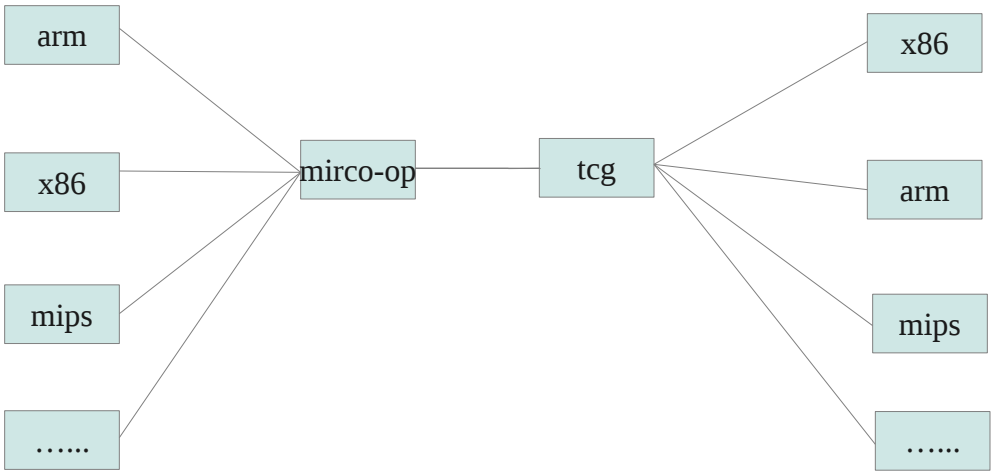
Qemu 代码分析

1.qemu 简介

qemu 是使用动态二进制翻译的 cpu 模拟器，它支持两种运行模式：全系统模拟和用户态模拟。在全系统模拟下，qemu 可以模拟处理器和各种外设，可以运行操作系统。用户态可以运行为另外一种 cpu 编译的进程，前提是两者运行的 os 要一致。qemu 使用了动态二进制翻译将 target instruction 翻译成 host instruction，完成这个工作的是 tcg 模块。为了移植性和通用性方面的考虑，qemu 定义了 mirco-op，首先 qemu 会将 target instruction 翻译成 mirco-op，然后 tcg 将 mirco-op 翻译成 host instruction。

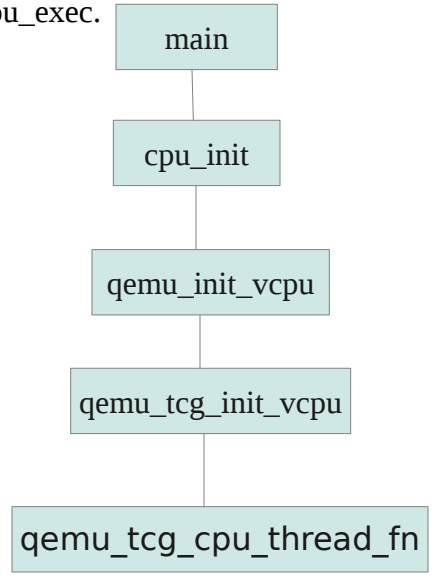
Qemu 代码翻译流程



2.qemu 代码执行流程：

1.初始化部分流程：

这部分主要是创建了一个为执行 tcg 翻译和执行的线程，它的函数是 qemu_tcg_cpu_thread_fn,这个函数会调用 tcg_exec_all，最后 cpu_exec.



2.执行主函数(cpu_exec)

主要是处理中断异常，找到代码翻译块，然后执行。

```
for(;;) {
        process interrupt request;

        tb_find_fast();

         tcg_qemu_tb_exec(tc_ptr);
}
```

qemu 会将翻译好到代码块暂存起来，因此首先会去查看该 pc 对应的代码是否已经翻译，如果已经存在直接返回，否则就进入 tb_find_slow，进行翻译。

```
139 static inline TranslationBlock *tb_find_fast(CPUArchState *env)
140 {
141     TranslationBlock *tb;
142     target_ulong cs_base, pc;
143     int flags;
144
145     /* we record a subset of the CPU state. It will
146        always be the same before a given translated block
147        is executed. */
148     cpu_get_tb_cpu_state(env, &pc, &cs_base, &flags);
149     tb = env->tb_jmp_cache[tb_jmp_cache_hash_func(pc)];
150     if (unlikely(!tb || tb->pc != pc || tb->cs_base != cs_base ||
151                  tb->flags != flags)) {
152         tb = tb_find_slow(env, pc, cs_base, flags);
153     }
154     return tb;
155 }
```

进入 tb_find_slow 后会调用 tb_gen_code，首先分配 TranslationBlock 描述符，将要翻译的 pc 等信息记录下来，然后调用 cpu_gen_code，这个函数完成代码翻译工作。qemu 将翻译好的代码存在一个缓冲区里面。

```
1029 TranslationBlock *tb_gen_code(CPUArchState *env,
1030                               target_ulong pc, target_ulong cs_base,
1031                               int flags, int cflags)
1032 {
1033     TranslationBlock *tb;
1034     uint8_t *tc_ptr;
1035     tb_page_addr_t phys_pc, phys_page2;
1036     target_ulong virt_page2;
1037     int code_gen_size;
1038
1039     phys_pc = get_page_addr_code(env, pc);
1040     tb = tb_alloc(pc);
1041     if (!tb) {
1042         /* flush must be done */
1043         tb_flush(env);
1044         /* cannot fail at this point */
```

```
1045        tb = tb_alloc(pc);
1046        /* Don't forget to invalidate previous TB info.  */
1047        tb_invalidated_flag = 1;
1048    }
1049    tc_ptr = code_gen_ptr;
1050    tb->tc_ptr = tc_ptr;
1051    tb->cs_base = cs_base;
1052    tb->flags = flags;
1053    tb->cflags = cflags;
1054    cpu_gen_code(env, tb, &code_gen_size);
1055    code_gen_ptr = (void *)(((uintptr_t)code_gen_ptr + code_gen_size +
1056                             CODE_GEN_ALIGN - 1) & ~(CODE_GEN_ALIGN - 1));
1057
1058    /* check next page if needed */
1059    virt_page2 = (pc + tb->size - 1) & TARGET_PAGE_MASK;
1060    phys_page2 = -1;
1061    if ((pc & TARGET_PAGE_MASK) != virt_page2) {
1062        phys_page2 = get_page_addr_code(env, virt_page2);
1063    }
1064    tb_link_page(tb, phys_pc, phys_page2);
1065    return tb;
1066 }
```

在 cpu_gen_code 里面首先是将 target instruction 翻译成 micro-op，然后将 mirco-op 翻译成 host 机器码。

```
54 int cpu_gen_code(CPUArchState *env, TranslationBlock *tb, int *gen_code_size_ptr)
55 {
56    TCGContext *s = &tcg_ctx;
57    uint8_t *gen_code_buf;
58    int gen_code_size;
59 #ifdef CONFIG_PROFILER
60    int64_t ti;
61 #endif
62
63 #ifdef CONFIG_PROFILER
64    s->tb_count1++; /* includes aborted translations because of
65                       exceptions */
66    ti = profile_getclock();
67 #endif
68    tcg_func_start(s);
69
70    gen_intermediate_code(env, tb);
71
72    /* generate machine code */
73    gen_code_buf = tb->tc_ptr;
74    tb->tb_next_offset[0] = 0xffff;
75    tb->tb_next_offset[1] = 0xffff;
76    s->tb_next_offset = tb->tb_next_offset;
77 #ifdef USE_DIRECT_JUMP
```

```
 78     s->tb_jmp_offset = tb->tb_jmp_offset;
 79     s->tb_next = NULL;
 80 #else
 81     s->tb_jmp_offset = NULL;
 82     s->tb_next = tb->tb_next;
 83 #endif
 84
 85 #ifdef CONFIG_PROFILER
 86     s->tb_count++;
 87     s->interm_time += profile_getclock() - ti;
 88     s->code_time -= profile_getclock();
 89 #endif
 90     gen_code_size = tcg_gen_code(s, gen_code_buf);
 91     *gen_code_size_ptr = gen_code_size;
 92 #ifdef CONFIG_PROFILER
 93     s->code_time += profile_getclock();
 94     s->code_in_len += tb->size;
 95     s->code_out_len += gen_code_size;
 96 #endif
 97
 98 #ifdef DEBUG_DISAS
 99     if (qemu_loglevel_mask(CPU_LOG_TB_OUT_ASM)) {
100         qemu_log("OUT: [size=%d]\n", *gen_code_size_ptr);
101         log_disas(tb->tc_ptr, *gen_code_size_ptr);
102         qemu_log("\n");
103         qemu_log_flush();
104     }
105 #endif
106     return 0;
107 }
```

qemu 将 target 翻译成中间码时，将操作码和操作数分开存储，分别存在 gen_opc_buf 和 gen_opparam_buf 中。翻译的过程就是不断向 gen_opc_buf 和 gen_opparam_buf 中填充操作码和操作数。

接下来就是 tcg_gen_code

```
2175 int tcg_gen_code(TCGContext *s, uint8_t *gen_code_buf)
2176 {
2177 #ifdef CONFIG_PROFILER
2178     {
2179         int n;
2180         n = (gen_opc_ptr - gen_opc_buf);
2181         s->op_count += n;
2182         if (n > s->op_count_max)
2183             s->op_count_max = n;
2184
2185         s->temp_count += s->nb_temps;
2186         if (s->nb_temps > s->temp_count_max)
2187             s->temp_count_max = s->nb_temps;
2188     }
2189 #endif
```

```
2190
2191     tcg_gen_code_common(s, gen_code_buf, -1);
2192
2193     /* flush instruction cache */
2194     flush_icache_range((tcg_target_ulong)gen_code_buf,
2195                        (tcg_target_ulong)s->code_ptr);
2196
2197     return s->code_ptr -  gen_code_buf;
2198 }
```

tcg_gen_code 的工作是将中间码翻译成 host 机器码，它的主要函数是 tcg_gen_code_common.

```
2045 static inline int tcg_gen_code_common(TCGContext *s, uint8_t *gen_code_buf,
2046                                       long search_pc)
2047 {
2048     TCGOpcode opc;
2049     int op_index;
2050     const TCGOpDef *def;
2051     unsigned int dead_args;
2052     const TCGArg *args;
2053
2054 #ifdef DEBUG_DISAS
2055     if (unlikely(qemu_loglevel_mask(CPU_LOG_TB_OP))) {
2056         qemu_log("OP:\n");
2057         tcg_dump_ops(s);
2058         qemu_log("\n");
2059     }
2060 #endif
2061
2062 #ifdef USE_TCG_OPTIMIZATIONS
2063     gen_opparam_ptr =
2064         tcg_optimize(s, gen_opc_ptr, gen_opparam_buf, tcg_op_defs);
2065 #endif
2066
2067 #ifdef CONFIG_PROFILER
2068     s->la_time -= profile_getclock();
2069 #endif
2070     tcg_liveness_analysis(s);
2071 #ifdef CONFIG_PROFILER
2072     s->la_time += profile_getclock();
2073 #endif
2074
2075 #ifdef DEBUG_DISAS
2076     if (unlikely(qemu_loglevel_mask(CPU_LOG_TB_OP_OPT))) {
2077         qemu_log("OP after liveness analysis:\n");
2078         tcg_dump_ops(s);
2079         qemu_log("\n");
2080     }
2081 #endif
2082
```

```
2083     tcg_reg_alloc_start(s);
2084
2085     s->code_buf = gen_code_buf;
2086     s->code_ptr = gen_code_buf;
2087
2088     args = gen_opparam_buf;
 2089     op_index = 0;
2090
2091     for(;;) {
2092         opc = gen_opc_buf[op_index];
2093 #ifdef CONFIG_PROFILER
2094         tcg_table_op_count[opc]++;
2095 #endif
2096         def = &tcg_op_defs[opc];
2097 #if 0
2098         printf("%s: %d %d %d\n", def->name,
2099                def->nb_oargs, def->nb_iargs, def->nb_cargs);
2100         //        dump_regs(s);
2101 #endif
2102         switch(opc) {
2103         case INDEX_op_mov_i32:
2104 #if TCG_TARGET_REG_BITS == 64
2105         case INDEX_op_mov_i64:
2106 #endif
2107             dead_args = s->op_dead_args[op_index];
2108             tcg_reg_alloc_mov(s, def, args, dead_args);
2109             break;
2110         case INDEX_op_movi_i32:
2111 #if TCG_TARGET_REG_BITS == 64
2112         case INDEX_op_movi_i64:
2113 #endif
2114             tcg_reg_alloc_movi(s, args);
2115             break;
2116         case INDEX_op_debug_insn_start:
2117             /* debug instruction */
2118             break;
2119         case INDEX_op_nop:
2120         case INDEX_op_nop1:
2121         case INDEX_op_nop2:
2122         case INDEX_op_nop3:
2123             break;
2124         case INDEX_op_nopn:
2125             args += args[0];
2126             goto next;
2127         case INDEX_op_discard:
2128             {
 2129                 TCGTemp *ts;
2130                 ts = &s->temps[args[0]];
2131                 /* mark the temporary as dead */
```

```
2132                 if (!ts->fixed_reg) {
2133                     if (ts->val_type == TEMP_VAL_REG)
2134                         s->reg_to_temp[ts->reg] = -1;
2135                     ts->val_type = TEMP_VAL_DEAD;
2136                 }
2137             }
2138             break;
2139         case INDEX_op_set_label:
2140             tcg_reg_alloc_bb_end(s, s->reserved_regs);
2141             tcg_out_label(s, args[0], s->code_ptr);
2142             break;
2143         case INDEX_op_call:
2144             dead_args = s->op_dead_args[op_index];
2145             args += tcg_reg_alloc_call(s, def, opc, args, dead_args);
2146             goto next;
2147         case INDEX_op_end:
2148             goto the_end;
2149         default:
2150             /* Sanity check that we've not introduced any unhandled opcodes. */
2151             if (def->flags & TCG_OPF_NOT_PRESENT) {
2152                 tcg_abort();
2153             }
2154             /* Note: in order to speed up the code, it would be much
2155                faster to have specialized register allocator functions for
2156                some common argument patterns */
2157             dead_args = s->op_dead_args[op_index];
2158             tcg_reg_alloc_op(s, def, opc, args, dead_args);
2159             break;
2160         }
2161         args += def->nb_args;
2162     next:
2163         if (search_pc >= 0 && search_pc < s->code_ptr - gen_code_buf) {
2164             return op_index;
2165         }
2166         op_index++;
2167 #ifndef NDEBUG
2168         check_regs(s);
2169 #endif
2170     }
 2171  the_end:
2172     return -1;
2173 }
```

大部分执行 default 分支，tcg_reg_alloc_op 主要是分析该指令的输入、输出约束，根据这些约束分配寄存器等，然后调用 tcg_out_op 将该中间码翻译成 host 机器码。

3.翻译代码块的执行

代码块翻译好之后，主函数调用 tcg_qemu_tb_exec，该函数会进入 tcg 的入口函数。

```
 100 #define tcg_qemu_tb_exec(env, tb_ptr) \
101     ((long __attribute__ ((longcall)) \
102       (*)(void *, void *))code_gen_prologue)(env, tb_ptr)
```

code_gen_prologue 是 tcg 的入口函数，在 tcg 初始化的时候会生成相应的代码

```
 273 void tcg_prologue_init(TCGContext *s)
 274 {
 275     /* init global prologue and epilogue */
 276     s->code_buf = code_gen_prologue;
 277     s->code_ptr = s->code_buf;
 278     tcg_target_qemu_prologue(s);
 279     flush_icache_range((tcg_target_ulong)s->code_buf,
 280                        (tcg_target_ulong)s->code_ptr);
 281 }
```

以 host 是 arm 为例，入口函数主要是保存寄存器的状态，然后将 tcg_qemu_tb_exec 传进来的第一的参数(env)给 TCG_AREG0，然后跳转至 tb_ptr，开始执行代码。同时代码执行的返回地址也确定了，返回后恢复之前保存的状态。

```
1881 static void tcg_target_qemu_prologue(TCGContext *s)
1882 {
1883     /* Calling convention requires us to save r4-r11 and lr;
1884      * save also r12 to maintain stack 8-alignment.
1885      */
1886
1887     /* stmdb sp!, { r4 - r12, lr } */
1888     tcg_out32(s, (COND_AL << 28) | 0x092d5ff0);
1889
1890     tcg_out_mov(s, TCG_TYPE_PTR, TCG_AREG0, tcg_target_call_iarg_regs[0]);
1891
1892     tcg_out_bx(s, COND_AL, tcg_target_call_iarg_regs[1]);
1893     tb_ret_addr = s->code_ptr;
1894
1895     /* ldmia sp!, { r4 - r12, pc } */
1896     tcg_out32(s, (COND_AL << 28) | 0x08bd9ff0);
1897 }
```

二 qemu 内存访问

1.qemu 物理内存分配：

qemu 分配给 target 的内存是以 RAMBlock 和 RAMList 来管理，内存主要分成以下几种类型：

1.IO_MEM_RAM:一般内存。
2.IO_MEM_ROM:ROM.

3.IO_MEM_UNASSIGNED:未指定

4.IO_MEM_ROMD:读的时候视为 ROM，写的时候视为装置。

1.通过 qemu_ram_alloc 申请空间：

```
ram_addr_t qemu_ram_alloc(DeviceState *dev, const char *name, ram_addr_t size)
{
    RAMBlock *new_block, *block;

    pstrcat(new_block->idstr, sizeof(new_block->idstr), name);

    // 检查 RAMList 中是否已有具有相同名称的 RAMBlock。
    QLIST_FOREACH(block, &ram_list.blocks, next) {
    }

    if (mem_path) {
    } else {
        // 指向宿主的虚拟内存位址。
        new_block->host = qemu_vmalloc(size);
    }

    new_block->offset = find_ram_offset(size); // 该 RAMBlock 在 RAMList 的偏移量。
    new_block->length = size; //该 RAMBlock 的大小。

    QLIST_INSERT_HEAD(&ram_list.blocks, new_block, next); // 將新增的 RAMBlock 加入
RAMList。

    ram_list.phys_dirty = qemu_realloc(ram_list.phys_dirty,
                                       last_ram_offset() >> TARGET_PAGE_BITS);
    memset(ram_list.phys_dirty + (new_block->offset >> TARGET_PAGE_BITS),
           0xff, size >> TARGET_PAGE_BITS);

    return new_block->offset; // 回传该 RAMBlock 在 RAMList 的偏移量。

}
```

2.通过 cpu_register_physical_memory → cpu_register_physical_memory_offset 向 qemu 注册 target 物理内存信息。

```
void cpu_register_physical_memory_offset(target_phys_addr_t start_addr,

                                         ram_addr_t size,
                                         ram_addr_t phys_offset,
                                         ram_addr_t region_offset)
{
    PhysPageDesc *p;
```

```
    for(addr = start_addr; addr != end_addr; addr += TARGET_PAGE_SIZE) {
        p = phys_page_find(addr >> TARGET_PAGE_BITS); // 用 target 物理位址 start_addr
查找 l1_phys_map
        if (p && p->phys_offset != IO_MEM_UNASSIGNED) {
        } else {
            p = phys_page_find_alloc(addr >> TARGET_PAGE_BITS, 1);
       }
    }
}
```

3.设备通过 cpu_register_physical_memory 注册 IO 物理地址空间。

```
static void cirrus_init_common(CirrusVGAState * s, int device_id, int is_pci)
{
    s->vga.vga_io_memory = cpu_register_io_memory(cirrus_vga_mem_read,
                                                  cirrus_vga_mem_write, s);
    cpu_register_physical_memory(isa_mem_base + 0x000a0000, 0x20000,
                                 s->vga.vga_io_memory);
}
```

4.PageDesc 维护着 tb 和 target 物理地址之间的关系，它也是一个典型的二级页表。

```
335 static PageDesc *page_find_alloc(tb_page_addr_t index, int alloc)
336 {
337     PageDesc *pd;
338     void **lp;
339     int i;
340
341 #if defined(CONFIG_USER_ONLY)
342     /* We can't use g_malloc because it may recurse into a locked mutex. */
343 # define ALLOC(P, SIZE)                                 \
344     do {                                                \
345         P = mmap(NULL, SIZE, PROT_READ | PROT_WRITE,    \
346                  MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);   \
347     } while (0)
348 #else
349 # define ALLOC(P, SIZE) \
350     do { P = g_malloc0(SIZE); } while (0)
351 #endif
352
353     /* Level 1.  Always allocated.  */
354     lp = l1_map + ((index >> V_L1_SHIFT) & (V_L1_SIZE - 1));
355
356     /* Level 2..N-1.  */
357     for (i = V_L1_SHIFT / L2_BITS - 1; i > 0; i--) {
358         void **p = *lp;
359
360         if (p == NULL) {
361             if (!alloc) {
362                 return NULL;
```

```
363         }
364         ALLOC(p, sizeof(void *) * L2_SIZE);
365         *lp = p;
366       }
367
368       lp = p + ((index >> (i * L2_BITS)) & (L2_SIZE - 1));
369     }
370
371     pd = *lp;
372     if (pd == NULL) {
373       if (!alloc) {
374         return NULL;
375       }
376       ALLOC(pd, sizeof(PageDesc) * L2_SIZE);
377       *lp = pd;
378     }
379
380 #undef ALLOC
381
382     return pd + (index & (L2_SIZE - 1));
383 }
```

2.softmmu

qemu 内存访问过程：guest virtual addr (GVA) → guest physical addr (GPA) → host virtual addr (HVA)。其中 GVA->HVA 由 qemu 负责完成，HVA->HPA 由 host 操作系统完成。tlb 的结构如下，addr_xxx 表示 GVA 地址，同时也表示了执行权限；addrend =gpa_base – gva_base;

```
typedef struct CPUTLBEntry {
    target_ulong addr_read; // 可读
    target_ulong addr_write; // 可写
    target_ulong addr_code; // 可执行
    unsigned long addend;
} CPUTLBEntry;
```

1.get_page_addr_code 会首先查看 tlb 是否命中，如果没有命中就 ldub_code 走 mmu 翻译这个分支，否则直接获取 hva。

```
316 tb_page_addr_t get_page_addr_code(CPUArchState *env1, target_ulong addr)
317 {
318     int mmu_idx, page_index, pd;
319     void *p;
320     MemoryRegion *mr;
321
322     page_index = (addr >> TARGET_PAGE_BITS) & (CPU_TLB_SIZE - 1);
323     mmu_idx = cpu_mmu_index(env1);
324     if (unlikely(env1->tlb_table[mmu_idx][page_index].addr_code !=
325                  (addr & TARGET_PAGE_MASK))) {
326 #ifdef CONFIG_TCG_PASS_AREG0
327         cpu_ldub_code(env1, addr);
328 #else
329         ldub_code(addr);
```

```
330 #endif
331     }
332     pd = env1->iotlb[mmu_idx][page_index] & ~TARGET_PAGE_MASK;
333     mr = iotlb_to_region(pd);
334     if (memory_region_is_unassigned(mr)) {
335 #if defined(TARGET_ALPHA) || defined(TARGET_MIPS) || defined(TARGET_SPARC)
336         cpu_unassigned_access(env1, addr, 0, 1, 0, 4);
337 #else
338         cpu_abort(env1, "Trying to execute code outside RAM or ROM at 0x"
339                   TARGET_FMT_lx "\n", addr);
340 #endif
341     }
342     p = (void *)((uintptr_t)addr + env1->tlb_table[mmu_idx]
[page_index].addend);
343     return qemu_ram_addr_from_host_nofail(p);
344 }
```

2.TLB 没有命中时，会通过 ldub_code，这个函数是由下面的宏产生。这个宏首先会在 tlb 检查下是否命中，如果命中直接从 hva 地址中返回，否则还是通过 mmu 来获取

```
 95 static inline RES_TYPE
 96 glue(glue(glue(CPU_PREFIX, ld), USUFFIX), MEMSUFFIX)(ENV_PARAM
 97                                         target_ulong ptr)
 98 {
 99     int page_index;
100     RES_TYPE res;
101     target_ulong addr;
102     int mmu_idx;
103
104     addr = ptr;
105     page_index = (addr >> TARGET_PAGE_BITS) & (CPU_TLB_SIZE - 1);
106     mmu_idx = CPU_MMU_INDEX;
107     if (unlikely(env->tlb_table[mmu_idx][page_index].ADDR_READ !=
108                  (addr & (TARGET_PAGE_MASK | (DATA_SIZE - 1))))) {
109         res = glue(glue(glue(HELPER_PREFIX, ld), SUFFIX), MMUSUFFIX)(ENV_VAR
110                                                     addr,
111                                                     mmu_idx);
112     } else {
113         uintptr_t hostaddr = addr + env->tlb_table[mmu_idx][page_index].addend;
114         res = glue(glue(ld, USUFFIX), _raw)(hostaddr);
115     }
116     return res;
117 }
```

3.在这个模板中会对 tlb 进行查询，看是否命中，如果命中，还要根据是 io 还是 ram 进行分别处理；如果没有命中，则需要通过 mmu 获取该虚拟地址所对应的物理地址，对 tlb 进行填充。

```
106 DATA_TYPE
107 glue(glue(glue(HELPER_PREFIX, ld), SUFFIX), MMUSUFFIX)(ENV_PARAM
108                                         target_ulong addr,
109                                         int mmu_idx)
110 {
111     DATA_TYPE res;
```

```
112     int index;
113     target_ulong tlb_addr;
114     target_phys_addr_t ioaddr;
115     uintptr_t retaddr;
116
117     /* test if there is match for unaligned or IO access */
118     /* XXX: could done more in memory macro in a non portable way */
119     index = (addr >> TARGET_PAGE_BITS) & (CPU_TLB_SIZE - 1);
120   redo:
121     tlb_addr = env->tlb_table[mmu_idx][index].ADDR_READ;
122     if ((addr & TARGET_PAGE_MASK) == (tlb_addr & (TARGET_PAGE_MASK |
TLB_INVALID_MASK))) {
123         if (tlb_addr & ~TARGET_PAGE_MASK) {
124             /* IO access */
125             if ((addr & (DATA_SIZE - 1)) != 0)
126                 goto do_unaligned_access;
127             retaddr = GETPC();
128             ioaddr = env->iotlb[mmu_idx][index];
129             res = glue(io_read, SUFFIX)(ENV_VAR ioaddr, addr, retaddr);
130         } else if (((addr & ~TARGET_PAGE_MASK) + DATA_SIZE - 1) >=
TARGET_PAGE_SIZE) {
131             /* slow unaligned access (it spans two pages or IO) */
132         do_unaligned_access:
133             retaddr = GETPC();
134 #ifdef ALIGNED_ONLY
135             do_unaligned_access(ENV_VAR addr, READ_ACCESS_TYPE, mmu_idx, retaddr);
136 #endif
137             res = glue(glue(slow_ld, SUFFIX), MMUSUFFIX)(ENV_VAR addr,
138                                                          mmu_idx, retaddr);
139         } else {
140             /* unaligned/aligned access in the same page */
141             uintptr_t addend;
142 #ifdef ALIGNED_ONLY
143             if ((addr & (DATA_SIZE - 1)) != 0) {
144                 retaddr = GETPC();
145                 do_unaligned_access(ENV_VAR addr, READ_ACCESS_TYPE, mmu_idx, retaddr);
146             }
147 #endif
148             addend = env->tlb_table[mmu_idx][index].addend;
149             res = glue(glue(ld, USUFFIX), _raw)((uint8_t *)(intptr_t)
150                                                 (addr + addend));
151         }
152     } else {
153         /* the page is not in the TLB : fill it */
154         retaddr = GETPC();
155 #ifdef ALIGNED_ONLY
156         if ((addr & (DATA_SIZE - 1)) != 0)
157             do_unaligned_access(ENV_VAR addr, READ_ACCESS_TYPE, mmu_idx, retaddr);
158 #endif
```

```
159         tlb_fill(env, addr, READ_ACCESS_TYPE, mmu_idx, retaddr);
160         goto redo;
161     }
162     return res;
163 }
```

4.以 arm 为例，tlb_fill 会通过 cpu_arm_handle_mmu_fault 对虚实地址转换进行处理，如果该虚拟地址没有对应的物理地址或者权限不够等情况，cpu 就会出现 page_fault 异常。

```
 76 void tlb_fill(CPUARMState *env1, target_ulong addr, int is_write, int mmu_idx,
 77                uintptr_t retaddr)
 78 {
 79     TranslationBlock *tb;
 80     CPUARMState *saved_env;
 81     int ret;
 82
 83     saved_env = env;
 84     env = env1;
 85     ret = cpu_arm_handle_mmu_fault(env, addr, is_write, mmu_idx);
 86     if (unlikely(ret)) {
 87         if (retaddr) {
 88             /* now we have a real cpu fault */
 89             tb = tb_find_pc(retaddr);
 90             if (tb) {
 91                 /* the PC is inside the translated code. It means that we have
 92                    a virtual CPU fault */
 93                 cpu_restore_state(tb, env, retaddr);
 94             }
 95         }
 96         raise_exception(env->exception_index);
 97     }
 98     env = saved_env;
 99 }
```

5. cpu_arm_handle_mmu_fault 里面主要是 page_walk，检查是否存在对应的物理地址和权限。如果有，更新 tlb;否则，保存出错信息。

```
2122 int cpu_arm_handle_mmu_fault (CPUARMState *env, target_ulong address,
2123                                int access_type, int mmu_idx)
2124 {
2125     uint32_t phys_addr;
2126     target_ulong page_size;
2127     int prot;
2128     int ret, is_user;
2129
2130     is_user = mmu_idx == MMU_USER_IDX;
2131     ret = get_phys_addr(env, address, access_type, is_user, &phys_addr, &prot,
2132                         &page_size);
2133     if (ret == 0) {
2134         /* Map a single [sub]page.  */
2135         phys_addr &= ~(uint32_t)0x3ff;
2136         address &= ~(uint32_t)0x3ff;
```

```
2137        tlb_set_page (env, address, phys_addr, prot, mmu_idx, page_size);
2138        return 0;
2139    }
2140
2141    if (access_type == 2) {
2142        env->cp15.c5_insn = ret;
2143        env->cp15.c6_insn = address;
2144        env->exception_index = EXCP_PREFETCH_ABORT;
2145    } else {
2146        env->cp15.c5_data = ret;
2147        if (access_type == 1 && arm_feature(env, ARM_FEATURE_V6))
2148            env->cp15.c5_data |= (1 << 11);
2149        env->cp15.c6_data = address;
2150        env->exception_index = EXCP_DATA_ABORT;
2151    }
2152    return 1;
2153 }
```