National School of Computer Sciences, Tunisia

Report on

I8086SIM

A simulator for the 16 bits microprocessor 8086 of Intel®

By Koutheir Attouchi

Table of Contents

1 . Overview of the simulator

1.1 . The Intel's 8086 microprocessor

Intel® 8086 is a 16 bits microprocessor (CPU) which enabled high performance computation. The processor:

- Can address up to 1 Megabyte of memory.
- Can address memory using one of 24 addressing modes.
- Have 14 word, by 16-Bit register set with symmetric operations.
- Can do bit, byte, word, and block operations.
- Does 8 and 16-Bit signed and unsigned arithmetic in binary or decimal including multiply and divide.
- Has several clock rates: 5 MHz for 8086, 8 MHz for 8086-2, 10 MHz for 8086-1.
- Can operate in *single processor configuration* or *multiple processors configuration*.
- etc...

The 8086 processor has a rich and mature instruction set, enabling efficient low-level assembly programming and rich high-level programming. This technical report presents a simulator application for the processor called I8086SIM.

1.2 . The simulator application

Currently, I8086 is a *simulator* application for 16 bits-based programs, this means that it tries to reproduce the same execution results for a given program as it might come out when executing the program on a real 8086 CPU. It does not execute it in *real-time*, nor does it handle *all* the features of the processor. I8086 was only built for simple demonstration purposes by ***Koutheir Attouchi*** in 2009.

2 . Specifications and limitations of the simulator

This section deals with the simulator application in details, presenting its capabilities and its known limits.

2.1 . Specifications and capabilities

I8086SIM is a set of 32-bits Windows-based programs written in C++ and inline x86 Assembly using Microsoft Visual Studio 2008 Service Pack 1 (Visual C++ version 9.0). It runs on Windows 2000 and later. I8086SIM is command line driven. It can disassemble and simulate the execution of a 16 bits program in an isolated virtual machine enabling user inputs to the program and printing the outputs it produces. The program and the simulator work in two independent consoles, enabling consistent display and parallel monitoring.

The simulator currently supports the following instructions (along with their equivalent mnemonics):
`MOV, PUSH, POP, XCHG, XLAT, LEA, LDS, LES, LAHF, SAHF, PUSHF, POPF, ADD, ADC, INC, SUB, SSB, DEC, NEG, CMP, AAA, DAA, AAS, DAS, AAM, AAD, CBW, CWD, MUL, IMUL, DIV, IDIV, NOT, SHL, SAL, SHR, SAR, ROL, ROR, RCL, RCR, TEST, AND, OR, XOR, REP, CALL, RET, JMP, JZ, JNZ, JL, JLE, JG, JGE, JB, JBE, JA, JAE, JO, JNO, JS, JNS, JPO, JPE, JCXZ, LOOP, LOOPE, LOOPNE, CLC, CMC, STC, CLD, STD, CLI, STI, HLT, INT*.`

* INT instruction is partially supported, only the following interruptions are handled:

- ✔ `INT 10h/AH=02h` when `BH=0`
- ✔ `INT 10h/AH=03h` when `BH=0`
- ✔ `INT 10h/AH=0Ah` when `BH=0`
- ✔ `INT 10h/AH=0Eh`
- ✔ `INT 10h/AX=1003h` when `BX=0`
- ✔ `INT 20h`
- ✔ `INT 21h/AH=01h`
- ✔ `INT 21h/AH=02h`
- ✔ `INT 21h/AH=06h`
- ✔ `INT 21h/AH=07h`
- ✔ `INT 21h/AH=09h`
- ✔ `INT 21h/AH=0Ah`
- ✔ `INT 21h/AH=0Bh`
- ✔ `INT 21h/AH=4Ch`

2.2 . Limitations

Some of the things I8086SIM cannot do are:

- ✗ No input or output to *I/O ports*.
- ✗ The simulated processor works in *single processor configuration*.

- ✗ The simulator has an internal disassembler, meaning that it will disassemble the program only for execution ability, but not to give the user an assembly source listing. This is not a 16 bits disassembler.
- ✗ The simulator outputs the virtual machine internal state only because it might be useful. User should never try to debug a program using I8086SIM. This is not a 16 bits debugger.
- ✗ The simulator has no video memory support, and it supports only the 80x25 character mode.
- ✗ The simulator uses some x86 assembly code and the Win32 API, so it is not portable and will only run on x86 machines. It is not ready for 64 bits processors.

3 . External structure

Now, we will describe the externally visible components of the simulator. I8086SIM is divided into two files: `I8086CL.EXE` and `I8086SIM.EXE`

3.1 . `I8086CL.EXE`

`I8086CL.EXE` is the Console Locker program. It is a solution to a limitation in Windows operating system.

3.1.1 . Background on single console limitation

Windows operating system limits the number of consoles one application can have to one, but permits many applications to share the same console. Internally, Windows holds for each console a reference count number, that is a number positive or null which indicates the number of applications using a given console in a given time. When the reference count reaches zero, the console is removed.

3.1.2 . Purpose of `I8086CL.EXE`

The fact presented in the background is problematic to the simulator because it must display two consoles.

`I8086CL.EXE` is a Windows GUI program, having no attached console by default, it can attach itself to an existent console to share it. This program is designed specifically to keep a console window open by constantly keeping the reference count strictly positive. `I8086CL.EXE` attaches itself to the console (tells the system it is one of the applications that share it) and stays open, but idle, so the console will remain alive because it is being used. When the console is no more useful, `I8086CL.EXE` is terminated and the system removes the console.

3.1.3 . Inputs

`I8086CL.EXE` is a command line program having the following syntax:

```
I8086CL.EXE ParentProcessID AttachedEvent DetachEvent
```

Where:

- `ParentProcessID`: an integer which uniquely identifies a running process in the system during all the period of execution of the process. This should be the process identifier (PID) of the parent process whose console should be locked.
- `AttachedEvent`: A system event object handle. The event should be in the reset state when given. When `I8086CL.EXE` attaches itself to the console successfully, it signal the event. When the parent knows the event is signaled, it can assume the console is locked, and so continue its work.
- `DetachEvent`: A system event object handle. The event should be in the reset state when given. The parent signals the event to indicate to the `I8086CL.EXE` that it should terminate itself. When this event is signaled, the program detaches itself from the console and exits.

3.1.4 . Outputs

This program does not output anything to the attached console. But its exit code is zero if execution was successful and non-zero to indicate a failure, in that case the returned positive integer is the error number.

3.2 . `I8086SIM.EXE`

This is the simulator program itself.

3.2.1 . Purpose of `I8086SIM.EXE`

`I8086SIM.EXE` is designed to:

- Load the executable file from disk to memory, then from memory to the virtual machine, applying segment relocations if necessary.
- Analyze the program code trying to ensure it is executable.
- Disassemble the machine code.
- Execute the instruction and control the flow of execution.
- Manage input and output.
- Display the state of the virtual machine while executing.

3.2.2 . Inputs

I8086SIM.EXE accepts as command line arguments the syntax:

```
I8086SIM.EXE [-h] [-s] filename
```

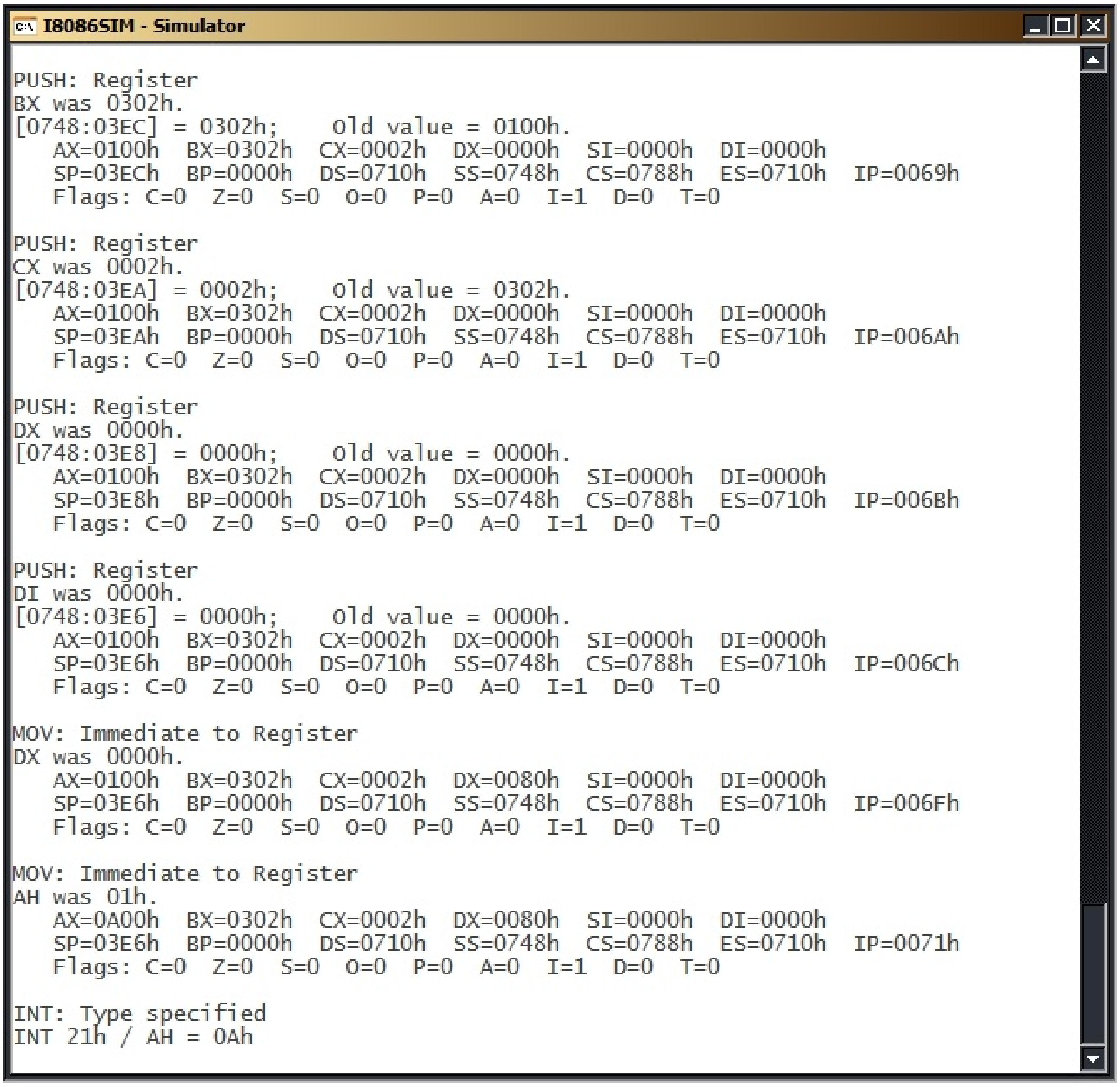
- `-H`: causes the simulator to display a usage help.
- `-S`: executes the program step by step and interrupt execution after each instruction.
- `filename`: the file name of the executable program to simulate, (executable program, not a source file). This can be a .COM file (single segment executable) or .EXE file (multiple segments executable). For consistency reasons, all other file extensions are rejected, even if the file contents is a real 16 bits program.

3.2.3 . Outputs

If I8086SIM gets the command line option `-H`, it outputs usage help. If it gets a program to execute, it starts simulating its execution by creating a new console for the simulated program for interaction with the user. The simulator console shows the internal execution state. If the execution is in the mode step by step (using the switch `-S`), the simulator screen stops after each instruction waiting for user input to continue. When execution is done, the simulator waits for user input before closing both consoles and terminating.

3.2.3.1 . The simulator screen in details

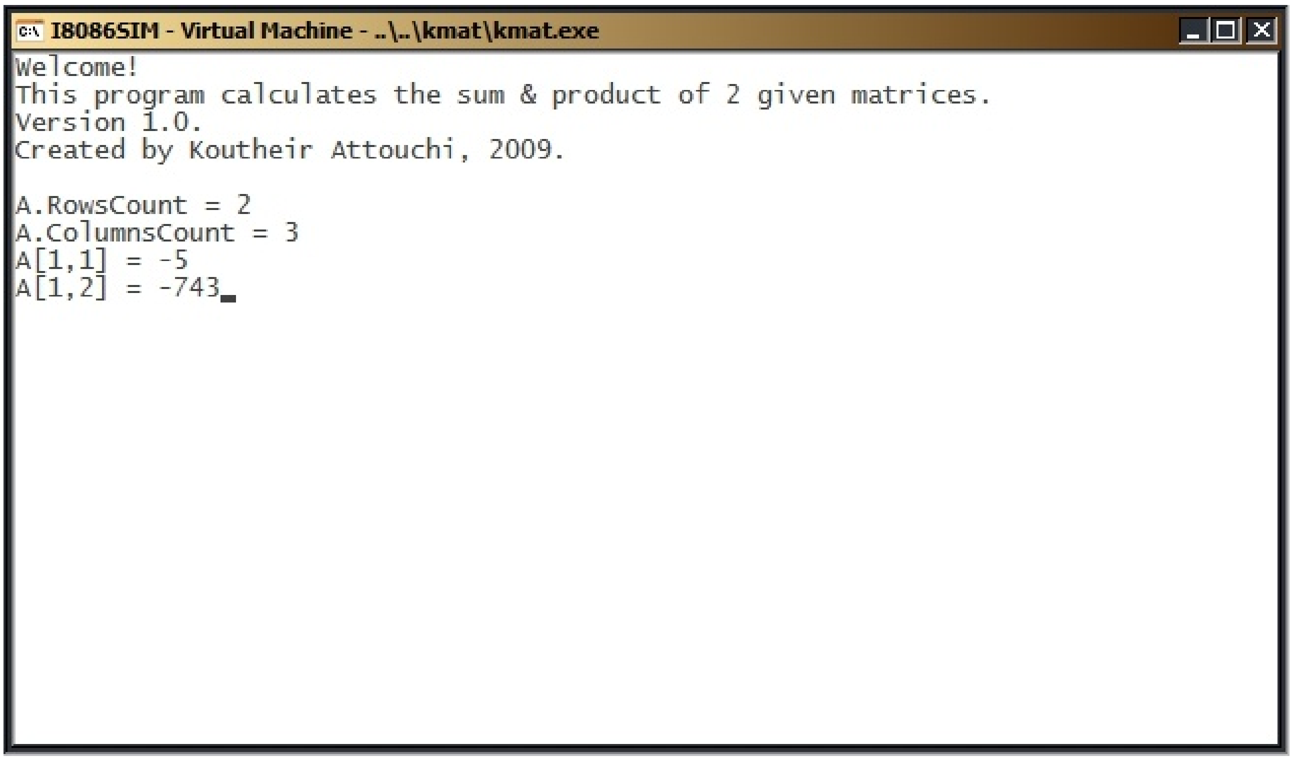
This console displays a part of the actual state of the virtual machine. It displays the registers contents, the flags, the touched memory locations, etc...



```
PUSH: Register
BX was 0302h.
[0748:03EC] = 0302h;    Old value = 0100h.
   AX=0100h  BX=0302h  CX=0002h  DX=0000h  SI=0000h  DI=0000h
   SP=03ECh  BP=0000h  DS=0710h  SS=0748h  CS=0788h  ES=0710h  IP=0069h
   Flags: C=0  Z=0  S=0  O=0  P=0  A=0  I=1  D=0  T=0

PUSH: Register
CX was 0002h.
[0748:03EA] = 0002h;    Old value = 0302h.
   AX=0100h  BX=0302h  CX=0002h  DX=0000h  SI=0000h  DI=0000h
   SP=03EAh  BP=0000h  DS=0710h  SS=0748h  CS=0788h  ES=0710h  IP=006Ah
   Flags: C=0  Z=0  S=0  O=0  P=0  A=0  I=1  D=0  T=0

PUSH: Register
DX was 0000h.
[0748:03E8] = 0000h;    Old value = 0000h.
   AX=0100h  BX=0302h  CX=0002h  DX=0000h  SI=0000h  DI=0000h
   SP=03E8h  BP=0000h  DS=0710h  SS=0748h  CS=0788h  ES=0710h  IP=006Bh
   Flags: C=0  Z=0  S=0  O=0  P=0  A=0  I=1  D=0  T=0

PUSH: Register
DI was 0000h.
[0748:03E6] = 0000h;    Old value = 0000h.
   AX=0100h  BX=0302h  CX=0002h  DX=0000h  SI=0000h  DI=0000h
   SP=03E6h  BP=0000h  DS=0710h  SS=0748h  CS=0788h  ES=0710h  IP=006Ch
   Flags: C=0  Z=0  S=0  O=0  P=0  A=0  I=1  D=0  T=0

MOV: Immediate to Register
DX was 0000h.
   AX=0100h  BX=0302h  CX=0002h  DX=0080h  SI=0000h  DI=0000h
   SP=03E6h  BP=0000h  DS=0710h  SS=0748h  CS=0788h  ES=0710h  IP=006Fh
   Flags: C=0  Z=0  S=0  O=0  P=0  A=0  I=1  D=0  T=0

MOV: Immediate to Register
AH was 01h.
   AX=0A00h  BX=0302h  CX=0002h  DX=0080h  SI=0000h  DI=0000h
   SP=03E6h  BP=0000h  DS=0710h  SS=0748h  CS=0788h  ES=0710h  IP=0071h
   Flags: C=0  Z=0  S=0  O=0  P=0  A=0  I=1  D=0  T=0

INT: Type specified
INT 21h / AH = 0Ah
```

3.2.3.2 . The program screen in details

This is the console of the simulated program. It displays its output and asks the user for input when the program wants so.



```
Welcome!
This program calculates the sum & product of 2 given matrices.
Version 1.0.
Created by Koutheir Attouchi, 2009.

A.RowsCount = 2
A.ColumnsCount = 3
A[1,1] = -5
A[1,2] = -743
```

4 . Internal structure of the simulator

In this section, we discuss the simulator from a programmer's point of view, dealing with its implementation details.

4.1 . Interprocess communication between `I8086CL.EXE` and `I8086SIM.EXE`

`I8086CL.EXE` and `I8086SIM.EXE` interact together as they execute in parallel, as follows:

- `I8086SIM.EXE` launches `I8086CL.EXE` passing it the necessary communication data and the console to lock, then it waits for it to start and lock the console.
- `I8086CL.EXE` locks the console then signal the parent to continue its execution.
- `I8086SIM.EXE` continues execution, while `I8086CL.EXE` is locking the console and waiting for termination request.
- `I8086SIM.EXE` signals that `I8086CL.EXE` should terminate itself after unlocking the console, then waits for it to terminate.
- `I8086CL.EXE` is terminated.

4.2 . Helper components

These are the helpers: the utility components that does not form the core of the simulator.

4.2.1 . `KConsoleLock` class

The `KConsoleLock` class is responsible of:

- Executing and terminating `I8086CL.EXE`.
- Managing synchronization with `I8086CL.EXE`.

It provides a simple interface enabling easy locking and unlocking of the active console.

4.2.2 . `K8086VM_COM` class

4.2.2.1 . Background on .COM files

.COM files are single segment MS-DOS executables. They contain directly the binary machine code, without any preliminary data describing the program. This makes it impossible to ensure whether a .COM file is really an MS-DOS program, and whether it is intact (not corrupt), because it has no signature or special attribute to distinguish it. This is why the specific interface of the virtual machine refuses to load a file whose extension is not .COM.

4.2.2.2 . Specific .COM files handling

This is the virtual machine interface specific for .COM files. It inherits the class `K8086VM` and overrides the way it loads files and the way it prepares their execution, to make the procedure suitable for .COM files structure.

This class must not be instantiated directly by declaring an object of its type. The only way is to have a pointer to an object of type `K8086VM_COM` which is created by calling the static function `CreateVMLoader` of the class `K8086VM`. That function returns a pointer to a `K8086VM_COM` object when it detects that the file you want to load is a .COM file.

4.2.3 . `K8086VM_EXE` class

4.2.3.1 . Background on .EXE files

.EXE files are multiple segments MS-DOS executables. An executable files has 3 parts in order:

- The MS-DOS header. Required, found once. May include the following data:
  - An MS-DOS signature.
  - Initial values of some registers.
  - Entry point of the program (main routine).
  - Checksum of the program (to ensure the file is not corrupt).
- The relocation sections. Optional, may be found multiple times, in sequence. Aligned on 16 bytes boundary.
- The executable segments (code, data,...). Aligned on 512 bytes boundary.

While this executable format has a field to ensure whether it is really an executable file, the problem remains the same as the field is documented as *optional* and many 16 bits compilers and assemblers just ignore it (or disable it by default) to speed up assembling time. So, just like the behavior with .COM files, `K8086_EXE` class will not load a file having an extension different from .EXE.

4.2.3.2 . Specific .EXE files handling

This is the virtual machine interface specific for .EXE files. It inherits the class `K8086VM` and overrides the way it loads files and the way it prepares their execution, to make the procedure suitable for .EXE files structure, such as ensuring the validity of the MS-DOS header signature, and applying segment relocations to the executable code in memory.

4.3 . Virtual machine components

These are the core components of the simulator.

4.3.1 . `K8086VMException` structure

4.3.1.1 . Purpose of exception handling

This structure enables encapsulating an error that resulted an exception to occur, along with its description. And permits displaying the description of the error when needed.

4.3.1.2 . `Structured Exception Handling` vs `C++ Exception Handling`

Structured Exception Handling is a *system-dependent* mechanism that enables applications running on Windows operating system to handle hardware and software exceptions (critical errors). For example, we can handle the exception "integer division by zero is impossible" when the program inside the virtual machine makes that operation. This way, the virtual machine does not crash due to a problem in the simulated program. Instead, it recognizes the error and reports it to the user, terminating the program as soon as possible.

C++ Exception Handling is the *system-independent* favorite way of handling exceptions in a C++ program. It enables the virtual machine to notify the execution controller about critic events, like executing an unsupported instruction.

In the I8086SIM, we use both of the presented mechanisms to provide an elegant robust way for catching errors as they occur, and to keep the simulated program isolated (the simulator is not affected by problems happening inside the simulated program).

4.3.2 . `KInstructionID` structure

4.3.2.1 . Background on machine code disassembling

The 8086 machine code has the following characteristics:

- An instruction is coded in a number of bytes ranging from 1 to 6.
- If coded on 1 byte, an instruction is uniquely identified by this byte, if coded on more than 1 byte, the first and second bytes ensure a full identification.

- A machine code byte sequence has two types of bits:
  - Fixed bits: these have always the same value, they serve as unique identifiers to the instruction.
  - Variable bits: these may have the value 0 or 1, they indicate some parameters for the instruction (example: to indicate byte or word instruction, register involved, memory involved, etc...).

This is why, considering only the fixed bits in a given machine code, the simulator can uniquely identify the instruction to be executed. Here, we define a *base code* as follows:

An instruction's base code is a sequence of bytes, each byte is formed of the fixed bits of the instruction, where variable bits are cleared (set to zero).

We check an instruction as follows:

a) Read the byte code.

b) Make a bitwise AND with a mask that will clear all its variable bits.

c) Compare the result with the corresponding base code byte of the supported instruction.

The mechanism used for a given machine code sequence is as follows:

a) The first byte of the machine code is checked against the first byte of the base code of a supported instruction. If it corresponds goto b), otherwise check the next instruction and go to a).

b) If the identified instruction is on one byte, go to c). Otherwise, the second byte of the machine code is checked against the second byte of the base code of the instruction. If it correspond go to c). Otherwise, check the next instruction and go to a).

c) Call the executer function of the identified instruction.

4.3.2.2 . Structure of the instruction identifier

This structure holds all the necessary data to identify in a unique way, a machine code for a given 16 bits instruction. It includes these fields:

- A brief description of the instruction.
- Size (exact or minimal) of the instruction: From 1 to 6.
- Bits check mechanism to identify the instruction: the validity masks.
- The procedure which can execute the instruction.

This is used when disassembling the machine code in order to execute the program.

4.3.3 . `KVMIO` class: Input/Output

This is the virtual machine input and output class. It is responsible of:

- Creating a console for the simulated program.
- Switching between virtual machine's console and simulated program's console.
- Reading and writing from and to the respective consoles.
- Repositioning the cursor on the console.

This class was created because the C++ Runtime does not support multiple consoles for the same program, so we reimplemented the functionality.

4.3.4 . `K8086VM` class: Virtual Machine

This is the virtual machine of the simulator. It is a complex class inside, but very minimal and easy to use from outside.

4.3.4.1 . Public interface elements

This is the interface used to execute an MS-DOS executable file using the simulator.

4.3.4.1.1 . Creating a virtual machine

The function `CreateVMLoader` is responsible of creating a new virtual machine to execute a given file. It tries to guess the file type (.COM or .EXE) and to give back the appropriate interface for it.

If successful, this function gives the caller a pointer to a virtual machine, which it can use to load and execute the file, before destroying the virtual machine. The caller need not to know the type of the file to load, the magic is done through polymorphism.

4.3.4.1.2 . Loading an executable file

`LoadFile` is the function you should call to load a file from disk to the virtual machine. It accepts the file name, and returns `true` if it was successful.

4.3.4.1.3 . Executing the program

`Execute` is the function you use to execute the loaded file. It gets some flags to control the execution process, and returns `true` if it was successful.

4.3.4.2 . Inner workings of the virtual machine

Here is where all the details are encapsulated. The real work is done inside these routines.

4.3.4.2.1 . Memory and Registry storage

Among the responsibilities of the class, there is the memory and registry storage. This means that the virtual machine controls the life time of these data, knowing where the registers and the 1 megabyte of memory is stored, when they are allocated, initialized and freed.

Among the functions designed for this purpose, we say:

- `K8086VM`: the constructor of the class. Where all data gets allocated and initialized.
- `~K8086VM`: the destructor of the class. Where all data should be freed.

4.3.4.2.2 . Execution control

The virtual machine must control the execution process, to be able to pause, resume, abort, etc... This is done through:

- `Execute`: Creates the console for the simulated program and loops until the end of the program to execute instructions, displaying the virtual machine state in each step.
- `ExecuteNext`: Disassemble the current instruction and call its corresponding executer routine.

- `CallExecuter`: Call the executer routine of the current instruction, guarding execution inside a SEH exception handler.

4.3.4.2.3 . Active state logging

There are functions which will display the current state of the virtual machine: the registers contents, the memory, etc... These are:

- `GetRegistersDump`
- `GetMemoryDump`

4.3.4.2.4 . Memory and Registers manipulation

The virtual machine should enable manipulating the registers and the memory directly or indirectly, this is done through:

- `ResetRegisters`: reset the contents of the registers to the default initial values.
- `GetFlagsIntoBitSet` and `SetFlagsFromBitSet`: Get or set the flags as a bit set respecting the layout of the processor.
- `GetDataAt` and `SetDataAt`: Get or set the data (byte or word) in a given location in memory.
- `GetRegister`: Get the address of the contents of a register (byte or word), enabling reading and writing from and to the register.
- `GetEffectiveAddress`: Get the address of the contents of a memory location, enabling reading and writing from and to memory.

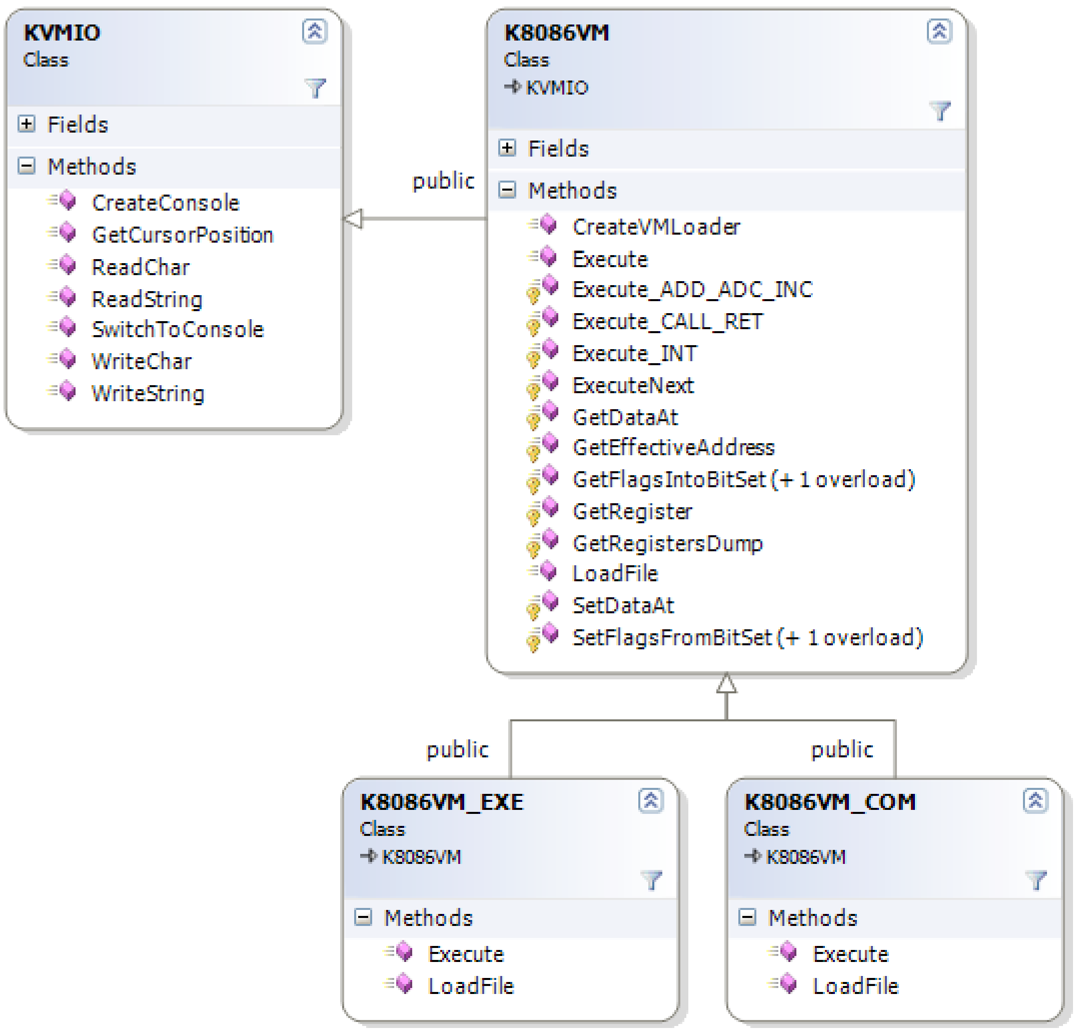
4.3.4.2.5 . Instruction execution

Here is where the instructions are fully disassembled, with all the parameters extracted. This is necessary for the execution which gets done here too.

The functions which do this all have the prefix "`Execute_`". Examples:

- `Execute_ADD_ADC_INC`
- `Execute_CALL_RET`
- `Execute_MOV`
- `Execute_NOT_SHL_SAL_SHR_SAR_ROL_ROR_RCL_RCR`

4.3.4.3 . Incomplete illustrative class diagram


KVMIO
Class
Fields
Methods
CreateConsole
GetCursorPosition
ReadChar
ReadString
SwitchToConsole
WriteChar
WriteString
public
K8086VM
Class
KVMIO
Fields
Methods
CreateVMLoader
Execute
Execute_ADD_ADC_INC
Execute_CALL_RET
Execute_INT
ExecuteNext
GetDataAt
GetEffectiveAddress
GetFlagsIntoBitSet (+ 1 overload)
GetRegister
GetRegistersDump
LoadFile
SetDataAt
SetFlagsFromBitSet (+ 1 overload)
public
public
K8086VM_EXE
Class
K8086VM
Methods
Execute
LoadFile
K8086VM_COM
Class
K8086VM
Methods
Execute
LoadFile