Adding new architecture to QEMU

Marek Vašut <marek.vasut@gmail.com>

June 1, 2017
Marek Vasut

- Contractor at multiple companies
- Versatile Linux kernel hacker
- Custodian at U-Boot bootloader
- Yocto (oe-core) contributor
- FPGA enthusiast
Structure of the talk

- How does a model computer work
- How to emulate a computer
- Introduction to QEMU
- Emulating with QEMU
- Userspace binary emulation
What do you need in a computer?

- CPU
- Memory
- Peripherals
Power up

- Power sequencing happens
- System brought out of reset
- CPU brought out of reset
- CPU is in defined internal state
- CPU starts its operation
CPU: Internal state

- Values stored in CPU’s registers
- CPU’s status register
- Interrupt status
- Cache configuration
- **Program counter (pc)**
- ...
CPU: How it works

1. Fetch instruction from memory (from pc)
2. Decode instruction
3. Perform action
4. Update internal state
5. GOTO 1

Implement the above in software, CPU emulator is done.
Memory

- Memory with boot program required (ROM, BIOS, …)
- Some fast read-write memory is useful
- NOTE: Any disks etc. are peripherals

Emulation:
- Naive: Allocate massive buffer, access with offset
- Less naive: Use MMU-alike approach
Memory: MMU

- Memory Management Unit
- Translates VA→PA
- Page granularity (usually 4kiB)
- Translate target PA to host VA
- Track host VAs in a linked list
Peripherals

- Separate IO space ⇒ separate insn (x86)
- Shared IO space ⇒ load/store insn (ARM)
- Intercept register access
- Call register handler upon access
- Peripheral has its own state machine
- WARNING: Peripheral can assert CPU interrupt
QEMU

- System-mode, emulates whole system (CPU, RAM, IO)
- User-mode, emulates runtime environment
- Supports about 20 targets
  - ARM, MIPS, PPC, x86, Sparc, xtensa, ...
- GPLv2 only (no GPLv3 code)
- Not timing-accurate
  - Emulates what CPU does, not how it does it
QEMU: CPU

- Tracks CPU’s internal state
- Dynamic binary translation, TCG
- Works like a JIT compiler
  Target insn → TCG micro insn → Host insn
- Faster than instruction interpreter
- Main loop in `cpu_exec()`, calls TCG
The cpu_exec() performs the following steps

1. Check if current PC is in code cache
   Yes: fetch Translation Block (TB) from code cache
   No: translate TB, insert into code cache
2. Execute TB
3. Optionally handle the fallout
4. GOTO 1
QEMU: Translation Block (TB)

Stream of insns ending with a Branch insn

TB is translated using gen_intermediate_code():
1. Fetch instruction at current PC
2. Decode instruction
3. Translate behavior into TCG micro insns
4. Append micro insns into current TCG context
5. If branch insn Then BREAK ; else GOTO 1
6. Optimize whole current TCG context
7. Translate TCG context to Host insn

TB is ready
QEMU: Executing TB

- TB is stream of insns, cannot be executed right away
- Handle like a C function:
  Add prologue and epilogue
- Prologue: set up execution env for TB
- Epilogue: clean up after TB
- The `cpu_exec()` calls Prologue-TB-Epilogue
- Prologue and Epilogue have significant overhead
QEMU: Chaining TB

- Check after returning from Epilogue if the next TB is in code cache
- If yes, QEMU can patch current TB to next TB directly
- Tightloop optimization

```
    cpu_exec()
    \-- Prologue
    \-- TB
    \-- Epilogue
    \-- Prologue
    \-- TB
    \-- Epilogue

  ⇒

    cpu_exec()
    \-- Prologue
    \-- TB
    \-- TB
    \-- Epilogue
```
QEMU: Translation pitfalls

- Anything with zero is special
- Division by zero, how is it handled in HW
  Can trigger exception or not
- Constant (zero) register (ie. on MIPS, Nios2)
  Must ignore writes
- Arithmetic with constant reg as destination
  Can trigger an exception (ie. div by zero)
- Load into constant reg
  Can trigger an exception (ie. MMU fault)
- Zero can be signed

Difficult instructions and exceptions handled by C-code Helpers:
- C function called from TB
static void divu(DisasContext *dc, uint32_t code, uint32_t flags)
{
    R_TYPE(instr, (code));

    /* Stores into R_ZERO are ignored */
    if (unlikely(instr.c == R_ZERO))
        return;

    TCGv t0 = tcg_temp_new();
    TCGv t1 = tcg_temp_new();
    TCGv t2 = tcg_const_tl(0);
    TCGv t3 = tcg_const_tl(1);

    tcg_gen_ext32u_tl(t0, load_gpr(dc, instr.a));
    tcg_gen_ext32u_tl(t1, load_gpr(dc, instr.b));
    tcg_gen_movcond_tl(TCG_COND_EQ, t1, t1, t2, t3, t1);
    tcg_gen_divu_tl(dc->cpu_R[instr.c], t0, t1);
    tcg_gen_ext32s_tl(dc->cpu_R[instr.c], dc->cpu_R[instr.c]);

    tcg_temp_free(t3);
    tcg_temp_free(t2);
    tcg_temp_free(t1);
    tcg_temp_free(t0);
}
QEMU: SoftMMU

- Used in system mode
- Uses two-level page tables
- Does Target VA → Target PA translation
- Does Target PA → Host VA translation
- Every memory access is translated
- Has TLB to speed up lookups

Notes:
- Code-cache is tagged by Target PA
- MMU flush unlinks TBs
QEMU: Peripherals

- Device registered in board init
- Callback triggered upon IO range access
- Device model tracks internal state
QEMU: Peripherals example: registration

```c
static Property altera_timer_properties[] = {
    DEFINE_PROP_UINT32("clock-frequency", AlteraTimer, freq_hz, 0),
    DEFINE_PROP_END_OF_LIST(),
};

static void altera_timer_class_init(ObjectClass *klass, void *data) {
    DeviceClass *dc = DEVICE_CLASS(klass);
    dc->realize = altera_timer_realize;
    dc->props = altera_timer_properties;
    dc->reset = altera_timer_reset;
}

static const TypeInfo altera_timer_info = {
    .name = TYPE_ALTERA_TIMER,
    .parent = TYPE_SYS_BUS_DEVICE,
    .instance_size = sizeof(AlteraTimer),
    .class_init = altera_timer_class_init,
};

static void altera_timer_register(void) {
    type_register_static(&altera_timer_info);
}

static void altera_timer_unregister(void) {
    type_unregister_static(&altera_timer_info);
}

type_init(altera_timer_register)
```
QEMU: Peripherals example: registration

```c
static void altera_timer_realize(DeviceState *dev, Error **errp)
{
    AlteraTimer *t = ALTERA_TIMER(dev);
    SysBusDevice *sbd = SYS_BUS_DEVICE(dev);

    if (t->freq_hz == 0) {
        error_setg(errp, ""clock-frequency" property must be provided.");
        return;
    }

    t->bh = qemu_bh_new(timer_hit, t);
    t->ptimer = ptimer_init(t->bh, PTIMER_POLICY_DEFAULT);
    ptimer_set_freq(t->ptimer, t->freq_hz);

    memory_region_init_io(&t->mmio, OBJECT(t), &timer_ops, t,
                         TYPE_ALTERA_TIMER, R_MAX * sizeof(uint32_t));
    sysbus_init_mmio(sbd, &t->mmio);
    sysbus_init_irq(sbd, &t->irq);
}

static void altera_timer_reset(DeviceState *dev)
{
    AlteraTimer *t = ALTERA_TIMER(dev);

    ptimer_stop(t->ptimer);
    ptimer_set_limit(t->ptimer, 0xffffffff, 1);
    memset(t->regs, 0, ARRAY_SIZE(t->regs));
}
```
static void timer_write(void *opaque, hwaddr addr,
        uint64_t value, unsigned int size)
{
    AlteraTimer *t = opaque;
    uint64_t tvalue;
    uint32_t count = 0;
    int irqState = timer_irq_state(t);

    addr >>= 2;

    switch (addr) {
    case R_STATUS:
        /* The timeout bit is cleared by writing the status register. */
        t->regs[R_STATUS] &= ~STATUS_TO;
        break;
    ...
    }

    if (irqState != timer_irq_state(t))
        qemu_set_irq(t->irq, timer_irq_state(t));
}

static const MemoryRegionOps timer_ops = {
    .read = timer_read,
    .write = timer_write,
    .endianness = DEVICE_NATIVE_ENDIAN,
    .valid = {
        .min_access_size = 1,
        .max_access_size = 4
    }
};
QEMU: Interrupts

- Device model calls `qemu_set_irq()`
- Current TB is unlinked from next TB
- Execution returns to `cpu_exec()`
- Exceptions handled in `cpu_exec()`
- Execution proceeds with next TB
QEMU: Board init

- Instantiates CPU
- Allocates memories
- Populates memories with content
- Instantiates device models and connects them
- Sets up default system state
QEMU: Board init example

1  memory_region_init_ram(phys_tcm, NULL, "nios2.tcm", tcm_size, &error_abort);
2  vmstate_register_ram_global(phys_tcm);
3  memory_region_add_subregion(address_space_mem, tcm_base, phys_tcm);
4  [...] cpu = cpu_nios2_init("nios2");
5
6  /* Register: CPU interrupt controller (PIC) */
7  cpu_irq = nios2_cpu_pic_init(cpu);
8  /* Register: Internal Interrupt Controller (IIC) */
9  dev = qdev_create(NULL, "altera,iic");
10  object_property_add_const_link(OBJECT(dev), "cpu", OBJECT(cpu),
11     &error_abort);
12  qdev_init_nofail(dev);
13  sysbus_connect_irq(SYS_BUS_DEVICE(dev), 0, cpu_irq[0]);
14  for (i = 0; i < 32; i++) {
15     irq[i] = qdev_get_gpio_in(dev, i);
16  }
17
18  /* Register: Altera 16550 UART */
19  serial_mm_init(address_space_mem, 0xf8001600, 2, irq[1], 115200,
20     serial_hds[0], DEVICE_NATIVE_ENDIAN);
21  [...] cpu = cpu_nios2_init("nios2");
22
23  /* Register: Timer sys_clk_timer */
24  dev = qdev_create(NULL, "ALTR.timer");
25  qdev_prop_set_uint32(dev, "clock-frequency", 75 * 1000000);
26  qdev_init_nofail(dev);
27  sysbus_mmio_map(SYS_BUSDEVICE(dev), 0, 0xf8001440);
28  sysbus_connect_irq(SYS_BUSDEVICE(dev), 0, irq[0]);
QEMU: User mode

- QEMU works as a user-mode virtual machine: Target binary runs on Host system
- TCG used to emulate Target code on host
- QEMU does Target VA → Host VA remapping
- Target sysroot is mandatory
  Any libraries are used from the target sysroot
- Signals and Syscalls
QEMU: Signals

- Sent to QEMU process
- Trapped and translated
- Host data structures converted to Target data structures
- Process is interrupted and Target signal handler invoked
QEMU: Syscalls

- Target sets up syscall data structures
- Target triggers CPU exception
- Syscall detected and translated
- Syscall invoked on Host system
- Syscall structures translated back
- Standard return from exception on Target
Thank you for your attention!

Contact: Marek Vasut <marek.vasut@gmail.com>