Emulation — 1

G. Lettieri

8 Oct. 2014

1 General strategy

Emulation is what we have done for the Manchester Baby example, but now we have to do it for a much more complex system, which also includes the I/O devices.

In emulation we implement the target CPU in software, and this gives us complete control on everything. We also implement all other pieces also in software: MMU, interrupts, I/O devices, ...

The general strategy for the implementation is as follows:

- write the emulator as a non-privileged program in the host system;
- define a data structure for each device (CPU, memory, MMU, each I/O dev, ...);
- write a CPU loop like the one in Fig. 1.

Since the emulator is a non-privileged program, it can only interact with the host hardware through the operating system libraries and primitives. We assume a Unix-like system with

- files, accessed through the `open`, `close`, `read`, `write` and `fseek` functions;
- processes and threads;
- the `select` system call.

In the Manchester Baby example we only had two devices: CPU and memory. Moreover, CPU only had two registers: the accumulator A and the instruction pointer CI.

1.1 Memory

To implement the Baby memory we defined an array of 32 `int32_t` entries. An array, or a buffer, is a good candidate for the implementation of the main memory also in the general case, but we have to consider some complications. In particular, the Baby memory was only word-addressable: each address was the

```
memory mem;
cpu_state cpu;

void cpu_loop()
{
  raw_instr ri;
  decoded_instr di;

  for (;;) {
      ri = fetch(cpu->ip);
      di = decode(ri);
      exec(di);
  }
}

void exec(decoded_instr di)
{
   switch(ri->opcode) {
   case ...
   case ...
   }
}
```

Figure 1: Emulated CPU pseudo-code.

address of a word, not of a byte (bytes did not even exist back then). Modern systems typically are byte addressed, they may support words of different sizes, and they may also allow *misaligned* reads and writes. The x86 architecture, in particular, supports all of these cases. Therefore, a more fitting candidate data structure for a modern main memory is an array of bytes.

1.2 CPU

The Baby emulator CPU state only consisted of two `int32_t` variables, one for A and the other for CI. A modern CPU will have many more registers, which we can put in a `cpu_state` data structure. Note that we already do something similar when we implement processes in a multiprogrammed kernel, but there is a difference: we now need to consider *all* the CPU registers available to the programmer, even those registers that are only accessible to privileged software (e.g., IDTR, CR0, CR2, ... in x86). This is because our emulator will need to run all of the software of the target system, including the system software.

A modern CPU will also have many more features that we need to emulate. In particular: interrupts, exceptions and protection.

1.2.1 Interrupts

We can implement interrupts by allocating a flag for the interrupt request and (if the target architecture needs it) a variable for the interrupt type. Then, the CPU loop must check the interrupt flag after the execution of each instruction. If the interrupt is set, we must load the interrupt handler address in the emulated CPU instruction pointer, and then start fetching again:

```
...
bool interrupt;
uint8_t  int_vector;

void cpu_loop()
{
   ...
  for (;;) {
       ri = fetch(cpu->ip);
       di = decode(ri);
       exec(di);
       if (interrupt) {
          ...
          /* obtain the new ip from
           * the interrupt descriptor table
           */
          cpu->ip = read_idt(int_vector);
       }
  }
}
```

We must also do all other things the target CPU does on interrupt acceptance. For the x86, these include saving the current (emulated) `EIP` and `EFLAGS` registers on top of the stack (the *emulated* stack in the emulated memory, pointed to by the emulated `ESP` register in the `cpu_state`), and many other things which we should recall from other courses.

1.2.2 Exceptions

Exceptions (e.g., division by 0, general protection, page fault, ...) are a bit more complex, since they may occur anywhere during the fetch, decode and execution of an instruction. If the language we are using supports it, we can use the `try ... catch` construct:

```
...
void cpu_loop()
{
  ...
  for (;;) {
    try {
      ri = fetch(cpu->ip);
      di = decode(ri);
      exec(di);
      if (interrupt) {
        ...
      }
    } catch (exception e) {
      ...
      cpu->ip = read_idt(e->type);
    }
  }
}

void exec(decoded_instr di)
{
  switch(ri->opcode) {
    ...
  case DIV:
    ...
    so = get_2nd_operand(ri);
    if (so == 0)
      throw exception(DIVISION_BY_ZERO);
    ...
  }
}
```

Figure 2: Emulation of exceptions in the CPU in the C language.

Note that, in x86, exceptions can also be raised by `read_idt` itself (gate not present, protection, even page fault), so we need to account for that also (this is left as an exercise for the reader).

The C language does not support exceptions. In this case we can use the `setjmp` and `longjmp` functions from the C standard library as follows:

```
...
#include <setjmp.h>
...
jmp_buf exc_jbuf;
exception exc_type;

void cpu_loop()
{
   ...
  for (;;) {
      if (setjmp(&exc_jbuf)) {
         ...
        cpu->ip = read_idt(exc_type);
      }
      ri = fetch(cpu->ip);
      di = decode(ri);
      exec(di);
      if (interrupt) {
         ...
      }
  }
}

void exec(decoded_instr di)
{
   switch(ri->opcode) {
      ...
   case DIV:
      ...
     so = get_2nd_operand(ri);
     if (so == 0) {
        ...
       exc_type = DIVISION_BY_ZERO;
       longjmp(&exc_buf);
     }
      ...
   }
```

```
}
```

First, we need to define a variable with type `jmp_buf` (the type is defined in the library). Then, we call `setjmp` with the address of our variable, which is `exc_jbuf` in the Figure. The functions stores in `exc_jbus` all the information needed to jump at the current program point and returns 0 (which means that the code in the `if` is skipped). Then, if/when we later call `longjmp` with `exc_jbuf`, the program will jump to the corresponding `setjmp`. This time, `setjmp` will return 1 and the code in the `if` will be executed. Note that we need to pass all additional information (like the exception type in the Figure) through global variables, since the stack is unwinded during the jump.

1.2.3 Protection

To emulate protection we simply need to implement in software all the checks performed by the target CPU. For example, the `SIDT` x86 instruction changes the CPU pointer to the interrupt descriptor table and it is, of course, a privileged instruction that can only be executed when the CPU is at the system privilege level. In our emulator we will need to do something like the following:

```
void exec(decoded_instr di)
{
   switch(ri->opcode) {
      ...
   case SIDT:
      ...
     if (cpu->privilege_level < SYSTEM)
       throw exception(GENERAL_PROTECTION);
    /* otherwise */
    cpu->idtptr = ...
     ...
   }
}
```

1.3 I/O

I/O devices are connected to the rest of the system via interfaces. Interfaces have a set of registers that are mapped in I/O or memory space. I/O registers look like memory locations, but the crucial difference is that whenever we write (or even read) from an I/O registers, actions take place, e.g., a character is printed, a message is sent, and so on. Therefore, in our emulator, we need a way to map I/O register accesses to functions, rather than simply to locations in memory.

In x86, I/O space can only be accessed via the `in` and `out` instructions, therefore we can emulate all accesses to I/O mapped interfaces with something like (assuming all operand sizes are the same):

```
void exec(decoded_instr di)
{
   switch(ri->opcode) {
      ...
   case IN:
      ...
     a = get_io_addr(di);
     v = io_input(a);
      ...
   case OUT:
      ...
     v = get_1st_operand(di);
     a = get_io_addr(di);
     io_output(v, a);
      ...
   }
}
```

The `io_input` and `io_output` functions might be implemented with another switch:

```
void io_output(operand v, ioaddr a)
{
   switch(a) {
      ...
   case 0x40:
           /* this is the timer */
            ...
   case 0x60:
           /* this is the keyboard */
            ...
   }
}
```

However, such a solution would be very inflexible. The PC platform, for example, can come with many different configurations of I/O devices and cannot be easily captured by such a static mapping of addresses. A much better solution is to have a data structure that maps I/O addresses to the data structures that represent the I/O devices. Then, we can do something like:

```
iomap io;

void io_output(operand v, ioaddr a)
{
    iodevice *iodev = io.search(a);
```

```
        if (iodev != NULL)
                iodev->set_register(v, a);

    }
```

But we also need to consider *memory* mapped interfaces, i.e., interfaces whose registers are given memory addressed and then are accessed via any instruction that can have memory operands. In this case we need to do something like the following, whenever an instruction tries to write memory (and similarly for reading):

```
    memory mem;
    mem_map mm;

    void mem_output(operand v, addr a)
    {
        iodevice *iodev = mm.search(a);

        if (iodev != NULL)
                iodev->set_register(v, a);
        else
                mem[a] = v;

    }
```

That is: first check whether the given address corresponds to an I/O interface, and only if this is not the case do a normal write into the (emulated) memory. The cost of this lookup can be mitigated if I/O is restricted to some fixed region of memory, since in that case we can do a quick check on the address value to understand if it corresponds to normal memory, without performing the (possibly expensive) lookup into the `mm` data structure.

1.3.1 Asynchronous events

The biggest problem with I/O devices is that they introduce asynchronous events in our emulation: "things" must happen in the devices while our program is executing the CPU loop.

As a first example, let us assume that our emulated CPU writes a character in the transmit buffer of an interface connected to a (very old) printer. The printer will start printing the character and reset the "buffer empty" bit in its status register, since now it cannot accept any other character. Concurrently, the CPU will continue to fetch and execute other instructions and, if it tries to read the printer status register, it will see the buffer empty bit at 0. At some later time, the printer will finish printing the character and it will set the buffer empty bit in the status register. If the CPU tries to read the status register now, it will see the buffer empty bit at 1. Therefore, the result of the status register

read should depend on an event which is asynchronous with respect to what our emulated CPU is doing. We can solve this particular problem by assuming that the emulated printer is very fast, so fast that the CPU is never able to see the buffer empty bit go to 0: the read from the status buffer can simply always return 1 in the buffer empty bit. This is possible if the action that we have to perform in response to the write to the transmit buffer can be implemented by a (mostly) non-blocking operation in the host system. For example, this is the case if we emulate the printer by simply writing the received characters in a file, or by showing them on the terminal.

Assume now, as a second example, that the CPU tries to read from the receive buffer register of the keyboard. We can emulate this by, e.g., reading from the terminal with a `read` system call. Now we have a different problem: the `read` system call may block the process waiting for the user to press a key on the keyboard (actually, it normally waits until the user has entered a complete line, but we can ignore this for now). But we cannot block our emulated CPU waiting for input, since this is not how the real target system works: in the target system, the read will complete in essentially constant time, returning whatever is the current content of the receive buffer register, and then the CPU will continue fetching and executing instructions. We can solve this particular problem by using *non blocking* I/O in the host system. This is an option that can be set on a file descriptor (including one connected to a terminal). If the option is set, any `read` will return an error if no input is available, instead of blocking. If the `read` returns error, we can complete the emulated input instruction by returning the previously read character.

As a final example, assume the program running on our emulated CPU is trying to read from the keyboard using interrupts. Now, we need to set our emulated interrupt flag as soon as the emulated keyboard has a new key available. But, again, our emulator only knows that the emulated keyboard has a new key when the `read` system call returns (without error). We have essentially two choices here:

- put the file descriptor corresponding to the emulated keyboard in non blocking mode, then periodically (e.g., after the CPU loop has executed a few instructions, or whenever it executes an `HLT` instruction) issue a `read` to check whether something new has arrived;

- use multi-threading; we can use a thread for the CPU loop, and one or more threads for the I/O devices.

(Actually we also have a third, less common one: use asynchronous I/O, if available.) If we have a separate thread for the emulated keyboard, we can simply block it in the `read` and set the interrupt flag (in shared memory) whenever the system call returns.

Most emulators, however, do not have a separate thread for each device. Another solution is to have just one thread for all the I/O devices. This thread is normally blocked on a `select` system call that checks all the file descriptors corresponding to all the devices. Whenever any one of the file descriptors is

ready, the `select` returns, the thread uses some data structure that maps the file descriptor number to an emulated I/O device, it performs the necessary actions on the devices (possibly setting the interrupt flag), and then blocks in the `select` again.