Chapter-7

Qemu Detailed Study

As discussed in chapter 6 QEMU is a machine emulator and thus can emulate a given number of processor architectures on machine in which it is running. For QEMU the emulated architectures is called the Target. And the real machine on which QEMU is running, emulating the target, is called the Host. The dynamic translation of virtual machine (target) code to Host code is done by a module in QEMU called the Tiny Code Generator or TCG for short. When it comes to TCG the term ‘target’ gets a different meaning. TCG creates the code to emulate the target thus the code created by TCG is called its target .Thus when it comes to TCG target means the generated Host code. Figure 7.1 clarifies this varied terminology.

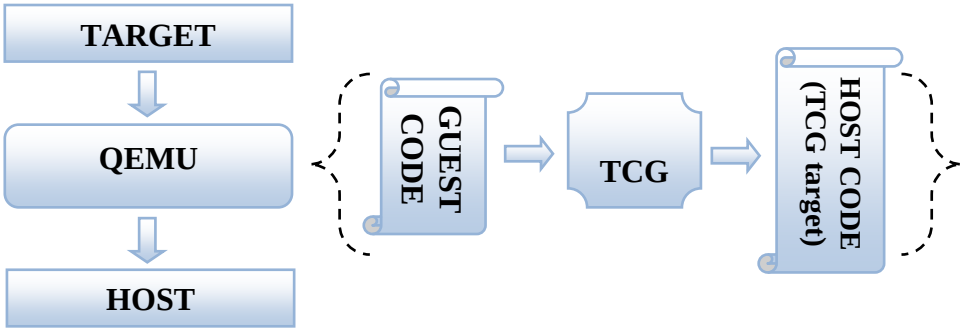

Figure 7.1: Use of term ‘Target’

Thus one may call the code (OS + USER TOOLS) being run by the emulated processor the guest code. QEMU functions by extracting the Guest code and converting it to Host specific code. The whole translation task thus consists of two parts: First a block of target code - Translation Block (TB) is converted into TCG ops - a kind of machine-independent intermediate notation, and subsequently the TCG ops for the TB is converted to Host code for the host's architecture by TCG. Optional optimization passes are performed between them.

7.1 Codebase

A clear understanding of the QEMU codebase is required to add new functionality that will extend the machine emulator to migrate its generated code to execute in remote nodes. The QEMU codebase has over 1300 files which are well organized into specific sections. Even though the code is well organized it is complex enough to leave any new developer perplexed. This section will throw light on the organization of QEMU codebase.

In this section the shallowest directory depth in the codebase will be represented by a ‘/’ , and consecutive directory depths will follow the usual Unix file path notations.

Start of Execution:

The major C files in the / that are important for the study are ; /vl.c,/cpus.c, /exec-all.c, /exec.c, /cpu-exec.c. The ‘main’ function where the execution starts is defined in /vl.c. The functions in this file sets up a virtual machine environment as per the given virtual machine specification such as size of ram, available

devices, number of CPUs etc. From the main function, after the virtual machine is set up, execution branches out through files such as /cpus.c, /exec-all.c, /exec.c, /cpu-exec.c.

Emulated Hardware:

The code that emulates all virtual hardware in the virtual machine can be found in /hw/. QEMU emulates a considerable number of hardware but detailed understanding of how the hardware are emulated is not necessary in this study.

Guest (Target) Specific:

The processor architectures currently emulated in QEMU are; Alpha, ARM, Cris, i386, M68K, PPC, Sparc, Mips, MicroBlaze, S390X and SH4. The code specific to these architectures necessary to convert TBs to TCG ops are available in /target-xyz/ where xyz can any of the above given architecture names. Therefore the code specific to i386 can be found in /target-i386/. This part can be called as the frontend of TCG.

Host (TCG) Specific:

The host specific code for generating the host code from the TCG ops are placed in /tcg/ . Inside TCG one can find /xyz/ where xyz can be i386 ,sparc etc which contain the code that converts TCG ops to architecture specific code. This part can be called as the backend of TCG.

Summary:

/vl.c :	The main emulator loop, the virtual machine is setup and CPUs are executed.
/target-xyz/translate.c :	The extracted guest code (guest specific ISA) is converted into architecture independent TCG ops
/tcg/tcg.c :	The main code for TCG.
/tcg/*/tcg-target.c :	Code that converts the TCG ops to host code (host specific ISA).
/cpu-exec.c :	Function cpu-exec() in /cpu-exec.c finds the next translation block (TB), if not found calls are made to generate the next TB and finally to execute the generated code.

7.2 TCG - Dynamic translation

As mentioned earlier in this document dynamic translation in QEMU before version 0.9.1 was carried out by DynGen. TBs were converted to C code by DynGen and GCC (the GNU C compiler) converted the C code into host specific code. The issue with the procedure was that DynGen was tightly tied to

GCC and created problems as when GCC evolved. To remove the tight coupling of the translator to GCC a new procedure was put in place; TCG.

The dynamic translation converts code as and when needed. The idea was to spend the maximum time executing the generated code that executing the code generation. Every time code is generated from the TB it is stored in the code cache before being executed. Most of the time the same TBs are required again and again, owing to what is called Locality Reference, so instead of re-generating the same code it is best to save it. Figure 7.2 summarizes the same. And once the code cache is full, to make things simple, the entire code cache is flushed instead of using LRU algorithms.

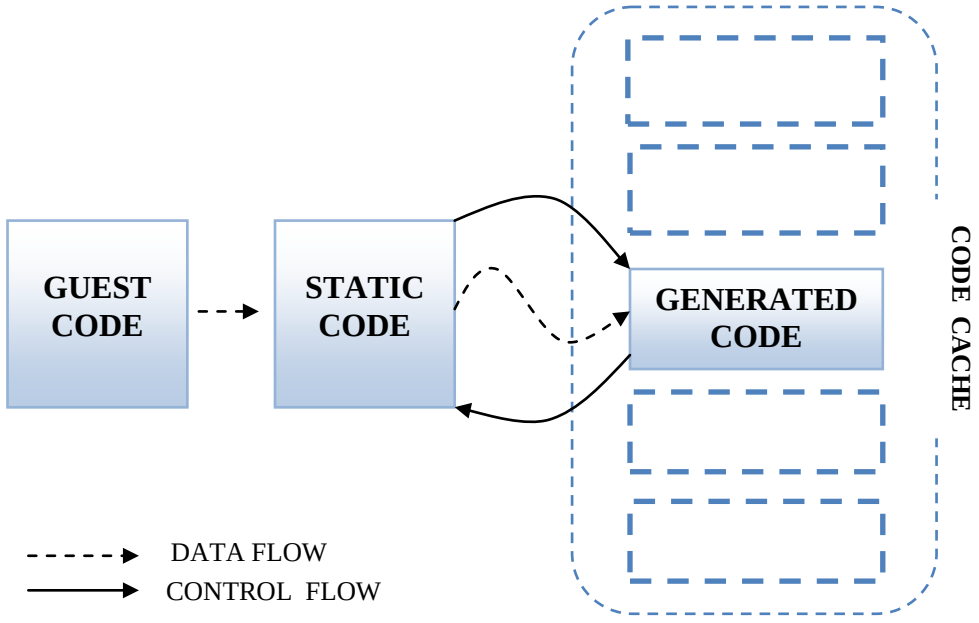

Figure 7.2: Jump to code cache

Compilers produce the object code from the source code before execution. In order to produce object code for a function call a compiler like GCC produces special assembly code that does what is necessary before a function is called and before a function returns. This special assembly code produced is called Function Prologue and Epilogue.

Function Prologue typically does the following actions if the architecture is having a base pointer and a stack pointer:

- Pushes the current base pointer onto the stack, such that it can be restored later.
- Replaces the old base pointer with the current stack pointer such that the a new stack will be created on top of the old stack.
- Moves the stack pointer further along the stack to make room in the current stack frame for the function's local variables.

Function Epilogue reverses the actions of the function prologue and returns control to the calling function. It typically does the following actions:

- Replaces the stack pointer with the current base pointer, so the stack pointer is restored to its value before the prologue.
- Pops the base pointer off the stack, so it is restored to its value before the prologue
- Returns to the calling function, by popping the previous frame's program counter off the stack and jumping to it.

TCG by itself can be seen to function as a compiler that produces object code on the fly. The code generated by TCG is stored in buffer (code cache). The execution control is passed to and from the code cache through TCG's very on Prologue and Epilogue, as illustrated in the figure 7.3.

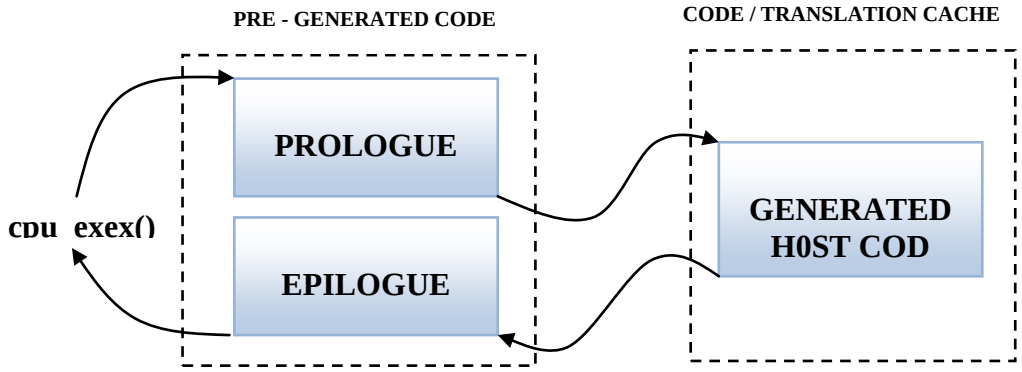

Figure 7.3: Use of Function Prologue and Epilogue

Following figures (7.4 – 7.7) illustrate how TCG functions. Brief descriptions of the functions seen in the figure are given in the next section.

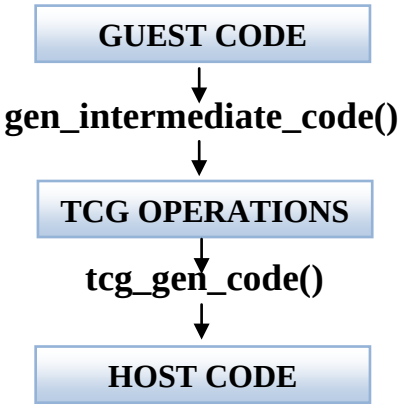

Figure 7.4: Dynamic translation - outline

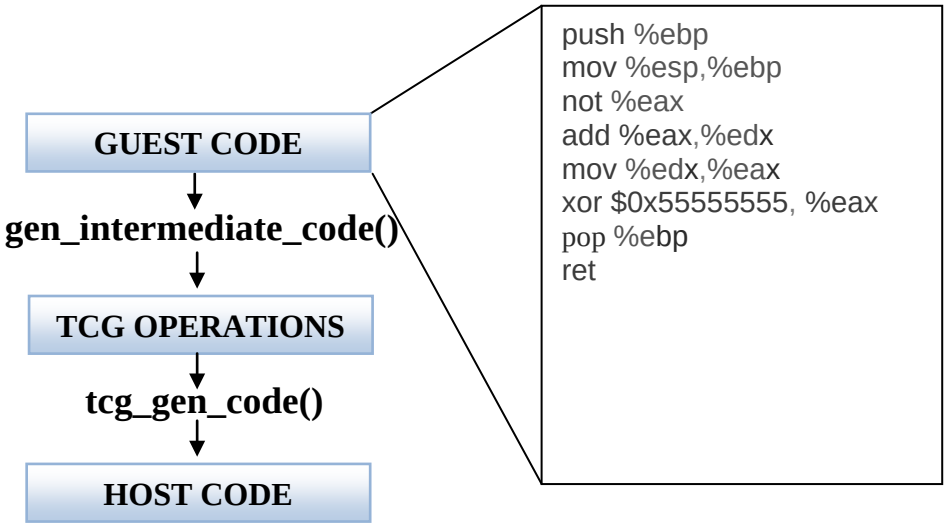

Figure 7.5: Dynamic translation – Showing guest code

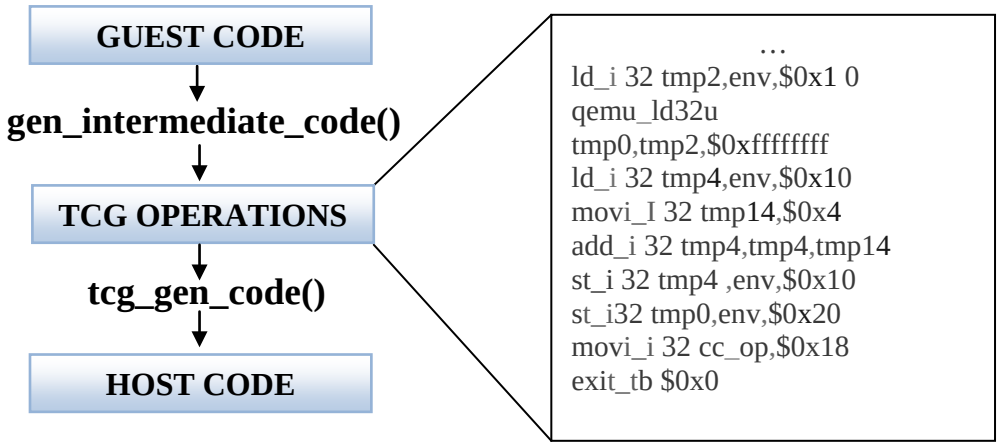

Figure 7.6: Dynamic translation – showing TCG ops

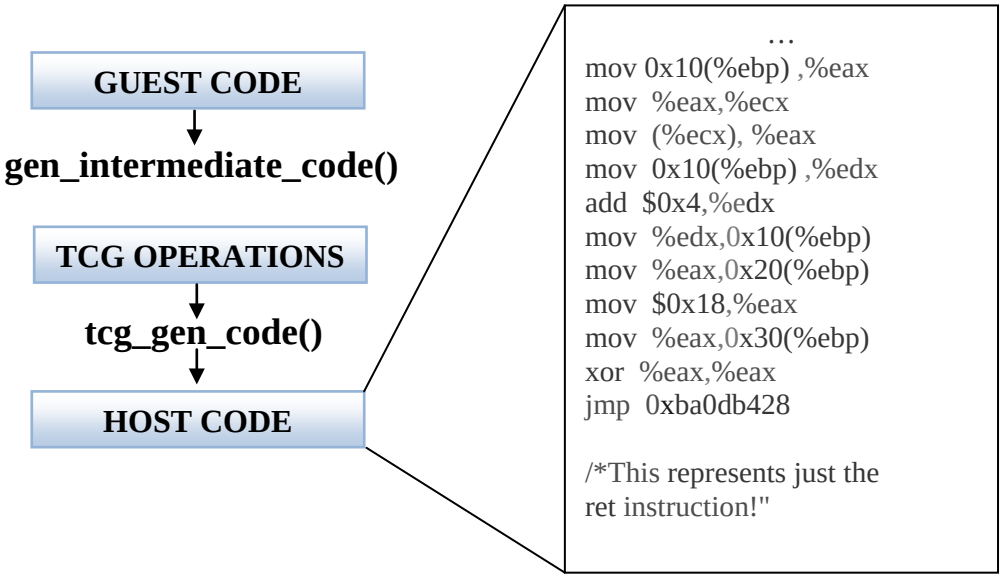

Figure 7.7: Dynamic translation – Showing Generated host Code (shown in assembly for readability)

7.3 Chaining of TBs:

Returning from the code cache to the static code (QEMU code) and jumping back into the code cache is generally slow. To solve this QEMU chains every TB to the next TB. So after the execution of one TB the execution directly jumps to the next TB without returning to the static code. The chaining of block happens when the Tb returns to the static code. Thus when TB1 returns (as there was no chaining) to static code the next TB, TB2, is found, generated and executed. When TB2 returns it is immediately chained to TB1. This makes sure that next time when TB1 is executed TB2 follows it without returning to the static code. Figure 7.8 (a-c) in the following page illustrates chaining of TBs.

7.4 Execution trace

This section will try to trace the execution of QEMU and specifically point out the location of specific files and declaration of functions called. This section will focus mainly on the TCG part of QEMU and will thus be key in finding the code sections that generate the Host code. A good understanding of code generation in QEMU will be necessary to help patch up QEMU in order to make the EVM.

The file/folder path notations are same as the ones used in the previous ‘codebase’ section but in order to specify the location of function declarations and define statements the same notations need to be augmented.

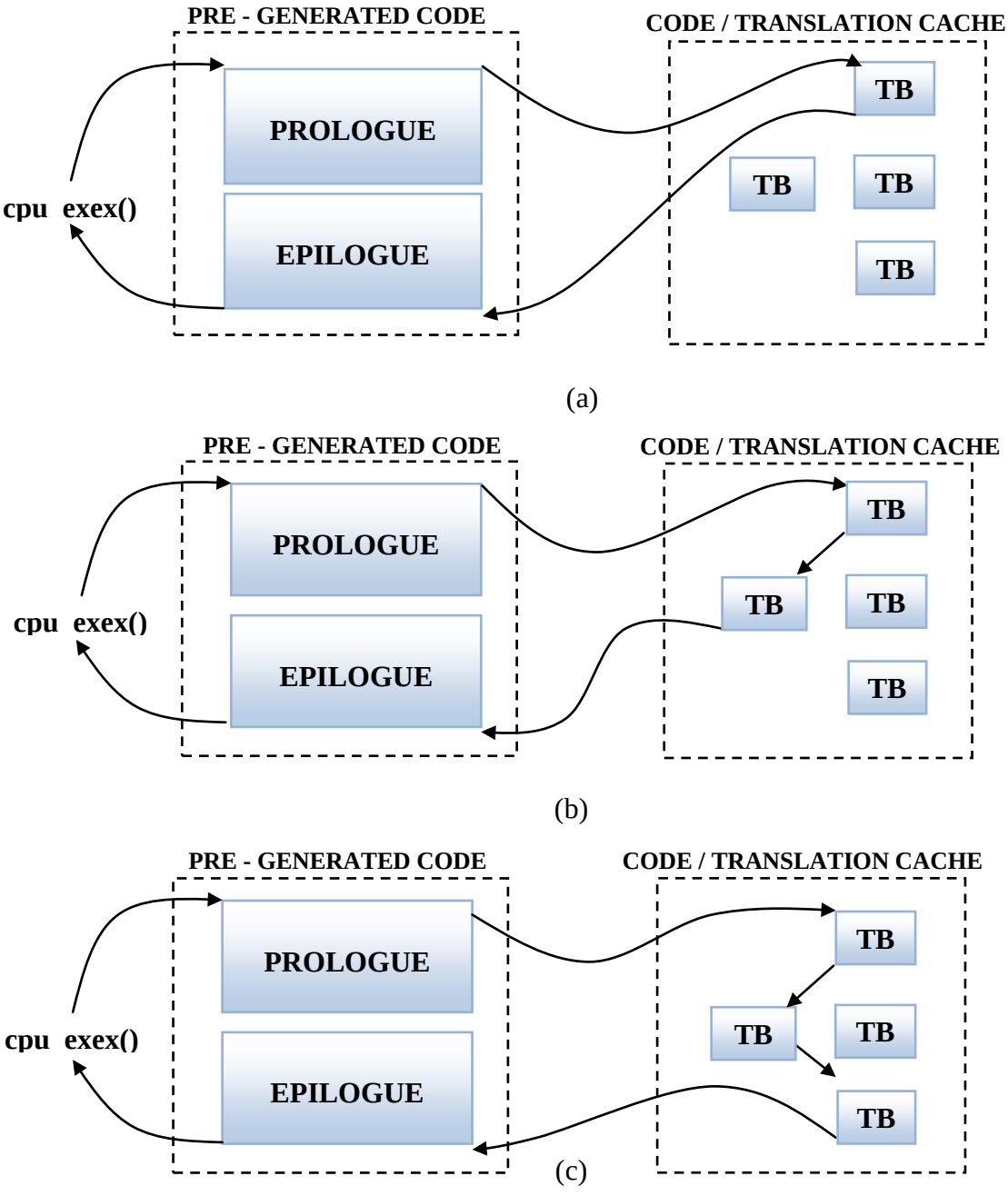

Figure 7.8: Chaining of TBs

Thus func1(...){/folder/file.c} would mean that declaration of func1() is in /folder/file.c the same goes with #define symbol_name{/folder/file.c}, var var_name {/folder/file.c}.

Similarly to highlight a particular piece of code the following convention is used.
:345
int max=MAX;
:
Suggests that ‘int max=MAX;’ is in line number 346 in the corresponding file.

main(..){/vl.c} : The main function parses the command line arguments passed during start-up and sets up the virtual machine (VM) based on the parameters such as size of ram, size of hard disk, boot disk etc. Once the VM is setup, main() calls main_loop().

main_loop(...){/vl.c} : Function *main_loop* initially calls qemu_main_loop_start() and then does infinite looping of cpu_exec_all() and profile_getclock() within a do-while for which the condition is vm_can_run(). The infinite for-loop continues with checking some VM halting situations like qemu_shutdown_requested(), qemu_powerdown_requested(), qemu_vmstop_requested() etc. These halting conditions will not be investigated further.

qemu_main_loop_start(...){/cpus.c} : Function *qemu_main_loop_start* sets the variable qemu_system_ready = 1 and calls qemu_cond_broadcast() which basically deals with restarting all thread waiting on a condition variable. This will not be further investigated here. Please. look in to /qemu-thread.c for more details.

profile_getclock(...){/qemu-timer.c} : Function *profile_getclock* basically deals with timing (CLOCK_MONOTONIC) and is not further investigated here.

cpu_exec_all(...){/cpus.c} : Function *cpu_exec_all* basically round robins the available CPUs (cores) in the VM. QEMU can have up to 256 cores. But all these cores will be executed in round robin fashion and thus do not fully mimic a multi-core processor in which all cores run in parallel. Once the next CPU is chosen it's state (CPUState *env) is found and the state is passed to qemu_cpu_exec() for continuation of execution of the chosen CPU from its current state, after checking a condition cpu_can_run().

struct CPUState{/target-xyz/cpu.h} : Structure *CPUState* is architecture specific and basically holds the CPU state like standard registers, segments, FPU state, exception/interrupt handling, processor feature and some emulator specific internal variables and flags.

qemu_cpu_exec(...){/cpus.c}: Function *qemu_cpu_exec* basically calls cpu_exec().

cpu_exec(...){/cpu-exec.c}: Function *cpu_exec* is referred to as the 'main execution loop'. Here for the first time a translation Block TB is initialized (TranslationBlock *tb) the code then basically continues with handling exceptions. Deep within two nested infinite for-loops one can find tb_find_fast() and tcg_qemu_tb_exec(). tb_find_fast() initiates the search for the next TB for the Guest and then generate the Host code. The generated Host code is then executed through tcg_qemu_tb_exec().

struct TranslationBlock {/exec-all.h}: Structure *TranslationBlock* contains the following; PC, CS_BASE, Flags corresponding to this TB, tc_ptr (a pointer to the translated code of this TB), tb_next_offset[2], tb_jmp_offset[2] (both to find the TBs chained to this TB. ie. the TB that follows this TB), *jmp_next[2], *jmp_first (points to the TBs that jump into this TB).

tb_find_fast(...){/cpu-exec.c} : Function *tb_find_fast* calls cpu_get_tb_cpu_state() which gets the program counter (PC) form CPUState (env) this PC value is passed to a hash function to get the index of the TB in tb_jmp_cache[] (a hash table). Using this index the next TB is found from tb_jmp_cache.

: 200

tb = env->b_jmp_cache[tb_jmp_cache_hash_func(pc)]

:

Thus it can be found that once a TB (for a particular PC value) is found it is stored in tb_jmp_cache so that it can be later reused from tb_jmp_cache using its index found using the hash function (tb_jmp_cache_hash_func(pc)). The code then follows to check the validity of the found TB, if the TB found is invalid then a call is made to tb_find_slow().

cpu_get_tb_cpu_state(...){/target-xyz/cpu.h}: Function *cpu_get_tb_cpu_state* basically finds the PC, BP, Flags from the current CPUState (env).

tb_jmp_cache_hash_func(...){/exec-all.h}: This is a hash function to find offset of TB in tb_jmp_cache using the PC as key.

tb_find_slow(...){/cpu-exec.c}: Function *tb_find_slow* is used when tb_find_fast() fails. This time an attempt to find a TB is made using physical memory mapping.

:142

phys_pc=get_page_addr_code (env, pc)

:

phys_pc should be the physical memory address of the Guest OS's PC, and it is used to find the next TB through a hash function.

:147

```
h=tb_phys_hash_func(phys_pc)
ptb1 = &tb_phys_hash[h];
:
```

The above ptb1 is supposed to be the next TB but its validity is checked in the code that follows. If no valid TB is found then new TB is generated through tb_gen_code() else if a valid TB was found then its quickly added to tb_jmp_cache at an index found by tb_jmp_cache_hash_func().

```
:181
env->tb_jmp_cache[tb_jmp_cache_hash_func(pc)] = tb;
 :
```

tb_gen_code(...){/exec.c}: Function *tb_gen_code* starts with allocating (tb_alloc()) a new TB, the PC for the TB is found from the PC of CPUState using get_page_addr_code() .

```
:957
phys_pc = get_page_addr_code(env, pc);
tb = tb_alloc(pc);
:
```

When this is done a call is made to cpu_gen_code() followed by a call to tb_link_page() which adds a new TB and links it to physical page tables.

cpu_gen_code(...){translate-all.c}: function *cpu_gen_code* initiates the actual code generation. In which there is a chain of subsequent function calls given as below.

gen_intermediate_code(){/target-xyz/translate.c}→
gen_intermediate_code_internal(){/target-xyz/translate.c → disas_insn(){/target-xyz/translate.c}

Function *disas_ins* does the actual conversion of Guest code into TCG ops through a long switch case of target(Guest) instruction and corresponding group of functions that end-up adding TCG ops to code_buff. After the TCG ops are generated a call to tcg_gen_code() is made.

tcg_gen_code(...){/tcg/tcg.c}: Function *tcg_gen_code* converts TCG ops to Host specific code. Check the previous section 'TCG- Dynamic Translator' to find out more on the functioning of TCG.

#define tcg_qemu_tb_exec(...){/tcg/tcg.g}: Once the next TB is obtained, through all the processes as detailed above, the TB need to be executed. The TB is executed through tcg_qemu_tb_exec() in /exec-cpu.c.

:644
next_tb = tcg_qemu_tb_exec(tc_ptr)
:

Infact tcg_qemu_tb_exec() is a macro function defined in /tcg/tcg.h

:484 (in /tcg/tcg.h)

```
extern uint8_t code_gen_prologue[];
:
#define tcg_qemu_tb_exec(tb_ptr) ((long REGPARM(*)(void *))
code_gen_prologue)(tb_ptr)
:
```

To understand what happens in the above line of code, a good knowledge of function pointers is required. The following lines will elaborate on understanding this.

It is well known that ***(int) var*** will explicitly convert a variable to type int. In the same sense (***(long REGPARM (*)(void *))*** is a type - pointer to a function that takes in void * parameter and returns a long. Here ***REGPARAM(*)*** is a GCC compiler-directive that causes parameter of a function to be passed through Registers instead of through the Stack.

The intention of ***((long REGPARM (*)(void *))*** would have been clear if the functions name appeared in it as ***((long REGPARM (*func_name)(void *))***. However it was used here without a function name (but serves the purpose). When an array name is used, the array base address is obtained, and thus points (pointer) to the array. Therefore ***(function_pointer) array_name*** will cast the array pointer as function pointer.

A function is called through its pointer as ***(*pointer_to_func)(args)*** therefore ***((long REGPARM (*)(void *))code_gen_prologue)(tc_ptr)*** does a function call. It is seen that a '*' is missing in the above function call, but one can test to see that ***(*pointer_to_func)(args)*** and ***(pointer_to_func)(args)*** are equivalent.

So the above explanation clarifies that code_gen_prologue, an array, is cast as function and executed. code_gen_prologue holds in it a function in binary form that takes in an argument tc_ptr and return a long which is the next TB. The function in code_gen_prologue is the Function Prologue (discussed in a previous section) that transfers control to the generated Host code pointed to by tc_ptr.