Jakub Radlak

Orbital Flight Simulator - educational program Virtual Machine Description

Warsaw, April 2024

Contents

Chapter 1

On-board computer model - Virtual Machine

The on-board computer is implemented as a sub-module of our simulation application. This is called a "Virtual Machine" with its own virtual memory, registers and instruction set. In addition, the VM has a byte-code interpreter and a translator from assembler to byte-code. This is all written in C++ and runs asynchronously in a separate thread. The application includes a simple assembly language code editor and offers the ability to load and save programs.

The figure 1.1 shows a general diagram of the Virtual Machine. In the following sections, we will discuss in detail the objects shown in this figure. That is: assembly code, translator, byte-code, interpreter, registers and main memory.

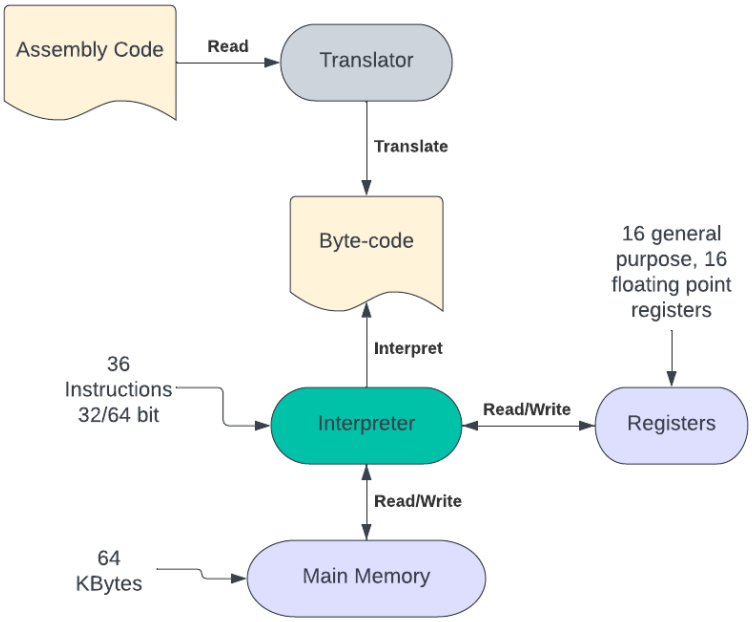

Figure 1.1: Virtual Machine Schema, source: self-elaboration

1.1 Main memory organisation

The virtual machine contains 64 kilobytes of random access memory. Memory has no specific structure: it is represented as single-dimensional array of bytes. The interpretation of a series of bytes in memory depends on the context of the instruction. For example: it may be a floating point number, an integer number, a string, e.t.c. The memory class written in C++ contains several methods that enables to store and fetch data. Listing (5.1) shows methods for storing and retrieving double-precision floating-point numbers. Each number of this kind is 8 bytes in size.

Listing 1.1: Fetch and store dword into memory

```
double Memory::fetchDWord(unsigned int addr) {
        assertConditions(addr + 8);
        unsigned char result[8] = { };
        memcopy(mem, result, addr, 0, 8);
        double val = *reinterpret_cast<double*>(result);
        leaseSemaphore();
        return val;
}

void Memory::storeDWord(unsigned int addr, double dword) {
        assertConditions(addr + 8);
        unsigned char* result
          = static_cast<unsigned char*>(static_cast<void*>(&dword));
        memcopy(result, mem, 0, addr, 8);
        leaseSemaphore();
}
```

The universal function *memcopy* copies a series of bytes between the requested locations in memory. Both methods use the *assertConditions* and *leaseSematore* functions, which provide multi-threaded security in concurrent resource access situations.

The example in Listing 1.2 shows a VM instruction that uses the *fetchDWord* function to fetch a number from memory and store it in a floating point register. We will talk about the instructions in the later sections of this paper.

Listing 1.2: Method that load from the memory into the register

```
void Instructions::fld(unsigned char* args) {
        unsigned char r_src_addr = args[1];
        unsigned char r_dst_addr = args[0];
        unsigned int src_addr = registers[r_src_addr];
        double value = memory.fetchDWord(src_addr);
        registers.fl(r_dst_addr, value);
}
```

There are regions of memory that have special interpretation and treatment. The figure 1.2 shows the main memory organization.

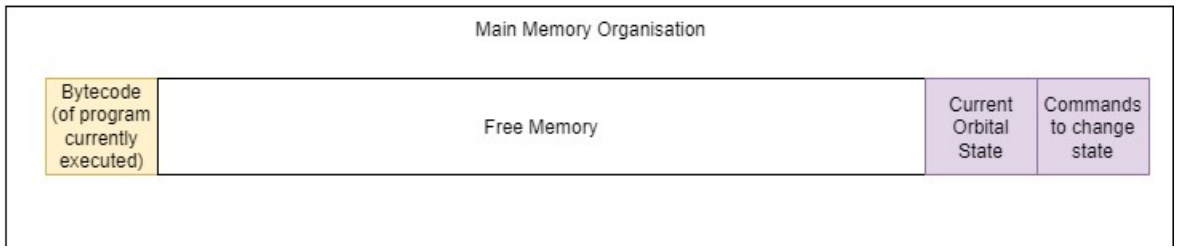

Figure 1.2: Map of the memory, source: self-elaboration

The initial part of the memory is used to store the translated byte-code of the currently executing program. Usually it is from a few to several thousand bytes depending on the complexity of the program. Byte-code is represented as binary stream of bytes and it is very compact. After byte-code there is the "free space" where programs can store their data, e.t.c.

Counting from the end, a few dozen bytes are used to store rocket flight telemetry information. This memory section is structured as follows:

```
Rotation:
        z       65528
        y       65520
        x       65512
Velocity:
        z       65504
        y       65496
        x       65488
Position:
        z       65480
        y       65472
        x       65464

        mass              65456
        thrust magnitude  65448
        altitude          65440
        timestamp         65432
```

The last few bytes are reserved to store temporary command data sent to the rocket.

1.2 Registers organisation

The virtual machine has 16 floating point registers and 16 general purpose registers. Floating point registers can store 64 bit double precision floating point numbers. The general purpose registers are 32 bits. So we can say that the VM has a hybrid 32/64 bit architecture and has a built-in FPU (Floating Point Unit).

In addition, there are three special-purpose registers:

- Zero Flag Register (ZF) - is used in compare and jump instructions. When the two compared values are equal, it takes the value 1.
- Carry Flag Register (CF) - is used in comparisons and jump instructions. When the first of the compared values is greater than the second, the register is set to 1.
- Program Counter (PC) - used to store the address of the currently executed instruction.

All arithmetic operations, comparisons and jumps are performed only on registers. Special VM instructions are needed to move data from memory to registers and vice versa.

1.3 Instructions set

We can divide instructions set into these seven categories:

- Data copy operations.
- Saving and loading data to and from memory.
- Arithmetic operations.
- Logical operations.

- Comparisons and conditional jumps.
- Unconditional jumps.
- Special instructions.

We will discuss each of those groups separately. Table 5.1 lists each supported operations. Instructions with the letter "h" at the beginning operate on floating-point registers. the target register is always specified first.

Table 1.1: List of instructions

Instruction name	Description	Arguments	Arguments size
Data copy operations			
mov, fmov	Moves data between two registers	ids of destination and source registers	2 x 8 bits
set, fset	Store value in the register	id of register, 32 or 64 bit value	1 x 8 bits + 32 or 64 bits
Load and store data from and to memory			
ld, fld, bld	Load value from the memory and store it into register	id of the destination register, id of the register in which address of value in memory is stored	2 x 8 bits
st, fst, bst	Store value from the register into memory	id of the register in which address of the destination in memory is stored, id of the register storing source value	2 x 8 bits
Arithmetical operations			
add, fadd, sub, fsub, mul, fmul, div, fdiv, mod	Add, subtract, multiply, divide and modulo two values and store result into the destination register	ids of the source and destination registers	2 x 8 bits
Logical operations			
vor, vand, vxor	Logical or, and, xor on two values and store result into the destination register	ids of the source and destination registers	2 x 8 bits
vnot	Logical not of single value	id of register which value must be negated	1 x 8 bits
vshl, vshr	Shift bits in left or right direction	id of destination register, id of register in which number of bits to shift are stored	2 x 8 bits

Comparisons and conditional jumps			
cmp, fcmp	Compare values of two registers. Sets ZF register to 1 if values are equal, and CF register if second register is greater than first	ids of registers to compare	2 x 8 bits
jz, jnz	Jump if ZF flag is set to 1 (jz) or is set to 0 (jnz)	id of register with address to jump	1 x 8 bits
jc, jnc	Jump if CF flag is set to 1 (jv) or is set to 0 (jnv)	id of register with address to jump	1 x 8 bits
jbe, ja	Jump if both ZF and CF flags are set to 1 (jbe) or CF is set to 1 and ZF is set to 0 (ja)	id of register with address to jump	1 x 8 bits
Unconditional jumps			
jmp	Unconditional jump	Address to jump	32 bits
jmpr	Unconditional jump	id of register in which address to jump is stored	1 x 8 bits
Special instructions			
cmd	Send command to the rocket	id of register in which command code is stored, id of floating point register in which value of command is stored	2 x 8 bits
halt	Stops Virtual Machine		
ftc	Fetch the rocket's orientation and physical data, and store it to memory		

1.4 Assembly language

The assembly language used by our on-board computer is a low-level language, similar in some respects to the Z80 or C64 assembly language described in [**?**] and [**?**]. Main difference is that our language supports floating point numbers and registers. The number of registers in our implementation is also much larger. The assembly syntax is very simple. Each non-blank row has the following structure:

```
(db str)|label: |(instr (label|(idreg1 [(,)idReg2|value])))

where
   a | b  means alternative
```

```
 (a)   means requirement
 [a]   means optionality

db     defines string of characters
instr  is instruction name
label  is any string of characters (excluding white spaces)
value  is any number
idReg1 is first register identifier
idReg1 is second register identifier
```

As it turns out, a simple language and a set of instructions listed above is enough to code any algorithm. This means it is Turing-Complete. Comparison statements and conditional jumps are enough to implement branches and loops of any kind.

For example, the program in listing 1.3 copies the string "Hello world" from program code memory to another address in VM memory.

Listing 1.3: Hello World assembly program

```
; store address of data in register (Hello world literal)
set r4, data

; set up registers for memory addresses
vxor r0, r0
set r1, 1
set r3, 256

print_loop:
  ; fetch byte from addres stored in r4
  bld r2, r4

  ; if zero, exit the loop
  cmp r2, r0
jz .end

; otherwise store byte in desired address in memory
bst r3, r2
add r3, r1

; move r4 on another character and go back
; to the beginning of the loop
add r4, r1
jmp print_loop

.end:
  halt
data:
  db Hello world
```

Lines starting with ";" are ignored and act as comments. Comments in Listing 1.3 describe the function of each block of code. "db" is not a instruction. This is a special directive for the translator, which means that the following string must be treated as a compact piece of memory and stored right after the translated program byte-code.

1.5 Translator to byte-code

Virtual Machine does not run assembly language code directly. Regardless of how cryptic the reading may seem, assembly language was designed to be used by a human, not a virtual machine interpreter. There are a two main reasons for this:

- Assembly language written as text is still very verbose compared to other methods of storing computer programs in memory. This is all the more important since we only have 64 kilobytes of main memory.
- The interpreter can become unnecessarily complicated if it has to translate and decode every single line of code individually.

We need to translate our assembly code into something more convenient for execution by a relatively simple interpreter. Therefore, our virtual machine has a supporting subsystem that is able to generate a stream of binary data corresponding to our assembly source code. This binary data is called byte-code.

The bytecode has a relatively simple structure: each line is translated one-to-one into the following stream of bytes:

```
opCode+argumentBytes

where
        opCode          is a numerical identifier
                        of the instruction

        argumentBytes   is a string of bytes representing
                        the arguments of the instruction

        +               is an empty space - there is no
                        byte separation between
                        nstrCode and argumentBytes
```

Each pair like above is stored next to other. It produces very compact stream: considering that each *opCode* is exactly one byte, length of *argumentBytes* is variable between 1 and 8 bytes. The interpreter knows the length of each *opCode's argumentBytes* because it uses a built-in table of instructions similar to the list we provided in the section above.

Listing 1.4 shows an example translator method. It translates an instruction class that takes a register identifier and a floating point number as arguments. Such an instruction could be, for example, fset: (fset f9, -1.27)

Listing 1.4: Translate instruction

```
void Translator::trnsl_fconstant_to_register(
std::tuple<unsigned int, unsigned int> instr, std::string line) {
  // decode opcode and arguments size
  line = trim(line);
  unsigned int opcode = std::get<0>(instr);
  unsigned int instrSize = std::get<1>(instr) + 1;

  // decode register number:
  int pos = line.find(" ") + 2;
  int pose = line.find(",");
  std::string regNumber = line.substr(pos, pose - pos);
  unsigned char reg = stoi(regNumber);

  // decode floating point constant number:
  pos = line.find(",") + 2;
  pose = line.size();
  std::string constant = line.substr(pos, pose - pos);
  double cnst = 0;
  if (std::isdigit(constant[0]) || constant[0] == '-') {
    cnst = stod(constant);
  } else {
    // label pointing to address
```

```
    cnst = labelDict[constant];
  }

  // calculate current instruction addres,
  // and copy opcode, registry and decoded number
  // into memory
  unsigned int addr = instr_addr - instrSize;
  code[addr++] = opcode;
  code[addr++] = reg;

  unsigned char* word = static_cast<unsigned char*>
  (static_cast<void*>(&cnst));

  Memory::memcopy(word, code, 0, addr, 8);
}
```

The comments in the 1.4 listing describe the functions of each section of code. The sample method in 1.4 is one of several types of decoding methods, all other types are listed below:

- trnsl_constant_to_register similar to trnsl_fconstant_to_register above but refers to integer numbers
- trnsl_register_to_register - translate instructions that operates on two registers
- trnsl_constant - translate instructions that operates only on constants or labels
- trnsl_register - translate instructions that operates only on one register
- trnsl_halt - translate halt instruction

1.6 Byte-code interpreter

The byte-code interpreter is a simple subsystem of the virtual machine, and on the other hand one of the most important. It contains the main execution loop. It sequentially fetches instructions one by one, decodes them, and executes them. The special register PC (Program Counter) indicates the memory address of the next instruction to be executed. PC is incremented on each iteration of the loop if there was no jump. Jump instructions can modify the PC register.

Thanks to the transfer of part of the work to the language translator, the interpreter may be simple and fast. Execution loop method is shown in the listing 1.5

Listing 1.5: Execution loop

```
void VMachine::executionLoop() {
  // fetching first opcode:
  threadFinished = 0;
  unsigned int opcode = memory->fetchByte(0);
  while (opcode != opcodes->getOpcode("halt") && !shouldStop) {
    while (pause); // wait

    // decode and execute instruction:
    unsigned int args_size = opcodes->getInstrSize(opcode);
    unsigned char* args = new unsigned char[args_size];
    Memory::memcopy(memory->mem, args, pc + 1, 0, args_size);
    instructions->call(opcode, args);
    delete[] args;

    pc = registers->pc();
    if (oldpc == pc) {
        // there was no jump - update program counter:
        pc += args_size + 1;
```

```
            registers->pc(pc);
        }
        oldpc = pc;

        // fetch another opcode
        opcode = memory->fetchByte(pc);
      }

      threadFinished = 1;
  }
```

Comments in the code in 1.5 describe the behaviour of the method. There is another interesting observation: conditional and unconditional jump instructions can change the program counter (PC) register. In this case, the execution loop method cannot increment PC - this is reflected in the "if" statement in the method.