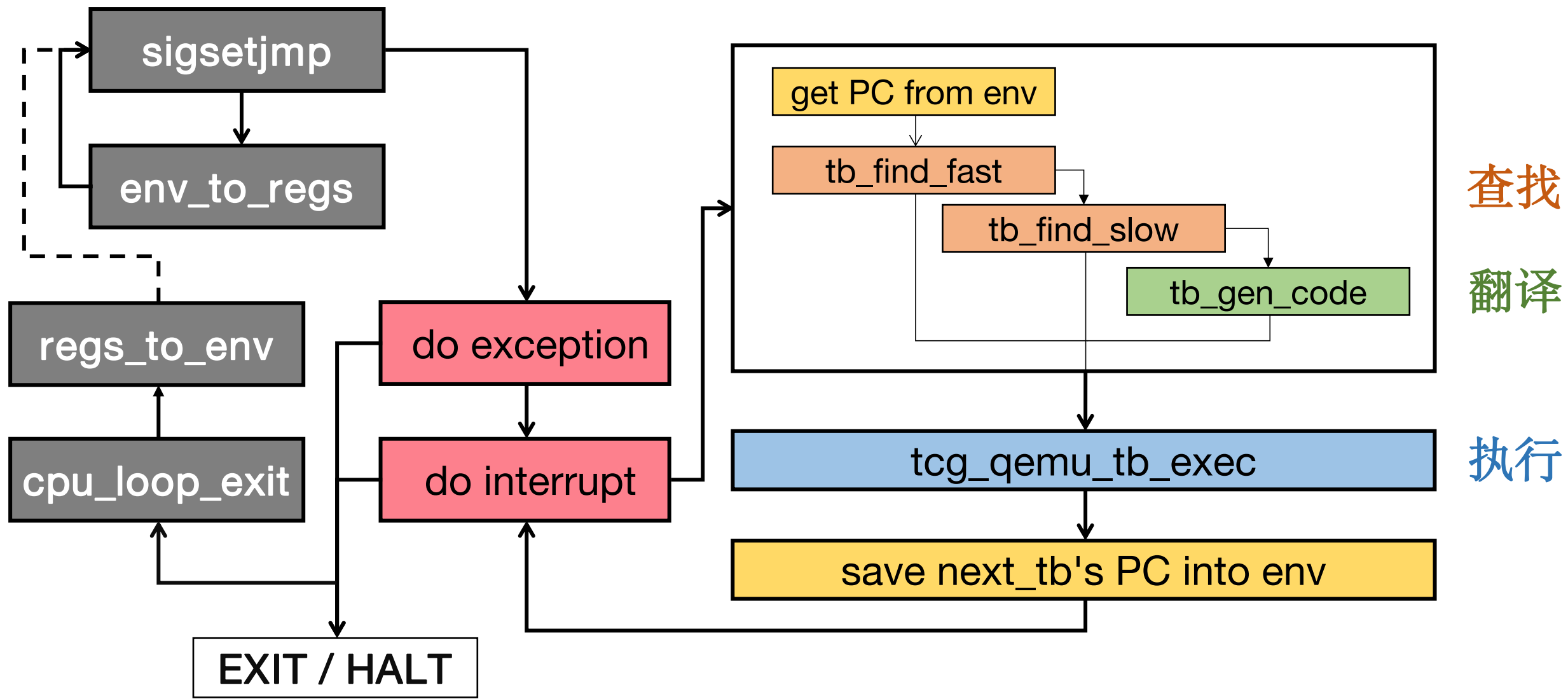
主循环: `cpu_exec(env)`

- `cpu-exec.c:277` `int cpu_exec(CPUState *env1)`
 - `TranslationBlock *tb`; `uint8_t *tc_ptr`; `unsigned long next_tb`;
 - `if (sigsetjmp(env->jmp_env, 1) == 0) {`
 - `if(env->exception_index >= 0) do_interrupt(env);`
 - `interrupt_request = env->interrupt_request; // DEBUG HALT HARD`
 - `tb = tb_find_fast(); // according to env->eip`
 - `tb->tb_lookup_cache[tb_lookup_hash(tb->pc)] = tb //TB_LOOKUP_OPT`
 - `tb_add_jump((TranslationBlock *)(next_tb & ~3), next_tb & 3, tb);`
 - `env->current_tb = tb; barrier();`
 - `tc_ptr = tb->tc_ptr`
 - `next_tb = tcg_qemu_tb_exec(tc_ptr);`
 - `tb = (TranslationBlock *)(long)(next_tb & ~3);`
 - `cpu_pc_from_tb(env,tb);`
 - `} else env_to_regs(env);`

for
(;;)

spin
lock

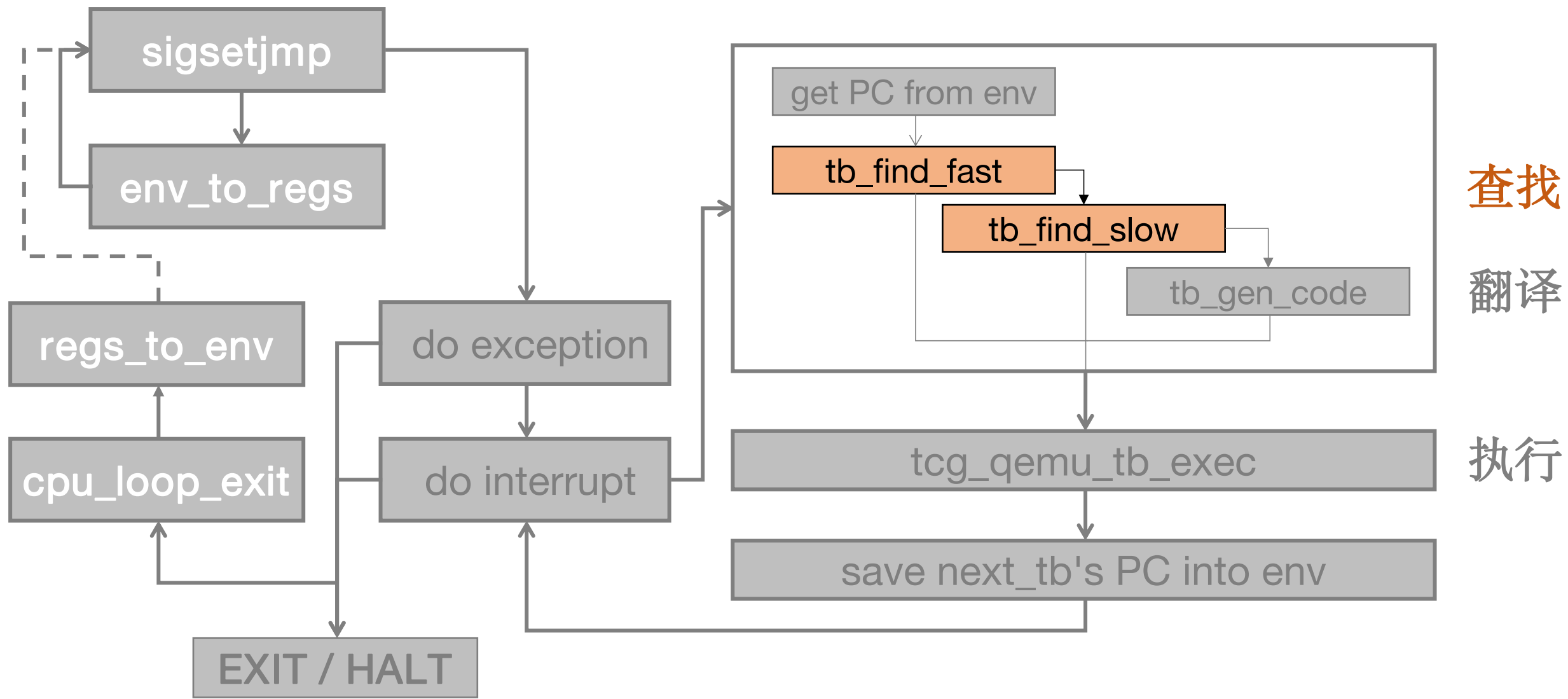
主循环: `cpu_exec(env)`



主循环: `cpu_exec(env)`

- `sigsetjmp(env, 1)` 将 CPU 状态保存到 `env` 中, 并返回 0
 - `pc, sp, s0-s7, fp, gp`
- `siglongjmp(env, 1)` 根据 `env` 保存的数据恢复 CPU 状态
 - 从而跳转到上一个 `sigsetjmp` 函数处, 并使其返回非 0 值
- `env_to_regs(env)` 将 `env` 保存到寄存器写入 CPU 寄存器中
- `regs_to_env(env)` 将 CPU 寄存器内容保存到 `env` 中
 - `CONFIG_DIRECT_REGS`时使用这两个函数来恢复/保存
- `cpu_pc_from_tb(*env, *tb)` 重置 `env` 中 `eip` 为 `tb` 的 `pc`
 - `env->eip = tb->pc - tb->cs_base`

主循环: `cpu_exec(env)`



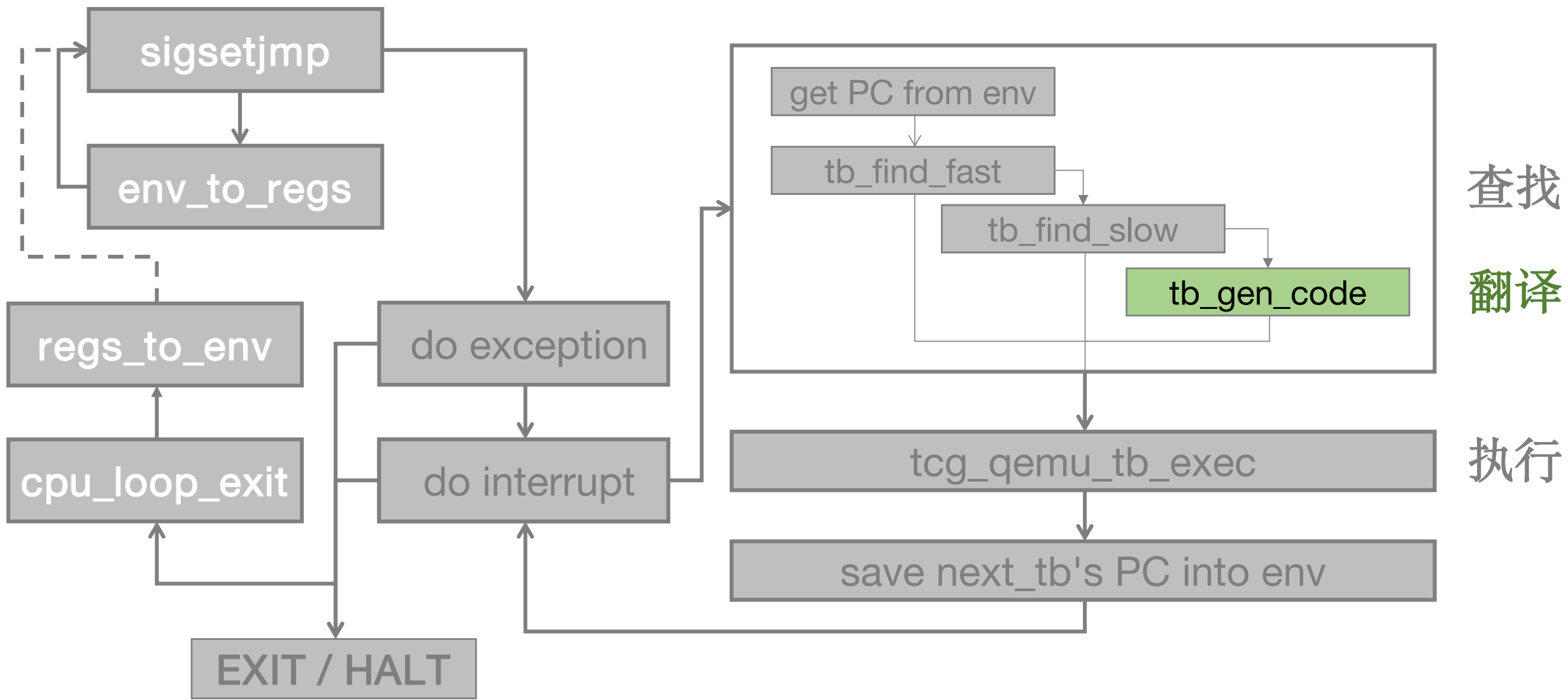
查找TB: `tb_find_fast()`

- `cpu-exec.c:232 TranslationBlock *tb_find_fast()`
 - `cpu_get_tb_cpu_state(env, &pc, &cs_base, &flags);`
 - `tb = env->tb_jmp_cache[tb_jmp_cache_hash_func(pc)];`
 - `if(no/wrong tb found)`
 - `tb = tb_find_slow(pc, cs_base, flags);`
- `target-i386/cpu.h:1000`
`cpu_get_tb_cpu_state(*env, *pc ,*cs_base, *flags)`
 - `*cs_base = env->segs[R_CS].base;`
 - `*pc = *cs_base + env->eip;`

查找TB: `tb_find_slow(pc,cs_base,flags)`

- `cpu-exec.c:177` `TranslationBlock *tb_find_slow(pc,cs_base,flags)`
 - **phys_pc** = `get_page_addr_code(env, pc);`
 - `phys_page1 = phys_pc & TARGET_PAGE_MASK; phys_page2 = -1;`
 - **hash** = `tb_phys_hash_func(phys_pc); ptb1 = &tb_phys_hash[hash];`
 - **tb** = `*ptb1`
 - `if(!tb) goto not_found;`
 - `// check pc & cs_base & flags & tb->page_addr[0:1] // 连续两页 or 只有一页`
 - `check OK goto found;`
 - **ptb1** = `&tb->phys_hash_next;`
 - `not_found: tb = tb_gen_code(env, pc, cs_base, flags, 0);`
 - `found: move tb to the head of the list in tb_phys_hash`
 - **env->tb_jump_cache[tb_jump_cache_hash_func(**pc**)] = **tb**;**

主循环: `cpu_exec(env)`

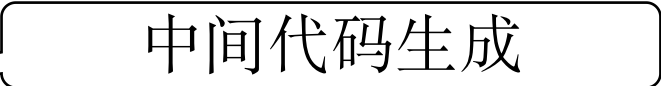



翻译TB: `tb_gen_code(env, pc, cs_base, flags, cflags)`

- `exec.c:1088 tb_gen_code(env, pc, cs_base, flags, cflags)`
 - `phys_pc = get_page_addr_code(env, pc);`
 - `tb = tb_alloc(pc);` // tbs 数组
 - `tb->tc_ptr = code_gen_ptr;` // 代码缓冲区顶部指针
 - `tb->cs_base, flags, cflags`
 - `cpu_gen_code(env, tb, &code_gen_size);`
 - `code_gen_ptr += code_gen_size;` // global
 - `tb_link_page(tb, phys_pc, phys_page2);`

调用翻译函数

翻译TB: `cpu_gen_code(*env, *tb, &code_gen_size);`

- target-i386/cpu.h: `#define cpu_gen_code cpu_x86_gen_code`
- translate-all.c:63 `int cpu_gen_code(*env, *tb, *code_gen_size)`
 - `TCGContext *s = &tcg_ctx;`
 - `tcg_func_start(s);`
 - **`gen_intermediate_code(env, tb);`**  中间代码生成
 - `s->tb_next_offset, tb_jump_offset, tb_next`
 - **`gen_code_buf = tb->tc_ptr;`**
 - `gen_code_size = tcg_gen_code(s, gen_code_buf);`  host 代码生成

翻译TB - 前端: `gen_intermediate_code(...)`

- target-i386/translate.c:9627

`gen_intermediate_code_internal(*env, *tb, search_pc)`

- // generate intermediate code in `gen_opc_buf` and `gen_opprarm_buf` for
- // basic block 'tb'. if search_pc is TRUE, also generate PC information
- // for each intermediate instruction.
- target_ulong **disas_insn**(DisasContext *s, target_ulong pc_start)
 - // convert one instruction
 - b = **ldub_code**(s->pc) // get next byte
- translate-all.c:43 uint16_t `gen_opc_buf`[OPC_BUF_SIZE]
- translate-all.c:44 TCGArg `gen_opparam_buf`[OPPARAM_BUF_SIZE]

翻译TB - 后端: tcg_gen_code(*s, *code_buf)

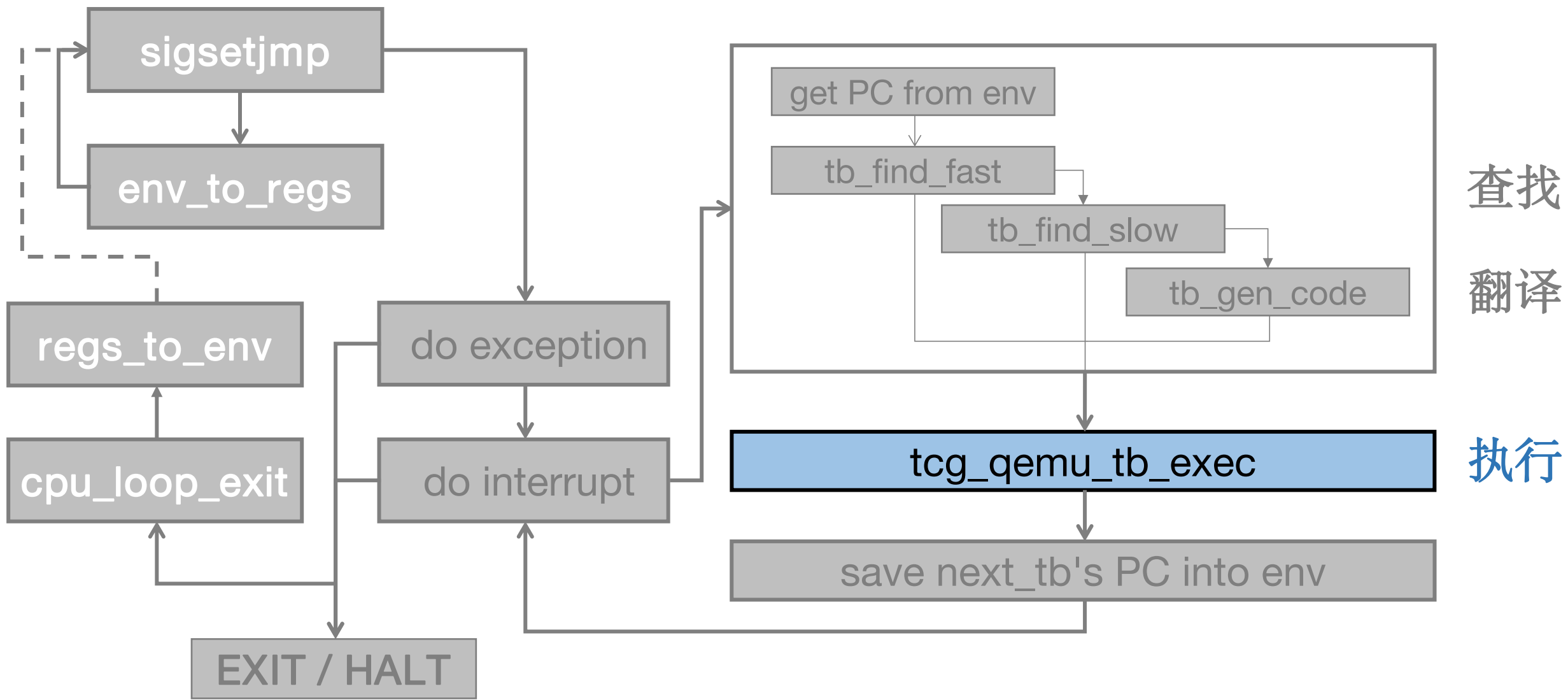
- tcg/tcg.c:2166 tcg_gen_code(*s, *gen_code_buf)
 - tcg_gen_code_common(s, gen_code_buf, -1);
- tcg/tcg.c:2015 tcg_gen_code_common(*s, *buf, long search_pc)
 - TCGOpcode opc;
 - args = gen_opparam_buf; s->code_buf = gen_code_buf;
 - opc = gen_opc_buf[op_index];
 - switch(opc)
 - default:
tcg_reg_alloc_op(s, def, opc, args, dead_iargs);
 - op_index++;

for(;;)

TCG翻译: `tcg_reg_alloc_op(*s, *def, opc, *args, ...)`

- `tcg/tcg.c:1648 tcg_reg_alloc_op(*s, *def, opc, *args, dead_iargs)`
 - `tcg_reg_alloc`
 - `tcg_regset_set_reg`
 - `tcg_out_op(s, opc, new_args, const_args)`
- `tcg/mips/tcg-target.c:1992 tcg_out_op(*s, opc, *args, *const_args)`

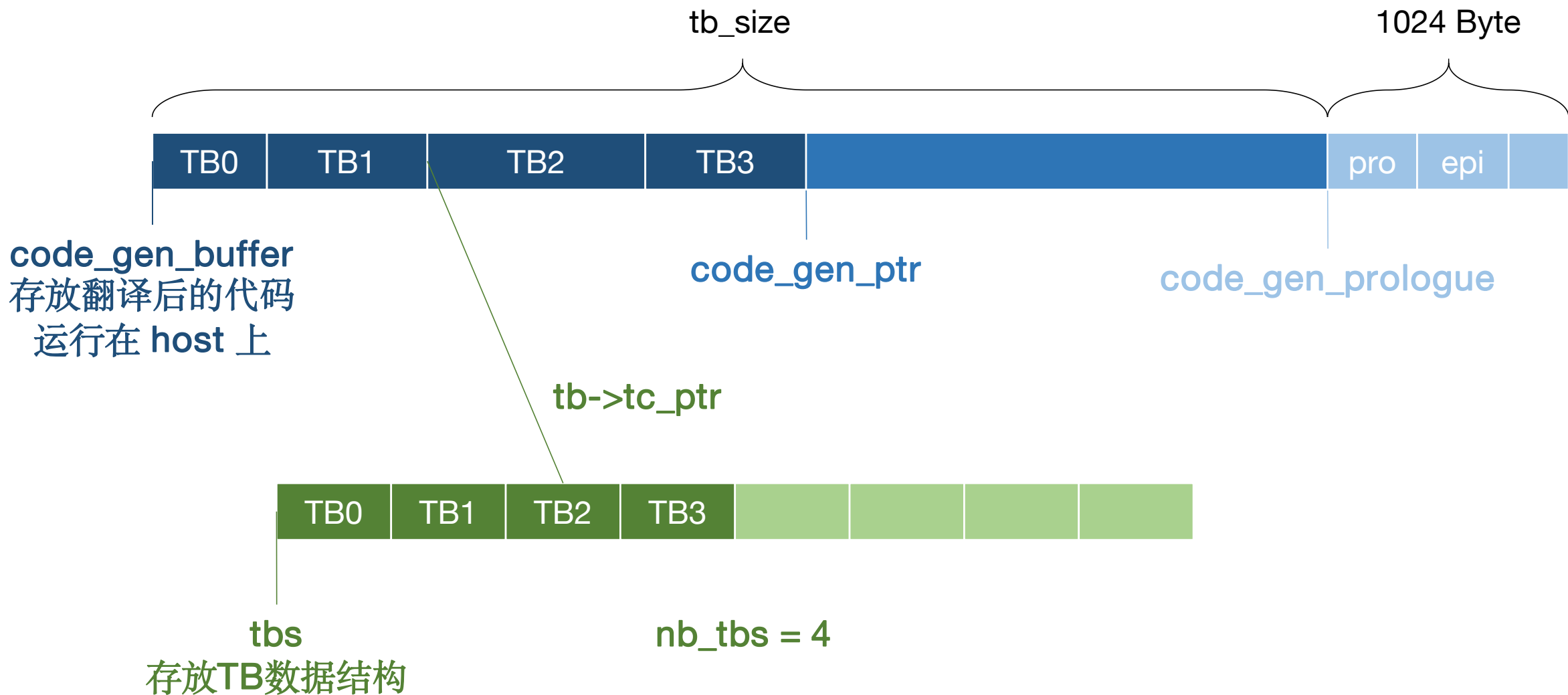
主循环: `cpu_exec(env)`



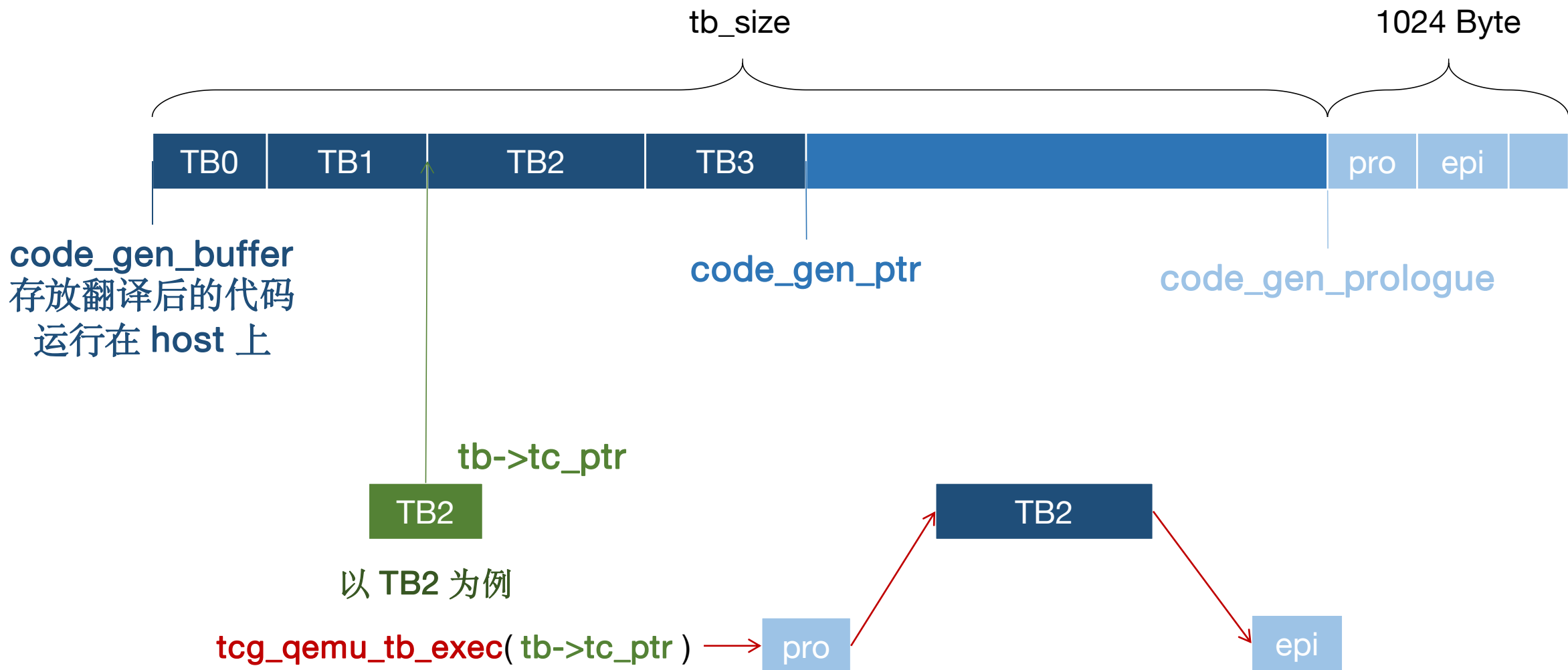
执行TB: tcg_qemu_tb_exec(tb_ptr)

- tcg/tcg.h:509 `#define tcg_qemu_tb_exec(tb_ptr) \`
`((long REGPARAM __attribute__((longcall)) (*)(void *))code_gen_prologue)(tb_ptr)`
- exec.c:117 `uint8_t *code_gen_prologue;`
- exec.c:118 `static uint8_t *code_gen_buffer;`
- exec.c:119 `static unsigned long code_gen_buffer_size;`
- exec.c:122 `static uint8_t *code_gen_ptr;`
- exec.c:507 `static void code_gen_alloc(unsigned long tb_size)`
 - `code_gen_buffer_size = tb_size;`
 - `code_gen_buffer = mmap(start, code_gen_buffer_size + 1024, ...); // linux only`
 - `code_gen_prologue = code_gen_buffer + code_gen_buffer_size;`

回顾: TB数据内存分配



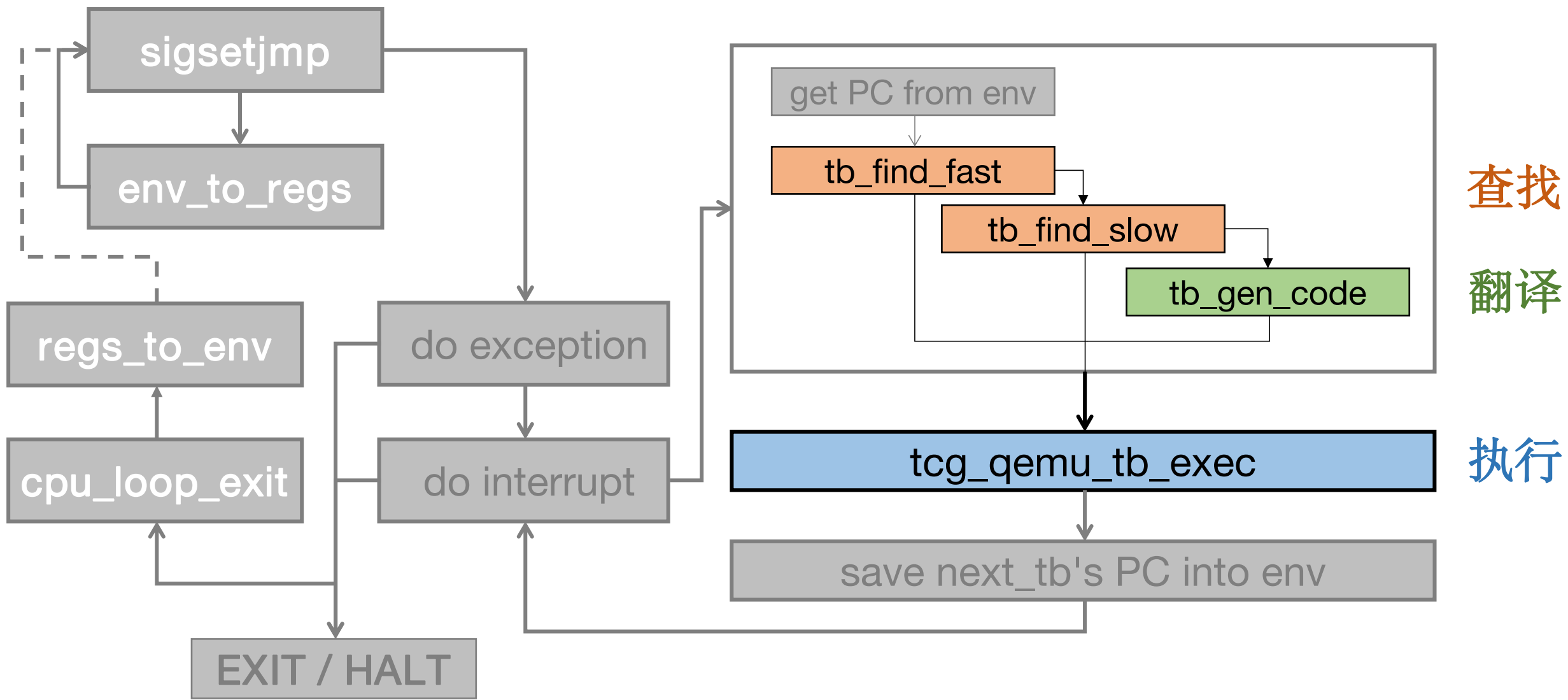
执行TB: 上下文切换



TB上下文切换: prologue & epilogue

- linux-user/main.c:2707 int main(...) tcg_prologue_init(&tcg_ctx);
- tcg/tcg.c:250 void tcg_prologue_init(TCGContext *s)
 - **s->code_buf = code_gen_prologue; s->code_ptr = s->code_buf;**
 - tcg_target_qemu_prologue(s);
- **tcg/mips/tcg-target.c:2536 void tcg_target_qemu_prologue(*s)**
 - /* TB prologue */ tcg_out_st(...) // save regs (s1-s7, gp, fp, ra) into Stack via sp
 - /* Call generated code */ tcg_out_opc_reg(s, **OPC_JR**, 0, **TCG_REG_A0**, 0)
 - **tb_ret_addr = s->code_ptr;**
 - /* TB epilogue */ tcg_out_ld(...) // restore regs via sp
- tcg/tcg.h:509 #define tcg_qemu_tb_exec(**tb_ptr**) \
((long REGPARAM __attribute__ ((longcall)) (*)(void *))code_gen_prologue)(**tb_ptr**)
- tcg/mips/tcg-target.c: tcg_out_op
 - **case INDEX_op_exit_tb: tcg_out_opc_j(s, OPC_J, (tcg_target_long)tb_ret_addr);**

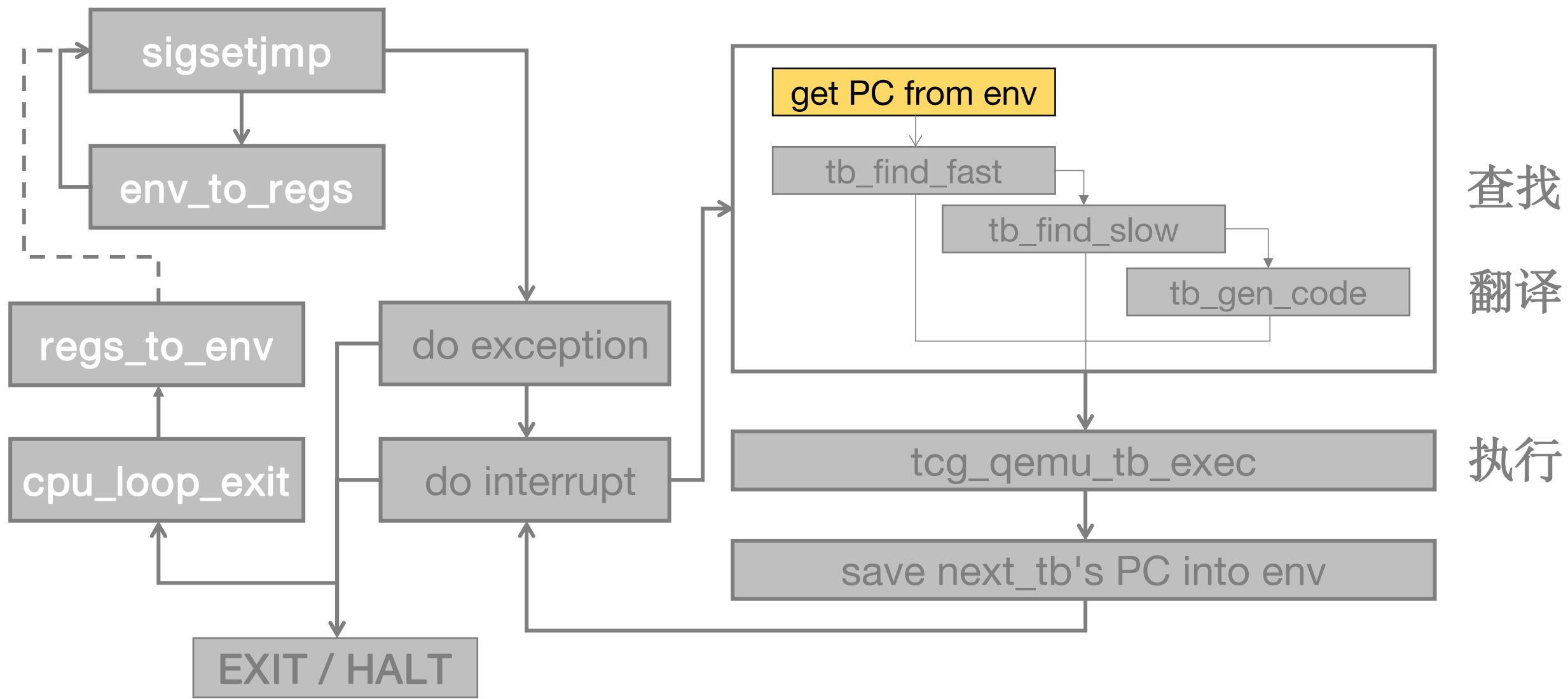
主循环: `cpu_exec(env)`



其他细节问题

- 第一个 **TB** 从哪来?
- 执行完一个 **TB** 后, 如何跳转到下一个 **TB**?

第一个 TB 从哪来



第一个 TB 从哪来

pc & cs_base

- cpu-exec.c:232 TranslationBlock *tb_find_fast()
 - **cpu_get_tb_cpu_state**(env, **&pc**, &cs_base, &flags);
 - tb = env->tb_jmp_cache[tb_jmp_cache_hash_func(**pc**)];
 - if(wrong tb found)
 - tb = tb_find_slow(pc, cs_base, flags);

- target-i386/cpu.h:1000

cpu_get_tb_cpu_state(*env, *pc ,*cs_base, *flags)

- ***cs_base = env->segs[R_CS].base;**
- ***pc = *cs_base + env->eip;**

第一个 TB 从哪来

cs_case
文件里定义好的

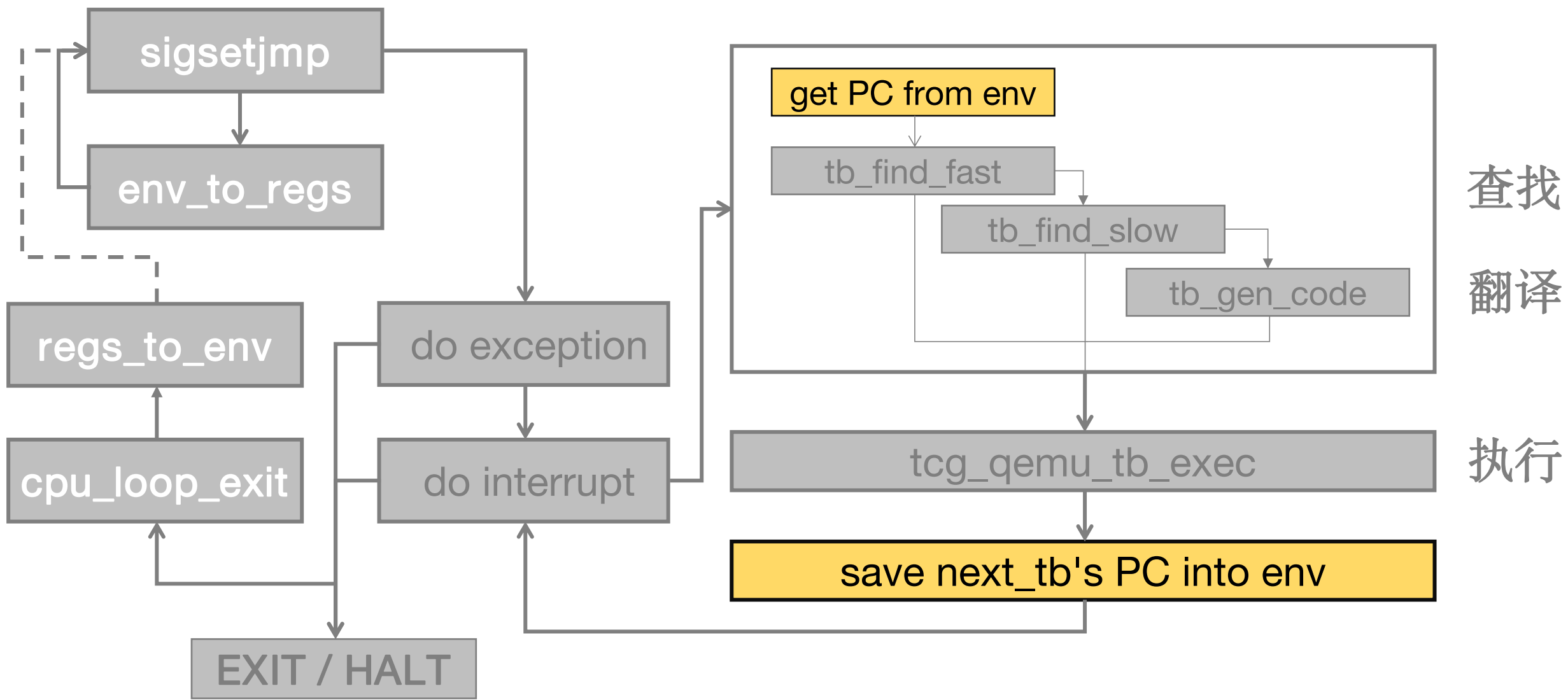
- linux-user/i386/syscall.h:
 - **#define __USER_CS** (0x23) // line 2
- linux-user/main.c:2707 int main(...)
 - cpu_x86_load_seg(env, R_CS, **__USER_CS**); // line 3198
- cpu-exec.c:483 void cpu_x86_load_seg(*s, seg_reg, **selector**)
 - **selector &= 0xffff;**
 - cpu_x86_load_seg_cache(env, seg_reg, selector, (**selector << 4**), 0xffff, 0)
- target-i386/cpu.h:783 static inline void cpu_x86_load_seg_cache(env, seg_reg, selector, **base**, ...)
 - SegmentCache *sc = &env->segs[seg_reg];
 - **sc->base = base;**
- 0x0230

第一个 TB 从哪来

PC 解析二进制文件

- linux-user/main.c:2707 int main(..)
 - ret = loader_exec(filename, target_argv, target_environ, regs, info, &bprm);
 - **env->eip = regs->rip; // line 3130**
- linux-user/linuxload.c:156 int loader_exec(...)
 - retval = load_elf_binary(bprm, regs, infop); // line 182
 - do_init_thread(regs, infop); // line 198 // **regs->rip = infop->entry;**
- linux-user/elfload.c:1575 int load_elf_binary(*bprm, *regs, *info)
 - load_elf_image(bprm->filename, bprm->fd, info, &elf_interpreter, bprm->buf);
- linux-user/elfload.c:1168 static void load_elf_image(..)
 - abi_ulong load_addr, load_bias, loadaddr, hiaddr;
 - load_addr = target_mmap(loadaddr, hiaddr - loadaddr, ...);
 - load_bias = load_addr - loadaddr;
 - **info->entry = ehdr->e_entry + load_bias;**

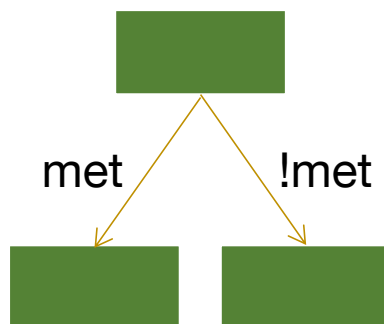
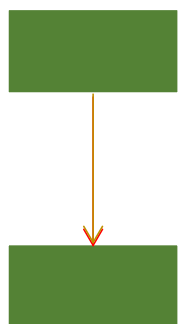
下一个 TB 到哪找



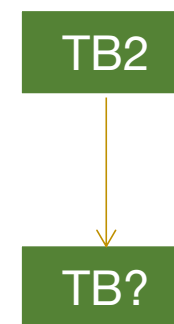
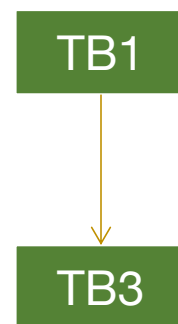
寻找 TB: TB 间跳转问题

- 跳转指令

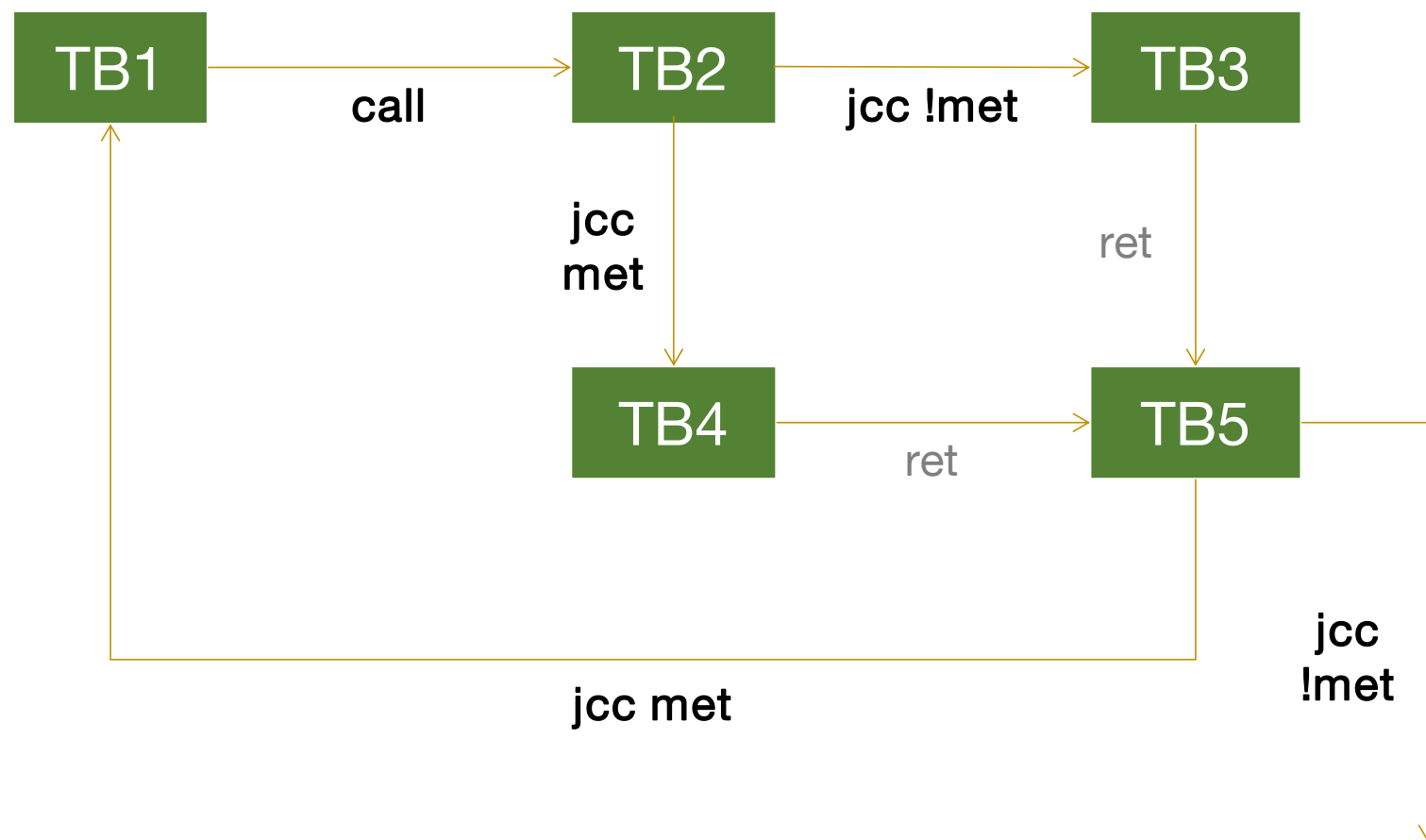
单向跳转	双向跳转
jmp call ret	jcc



直接跳转	间接跳转
jmp addr jcc addr call addr	jmp/jcc/call reg jmp/jcc/call *addrret ret



直接跳转 jmp/jcc/call



```
for( ... ){
    TB1;
    XXX( ... );
    TB5;
}

XXX( ... ){
    TB2;
    if ( )
        TB3;
    else
        TB4;
}
```

直接跳转 - 以 jmp im 为例

JMP—Jump

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
EB <i>cb</i>	JMP <i>rel8</i>	D	Valid	Valid	Jump short, RIP = RIP + 8-bit displacement sign extended to 64-bits
E9 <i>cw</i>	JMP <i>rel16</i>	D	N.S.	Valid	Jump near, relative, displacement relative to next instruction. Not supported in 64-bit mode.
E9 <i>cd</i>	JMP <i>rel32</i>	D	Valid	Valid	Jump near, relative, RIP = RIP + 32-bit displacement sign extended to 64-bits
FF <i>/4</i>	JMP <i>r/m16</i>	M	N.S.	Valid	Jump near, absolute indirect, address = zero-extended <i>r/m16</i> . Not supported in 64-bit mode.

- **case 0xe9: /* jmp im */**
- if (dflag) tval = (int32_t)insn_get(s, OT_LONG); else tval = (int16_t)insn_get(s, OT_WORD);
- tval += s->pc - s->cs_base;
- if (s->dflag == 0) tval &= 0xffff; else if(!CODE64(s)) tval &= 0xffffffff;
- **gen_jump(s, tval);**

直接跳转 - 生成中间代码

- target-i386/translate.c:9627

`gen_intermediate_code_internal(*env, *tb, search_pc)`

- `pc_ptr = disas_insn(dc, pc_ptr);`
- `if (dc->is_jump) break; // 遇到跳转指令后, 停止翻译`
- target_ulong **`disas_insn(DisasContext *s, target_ulong pc_start)`**
 - `b = ldub_code(s->pc) // get next byte`
 - `switch(b) {`
 - `case 0xe9: /* jmp im */`
 - **`gen_jump(s, tval);`**

直接跳转 - 生成中间代码

情况1: 未开启代码链优化 (jmp_opt) 或 pc 与 s->tb 不在同一页内

- **tcg_gen_movi_tl => 把 pc 写入临时寄存器 cpu_tmp0**
- **tcg_gen_st_tl => 把 cpu_tmp0 写入 env->eip**
- **gen_eob => exit_tb 指令, 值 0**

- target-i386/translate.c:3949 gen_jump(DisasContext *s, target_ulong eip)
 - gen_jump_tb(s, eip, 0)
- target-i386/translate.c:3937 gen_jump_tb(*s, eip, tb_num)
 - **gen_jump_im(eip)**
 - tcg_gen_movi_tl(cpu_tmp0, pc);
 - tcg_gen_st_tl(cpu_tmp0, cpu_env, offsetof(CPUState, eip));
 - **gen_eob(s)**

中间代码 exit_tb

- **gen_eob: End of Block**

- target-i386/translate.c:3893 void gen_eob(DisadContext *s)
 - tcg_gen_exit_tb(0);
 - s->is_jump = DISAS_TB_JUMP;
- tcg/tcg-op.h:2188 void tcg_gen_exit_tb(tcg_target_long val)
 - tcg_gen_op1i(INDEX_op_exit_tb, val)
- tcg/tcg-op.h:40 void tcg_gen_op1i(opc, arg1)
 - *gen_opc_ptr++ = opc; // INDEX_op_exit_tb
 - *gen_opparam_ptr++ = arg1; // 0

- 生成一条 exit_tb 中间指令，参数为 0，并标记 is_jump 来终止翻译

中间代码 exit_tb

- `gen_eob` \Rightarrow `exit_tb` 指令, 值 0 // 这个值就是 `args[0]`
- 优化前, 原始版本
 - `tcg_out_movi(s, TCG_TYPE_I32, TCG_REG_V0, args[0]);` // \$v0 存返回值 `args[0]`
 - `tcg_out_movi(s, TCG_TYPE_I32, TCG_REG_AT, (tcg_target_long)tb_ret_addr);` // \$at 存地址 `tb_ret_addr`
 - `tcg_out_opc_reg(s, OPC_JR, 0, TCG_REG_AT, 0);` // jr \$at, 转到 `epilogue`
 - `tcg_out_nop(s);` // nop, 延迟槽
- 开启优化 `CONFIG_EXITTB_OPT`
 - `tcg_out_opc_j(s, OPC_J, (tcg_target_long)tb_ret_addr);` // j `tb_ret_addr`, 直接跳转到 `epilogue`
 - `tcg_out_movi(s, TCG_TYPE_I32, TCG_REG_V0, args[0]);` // \$v0 存放返回值 `args[0]`, 位于延迟槽中
- `exit_tb`: 以 `args[0]` 作为返回值, 转到 `epilogue`, 最终返回到 QEMU 上下文

直接跳转 - 生成中间代码

情况2: 开启代码链优化 (jmp_opt) 且与 s->tb 在同一页内

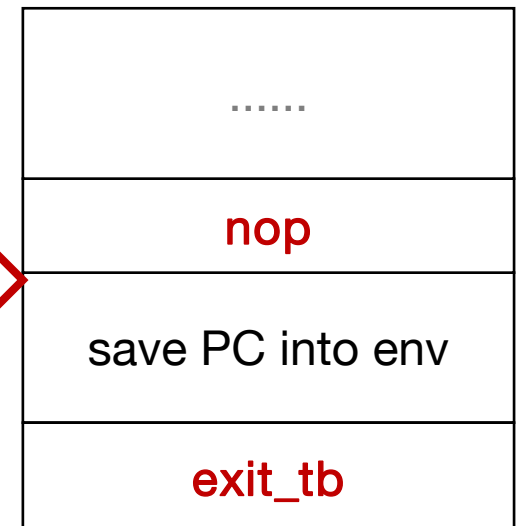
- **tcg_gen_goto_tb => goto_tb指令, 值 tb_num = 0**
- **tcg_gen_movi_tl => 把 pc 写入临时寄存器 cpu_tmp0**
- **tcg_gen_st_tl => 把 cpu_tmp0 写入 env->eip**
- **tcg_gen_exit_tb => exit_tb指令, 值 s->tb + tb_num**

goto_tb
save PC into env
exit_tb

- target-i386/translate.c:3949 gen_jump(DisasContext *s, target_ulong eip)
 - gen_jump_tb(s, eip, 0)
- target-i386/translate.c:3937 gen_jump_tb(*s, eip, tb_num)
 - tcg_gen_goto_tb(tb_num)
 - gen_jump_im(eip)
 - tcg_gen_movi_tl(cpu_tmp0, pc);
 - tcg_gen_st_tl(cpu_tmp0, cpu_env, offsetof(CPUState, eip));
 - tcg_gen_exit_tb((long)tb + tb_num)

中间代码 goto_tb

- if (s->tb_jump_offset) { /* direct jump method */
- s->tb_jump_offset[args[0]] = s->code_ptr - s->code_buf;
- s->code_ptr += 4;
- } else { /* indirect jump method */
- tcg_out_movi(s, TCG_TYPE_PTR, TCG_REG_AT, (tcg_target_long)(s->tb_next + args[0]));
- tcg_out_ld(s, TCG_TYPE_PTR, TCG_REG_AT, TCG_REG_AT, 0);
- tcg_out_opc_reg(s, OPC_JR, 0, TCG_REG_AT, 0);
- }
- **tcg_out_nop(s);**
- **s->tb_next_offset[args[0]] = s->code_ptr - s->code_buf;**



指向 goto_tb 后，从而跳过 goto_tb，返回 QEMU 上下文

中间代码 goto_tb

- 配置选项: `USE_DIRECT_JUMP` 和 `CONFIG_GOTOTB_OPT`
 - 开启时, 使用 `tb_jump_offset`
 - 关闭时, 使用 `tb_next`
- 为什么 `tb_jump_offset` 有两项
 - 因为一条跳转指令最多指向两个地址

<code>uint16_t</code>	<code>tb_next_offset[2]</code>
<code>uint16_t</code>	<code>tb_jump_offset[2]</code>
<code>unsigned long</code>	<code>tb_next[2]</code>

运行时动态绑定 TB 间跳转关系

```
#ifdef USE_DIRECT_JUMP
    s->tb_jump_offset = tb->tb_jump_offset;
    s->tb_next = NULL;
#else
    s->tb_jump_offset = NULL;
    s->tb_next = tb->tb_next;
#endif
```

中间代码 goto_tb

- if (s->tb_jump_offset) { /* direct jump method */
- s->tb_jump_offset[args[0]] = s->code_ptr - s->code_buf;
- s->code_ptr += 4;

s->tb_jump_offset[0] 

在 QEMU 得到下一个 TB 的地址时
可以在这里插入一条直接跳转指令

空的
nop
save PC into env
exit_tb

中间代码 goto_tb

- } else { /* indirect jump method */
- tcg_out_movi(s, TCG_TYPE_PTR, TCG_REG_AT, (tcg_target_long)(s->tb_next + args[0]));
- tcg_out_ld(s, TCG_TYPE_PTR, TCG_REG_AT, TCG_REG_AT, 0);
- tcg_out_opc_reg(s, OPC_JR, 0, TCG_REG_AT, 0);

在 QEMU 得到下一个 TB 的地址时
可以把地址写入 **tb->next[0]**
这三条指令会读取并跳转

把 s->tb_next[0] 的地址写入 \$at

把 \$at 指向的内存数据读到 \$at 中

jr \$at

nop

save PC into env

exit_tb

TB 间跳转问题

- 跳转指令

- 单向跳转 / 双向跳转
- 直接跳转 / 间接跳转

- 中间代码

- `exit_tb` 保存 PC 到 `env`, 并返回 QEMU
- `goto_tb` 直接跳转到下一个TB, 不返回 QEMU
 - 使用 `tm_jump_offset[2]` 或 `tb_next[2]`

- a

已有的优化措施

- 优化直接跳转
 - 代码块链接: `goto_tb`
- 优化间接跳转
 - 加速地址查找: 基于 CAM 的 SPC 到 TPC 的映射查找
 - 间接地址预测: 代码块中预留预测槽位
 - 间接跳转分布式查找表, 增加局部性
 - 硬件 TLB 转换 GVA 到 HVA, 避免软件模拟的低效率
- 优化寄存器访问
 - 全寄存器模拟, 即直接使用 CPU 的寄存器, 而不是读写 `env`

总结

- **QEMU 结构**

- 动态翻译: TCG, 中间代码与某种指令集的翻译等
- 基本块: 查找、翻译、执行, 跳转指令 (直接跳转、间接跳转) 等
- 地址问题: QEMU 模拟的地址翻译 (GVA-GPA-HVA) 等

- **现有的优化措施**

- 主要是针对间接跳转进行优化, 如硬件CAM、硬件TLB

- **后续工作**

- 转变看代码的思路, 不是看代码怎么写, 而是看为什么这样写
- 如何将动态翻译和静态翻译结合, 已有的框架是怎么实现的
- 尽快将原型系统跑起来, 目前仍有诸多问题
 - 编译后运行报错、内核起不来