Emulation

G. Lettieri

Oct. 2018

1 General strategy

The program we have written for the Manchester Baby is an *emulator*. An emulator is a program that is built around a main loop that mimicks the actions of the target machine processor: fetch a single instruction, decode it, execute it and start again. The execution phase typically contains a big switch with a case for each possible instruction of the target processor. By emulating every instruction, we are able to run any software that was written for the target machine. This also gives our emulator complete control on all the things that target software tries to do.

Let us now look at how we can build an emulator for machine with a more complex CPU and 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, the CPU only had two registers: the accumulator A and the instruction pointer CI.

```
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.

1.1 Memory

To implement the Baby memory we defined an array of 32 `int32_t` entries. An array 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 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);
```

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

```
        if (so == 0)
          throw exception(DIVISION_BY_ZERO);
        ...
    }
}
```

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_jbuf` 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.

If we go back to our state machine abstract model, we se that now the T-state contains the state of the target CPU (the contents of its register, the interrupt flag, and so on), the state of the target main memory, and also the state of the target printer. This latter state is made up of the contents of the interface registers (in our case, the transmit and status registers), and the state of target I/O device itself. It is no longer true that this T-state changes only when the CPU executes an instruction, i.e., the T-next function should take into account also events that occur inside the printer. Since these events occur concurrently with the execution of the instructions, things may become very complicated very quickly: we should take a new snapshot (modelled as a T-state change) also when the printer becomes ready to accept a new character, but this may occur while the target processor is in the middle of the execution of an instruction. Note that this is different from exceptions, since these are only raised in precise points in our program, while the change in the I/O device state may occur at any moment.

A simple way to think of the new system without disrupting too much of what we have already done, is to model it as a *non-deterministic* state machine. Like before, we take a new snapshot only when the CPU has completed the execution of an instruction (possibly aborted by an exception), but whenever the snapshot captures a 0 in the buffer-empty bit of the printer status register, we allow for *two* possibile next states: in both states the CPU will have executed the next instruction, but in one state the buffer-empty bit is still 0, and in the other state the bit has become 1. Each execution of the software in the target system may take one path or the other.

In general, our virtual machine will also be non-deterministic, so each V-state may also have more than one next state. Assume now that we start from a V-state v that is interpreted as a T-state t. We say that our virtual machine preserves the interpretation if, *for each* V-state v' which is a next-state of v, *there exists* a T-state t' among the next-states of t such that t' is an interpretation of v'. In other words, each execution of our virtual machine must correspond to at least one *possible* execution of the target machine.

Note that this definition allows our virtual machine to *never* emulate some target executions. This may or may not be acceptable, depending on context. In the printer emulation example, it may be acceptable: we can assume that the target 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. From an implementation point of view, this is possible if the action that our virtual machine has 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). Blocking the emulated CPU for an unbounded amount of time may not be acceptable. Assume, for example, that the software running in the target is a videogame: the enemies must continue to move even if the player is not moving her character. 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 option: 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.

2 I/O examples

We now sketch the implementation of some I/O interface we already know. It is a useful exercise to fill the missing details and to think at the emulation of other interfaces.

We can implement each I/O device as an object, or a set of objects. The real device in the target system is connected to the CPU via an interface consisting of a set of registers, with may reside in a separate address space (e.g., the I/O address space of IA32 architectures, or the configuration space of PCI devices), or just in memory space. In all cases, the emulated CPU will access the emulated device via a (at least) a pair of functions provided by the object that implements the device, which we can define as follows (assuming 8 bit registers only):

```
class IOdev {
public:
        // get the contents of register at address a
        virtual uint8_t get_register (address a) = 0;
        // write v into register ad address a
        virtual void set_register (address a, uint8_t v) = 0;
}
```

These functions are the interface between the device object and the emulated interface. The object will implement them to complete the I/O emulation. In order to do that, the object has to interact with the host operating system. Therefore, there will be another interface, between the I/O object and the host OS. The details of this latter interface depend on the kind of device we are emulating *and* the way we are using the host resources to emulate it.

It is often useful to split the code that emulates a device into two parts:

- a *frontend*, that only depends on the device we are emulating (e.g., which registers the devices has and what they do);
- a *backend*, that depends on how we are using the host to emulate the device (e.g., we are using a file, a terminal, . . .).

The frontend implements the interface between the emulated CPU and the emulated device (the *CPU-frontend interface*), while the backend implements the interface between the emulated device and the host (the *host-backend* interface).

In some emulators the backend and frontend are implemented as two independent objects, interacting through yet another interface: the *frontend-backend* interface. This setup enhances the flexibility of the emulator since now we can attach one of several equivalent emulated devices (e.g., Ethernet cards from Realtek, Intel, Novell, . . .) to any of several possible host resources (e.g., sockets, software bridges, tap devices, . . .). This interface is also dependent on the kind of emulated device (e.g., video adapter, network interface, block device, . . .).

2.1 Multi-threading

If the emulator uses several threads (e.g., one for the CPU loop and another for the I/O `select()` loop), then the I/O device object will be accessed concurrently and will need to be protected from interference problems with any of the techniques you already know or you are studying right now.

Be careful not to mistake the frontend/backend separation with the CPU-thread/IO-thread one: it is generally *not* the case that the frontend is only accessed by the CPU-thread and the backend only by the IO-thread. Instead, depending on the emulator and the kind of device, it may well be the case that *both* frontend and backend are accessed concurrently by both threads. In a typical interaction, the CPU thread executes an I/O instruction that accesses a register in the I/O device and it calls the corresponding `set_register` (or `get_register`) method on the device. This method will update the state of the object (to match the state of the target system) and then will call into the backend (using the frontend-backend interface) to complete the I/O operation, all in the same thread. Concurrently, the I/O thread may exit from the `select()` and find out that the same device needs to be updated (e.g., a new key has been pressed on the keyboard). Then, it will call some function of the backend (from the host-backend interface), which will in turn call some function in the frontend (from the frontend-backend interface). All these function calls occur in the context of the I/O thread. In the following we will omit the solution to the concurrency problems: assume that each interface function is protected by per-object mutual exclusion (it is *synchronized* in the Java sense).

2.2 Hard Disk

Let us consider a simplified and abstract version of the ATA interface, with:

- a set of registers SN1, SN2, SN3, SN4 to specify the Sector Number (we assume there are four of them since a sector number does not fit into a single one);
- one CMD register to specify the operation we want the device to perform (e.g., sector read or sector write);
- a BR register that gives us access to an internal buffer of the interface.

Assume all registers are 8 bits wide and a sector is 512 bytes. A write operation is started as follows:

1. write the sector number in the SN1, . . . , SN4 registers;
2. write the code for “write sector” in the CMD register;
3. perform a sequence of writes in the BR register to fill the internal buffer with the data we want to write.

Only when the buffer is full the interface starts the write operation, ordering the device to move the read/write head to seek for the desired sector, and then start writing the buffered data to the disk plate.

To emulate this interface, the corresponding (frontend) I/O object in our emulator program will look like the following (assuming C++):

```
class HDFrontend: public IODev {
        enum { iSN1, iSN2, ... };
        static const uint8_t WRITE = ...;
        static const uint8_t READ = ...;

        uint8_t SN1, SN2, SN3, SN4;
        uint8_t CMD;
        uint8_t buf[512]; // the internal buffer
        int next; // next read/write position in the buffer

        /* we assume split frontend/backend */
        HDBackend *be;

public:
        HDFrontend(HDBackend *be_): next(0), be(be_) {}

        void set_register(address a, uint8_t v) {
                /* the interface may be mounted at several addresses,
                 * but the the lower bits of the address always
                 * identify the same register
                 */
                int index = a & ADDR_MASK;
                switch (index) {
                case iSN1:
                        /* just store it for later */
                        SN1 = v;
                        break;
                case iSN2:
                        /* just store it for later */
                        SN2 = v;
                        break;
                 ...
                case CMD:
                        /* just store it for later */
                        CMD = v;
                        break;
                case iBR:
                        if (CMD != WRITE) {
                                /* an error */
                        }
```

```
            buf[next++] = v;
            if (next == 512) {
                /* compute the sector number */
                uint32_t sn = SN1 | SN2 << 8 | ...;
                /* start the write */
                be->write_sector(sn, buf);
                /* reset */
                next = 0;
            }
            break;
        ...
        }
    }
    ...
};
```

The backend may use a file in the host system to emulate the target hard disk. It needs to translate the sector number into a file offset and perform the write.

```
class FileBackend: public BlockDeviceBackend {
    int fd; // file descriptor
    size_t hdsize;
    ...
public:
    FileBackend(const char *filename, size_t hdsize_):
        fd(0), hdsize(hdsize_)
    {
        /* create if it does not exist, but do not O_TRUNC! */
        fd = open(filename, O_RDWR|O_CREAT, 0660);
        if (fd < 0) {
            /* cannot continue, throw error */
            throw ...;
        }
    }
    ~FileBackend() {
        close(fd);
    }
    void write_sector(int sn, const uint8_t *buf) {
        if (outsideHD(sn)) {
            /* do not write, update internal state */
            ...
            return;
        }
        off_t offset = sn * 512;
        lseek(fd, offset, SEEK_SET);
        write(fd, buf, 512);
        }
         . . .
};
```

The *BlockDeviceBackend* should define the frontend-backend interface for block devices. It should contain the declaration of the write_sector () and, of course, read_sector () functions. A *FileBackend* is only one of several possible *BlockDeviceBackends*, other options being a disk partition in some host hard disk, an host device such as a DVD, ...

Note the difference in the error handling in the constructor and in the write_sector () function. In the former case, our emulator is not able to open the file: this is an error in the host and the emulator most likely cannot continue. In the latter case, the frontend passed the backend an invalid sector number (one that is outside the emulated hard disk): this is an error *in the guest*, and now we should emulate the behavior of the target hard disk in the same scenario (set some error bit in its status register).

2.3 Video

Let us briefly recall how VGA-compatible video adapters work. If the adapter is in text mode, the display is organized in rows and columns of characters and there is an area of memory that describes the contents of each character. In the memory area, each character is represented by a pair of bytes, one byte for the ASCII value of the symbol that must be displayed and another byte for the foreground/background colors and optional blinking. In the simplest to use graphics modes, the display is organized in rows and columns of pixels and there is a much bigger memory area where the programmer can set the color of each pixel.

The main point to note is that programs show output on the video by simply writing in the video memory area, where each write modifies only one or two characters or a few pixels (possibly just one). If we map each of these writes to a function, video updates are going to take a lot of time. Instead, we can draw inspiration from the target hardware itself: writes into video memory do not instantly cause video updates; instead, the video adapter periodically (with a frequency of, say, 60 times a second) scans the entire video memory to generate the corresponding signals for the display. We can adopt the same strategy for our emulator: we implement video memory with a plain buffer, so that writes to emulated video memory translate to writes into the buffer, with no other immediate side effect (and, therefore, no function to call). At the same time, we arrange for the I/O tread (or a dedicated thread) to periodically read the buffer and, e.g., draw the corresponding contents in a window on the host display.

2.4 Interrupt Controller

We know that I/O devices do not send interrupts directly to the CPU, which typically only has one interrupt request pin for I/O. Instead, the target system may have an interrupt controller which collects all interrupt requests coming from the I/O devices, prioritizes them and then sends them to the CPU one at a time, possibly with a number that identifies the interrupt source. This is how things work for IA32 processors coupled with the APIC. If interrupts are enabled in the CPU, it responds to the interrupt request from the APIC with a jump to an interrupt handler routine. When the routine completes, it has to write an EOI word in an APIC register: the APIC is then enabled to pass the CPU lower priority interrupts that may have been queued in the meantime.

In our emulator, the interrupt controller is another *IOdev* object, since the CPU must be able to read and write into its registers (in particular, the EOI register). At the same time, the interrupt controller is used by other objects: when an emulated device want to raise an interrupt, it cannot simply set the CPU interrupt flag, but it has to notify the controller instead. This notification can be implemented by a call to a method in the interrupt controller interface. This method will update the internal state of the controller and either set the interrupt flag, or queue the request internally if it cannot pass it right away (e.g., because there are higher priority requests pending). The set_register () method of the interrupt controller object, when it identifies a write to the EOI register, can then inspect the internal queue and pass other interrupts as needed. Note that, in a multi-threaded emulator, the actions corresponding to an interrupt request will be performed in the I/O thread, while the actions following the EOI are executed by the CPU thread.

3 Virtual Memory

One of the most complex part of virtualization is the management of virtual memory. Here, we are talking about virtual memory *in the guest*. If the target system includes a MMU, the target system code will try to program it to implement its own virtual memory.

Assume, for example, that we use an array of bytes, call it *Mem*, to emulate the target physical memory, and assume that our CPU loop is trying to execute the following IA32 instruction

```
        movl 4100, %eax
```

If pagination is not enabled, the emulator will correctly read the operand at Mem[4100], since the target CPU will load the operand from the corresponding address 4100 in its physical memory. If pagination is enabled, however, address 4100 is virtual and the target CPU will not load the operand from address 4100. Instead, he address will be translated into something else, say 8196, by the MMU, and then address 8196 will be loaded from memory. Accordingly, our emulator must read Mem[8196] instead of Mem[4100]. We need to make a distinction among several kinds of addresses:

- address 4100 in the example is a *guest virtual address*; it is an address that needs to be translated according to the rules set up by the guest MMU;

- address 8196, the result of this translation, is a *guest physical address*; it is the address of a location in the guest physical memory;
- &Mem[8196], i.e., the address of the location the emulator will actually read, is a *host virtual address*, if pagination is enabled in the host (as it is usually the case);
- finally, the host virtual address will be translated to an *host physical address* by the host MMU.

The translation from guest virtual to guest physical is performed by the guest MMU and we need to emulate it. We need to provide a translate_address () function that returns a guest physical address when given a guest virtual address, then we need to call this function whenever the CPU loop needs to access guest memory. Recall that these accesses not only occur during instruction operands loads and stores, but also during instruction fetches, since instructions are in memory.

There is also a translation, that must not be overlooked, from guest physical to host virtual addresses. In this simple example the translation is simple: we use the guest physical address as an index in the Mem array. In the example, we translate 8196 to whatever the value of &Mem[8196] is. Note, however, that if the memory data structure we use is more complex, this translation will accordingly be more complex. We already know of a source of complexity here: if the target system has memory mapped I/O devices, before accessing memory we first need to check whether the address maps to any I/O device.

Finally, the host virtual address is translated into an host physical address. In our example, we have nothing special to do to make this translation happen: the emulator is an unprivileged program in the host system and it typically has no control at all on its virtual memory, which is transparently managed by the host kernel.

3.1 Translating guest virtual addresses

To implement the translate_address () function we need to implement in software the actions performed in hardware by the target MMU. Assuming an IA32 system, we need to read the (guest physical) address in the cr3 field of the emulated CPU data structure and read the emulated physical memory at offset

```
CPU->cr3 + 4 * ((virtual_address & 0xFFC00000) >> 22)
```

to obtain the page table descriptor, an so on. If we find some reset P bit, we also need to throw a page fault exception.

We can speed up this process, again, by drawing inspiration from the hardware. The target MMU includes a TLB, which caches recently used translations. We can also define a software TLB of sorts, where we put recently used translations in a form that is more easily accessed. Note that the hardware TLB is useful because it is a much faster memory compared to the main memory containing the actual page tables. The software TLB, instead, is a data structure in our program just like the emulated physical memory is. The software TLB can be still useful, however, if it allows us to complete the translation from guest virtual address to guest physical address using fewer instructions.

A simple TLB is an array of entries like the following:

```
struct TLB_entry {
        uint32_t  guest_virtual;
        uint8_t*  host_virtual;
};
```

where we directly cache the mapping from guest virtual address to host virtual address (in the above example, we would cache the mapping from 4096 to &Mem[8192]: recall that mapping are between pages, while the offset in the page, which is 4 in the example, simply has to be added to the resulting pointer). We can use NULL in the second field to mark an empty entry.

Assume we keep an array of, say, 1024 TLB_entry's. When we need to translate a guest virtual address, we first use an hash function to obtain an index into the array from the address, we check that the corresponding entry is not empty and it actually contains the address we are looking for. If this is the case, the host_virtual field already gives us a pointer into the emulated memory. Otherwise, we complete the translation in the ordinary way, then store the resulting pointer in the entry (replacing any previous one). The hash function may be very simple: since programs typically access their memory in order, we can simply use the 10 lower order bits of the guest virtual page address.

4 The IA32 case study

We conclude this notes on emulation by focussing a little more on the emulation of the IA32 and AMD64 architecture.

Several features of the IA32/AMD64 architecture make its emulation particularly inefficient. The worst offenders are:

- the incredibly complex instruction format;
- the EFLAGS/RFLAGS register.

4.1 The IA32 instruction format

The IA32 instruction format has evolved over time, starting with 16 bits processors, moving to 32 bits, then adding new sets of instructions in several rounds. The format was variable length already in the first processors, and this forces any emulator to have a combined fetch&decode function. In the current processors, the format is as follows:

1. optional 1–4 prefix bytes;

2. 1–3 opcode bytes;
3. optional Mod R/M byte;
4. optional SIB byte;
5. optional 1/2/4 bytes displacement;
6. optional 1/2/4 bytes immediate operand.

Prefix bytes are used for several purposes, among them we can mention:

- the REP, REPE, REPNE, ... prefixes for string instructions;
- the LOCK prefix, to lock the bus and turn instructions like XCHG into atomic operations;
- some prefixes to change the default size of "large" operands. Operands may be either "short" (8 bits) or "large" (either 16 or 32 bits depending on several things, including these prefixes).

The opcode may include three bits that encode a register operand, so that instructions like pushl %**eax** may be encoded in just one byte. The "Mod R/M" byte encodes, in complex ways, the mode of one of the two operands, which may be either register, immediate, or memory, the latter in several formats. The "SIB" byte encodes the Scale, Index and Base of memory operand expressions like 16(%**ebx**, %**ecx**, 4). The "16" displacement in this expression is encoded in the "displacement" field. The optional "immediate" field encodes the value of operands such as $1000.

The encoding is not regular at all, since several special cases have been added over the years to add new features and new instructions. The Bochs file that implements the fetch&decode function for IA32 is more than 2000 lines long!

Of course, all this complexity adds to the time needed to decode each IA32 instruction.

4.2 The AMD64 instruction format

The AMD exstension of the IA32 architecture to 64 bits, later adopted also by Intel, reuses much of the IA32 instruction format with an additional byte prefix: the REX prefix. Some instructions in AMD64 (most notably the PUSH/POP instructions) default to 64bit operands. All the others, however, only use 64 bits operands when prefixed by REX. This adds another byte to the encoding of most instructions.

Moreover, AMD64 brings the number of general purpose registers from 8 to 16, but the IA32 machine code only had 3-bits fields for registers names (e.g., in the SIB byte). The REX byte also provides the missing bits for those fields.

4.3 The IA32 EFLAGS (AMD64 RFLAGS) register

The main problem with the EFLAGS register is that almost all instructions update it: this includes the arithmetic instructions, the compare instructions, the logic instructions, the shift and rotate instructions. Depending on the emulator architecture, and on the host CPU, the emulator may need to compute all the flags in software. Some of these are both complex to compute, and little know and used:

- the PF (parity) flag is set if the number of set bits in the least significant byte of the result is even;
- the AF (adjust) flag is set if there has been a borrow/carry out of the four least significant bits of the result (used in BCD arithmetic).

Our emulator will have to compute all these things, for many instructions.

Note that, even if the EFLAGS bits are updated very often, they are very seldom read. Essentially, they are only read by conditional jump instructions. Therefore, there is room for optimization here, since we can think to compute these flags in a “lazy” way, delaying their computation until they are actually needed, which in many cases means never.

We don’t explore this optimization here, since we are going to introduce a better technique that allows us to address this and other problems: *binary translation.*