Study of the techniques for emulation programming

(By a bored and boring guy)

The Author: Victor Moya del Barrio

Director: Agustin Fernandez

Computer Science Engenieering – FIB UPC

18 June 2001

Contents

Index of figures.

Chapter 1. Introduction.

1. Motivation and Purpose of the Study.

I still remember the first emulator I ever tested, it was Marat Fayzullin's VGB-DOS, an early GameBoy emulator for MS-DOS, ... or it was FMSX-DOS, the MSX emulator by the same author? In any case that was my first introduction to emulation. Since that date, in the last six years, I have been using, playing and enjoying with console, home computer and arcade machine emulators. And perhaps also learning a bit, about how those machines worked, about how (good, very good) the games were and how an emulator works.

I planned many times how I could introduce myself in the world of emulation. Programming an emulator was the first choice in any case, I have been more or less a programmer for almost ten years now so it made sense. I always have enjoyed with the hardware internals and assembler programming. And although now I'm a (modest one) emulator programmer I want also to contribute in another way (because who will actually play with my boring emus?) to the emulation scene.

In the time I started to learn about how an emulator is programmed I found that there was a limited amount of documentation. Documentation about how an emulator should be programmed or about the characteristics of the machines to be emulated. Thinking about how to contribute to the emulation I found that the second part would mean the construction of a database with all the information available of a large amount of computers and systems. This wasn't very suited neither with a university project for with my own preferences, and there are a few sites in Internet which more or less serve to that purpose. The first part meant to study and learn the various techniques and knowledges related to emulator programming. Since I like a lot learning and also writing I thought that could be my way to contribute. So I thought it would be nice to have a large, ordered and precise documentation about how to program an emulator. That is the purpose of this document and my work in the last months.

After about a year of study, programming and thinking (well, only sometimes) I think now that a 'complete' document about emulation programming is almost impossible. There are too many specific techniques, too many ways to implement things and many kinds of hardware to emulate. And for each kind of hardware there are ways best suited than others. Nor to say that every programmer has it owns ideas and preferences, and they can be all correct. So I think that finally this document will be just generic, rather complete, but specific in some important points, about programming emulators. Perhaps it will not arrive to be a real reference about emulation programming but I hope it will be useful for someone, like many other documents from other people were useful for me.

Talking about motivations I would want also talk about what is the motivation behind emulation. In fact there is not a single motivation and perhaps every person has it owns particular motivations. The motivations are diverse. To remember old games or old systems from our childhood (or even our first work). To know about old games and systems which we were unable to know about or use in the old times. To use and test systems which we did not know about anything before. To preserve our memory. To use programs from other systems in our own system. To provide a protective layer over the real hardware. To learn about the internals of other computers.

All those motivations can combine to start an emulator. Many of the emulators which can be found around the web were implemented by programmers for fun. Because they wanted to use again games from those systems in their new computers and they wanted to give access to everyone (in most of cases for free) to old programs and games. The 'emuscene' has grown in the last five years and now emulators for almost all the systems which existed can be found.

2. What do we call an emulator?

The standard definition for emulation is “try to be equal or better than someone or something”. An emulator is thus someone or something who or which emulates someone or something else.

Emulation in computers is the same, to emulate the behaviour of a hardware device in software or with a different hardware, or to emulate the behaviour of a piece of software either with another hardware or software. That is still a too general definition, because you can emulate from an OS to a sound card. And the techniques used are absolutely different. Emulating a hardware device with another hardware device just has to care about to output the same values than the original hardware for the same input values. That is a task for electronic or VLSI design. But if you are emulating an OS over another OS that is a software problem.

The kind of emulators which this document talks about are software emulators of computers. A software computer emulator has to emulate all the components in a real computer using software programs over another computer. The typical, Von Neumman, computer architecture is formed by one or more CPUs which are the core of the computer and perform calculations, a bus, to which are attached devices and memory, and the memory and devices. All those components must be emulated by software. We will talk in this document about how each of those components could be emulated.

This definition is related with the concept of virtual machines (VM) like Java and others. A virtual machine is any kind of computer which does not exists as a real hardware but it is implemented by software over a real computer. A virtual machine has many uses like providing code portability or hiding specific hardware characteristics to the software (for example the OS provides a virtual machine to the user programs). And of course a virtual machine can be implemented to emulate another computer over our computer. So an emulator can be viewed as a virtual machine.

With the purpose of being a bit more specific about what kind of emulators we are going to talk in this document it is necessary to talk about our target and source computers. The target computer uses to be a standard x86 based home PC running Windows 9X or DOS. This has changed lately and emulators are being ported now to another PC systems like Macs and also to videogames consoles like DreamCast or PlayStation. But the standard system is still a PC. The target machine determines the maximum capabilities of the system in terms of calculation power, and thus the maximum hardware that will be emulated correctly. A good approach for knowing the minimum hardware that is needed for emulating a machine is multiplying the emulated machine capabilities by 10.

Our source computers can be divided in three groups: old or even new home computers, for example Sinclair Spectrum, MSX, Commodore 64, Amiga or even Macintosh or PC; videogame consoles like Sega Master System, Nintendo NES (8-bit), Super Nes, Mega Drive or even PSX, N64 and DC; and video arcade machines like the old Space Invaders, Pacman, more modern SNK Neo Geo or Capcom CPS1, to the newer Sega Model 1 and other 3D arcade machines.

All those computers share some characteristics, some more than others do of course. All of them use cheap chips from the 8-bit era (6502, Zilog Z80, Intel 8080 and 8085, Motorola 6809), or from the 16-bit era (Motorola 68000 and other versions, i8086) and the more modern ones use even middle level RISC CPU (Mips, SH, Nec, ARM). Videogames consoles, arcade machines and some of the home computers more related with videogames have powerful (in comparison with the CPU) hardware for producing graphics and sound. And they only use a few more hardware for user input. Home computers use more different hardware devices such keyboards, disk drives and communication ports.

Arcade machines use to be more powerful versions of the videogame consoles, with more memory and extended capabilities. While a videoconsole can have hundred or thousands of games an arcade machine has perhaps just ten to twenty games. The arcade machine hardware can also change a bit from one game to the another while the console remains equal. So for an arcade machine is easier to test all the games to see if they work correctly and correct problems in the emulator. In a console that is almost impossible because it would be very time expensive to test thousands of games. Therefore console emulators usually don’t support all the games.

Another characteristic of the machines to be emulated is the lack of ‘official’ information about its real architecture and the characteristics of its hardware components. In almost all cases those machines are proprietary. The information for building an emulator for those machines will come then from other sources like schematics or reverse engineering.

The performance of our emulators is very important because both videogame consoles and arcade machines are 'real time machines' (the games are time dependant) and game programmers use many times the 100% of the capacity of the system. The game must run exactly at the same speed in the emulator than in the original system. That means that if the emulator is too fast it must synchronize with the original timing, and if the emulator is slow try to reduce the calculations (for example reduce the number of frames displayed) to keep the original speed of the emulated machine. For computer emulators this problem depends of the kind of programs that the computer is running, for example a word processor is not a real time application, while a game uses to be.

Lately as PCs grown in power and GHz more modern machines are beginning to be emulated. Such computers, like Nintendo 64, DreamCast or 3D arcade machines use powerful CPUs up to 200 MHz and powerful 3D graphic chips. Those machines require more advanced techniques to be used to achieve enough performance. They require a fast and accurate translation of the 3D directives of the machine 3D hardware to the 3D PC directives (Direct3D or OpenGL). They also need from the more efficient possible CPU emulation techniques like dynamic binary translation (or dynarec in short).

3. Small History of Emulation.

It could be said that the first emulator was created when the first computer was to be replaced by a new computer which was compatible with the first one. The programs on the old computer should be ported to the new one. There are diverse ways to port those applications: if the source code was available recompile them in the new machine, but this it is not always an easy task because of the difference between the machines; to rewrite the applications in the new machine; to translate directly the application binary to the object code of the new machine, translating system calls at the same time; and the last to build an emulator of the old machine and run it in the new one.

The first emulator as we have defined them was an IBM emulator created by the 60th decade.

In next years, last 80 and early 90th with the rise of the RISC architectures it was needed to port many applications from old CISC architectures to the new RISC machines. That time binary translators (most of them in static time) were investigated and produced to perform this change.

The first emulators were designed in the early 90 for systems like the Amiga and they were more or less used. But the systems those days were not enough powerful to handle real emulation at full speed. There were C64 emulators or PC emulators for example.

Later in the middle 90 it started what now it is called the 'emuscene'. The first emulators were from old 8-bit machines but with the time and the increasing power of the PCs now there are emulators for 32 bit and 64-bit machines.

Lately emulation and dynamic binary translation has moved to the professional field with researchs for dynamic translators for the new IA64 architecture or the Crusoe's Transmeta processors.

Transmeta processors uses a layer of 'emulation' or dynamic binary translation to provide a x86 compatible service to the applications and OSes running over them. The real CPU is a VLIW with explicit ILP. A software layer translates on the fly (dynamic translation), at run-time, the x86 code into VLIW instructions. This software is called Code Morpher and it has a profile system, an interpreter, a binary translator and an optimizer.

Other commercial products which use emulation or binary translation are Ardi's Executor, a MAC emulator for x86 and FX!32, an impressive static translator of x86 NT applications for Alpha NT. Lately two commercial console emulators, both for PSX, Bleem from Bleem Co. and Virtual Game Station from Connectix. Bleem emulates the Sony PlayStation for PCs and VGS for MACs and PCs. Both companies are in legal issues with Sony. Emulating a proprietary system is a task related with reverse engineering and has more or less the same legal problems.

4. Related topics.

Emulating a computer, and in this case computers which could be named ‘multimedia computers’ is a hard task which means to have a lot of knowledge about all the aspects of the emulated computer, the target computer and the way the emulation must be performed.

The number of aspects of computing that are implied in such a program are very large. Knowledge about graphic generation in computers, both 2D and 3D. Sound generation. Computer architecture to know how a computer is internally built and all their parts work together. Knowledge about ICs and electronics to reverse engineering schematics of the system. Reverse engineering of the programs to see how the real hardware works.

An emulator is based in the theory behind Turing Machines. A Turing Machine is an automata which can read from a ‘tape’, which is its memory, and perform some operations or manipulations. Turing Machines are useful for modelling, using mathematic theory, computers and the problems which can be solved with computers. One of these problems or characteristics of a Turing Machine is the ability of every Turing Machine of ‘emulating’ the behaviour of a different Turing Machine with his own resources. Emulators are thus based on this mathematics capability of Turing Machines.

Emulation is directly related with the design of Virtual Machines, like Java. As it was said in the previous section an emulator can be viewed as a form of virtual machines which allows programs (in our case usually videogames) from an old computer run in our home PCs today. A lot of the techniques used in the design and implementation of fast and efficient virtual machines can be also applied to the implementation of an emulator, and so in the other way. The implementation of virtual machines has risen in interest since Java became popular and it began to be used commonly for applications which would run multisystem, like programs in web pages.

Java as a full virtual machine, with a virtual processor which uses bytecode and has other ‘devices’ is really near to our idea of emulators. In fact a Java VM can be easily viewed as an emulator for a ‘standard’ machine that does not have real hardware implementation (there are some designs with ICs which directly execute Java code, but those are just exceptions) but it is fully specified.

A topic which is related with virtual machines is simulation. Simulators of non-existent and existent hardware are designed and used for profiling information about how the system works internally (for example in terms of cache usage). This profiling is impossible or very hard to implement using the real hardware therefore software simulators, which could be understood as even more accurate emulators (increased accuracy emulators, for example in terms of electronic accuracy), are very used for testing purposes.

Binary translation is one of the techniques which can be used for emulating the CPU. Binary translation translates directly binary code (a program) from one CPU to binary code which can run in another CPU. This process has many resemblances with the way a compiler works, just that the source code in this case is directly machine code. Therefore, although there are important difference between translation and compilation, a lot of techniques from compiler theory can be used or modified to be used in emulation.

Since many of the computers which can be emulated are proprietary designs, and there is not free information about their characteristics, reverse engineering is mandatory. Reverse engineering in this case can come from different points of view. Reverse engineering the hardware, for example use the schematics from a computer board to discover the memory map of a computer or what kind of CPU uses. Or reverse engineering the software, to know how the programs that run in the computer access the hardware. If the system which we are going to emulate has never been emulated and the amount of information about it is small reverse engineering techniques must be used.

The kind of machines we are emulating have a strong point in graphic and sound generation, so it is needed a good knowledge of the basis of these topics. It must be known how the graphic system works in the emulated machine, for example old 2D graphic system from arcade and videoconsole used tile engines which are a lot of different from PCs graphic system. That means that those graphic systems must be emulated by software. Or if there is a hardware in the target processor which can be used for the emulation this hardware must be know, for example when translating 3D graphic directives. Same it happens with sound. The different ways sound can be produced: FM synthesis or PCM sound for example must be implemented in the hardware of the target system.

Emulation is also a computer architecture problem and even an operating system problem because the knowledge of these topics is needed to understand the emulation and implement it, and also because emulation can be applied in these field. For example it could be implemented an OS whichh would run programs from different source machines. The OS would detect what kind of program was to be run and use the properly virtual machine or emulator to run it. It also possible to think in an OS which could provide facilities for emulation, for example a direct access for the MMU could be useful for emulating the memory.

There is also a legal perspective involved with emulation. There are two aspects, if the implementation of an emulator of a copyrighted system is illegal, and if the programs for that system can be used and copied in a different system than the one they were designed for. The last legal problem is related with the fact that the media where almost all old games or programs were stored can not be used with modern computers. For example all old videogame consoles and arcade machines used ROMs for storing the data and code of the games. These ROMs must be read with an electronic device and stored in files. These files are the ones which emulators use as input programs. The copy and use of these files is in a grey legal area.

The other legal aspect is related about how the information for building an emulator is obtained, when this information is protected. Some of the older system had protected info but the time passed and the information, although still protected, could be obtained and was used for building the emulator. In this case the system is no longer in use so although there would be a legal restriction the damage to the proprietary company is low. But sometimes the information is stolen and become public when the system is still alive. An emulator using this information could not be legally sold. The other way to obtain the information is reverse engineering the system. The legal aspects about reverse engineering can be applied here. Reverse engineering has a lot of legal backup and it is being used and abused for centuries.

Chapter 2. Introduction to the process of emulation.

1. Basic Structure/Algorithm of an Emulator.

An emulator tries to duplicate the behaviour of a full computer using software programs in a different computer. An emulator thus has to be designed taking into account the internal architecture of a computer.

All modern computers are based in the Von Neumann architectures. The Von Neumann architecture is based in a bus to which the CPU, the memory and the IO devices are attached.

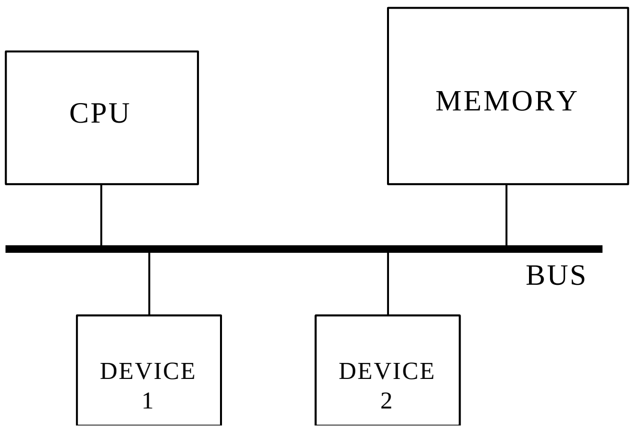


Figure 1. Von Neuman Architecture.

The CPU is the part which controls the computer, performs most of the calculations and uses to make the hard work of the system. As its name says (Control Processor Unit) it is the central part of the computer.

The buses are groups of electric lines which connect the CPU and all the other devices in the computer. In some cases more than a bus can exist. The typical bus can be divided into three parts: address bus, which carries the information about what device (or part of the device) must be accessed; the data bus which carries data from the devices to the CPU and from the CPU to the devices; and finally the control bus which carries additional signals and control information to help to arbitrate the access to the bus. The bus is shared by all the devices and because of that only one device can be accessed by the CPU at a time or two devices can share information while the CPU does not have access to the bus.

The memory is the device which serves as primary data storage. There are a lot of types of memory: RAM (memory which can be read and written), ROM (memory which can be only read) are the basic types, but there are diverse variants of those basic memory types. Here is where reside the program code and data which the CPU executes and with which performs calculations.

The IO devices (input/output) serve to various purposes and include a large number of devices. From secondary storage devices (hard disks, CD-ROMs), to graphic display devices or sound generation devices, to input devices (keyboards, mouse or gamepads) or communication hardware (network cards), and any other kind of hardware which can be controlled by the CPU.

The CPU reads instructions (opcodes) from the memory, those instructions tell the CPU to get data from the memory or from the CPU internal memory (registers) and perform calculations. Then the obtained result is written to the registers or to the memory. The CPU accesses the devices to get information from the outside world (for example the gamepad inputs from a player in a videoconsole) and uses the output devices to send information to the user (for example a melody or the image in a videogame).

Therefore the CPU is the ‘core’ of the computer. In an emulator the emulation of the CPU will be the core of the emulator too.

The emulation can be divided into parts. The emulation of the CPU is very different from the emulation of the sound or graphic system. The CPU fetches bytes from the memory and executes the functions which those bytes (opcodes) mean. A graphic device uses the video memory and commands sent by the CPU to generate an image in a TV or in a monitor screen. A sound device uses data and commands to generate an electrical signal which is fed into speakers to produce sound. The algorithms involved in graphic generation or in sound generation have nothing to do with the algorithms involved in CPU emulation.

The main part of the emulator is a loop because the way a computer works can be viewed as a repetitive task. The CPU is doing any time a fetch, decode and execute loop with the instructions. As the CPU is used as the central part of the emulation the emulation of all the other devices is instructed following the emulation of the CPU. In fact all the devices work in parallel, the CPU is executing instructions and at the same time the graphic hardware is generating the screen image and the sound hardware playing sound. But almost all emulators use to be implemented for monoprocessor machines so the devices can not be emulated in parallel.

In any case parallel models of emulators are hard to implement because of the amount of synchronization that it is needed between the different emulated devices. A threaded approach can be still worst, in a monoprocessor environment, for most of the devices because of the same reason. In any case as this document is intended for emulators which run in PC x86 systems or similar, we won’t go into the problem of the implementation of parallel or threaded emulators.

The main loop of an emulator could be something like this:

```
while (¡stop_emulation)
{
    executeCPU(cycles_to_execute);
    generateInterrupts();
    emulateGraphics();
    emulateSound();
    emulateOtherSoftware();
    timeSincronization();
}
```

Figura 2. Basic Emulator Algorithm.

The CPU is the core of the emulation and it is used to mark the time of the emulation. Many computers have hardware which introduce time into their system (for example timers and interrupts driven by timers), but the main method to know about the time in a computer is the same executed instruction time in the CPU (counting the CPU cycles). That is the way the emulator main loop takes into account the time.

We know the CPU we are emulating, we know the speed in MHz of the CPU we are emulating and we know how many cycles takes each instruction of the CPU to execute. Using the MHz we know how many cycles can execute the CPU in a time interval (for example on a second which is would

CPU_FREQ). So we can tell the CPU emulation to execute a number of CPU cycles and then stop and return to the main loop and we know how much time would have passed in the real system (or a good approximation of this time). It modern CPUs though there are a few things which has to be taken into account before, as the cache misses, the pipeline stalls and similar concepts, which can modify and make complex the calculation of the number of cycles of a given instruction.

There are periodic tasks which has to be emulated. These tasks include interrupts like keyboard interrupts, vertical retrace interrupts (related with the video display) and timer interrupts. Using the knowledge about the CPU execution time we stop the CPU emulation periodically and those conditions and interrupts are checked. They must be generated outside of the CPU emulator core. In the main loop the conditions for generating an interrupt are tested and if the conditions are satisfied an interrupt is signalled to the CPU core. When the CPU emulation is restarted, and taking into account interrupt priorities and if the interrupts are disabled, the CPU core starts the emulation of those interrupts.

There are other devices in the computer which must be emulated as well. Since this algorithm is sequential and single threaded the emulation of those devices must be multiplexed with the emulation of the CPU. When the CPU emulation stops start the emulation of the other devices: graphic hardware, sound hardware, gamepads or anything else.

The number of CPU cycles which must be executed in each turn of the block depends upon de computer which is being emulated. It is related with the frequency of the more frequent event which must be tested or triggered by the emulator. Or by the hardware which must be more accurately executed in relation with the CPU (for example if the sound hardware must rely in registers which are frequently modified by the CPU).

Other hardware is also emulated, keyboard, networks connections and so.

As we said on early sections an emulator is a real time task in the sense that it needs (most of the times) to emulate as exactly as possible the real time of the emulated computer. For example if a computer runs 2 millions of instructions and 60 frames (images) in a second the emulator should emulate the same number of instructions and frames in a second. That means either to slowdown the emulator, if the emulator runs to fast in the target machine, or to speedup it skipping some calculations (for example do not to process all the frames or reduce the audio quality), to synchronize the emulator with the expected time behaviour in the real system.

Many of our target computers for being emulated have very interesting characteristics. They are intended for entertainment, their main components are thus related with graphic and sound generation. In a typical computer or workstation the more calculation power relies in the CPU or CPUs of the machine, but in many of our target computers the most of the calculation power is in the graphic and sound hardware. That means that not only the CPU emulation must be taken into account and implemented in a fast way, but also the graphic and sound hardware. In some cases the percentage of CPU needed for emulating them is a lot of larger than for emulating the CPU.

One of the reasons for that was the fact that the old videogame systems used very low CPUs (for example the 8-bit CPUs) which could not carry many of the work. Those computers were very limited in memory and calculation power. Graphic and sound hardware was created to try to help the CPU and reduce the needed amount of memory. Tile based graphic hardware or PSG and FM sound synthesis hardware were designed with this intention. Those hardware systems made a trade off between increased calculation and additional hardware and less CPU and memory usage (and therefore less data movement between CPU and the graphic and sound hardware). Of course this fact now implies that the emulation will be more difficult because this complex hardware must be implement in software.

Another aspect to take into account with emulation is the level of accuracy that the emulator needs. The kind of emulator we are thinking about will not need to be accurate in a very low level (for example at the internal CPU workings, cache function or bus traffic) because the intention is not to profile and simulate the computer. We want to emulate the nearer possible to the real computer the external behaviour of the videogames (or programs) executed in the emulator. The intention is that the emulator will sound and look like it was the real system. That means that some the accuracy sometimes must be sacrificed in terms of the performance, because one of the most important things to emulate in an emulator is the ‘feel of time’, the emulator must run exactly at the same speed as the real system.

But the accuracy must not be so harmed that the image or sound are corrupted or becomes very different from the real system.

In the next sections of these chapter it will be introduced the topics that will be discussed in the document. First we will start with the emulation of the CPU because is the core of the emulation. There are two basic different ways of emulating the CPU, interpreting (emulating) the CPU instructions by reading opcodes and translating instructions from the emulated CPU to instructions in the target CPU. The different algorithms and techniques involved, and the different variants that these two ways of emulating the CPU have will be discussed in two different sections.

It will be discussed how much accuracy the CPU emulation needs, how it is emulated the communication between the CPU and the hardware devices, the emulation of the memory subsystem and the interrupts driven by the hardware.

Then it will begin the study of the graphic and sound hardware which are so important in the system we want to emulate. The different kinds of hardware that exists and are used will be described. Basic algorithms and explanations of the basic concepts about sound and graphic emulation will be introduced.

At last it will be introduced the concepts behind the testing of the different parts of the emulator and the full emulator. Emulators are complex programs which are really hard to test. The process is many times made at hand without any help of automatic tools. It will be discussed the introduction of such tools in the process and advices to help in the mechanisation of this task.

Another really important aspect about the emulators is how the information about the emulated system can be obtained. Either from known sources in Internet which contains information about the system (for example most of the CPU information can be found in the web pages of the corporations which make it), or using reverse engineering techniques when the information is not available.

2. The CPU emulation core.

The CPU is the core of the computer we are emulating, and the emulation of the CPU is the core of the emulator. Therefore the emulation of the CPU is one of the most important tasks. In some systems it will be the part of the emulation which will need more of the target CPU power so optimize its emulation is mandatory.

In the basic algorithm there is only one CPU emulated but some of the systems we will want to emulate have more than a CPU. In many cases there is a master CPU and one or more slave CPUs in other cases the different CPUs work more like a multiprocessor system. In terms of the basic algorithm this just implies to add more 'executeCPU(N)' functions which will interlace the emulation of the different CPUs. But in some cases there is needed additional work to add the implementation of the hardware which implements the synchronization and communication between the CPUs.

The aspects of the CPU we want to emulate are just the related with the calculations which performs and the way it operates. We only want to emulate (most of times) the execution of program instructions by the CPU. Modern CPUs are very complex internally, they can be superscalar, pipelined and out-of-order but in most of cases those aspects of the CPU should not be emulated. Only the aspects of the CPU that have an impact in the way the computer work are important.

There are two basic ways to emulate a CPU. We can get the source code (raw bytes), and in a fetch-decode-execute loop read each byte, decode what CPU instructions mean those bytes and execute the function of the decoded instruction. This kind of CPU emulation is called just emulation or interpreted emulation (in short just interpreters or emulators). This is the more basic form of CPU emulation, the faster and easier way to write an emulator, but it is also the slower in CPU time.

The other way is to get the native code and translate it into new code for the target CPU. This process is called translation or binary translation (BT). In some cases is wrongly called dynamic recompilation or dynarec in short (a discussion about the terminology would be fun but that is not the purpose of this document). The translation can be made in static or in dynamic (related with execution time), and thus there are dynamic binary translation (DBT) and static binary translation (SBT). Since the kind of work that emulators will carry is in most of time dynamic and they can run a lot of different programs the dynamic approach is the more (or just only one) used.

Now it will be introduced some concepts about the two ways of emulating the CPU.

2.1. CPU emulator: Interpreter.

An interpreter is the easier way to emulate a CPU. It just reproduces how a basic CPU works. A basic CPU reads bytes from an address of the memory pointed by a special register (PC or Program Counter). These bytes contain the information about the instruction that the CPU must execute (that is what the CPU has to do). The CPU must decode these bytes and decide what it has to do. Then it performs the action commanded, updates de PC counter and reads another byte or bytes.

That is how it works an emulator interpreter. At first it reads a byte (or the number of bytes which form an instruction in the emulated CPU). Then it decides which operation must be performed, and at last it executes the functions which correspond to that instruction. One of the easier implementations is a switch/case statement for each of the different opcodes (bytes which define an instruction).

```
switch(memory[PC++])
{
        case OPCODE1:
                opcode1();
                break;
        case OPCODE2:
                opcode2();
                break;

        ....

        case OPCODEn:
                opcodeN();
                break;
}
```

Figura 3. Interpreter emulator.

An interpreter emulator is slow compared with other forms of CPU emulation. The overhead is due to the decode and the scheduling of the instructions. One of the ways to improve the performance of an interpreter is use to assembly coded emulators. An emulator implemented in C or in another high level language has a lot of overhead because of the difficulty of implementing some basic CPU instructions. For example calculations with flag are very expensive and it is hard to use the capabilities of the target CPU at full. An assembly emulator solves this problem.

But then the emulator is not portable without rewriting the CPU core. There is a trade off between the portability of an emulator and its performance, and therefore the minimum requeriments of the target CPU. It is also important to take into account if the computer to be emulated can be easily emulated in the standard target processor and if it is really needed to implement it in assembly.

From the basic interpreter there are other implementations which try to increase its performance. Since one of the more important overheads in interpreters is the decoding of the instruction the first optimization is pointed to that. Using the property of code locality, that is, that the same code is executed more than once (loops) it would make sense to store the information about the decoded instructions. And then use this information to avoid decoding the same instruction more than once. These kinds of interpreters are called threaded emulators.

Threaded emulators are based in a form of code called threaded code. The flow of execution is pointed by an array of pointers to function. A threaded emulator just decodes the first time the instruction and stores in an array a pointer to the function or code which implement the instruction. The next time the same instruction is found the array is accessed and a direct jump to the code is performed.

Not all C compilers are capable of do such a thing and need special features or the use of inline assembly to implement it. Other compilers have been modified to permit that or implement special features which can be used: for example GCC implements labels and the goto statement.

2.2. CPU emulator: Binary Translation.

Faster than just to get each opcode from the emulated CPU and execute a function or code, which implements the function of this instruction, would be to translate that opcode to an opcode in the target CPU. That is what binary translation intends to do.

Binary translation means to get the native code for the emulated CPU and translate this code, using techniques related with compilation for example, into code for the target CPU. The translated code is stored and executed every time that is called the emulation of the CPU.

The translation is performed in blocks of code, sometimes related with the concept of basic block in compiler theory sometimes not. The reasons for translating in blocks rather than translating the full program at once are diverse. From the fact that not all the code can be known at first, to the fact that there must be points for stopping the emulation and to ease the process of translation, which is faster working in a block base.

There are two ways of translating the code: statically o dynamically. Perform the translation statically means that before start using the program in the emulator the program is fully translated to the target CPU. It works like if the program was source code from a high level language that was being compiled by the compiler of this language. The translation is performed dynamically when the code for the emulated CPU is translated as it is being executed, on-the-fly.

Static translation is not very used in emulators, the idea behind an emulator is to be dynamic rather than static. And in most of cases a static translation is not possible or hard to implement. For example to statically translate the thousands of possible programs that a computer can execute would be a problem. In other machines, like arcade systems this technique could be useful. We will just introduce the concept of static translation to see the aspects which are similar and different from dynamic translation. We will also discuss how this technique could be used with other techniques to help in the emulation.

Dynamic binary translation, sometimes wrongly called dynamic recompilation or dynarec (dynamic compilation has nothing to do with binary translation), is a very useful technique for emulating the CPU. The code is being translated in blocks as it is being executed. It is also called just-in-time compilation (the way it is called in Java nomenclature) or on demand translation. The code is only translated when it is really needed. When the code has been translated the first time is stored in a translation cache and reused any time later the same code has to be executed.

The process of translation can be slow compared with the time of just executing the code or interpreting it, but if it is only done once (or it is done before the execution, like in static translation) and the code is executed many times the final performance is increased. Translated code is much faster than just an interpreter is because the possibilities of the target CPU are better used than in an interpreter. The overhead due the decode of each instruction is only done once and the code to link the execution of each instruction with the next one (the execution loop) has nnot to be emulated.

There are a lot of topics related with the process of translation in both static and dynamic translation. The process of translation is not easy and includes the use of diverse techniques related with compilation. We will discuss topics like the use of intermediate representations in the translation process, the difference between the source and target ISA (Instruction Set Architecture), how the blocks are used for translating, register allocation, optimizations which can be performed and other things related.

But binary translation has also problems that an interpreter emulator does not have. Binary translation can not easily work with self-modifying code for example, while a standard interpreter is not affected. A static translator is not able to detect or translate self-modifying code or dynamic generation of code (dynamic compilation, for example) because this code is only created in execution time, and in a static translator the translation is performed before the execution. A dynamic translator could detect memory regions with translated code that are modified. But the overhead of the process of retranslating many times the same piece of code, and the process of detection of those modifications, could decrease a lot the performance of the emulator.

3. The Memory Emulation Subsystem.

The emulation of the memory and communication with the devices (IO or input/output) is also very important for the performance of the emulator. The memory for example is accessed very frequently, in some old architectures with a small register bank it can be accessed in each instruction. The IO ports are less accessed but as they are used for communicating with the other devices in the machine their implementation use to be slower.

The emulation of the memory and the IO with the devices is also the emulation of the bus of the computer, because it emulates the communication between the CPU and the other devices in the computer. The real behaviour of the bus usually is not needed to be emulated (arbitration, limited access and so), but sometimes it must be taken into account to for accuracy of the emulation. For example in the emulation of the Sega Mega Drive the time while the bus and the memory is doing DMA (Direct Memory Access) with the VDP (the graphic hardware) must be taken into account for counting the number of CPU cycles while it is stopped.

There are a lot of memory types, RAM and ROM are the basic types. RAM is the memory which is used for storing temporal calculations because can be read and written. ROM only contains just static data which must be used as it is. ROM contains for example program code and static graphic and sound data.

In a computer not all the memory address space is accessed in the same way. For example a region of the address space can be attached to ROM memory, another to RAM and some addresses to device registers. This mapping of the address space with the real hardware behind the address has to be emulated. It is called the memory map which is usually represented as a list of regions of memory and the kind of memory they access.

The access to memory can also have attached additional hardware to increase their capabilities. Memory Management Units of the modern CPUs, which translate virtual addresses to physical address for example. Or bankswithching hardware which provides a way to access more memory than the maximum amount of addressable memory for the architecture (used most in 8-bit processors with just 64 KB of addressable memory space).

These additional capabilities has to be emulated and be emulated very fast because the memory is very frequently accessed. We will talk about the different ways it can be emulated.

The emulation of the communication with the devices, IO emulation, is implemented with the memory emulation for two reasons, they work in a similar manner and in many computers IO is already mapped into the memory address space. A list with the IO address is provided to the CPU emulation which for each memory or IO access tests if it is a mapped device. Then it calls a function for communicating with the device.

Another aspect to take into account that is related to both memory and CPU is endiannes. The endiannes is the way the multi byte data (words) are stored in memory. There are two ways: little endian and big endian. In little endian the less significant bytes are stored in lower memory addresses and the higher bytes in higher addresses. In big endian are the inverse, lower bytes in higher addresses and higher bytes in lower addresses.

A	B	C	D

Big Endian. High order bytes are stored first.

D	C	B	A

Little Endian. Low order bytes are stored first.

Figure 4. Endiannes.

This has to be emulated either in the CPU emulator core, implementing multibyte access, or in the way the emulated memory is organized, byteswapping the memory, or with both methods. If both the emulated CPU and the target CPU use the same endianness this problem does not exist and the memory access can be performed directly. But if they use different endianess it can mean a big impact in the performance of the emulation. Therefore trying to implement the best way to perform the data conversion is very important for the performance of the full emulator. We will talk both in the CPU and memory chapters about this topic.

4. CPU and emulated devices communication. Interrupts and Timing.

We said that time in emulation is very important. Both the time that the user of the emulator can note (how fast the program or the game runs) and the internal time which controls the emulation of the different devices of the computer.

The external time is important because we want to emulate the computer exactly as it was the real computer. The games or programs must run at the same speed, it would not be good to play with a game which runs too fast or too slow.

If the emulator runs too fast then the amount of emulation performed must be controlled. We must count how many cycles executes the emulated CPU and know the number of cycles that executes the real CPU in the same time interval. Then when the emulated CPU reaches the number of cycles executed by the real CPU the emulation is stopped until the time interval is also reached.

If the emulator is too slow there is a problem because the time can not be as easily controlled in this case. A way to deal with the problem would be to reduce the amount of emulation when the time interval is reached. But this does not work for all the emulated devices, for example the CPU must execute all the instructions and in the correct time. But other devices can be emulated with less accuracy and reduce the amount of time needed. That is usually done with the graphic emulation.

The graphic emulation uses to emulate a fixed number of frames (images) each second (fps) but this number uses to be greater that the amount of fps the human eye can note. A way to reduce the calculations is reduce the number of fps. If the reduction is too big the animation becomes very bad and jerky, but, if it is just a small reduction, time could be saved for the emulation of other devices while preserving most of the feeling of the real machine. That is how emulators use to work.

Something similar could be done with sound but this is more difficult to be done and it is easy to be discovered. In any case these kind of time control is limited. If the emulator is too slow (the target machine is not able to emulate properly the emulated machine) there is no way to provide a real timing.

The internal time is important because in the real computer the CPU and the other devices are working in parallel while our emulators work sequentially emulating each device in turns. There could be problems of synchronization between the CPU and the graphic and sound hardware if the CPU emulation executes too many instructions before emulating another step the graphics and sound.

For example the CPU could modify two times the same register of the graphic hardware because the programmer knew that the graphic hardware was already displaying a frame. The modification would modify a limited number of lines of the image. But if the CPU emulation executes the two modifications at the same time slot and then it starts to emulate the graphic hardware, it will not notice that there were two changes. The output image will not be the same than in the real system. There are various solutions for this problem: store buffers with the modifications of hardware registers and perhaps a time-stamp or to reduce the time slice for the CPU. Reducing the time slice for example increases the overhead due to the restart of the CPU emulation. We will talk about all those things.

The internal timing can be controlled with the time slice for the CPU emulation, that is, the number of cycles it must emulate before stoping for emulating the other hardware. Also how often the emulation of the other devices is updated, for example the graphic hardware could be emulated after each frame or after each line.

Hardware interrupts are many times related with timing. Most of the computers we are going to emulate use interrupts for timing, for example is very usual a vertical retrace or vsync (video synchronization signal) interrupt. This interrupt is generated by the video hardware at the end of each frame. Those interrupts are thus very important for the correct emulation of the computer because must be generated in the correct moment for good emulation accuracy.

There are other hardware interrupts in our emulated machines which have to be emulated. The hardware interrupts are a different mechanism from IO for the communication between the CPU and the devices. Rather than the CPU reading and writing IO registers the hardware devices send a signal to the CPU telling that something has happen (it has finished a task or an error has occurred). Then the CPU stops what it was doing and executes an interrupt routine which check the state of the device and sends new orders to the device. The first form of communication is called polling (the CPU keeps asking for the state until it changes) and the last interrupt driven.

The correct emulation of the interrupts is needed for accuracy and a good emulation. Many times the emulated program is stopped in a loop waiting an event which is triggered by a hardware interrupt so it is important that the interrupt would happen in the correct time.

Time emulation and interrupts are very related with CPU emulation and the mechanisms for its emulation are implemented both inside and outside the CPU core. For example the core must have an interface for signalling interrupts and an internal interrupt priority policy. But the interrupts are triggered from outside. The CPU must also count the number executed cycles but the number of cycles to execute and the time synchronization is decided outside.

The emulation of the timing and interrupts with an interpreter emulator and with binary translation is someway different and we will talk about that.

Exceptions are also another way of interrupting the CPU but those are internal to CPU rather than external (sometimes they are from hardware inside the CPU chip like the MMU) and their emulation is more related with CPU emulation but we will talk about them too.

5. Graphic Hardware Emulation.

Graphics is the main form that the computer can communicate with the user. And in computers which are used for playing videogames that is even more important. In these computers the graphic hardware is very important and uses to be many times more powerful than the CPU. It provides facilities to help the CPU to perform the animation and reduce the communication and the CPU calculations.

Therefore the graphic hardware is hard to emulate. It has to be also emulated very fast because it is also used very frequently and it must perform a lot of work.

There are several kinds of graphic hardware, from the easiest hardware implementation of a video device: a bitmap with a bit, a byte or bytes for each pixel in the screen to 3D hardware devices which perform complex floating point calculations, texture management and so.

We will just introduce about the principles of the 3D hardware, how it works and how it could be emulated in a PC or similar systems. The 2D hardware will be explained largely, mainly the tile and sprite based engines which are the most used in old videoconsoles, arcade machines and some home computers. The PC graphic hardware will be also explained because it is the hardware we will use for emulating the other hardware and it is important to know about it.

We will introduce basic algorithms for emulating tile graphics and some of the effects that the hardware can include. Also we will talk about some effects which can be added to enhance the original graphics in the video modes with more resolutions than the original. Basic concepts about graphics will be also introduced.

Basically the old computers used three different kind of 2D hardware, plain bitmaps in the old arcades, tile/sprite based engines (and combination of both) and vector graphics (something similar to modern 3D hardware but just with lines). The more important are tile/sprite engines and will be explainer further and with some particular examples. Tile/sprite engines provided an easy way to reduce the amount of communication between the main memory and the video memory because it is based in reusing the same graphics for many images. The image is also formed with a character table which can be easily modified.

6. Sound Hardware Emulation.

In many aspects sound and graphic hardware emulation are very related. Of course the algorithms implied in the emulation are different (is very different the way a sound is represented or produced from the way an image is), but both graphic and sound hardware are very important in the computers we want to emulate. Both have different types of hardware which try to reduce the calculations in the CPU computer and reduce the communication between the CPU and the hardware.

There are even more different types or ways of producing sound that in the case of the graphic hardware. The more important types of hardware related with the machines we want to emulate are: PSG and FM, MIDI and Wavetable sound generation, sample based systems (PCM, ADPCM) and DSPs.

PSG (Programmable Sound Generator) and FM (Frequency Modulation) produce tones with more or less complexity but the amount of information they need (the bandwidth with the memory or the CPU) is very low. A PSG just generates clean tones, they implement volume changes and some implement simple envelope forms. FM tries to emulate someway-musical instruments and it has more options than a PSG. They work combining multiple sinus waves to produce harmonics. They can modify the more important aspect of the tone and the envelope. The sound does not sound as a real instrument (even more with percussion instruments) but it is a lot of better than a PSG. The PSGs are used in the early 8-bit era systems, FM was used in the 16-bit era systems.

MIDI and Wavetable based hardware works a bit similar but produces sound which are more nearer to real music instruments, but they use to be more powerful hardware and are not used so frequently in the systems we want to emulate.

Sample based hardware just outputs digital sound to a DAC (Digital to Analog Converter) to produce sound. They need a lot of more bandwidth and memory because with the sample rates needed for a good sound quality the amount of data for each second of sound is very large (for example, 44,1KHz 16-bit samples and stereo –two channels-, the standard CD quality, a second means 172KB of data). The basic hardware just outputs raw digital data to a DAC without any change, the more advanced hardware uses diverse forms of compressed samples (to reduce the amount of memory needed), mixes multiple samples by hardware and adds envelope and other sound effects. The more advanced sound hardware uses DSP (Digital Signal/Sound Processor) to generate the output sound. This kind of hardware is based in interrupts and timers. The sample is placed in a buffer and when it is signalled to start the chip starts to send the data to the DAC, when it reaches the end of the buffer either generates an interrupt or loops to the start of the buffer.

As important as the hardware we have to emulate is the hardware we can use to emulate it. In PCs the basic hardware is sample-based hardware, but most of the sound cards have FM and PSG capabilities (the Sound Blaster 16 standard uses a YM2612 FM Sound Processor). Some also have MIDI and Wavetable capabilities. In MS Windows is difficult to use the FM and in most of cases does not produce the correct sound so the standard way is to emulate the sound producing samples, mixing them and sending to the sound hardware.

Sound uses to be the last option to be emulated in many emulators because is the less important for using the games or programs. A game can not be played if there is no image, but can be played (of course with a loss in the game quality) without sound. In many case is the last thing emulated, decreases a lot the emulator performance (if the emulation is accurate and the hardware complex) or it does not sound correctly. Perhaps it is one of the hardest things to emulate.

We will introduce the basic concepts about sound programming and PSG and FM sound generation will be explained and how they can be implemented using sample based hardware. Some aspect about sample based hardware will be also explained and finally some comments about other kinds of sound hardware.

7. Other devices.

The graphic and sound hardware with the CPU is not all the hardware we have to emulate. We have also talked about the memory emulation and the interrupts. But there are other hardware devices which have to be emulated. Videoconsoles and arcade machines have gamepads, joysticks, buttons, dips (for setting the arcade options), and other input devices (or even output). Home computers have a larger list of additional hardware: disk drivers, tapes, communication ports, mouse, and others.

And it is not just the hardware a common user could know about (input devices, output, communication, and secondary storage) but also small chips which perform minor or major tasks in the computer systems. For example chips for timers, chips which control interrupt priorities, chips which encrypt and decrypt data, chips for DMA (Direct Memory Access) and similar stuff.

All those devices or chips must be emulated in most of case so the emulator could work properly. So we will introduce some of the most common ones and we will show how they can be emulated. The main ones will be input devices, mainly keyboards and gamepads, a bit about secondary storage units, and a bit about other devices which are hidden but are important like DMA control chips.

8. Testing the emulator.

At last what we want is a working emulator that works as similar to the original machine and fast as possible (so it would need the less powerful machine possible). But to accomplish that a lot of code must be written and the most important tested.

Testing is very important in emulation and very hard too. For example to properly test a CPU emulator it would mean to generate all the possible instructions which can receive the CPU and compare the result in the emulator with the real result. It will be also needed to test combinations of instructions because of side effects (flags for example) of the instructions. The interrupt system, the exceptions, the access to memory or to the other devices, if the cycle count is being done correctly, everything should be tested and work well. Something similar happens with the sound and graphic hardware and all the other devices.

As it can be seen is a hard task, or in some cases an impossible task. There are additional difficulties, for example many CPUs have undocumented (more likely non-official) features that sometimes were used by the programmers so they must be also emulated (and discovered). Sound is very hard to properly test because you would need a spectre analyzer (and perhaps also a sound expert for the better accuracy). Graphics are easier because you can easily see that there is something wrong. The hard part is to discover what is the error.

The different types of implementation of an emulator needs different approaches with testing and have different problems. For example binary translation is harder to test and debug than just an interpreter.

The usual techniques for testing computer programs include the black box approach, the white box, the joint approach and a lot of others. In this chapters we will just introduce some advises and easy techniques to help deal with the testing of some of the parts of the emulation. For example the CPU.

More commonly the task of testing and debugging an emulator means to test many programs or games for the emulator and see if or how they work. Then debug them using debuggers included in the emulator and disassemblers. It is a task of reverse engineering the program and the emulated system. In fact as many times there is a lack of information about the emulated system the task of testing and debugging the emulator becomes the task of reverse engineering the system.

We will just introduce those topics but they would need further talk because is the hardest task involved with emulation and mean a lot of hours of patient work.

9. Searching for information and other aspects of emulation.

Write the code of the emulator is not the only task which is important to build an emulator. As we said in the previous section the debugging and testing are important. Know about how to reverse engineering the system or a program can be necessary to know how it works a computer when there is not enough information about it.

So there are other topics that would be interesting to discuss or just introduce for completeness. In this section we will see some of those topics.

For example, it is also important to obtain information about the system that we want to emulate. The CPU, the sound and graphic hardware, the other chips which need to be emulated. The information about the CPU uses to be easy to be found. For most of the proprietary systems information about the other hardware (sound, graphics) is hard to obtain. If the system is old is easier because a lot of people would have worked with the system and would have written documents about it. Or even someone would have programmed an emulator. For newer computers the only way is to try to get development kits (but they use to be expensive) or find some info in programming sites for the systems. For example demo coders are a good source of information about new systems.

Another useful thing to know about is to read schematics. Schematics are the diagrams that represent components of the computer system and are useful to find what chips it uses the system and how are them connected.

There are also legal aspects related with emulation, but this is not a legal text so we will just introduce the basics.

And finally it will be discussed how, why and if an emulator could be a commercial product and do not just a product of the freeware scene. Or in marketing terms: how could I sell an emulator?.

Chapter 3. CPU emulation: Interpreters.

1. Basic interpreter CPU emulator.

We will introduce first how a basic interpreter emulator is implemented. The basic CPU emulator is based in a fetch-decode loop and, in this case, we will consider that it is implemented using a high level language. In other sections it will be told about CPU emulators implemented in assembly or which use other techniques more advanced for increase the performance of the emulation (threaded emulators).

An interpreter CPU emulator works in a similar manner than a real CPU (a basic one), and also similar to how an interpreted language is executed. It gets the code of the program and it decodes it into instructions to be executed, then it calls a function which implements those instructions.

Most of the CPU cores are implemented in C although lately there are implementations in Java, C++ or even Visual Basic (although some of those languages are not well suited for CPU emulation). The performance of a CPU emulator is limited by the difficulty of translating the internal behaviour of a CPU (even more if the CPU is old) with high level instructions. For example side effects of the instructions like flags are hard to calculate in a high level language.

We will begin explaining the basic fetch-decode loop which is the main loop of the CPU emulator, and the different ways it can be implemented. Later we will introduce the emulation of the CPU instructions and some of the harder aspects to implement in a high level language (flags for example). The next aspects to talk about will be the interaction with the memory emulation and the interrupt emulation. At last we will see a standard interface CPU emulators.

In Appendix A we can found an extended explanation about how a basic CPU emulator for a basic CPU can be implemented. Appendix A is a tutorial about a Space Invaders emulator. The Space Invader machine uses an Intel 8080 CPU. The tutorial explains the implementation of the different parts of the i8080 emulator core using C.

The CPU status.

A CPU must keep information between the execution of an instruction and the next. The minimum that is needed is a register pointing to the next instruction to be executed. This register is the PC or Program Counter (some CPUs could change the name, for example in x86 is called IP).

The more basic CPUs (and also Virtual Machines like Java), have at least two registers, the PC and the SP. The SP, or Stack Pointer, is a pointer to the memory. It is used to keep a stack data structure, that is, a FIFO (First In First Out) structure which is useful for retrieving the last data added to the structure. The SP is decremented and incremented as new values are pushed or popped to the stack. Those CPUs, called stack machines, perform the operations with the top values in the stack and put the result in the top of the stack.

The common CPU has more registers. CISC kind CPUs use to have only a few registers which many times are specialized for some task. There could be general purpose registers, the accumulator register (where the result of all operations is stored), address registers and other types of registers. Examples of such CPUs are the i8080, the 6502, and the other 8bit and 16bit CPUs.

RISC CPUs use to have large register banks. If the RISC CPU implements floating point operations there will be two different registers banks, one for the integer calculations (GPR or General Purpose Registers) and one for the float point calculations (FPR of Floating Point Registers). CISC CPUs with float point capabilities also use different registers for storing integers and floats.

There are also other register which hold information about the state of the CPU, for example flags (modified by the result of the operations), the state of the interrupts, exceptions, CPU configurations and others.

The full state of a CPU must also be maintained in a CPU emulator. We need a structure, the CPU context, which will store the full state of the emulated CPU. This structure will be modified by the emulated instructions. The usual implementation is a C struct (or record in Pascal) which will have fields for all the aspects of the CPU state. It will have, for example, an array of integers for the GPR bank, an array of floats for the FPR bank and booleans for the flags and the CPU status bits. It can also have other fields not directly related with the CPU, for example a cycle counter to store the number of executed cycles since the last execute() call. This kind of data is using for profiling and control purpose inside the emulator.

```
typedef union
{
        UINT32 w;       /*  Access it as a double 16-bit register  */
        UINT16 w;
        struct
        {
                UINT8 l,h;      /*  Low (l) and High (h) bytes of the register  */
                UINT16 pad;             /*  Little-Endian padding  */
        } b;            /*  Accessed as two single byte registers  */
} i8080Reg;

struct i8080Context
{
        i8080Reg AF;            /*  Register AF (Accumulator + Flags)  */
        i8080Reg BC;            /*  Register BC (B + C)  */
        i8080Reg DE;            /*  Register DE (D + E)  */
        i8080Reg HL;            /*  Register HL (H + L)  */

        UINT16 PC;              /*  Program Counter  */
        UINT16 SP;              /*  Stack Pointer  */
        UINT32 flagC;           /*  Carry flag  */
        UINT32 flagZPS;         /*  Zero, Parity and Negative flags  */
        UINT32 flagAc;          /*  Auxiliary Carry flag (4-bit carry)  */

        UINT32 halted;          /*  CPU is in Halt state.  */
        UINT32 intEnabled;      /*  Interrupts are enabled or disabled  */
        UINT32 intPending;      /*  Stores if there is a pending interrupt */
        UINT16 intAddress;      /*  Stores the start address for the interrupt handling rutine  */
        UINT16 NMIAddress;      /*  Stores the start address for the NMI rutine  */
        UINT32 NMIPending;      /*  Stores if there is a pending nmi  */

        UINT8 *mainMemory;      /*  Pointer to the main memory  */

        struct readMemoryHandler *readMemoryHandler;    /*  List of memory handlers for reading  */
        struct writeMemoryHandler *writeMemoryHandler; /*  List of memory handlers for writing  */
        struct readIOHandler *readIOHandler;            /*  List of IO handlers for reading  */
        struct writeIOHandler *writeIOHandler;          /*  List of IO handlers for writing  */

        UINT32 numMemoryBanks;          /*  Number of banks of memory  (0 - 64) */
        void *pBankList;        /*  List of pointers to the bank memory regions  */

        UINT32 cycleCount;      /*  Number of executed cycles */
        UINT32 opcodeStatistics[256];   /*  Number of times an opcode has been executed  */
        UINT32 tracePC[256];    /*  Stores the last 256 PCs  */
        UINT32 traceIndex;              /*  Last entry in the PC trace buffer  */
};
```

Figure 5. CPU context structure (i8080 emulator).

There are some difficulties which must be taken into account. For example in some architectures the same register can used for performing calculations in different data sizes. In x86 it has 32 bit registers that can be used as 32-bit registers, as 16-bit register or as two 8-bit subregisters (EAX, AX and AH/AL). Therefore we will have to take into account that the data type for a register should admit fast accesses to diverse data sizes.

A good solution using C is to implement registers as unions, each field in the union will be associated with one of the register uses. In C it can be also used its capability with pointer to do the same work (it is easy to access any data as raw bytes). Other high level languages could not admit those facilities. In any case the idea is to implement a fast access to the more common register use. The other register uses would have slower access but the performance lost would be less. What has to be avoided is to have to convert the register from one format (use) to another very frequently because the performance lost would be too big.

```
typedef union
{
        UINT32 w;                    /*  Access it as a double 16-bit register  */
        UINT16 w;
        struct
        {
                UINT8 l,h;           /*  Low (l) and High (h) bytes of the register  */
                UINT16 pad;          /*  Little-Endian padding  */
        } b;                         /*  Accessed as two single byte registers  */
} i8080Reg;
```

Figure 6. Multi size registers emulation using C union data type.

```
long reg;

...
((char *) &reg)[0] = 0x01;
...
((int *) &reg)[i] = 0x0212;
...
reg = 0x1234afed;
...
```

Figure 7. Multi size registers emulation using pointers and type cast.

The flags and the other data stored in the CPU state registers usually can be better implemented as standalone fields in the CPU context. They use to be accessed separately rather than all of them at a time (for example a conditional branch checks one or two flags and not the full status word). The CPU state fields, for example one bit that would enable or disable interrupts, are rarely accessed reading the full status word. When the CPU status word is accessed as a full (for example when it is saved or restored from the stack) a conversion (expansion or compression must be performed). It this kind of access is too frequent it would be better to change the implementation the inverse one (all flags and status fields in a single CPU context field).

S	Z	X	Ac	X	P	X	C

i8080 PSW (F)

a) Flags stored as a single state word

```
i8080Reg AF;            /*  Register AF (Accumulator + Flags)  */
```

b) Flags stored as different fields

```
UINT32 flagC;           /*  Carry flag  */
UINT32 flagZPS;         /*  Zero, Parity and Negative flags  */
UINT32 flagAc;          /*  Auxiliary Carry flag (4-bit carry)  */
```

Figure 8. Examples of CPU flags.

The CPU context needs additional data that is needed by the CPU emulator to work. Some of the CPU state is implicit but the emulator must store it, for example the status of the IRQs (hardware interrupts), if the CPU is halted, if the CPU has lost the access to the bus. The emulator also needs information about how to emulate the memory (the memory map). We will talk about the memory map in the memory emulation section.

Finally the CPU context can store diverse statistic data. One data which is almost mandatory is a cycle counter. We need to count how many cycles have been executed since the last call to execute() or since the emulator was started (or reseted). The other data is related with the profiling of the code. The CPU emulator could be used for gathering information about the executed code for simulation purpose or for a second pass binary translator. A good place to store all the profiled data is the CPU context.

Finally, many of the CPU cores can be used to execute of multiple instances of a real CPUs with the same code, for example if the emulated machines uses 2 CPUs of the same type (multiprocessor). In a sequential single-threaded environment this just implies that the CPU context must be saved and restored when the emulation of one CPU ends and the emulation of the other starts. In a parallel or multi-threaded environment it is also needed that the CPU emulator code is reentrants. The emulator should access to different CPU context structures for each emulated CPU (through pointers for example), and all the sensible data would be stored in the context.

The fetch-decode loop.

The fetch-decode loop is the main loop of the CPU emulator. A real CPU works in this way: it gets a byte or some bytes from the memory which are located in a position pointed by a special register (commonly called PC or Program Counter). Then these bytes are used to decide which instruction must execute the CPU. And when it has decided what it has to do it executes the function and reads another byte or group of bytes.

The byte or group of bytes which define a single instruction in a CPU are usually opcode or operation code. Some times the term opcode is used for all the bytes which form the instruction, including offsets, extended fields and other literal data which is passed inlined to the instruction. In other cases the opcode only means the part which really defines the instruction to be executed.

a) i8080 instruction (1 byte).

1000 1XXX	ADD REG
27	DAA
CC PP QQ	CZ LABEL

b) Z80 instruction (extended opcode).

ED 43 PP QQ	LD (addr), A
DD 0111 0sss YY	LD (IX + disp), reg
FD 0111 0sss YY	LD (IY + disp), reg

c) m68000 instruction.

5 bbb1 0 0ddd	SUBQ.B data3, Dn
5 bbb1 00ff ffff	SUBQ.B data3, dadr

d) MIPS instruction.

SPECIAL 00000	rs	rt	rd	00000	ADD 100000
6	5	5	5	5	6

ADD rd, rs, rt

ADDI 00100	rs	rt	Immediate
6	5	5	16

ADDI rt, rs, immediate

Figure 9. Opcode examples.

As we have said when a CPU executes an instruction it passes a number of phases. Those phases are usually called fetch phase, decode phase and execution phase (basically, in a more complicated CPU there are more phases and other different phases). In the fetch phase the CPU gets the code data from the memory pointed by the special register PC and stores that data, the opcode, in an internal register. Fetching could be considered as reading code, rather than read, which would be reading data. In the decode phase the CPU uses fetched data to decide which actions (which instruction) must be executed and sends the proper signals to the functional units. In the execution phase the CPU executes the actions that must be done for the opcode read. Our CPU emulator will work in a similar manner.

The fetch-decode loop is the part of the CPU emulator which implements the CPU fetch and decode phases. The fetch is implemented just reading from the emulated memory at the position pointed by the emulated PC a given number of bytes. Then those bytes, which we will call the opcode, will be decoded. Our goal is to perform the decoding as fast as possible because that is a task which is very frequent (any instruction must be decoded!). Decoding means to decide which function must be executed for the given opcode.

```
while (executed_cycles < cycles_to_execute)
{
      opcode = memory[PC++];
      instruction = decode(opcode);
      execute(instruction);
}
```

Figure 10. CPU core main loop.

The number of bytes to read on each fetch depends upon the processor. Some CPUs have fixed length opcodes and then the size in bytes of an opcode is always the same, that eases the task of fetching because you already know how many must be read for each instruction. This uses to happen with RISC CPUs. Other CPUs, mostly old CISC ones, have variable length opcodes. In this case the size in bytes of each opcode depends upon the type of each instruction, some will need more and other less bytes.

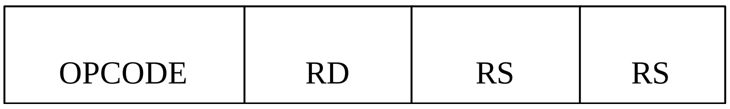
That means that the fetching must be done in two or more steps. At first it is read the number of bytes needed to decode a simple instruction or too differentiate between the different kind of instructions. Those instructions which need more bytes to be fully decoded or executed will perform more memory reads from the address in the PC. The additional fetching is implemented in the same functions which emulate those extended size instructions.

Each time a byte is fetched from the memory (code is read) the PC (which points to the next byte of code) must be updated. At the first step of the fetching, in the main loop, the PC is incremented in a fixed amount (the basic opcode size). The instruction implementation (in CPUs with variable length instructions) must update itself the PC if it performs more code reads. The PC is also affected by control flow instructions (jump, branch, call and return instructions) and by hardware interrupts and CPU exceptions.

The decoding can be performed in different ways. It depends upon how the instruction is encoded in the opcode for the emulated CPU and also in the capabilities which of the language we are using. RISC kind CPUs for example can be more easily decoded because they have fixed opcode formats which different fields which define the different kind of instructions, the registers to use, literals and so. They also, as it has been said, use a fixed in length opcode format.

CISC instructions in the other way do not use to have standarized fields in the opcodes for each information. And they have instructions in various sizes, extensions of the normal opcodes, prefixes and other weird stuff.

On RISC CPUs or in any CPU where the different bits in the opcode can be grouped to form different information (opcode, registers, literals, special opcodes) the first thing to do could be to differentiate (using macros, or copying them to different variables) the different fields in the opcode. The use the field which determines the instruction type to determine which instruction to execute. The other fields could be used for further decoding (if that type of instruction has different final instructions) or to be used for the instruction implementation (where to get the data, where to store the data, data sizes, etc).



```
       OP RD, RS, RT

  RISC kind instruction.

/* fetch */
opcode = fetch(PC);

/* first decode */
opcfield = OPCODE(opcode);
destreg = DESTREG(opcode);
sreg = SREG(opcode);
treg = TREG(opcode);

/* last decode, execute */
execute(opcfield, destreg, sreg, treg);
```

Figure 11. RISC instruction decoding (fixed lenght).

On CISC machines where it is hard to find the different fields or there is not a standardized encoding format for all the instructions the decoding uses to be done with all the read opcode.

```
/* fetch */
opcode = fetch(PC);

/* decode and execute */
execute(opcode);
```

Figure 12. CISC instruction decoding (variable lenght).

The process of determining which function or code must be executed to emulate each instruction can be implemented in different manners. The first we could propose is an array of if statements, but those are very inefficient because the last opcode will take N (N number of opcodes) times the time of decoding of the first one. The better implementation is to use indexed or indirect jumps.

```
if (opcode == OPC1)
{
        /*  emulate the opcode 1  */
}
else if (opcode == OPC2)
{
        /*  emulate the opcode 2  */
}
...
else if (opcode == OPCi)
{
        /*  emulate the opcode i  */
        if (subopc == OPCiSUBOP1)
        {
                /*  emulate subopcode 1 of opcode i  */
        }
        ....
        else if (subopc == OPCiSUBOPN)
        {
                /*  emulate subopcode N of opcode i  */
        }
}
...
else if (opcode == OPCN)
{
        /*  emulate the opcode N  */

}
else
{
        /*   Wrong opcode.  Illegal instruction.  */
}
```

Mean number of condition tests for decoding an instruction: N/2.

Figure 13. Decoding using conditional (if) statements.

In a high level language an indirect jumps is not direct to implement. The first approach is to use a switch statement, with a case for each of the possible opcode values and hope that the compiler is intelligent enough to translate it into an indirect jump using a jump table, rather than an array of ifs.

```
switch (opcode)
{
        case OPC1:
                /*  emulate opcode 1  */
                break;
        case OPC2:
                /*  emulate opcode 2  */

                if  (subopc == OPC2SUBOPC1)
                {
                }
                ...

                break;
        case OPC3:
                /*  emulate opcode 3  */
                break;
        ...
        case OPCN:
                /*  emulate opcode N  */
                break;

        default:
                /*  Wrong opcode.  Illegal instruction.  */
                break;
}
```

The decoding is performed (good compilers) through an access to a jump table. Just one memory access (tests) needed.

Figure 14. Decoding using swith-case statement.

If we know the compiler is not making its work correctly the next step would be to implement the jump table at hand as an array or table of functions. The table is indexed by the opcode value and each entry has a pointer to a function. The pointer is retrieved and it is performed a function call.

Yet another option could be to use labels and the goto directive in those language which permit them (for example some C implementations like GNU C).

The easier implementation is to use a switch-case statement, but perhaps it is the slower due to problems with the code generated by the compiler. The function table could be faster but it could suffer from procedure call overhead. The label implementation could hurt the optimizations performed by the compiler (the program is jumping to any place). Those problems and advantages should be taken into account when implementing the process of decoding.

Many times, although the decoding could be implemented in different steps, for example if there is an opcode field and a function opcode, it is better to perform the decoding in a single step. The CPU emulation must be as fast as possible so we could trade memory space for time and use large function tables or switch statements and a lot of functions for each possible opcode. For example in a CPU with 16 bit opcodes, for example de Motorola 68K, the best option could be to use the full 16 bit opcode as the function table index. That would mean 64K table entries and some hundred or thousand of functions. But the trade-off uses to be good. It must be taken into account that there could be problems with cache usage with those large tables and so many functions.

About decoding and dispatching the instructions we will talk more because it is one of the points it is spent a lot of the effort to optimize CPU emulators. Threaded code/interpreters are used to reduce the overhead of the decoding.

Instruction emulation.

That uses to be more or less a mechanical phase in the implementation of the emulator, most of the instructions are quite simple and common among all the CPUs. For example you always will find an addition instruction, an and instruction or jump instruction for example. Most of times those instructions are easy to implement. This happens more in RISC CPUs because of the RISC design philosophy (simple instructions). In the case of CISC CPUs there can be more complicated instructions which could be a bit hard to implement, for example.

Basically the task here it is to take the ISA (Instruction Set Architecture) of the CPU, which can be found most of times in the official web page of the CPU manufacturer, and reproduce in the language you are using the algorithm (most ISA manuals include the full algorithmic description of the instruction) or the function of the instruction to be emulated.

ADD M

[A]   [A] + [[HL]]
Add to A

Figure 15. i8080 ADD M instruction description (i8080 datasheet).

In the Appendix A, the Space Invaders tutorial, it can be found the explanation of the implementation of some basic types of instructions. Go there for see some examples on this topic. The basic structure of the functionality of an instruction is get input data, operate with that data, store the result somewhere, update the PC and the timing. Some instructions implement all of those parts some just some. For example the more basic instruction, a nop (no operation), just increases the PC and updates the timing. Some instructions, as we said in the previous section, will have to perform some additional decoding, most of time related with where to get the operands from or put the result to, so it is also implemented here.

```
instruction(operands)
{
        get operands
        perform calculations
        store result
        update time
        return to the main loop/fetch next opcode
}
```

Figure 16. Instruction basic algorithm.

The instructions that perform memory access must contain the code to handle the memory access, for example they will have to scan the memory map for special address. It must also had the code for performing data conversions, for example big endian data to little endian. In a next section we will talk about the emulation of the memory in the CPU core.

Although the implementation of the instructions is an easy task it has to be taken into account that we want the maximum performance from the CPU emulator. So we will be aware to implement them the in the more efficient possible way that the language we are using is capable. Something that would be interesting to taken into care is what instructions are the more commonly executed in our CPU and try to spend the most time optimizing them. For example it would not make the effort to optimize a very complex instructions which is almost never used in the programs we are going to execute. It happens many times that the real set of instructions which are used by the programs for a CPU is quite smaller than the full ISA.

It is also important to pay attention to all the side effects of an instruction, which registers modifies but the given operands (implied operands), if it changes the flags or if it raises some exception (if we are emulating exceptions). That also implies to know if the specifications we are using are accurate enough or they have errors. Many CPUs have undocumented features (instructions) which can and are being used by the programmer. So we must know about them and if the programs we want to run use them we have to implement them correctly. It is also important to know the real amount of cycles it takes the instruction to execute if we want to implement accurate timing. In some case (for example multiplication or division instructions) this can be hard to determine though, and we will have to think if the accuracy level needs the exact timing.

About the complex instructions that can be found in some CPUs as noted before if it is an unused instruction the best is just to implement it so it just works and do not care any more. But some of those instructions can be very used, in this case perhaps it will pay the effort to find an efficient implementation, for example on Intel CPUs (8080,8085, x86) we can found string instructions which are commonly used. It can make a big difference an implementation which just copies the standalone instruction (the version for only one iteration) and we just put a jump to the start of the instruction than an implementation that tries to optimize to the maximum the loop.

There are some functionality or instructions which are hard to implement in a high level language, or even on some CPUs, because they work in an higher level. Flags are a good example and performance lost one. We will talk about them in the next section. The solution could be to use assembly rather than the high level language for this instruction. Other hard to solve problems would be if the target has a bigger word size (emulate a 64-bit CPU in a 32-bit CPU) or if the floating point specifications differ.

Not all the CPUs are fully IEEE 754 compliant, some implement the calculations in different data sizes (Intel 80bits, most RISC in either 32 and 64 bits), some use a different size internally for some operations (Power and PowerPC mul/div and sub/add instructions). An accurate emulation of the FP instruction can be really hard, and not a good idea, if the two CPUs (or the source CPU and the language used) are too different. We will not go into FP emulation in this document.

Another example, which is becoming more interesting lately, is the vector instructions, the same operations applied to multiple data or SIMD (Intel). Not only new PIII and P4 x86 based CPUs support this kind of instructions but also the PS2 MIPS has two additional vectorial coprocessors so it is something to begin to take into account (of course you could also want to emulate some of the old vectorial supercomputers just for fun). This is a good example of a harder to implement type of instruction if the language or the CPU does not have vectorial capabilities and also a good example of the great increase of performance if it is efficiently implemented.

If we want to emulate a CPU which supports protected mode, and thus the execution of a full OS, we will have to deal some times with complex operations which can be related with MMU management and other complex stuff. Those can be perhaps the hardest to implement. In fact is something rare unless you want to emulate something like a 486 or a Pentium based computer. The consoles that have modern CPUs use to do not OSes and those complex instructions but there could be exceptions. I will not talk about this topic.

```
case 0x8D:  /* ADC L  */
      AddC(tC.HL.b.l)
      UpdateTime(4)
      break;

case 0xFC:  /*  CM ppqq  */
      if ((tC.flagZPS & FlagS) != 0)
      {
            Call
            UpdateTime(17)
      }
      else
      {
            tC.PC += 2;
            UpdateTime(11)
      }
      break;

case 0x2F:  /*  CMA  */

      tC.AF.b.h = ~tC.AF.b.h;
      UpdateTime(4)
      break;

case 0xB6:  /*  ORA (HL)  */

      temp8 = readByte(tC.HL.w);
      Ora(temp8)
      UpdateTime(7)
      break;
```

Figure 17. Examples of the implementation of some i8080 instructions.

Flags.

As we said in the previous section flags is one of the harder tasks to emulate using a high level language (or a CPU that does not have those flags). Flags or condition codes are single bit variables or registers which are set after some arithmetic or logic instructions. They were very common in old 8-bit and 16-bit CPUs and they are still in use in many modern CPUs (x86, Power). The most commons are Carry flag which means if the last addition has produced a carry, Zero flag if the result is zero, Sign flag with the sign of the result and some other (x86 has Overflow and Parity). Some CPUs use other approaches like setting complex condition codes (combinations of the already mentioned) like Little Than, Greater Than, and so (Power/PowerPC). Some even raise exceptions rather than set flags (MIPS).

Implementing the calculation of a flag uses to mean to perform additional calculations to the real calculation performed by the instruction. And most of time a single instruction changes more than one flag, so the number of calculations increases. For example an 8080 addition set all the 8080 flags (Carry, Sign, Zero, Parity and Auxiliary Carry). For example, in a direct slow implementation, it could mean up to 5 ifs (and the jumps, which are really a pain to all the modern CPUs). The parity is hard to calculate using all algorithm (it is better to use tables) the Carry and Auxiliary Carry means to reproduce two times the addition. All those task are easily performed by the ALU (Arithmetic-Logic Unit) hardware of the CPU in parallel with the addition but are hard to implement by software.

One solution to the problem would be to use assembly if the way our target CPU implements flags is similar to the source CPU. For example x86 is based in the 8080 family and the flags are almost identical. In most of cases a 8080 addition can be directly translated to a x86 addition (of the same data size) and a 'lahf' instruction which retrieves the flags.

Flag calculation in arithmetic and logic instructions, which are with memory instructions (those also have their own problems) the more common ones, can suppose up to a 30% of the time emulating the CPU. That implies that it is a main target for optimization.

Rather than using assembler a good approach with many flags is to change instructions or CPU cycles for memory space. Most of the flags can be calculated in parallel, or most exactly precalculated, using tables. Usually the tables are just for 8-bit results because of the amount of memory needed for just a 16-bit result, but the more problematic CPUs are also 8-bit so it is not a problem. Zero, Sign and Parity flag can be calculated easily this way. In any case is a good idea to search for implementations which does not use conditional or ifs because in most languages that it is translated to jumps and jumps are very expensive on modern CPUs.

You can found more examples in the SI tutorial in Appendix A.

```
/*  Build the table calculate Z, P and S flags.  */
for (i = 0; i < 256; i++)
{
      ZPSTable[i] = ((i == 0)?FlagZ:0) | ((i & 0x80)?FlagS:0) |
              (hasEvenParity((UINT8) i)?FlagP:0);
}
```

Creation of the precalculated flags table.

```
/*  Calculate 8-bit add carry  */
#define CalcFlagC(value1, value2) tC.flagC =
      ((((UINT16) value1 + (UINT16) value2) & 0x100)?1:0);
```

Calculation of the 8-bit add carry.

```
/*  Calculate Z, P and S flags  */
#define CalcFlagZPS(value) tC.flagZPS = ZPSTable[value];
```

Calculation of the zero, parity and sign flags.

Figure 18. Flag calculation in the i8080 emulator.

Memory.

The emulation of the access to memory can in some cases, like the calculation of the flags, suppose a significant amount of the time wasted in emulating the CPU. It is because of the combination of the same two conditions: the memory instructions are very common (memory access is basic for any CPU but even more for old CPUs with few registers) and the complex way some memory access must be translated.

The main problem with the emulation of the memory is how to decide what kind of memory or device is mapped behind a given address. It could be a ROM or normal memory (RAM) or a memory mapped IO register. Every time a memory access is performed, either a read or a write, the CPU emulator must checks what kind of access is.

If our emulated machine would be using just plain read-write memory, with no IO mapped registers in its address space, and without special regions of memory we could just emulate the memory access with an byte array of the size of the memory. Every access to memory will be then implemented like an access to a position in this memory buffer.

No special memory features: Direct access to a buffer.

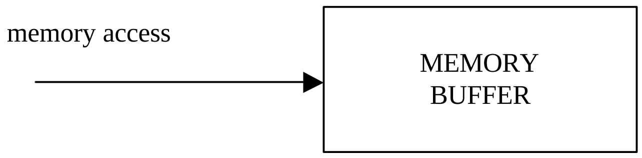


Figure 19. Direct access to the memory buffer.

However that does not use to happen. We will have a list of regions of memory which are special, each region will have something associated, either a function, a pointer to a buffer or something else. The CPU emulator will have to scan this list for the given address and perform the action associated with the proper region.

Special memory features (IO, banks ...). Access throug a list of region handlers.

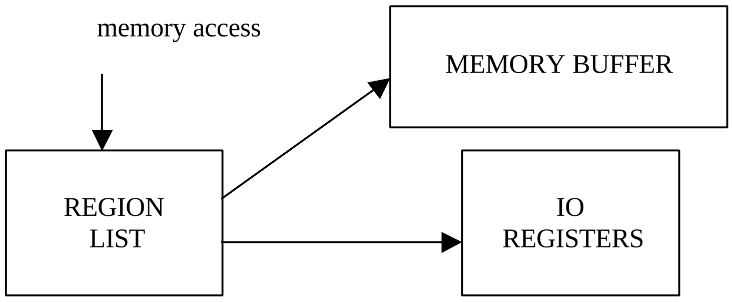


Figure 20. Access through a list of memory regions.

Usually there are just two separate lists for read access and write access. In some cases is needed another list for fetch (read code or write code) access, a fetch access is any access which is performed relative to the PC. Each emulated instruction which accesses the memory will have to scan one or more of those lists (if it performs both a read and a write for example). That is the reason because it is a so wasteful task, because if the list of regions is large it can consume many cycles. By the other way we are adding code which many times is unnecessary because almost all access use to go the main memory buffer.

In CPUs which have separate a separate address space for IO we will have two more lists, read IO and write IO, but in this case, as the IO access is already slow in computers, the performance lost is less. And the IO operations are not so common like normal read/writes to memory.

Old computers use to have a plain memory system but as the machine become more complex the memory system become more complex too. In computers with small address spaces (16-bit address space in old 8-bit CPUs) sometimes they need to access more memory than they can map directly in their address space (they use ROMs like the main source of data and programs). The simpler mechanism to solve this problem is to have regions of the address space which can map different pages of the physical memory, those regions are called banks, and the system banking or bankswitching. A direct implementation of this would be to have an array of pointers for each bank which will contain the address for the mapped page.

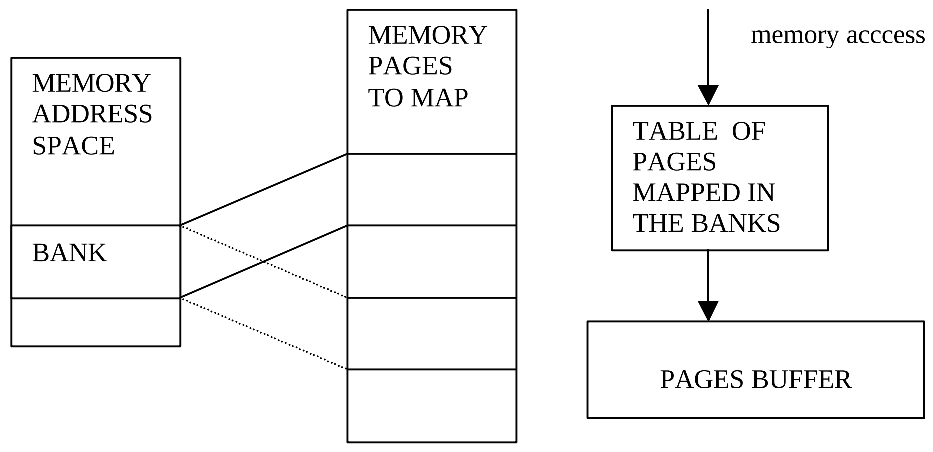


Figure 21. Memory banks and emulation.

In modern CPUs the memory management system become even more complex with additional capabilities to support virtual address, protection and other features needed for OSes. Those CPUs implement MMUs (Memory Management Units) to manage the access to memory and mapped IO registers. The emulation of such hardware implies a great loss in performance. One good solution could be try to use the own MMU of the target CPU or OS memory subsystem to handle the access to memory.

Some CPUs limit multibyte access to aligned address and they raise exceptions if a unaligned access is performed. This must be also checked in the emulation. Just another problem with memory is the endianness. If the emulated system and the target system work in different endian modes it is needed some kind of conversion of the multibyte data. Usually the conversion can be performed each time a read or a write is performed or, if possible, the best at start-up when the data is loaded into the emulator.

We will talk further about memory emulation in another chapter.

Interrupts.

Interrupts are basically mechanism for interrupting the task that is actually performing the CPU (the code is executing) because something has happened and another part of the code must be executed to handle this event. There are two kind of interrupts, hardware interrupts (usually also called just interrupts or IRQs) or CPU exceptions. Hardware interrupts are generated by hardware outside the CPU and arrive to CPU through special PINs of the control bus. CPU exceptions happen when the CPU detects a problem in the execution of the instruction.

The CPU exceptions are full responsibility of the CPU emulator which have to implement them. Such exceptions can be for example divide by zero exception, when a division instruction is executed with a zero as divisor, or illegal opcode exception when it is fetched data which can be correctly decoded. Another important exception in modern CPUs is the memory fault exception which is produced by the MMU.

The exceptions are implemented adding additional testing to the implementation of some, or all, of the emulated instructions. The emulation of some exceptions can reduce a lot the performance of the emulation. A good example of such exceptions is the memory exception which must check any memory access (we already told that MMU emulation is a hard task). Another good one could be the check for misaligned access. Most of time the exception check will have a negative result but if the programs we will execute need of them we need to implement them. In some cases it could make sense to do not implement such testing if they are rarely or never used. Exceptions use to be used for detect problems but if we know that the program already works correctly perhaps the checks are unnecessary.

The emulation of the hardware interrupts is not a main task of the CPU. The different hardware devices are the responsible of signalling the interrupts, and it will be the emulation of those devices which will take care of when and how to signal the interrupts correctly. The CPU just must receive and answer correctly to the interrupts. It is the CPU core thus which has to implement this response and provide a mechanism that permits the external devices (which their emulation is also implemented externally to the CPU core) to send signals to the CPU core.

For these task there will be special functions will be tell the CPU core that an interrupt has been signalled. In some cases those functions will return if the signal has been received or denied (acknowledge). The functions can either be signalled when the CPU emulation is stopped (remember the main loop, we emulate the different parts sequentially) or when the CPU is in execution (signalled by the memory handlers). The second option is rare though. The signal must be stored in the CPU context (or status) for the next time the CPU is emulated. When the CPU emulation starts again the pending interrupts information is processed and the opportune actions are taken.

The CPU could implement different mechanism for deciding if a signalled interrupt must be served or not and to decide between different interrupts (for example if a new interrupt is received while another is being served). This is implemented by interrupt levels and interrupts enabling/disabling flags. Those must be emulated in the code at the start of the execution of the CPU emulation.

An interrupt is served stopping the execution of the code at the actual PC, saving the PC and sometimes some of the CPU status in memory (in the stack or in special registers), and jumping to a new address which must have special code for handling the interrupt (interrupt or exception handler). After the interrupt handler finishes its execution the CPU returns to the previous address and restores the CPU state. Exceptions work in the same way but it is the CPU which produces the interrupt, and the returning instruction can be the same it has produced the exception while a interrupts just happens after the full instruction has been executed.

The way of obtaining the address to jump to differs from one CPU to another. Modern CPUs use vector driven interrupts and exceptions, an array of jump address somewhere the CPUs know and indexed. Others use fixed address and some receive the address from the signalling device (one the Z80 interrupts modes).

We will talk further about interrupts (mainly in the device side) in other chapters.

Core interface.

In most emulation projects the CPU emulator core is implemented to be a separated part of the emulator which just gives a standard interface to the main emulator to use it. We will see how the CPU core interface looks like. The main reasons for such separation between the emulation of the CPU and the rest of the hardware is not only by the means of program modularity but because many CPU cores are used by more than one emulator. In fact it happens that many (or almost all) machine emulators are implemented using already implemented, working public CPU emulators.

The interface for the different CPUs it is also something that can be more standard that the interfaces for other parts of the computer. For example an interface for sound and graphic hardware is harder to standardise (but it is possible, M.A.M.E. – Multi Arcade Machine Emulator - just standardised everything).

Basically there are two parts in the interface of a CPU emulator (or by the way in the interface of anything), the data part and the functions part (something like Object Oriented but without any need of being OO). The data part is the CPU context which contains the information, as we said, about the CPU status and state of execution. This data must be read and written (or some parts of the data in any case). There is also good idea to implement an easy way for switching between different context to emulate multiCPU (of the same type) machines.

The functions are the way we can put to work the CPU emulator. The main functions we need are: Init(), Reset(), Execute(), GetContext(), ResetContext(), InterruptSignal() and perhaps Stop().

The Init() functions would initialize the internal data structures of the CPU core. For example if the jump table for the instructions was compressed it should be decompressed or if the code to be executed must be prearranged in some way. It is a good time to create tables for handling flag calculations and the implementation of some instructions (if they are not statically calculated). This function is only called once at the start of the emulation.

The Reset() function must reproduce the effect of a reset signal sent to the CPU. Basically it initializates the CPU so it could begin to execute code. The usual task are to put the registers and CPU flags to default values and to put in the PC and the SP (the Stack Pointer) the start-up values from where to start to execute code. This function must be called before the emulation is started or any time a hardware reset must be emulated.

The GetContext() and SetContext() are used to set and retrieve the CPU status. The context includes all the CPU status and other information which the CPU core needs to perform the emulation. At the start-up process the emulator needs to create an empty context and add the data for the emulation of the memory. That is basically implemented with a main memory buffer pointer which is the default for memory access, and a memory map or lists of regions for those address which a need special implementation. The CPU interface must provide prototypes for the memory handler functions if the memory maps permit this option.

The GetContext() is used either for debugging purpose or for performing context switching in multi CPU machines. The SetContext() function can be called any time something (normally related with the memory) must change in the CPU. For example in a banked system if there is a change in the page mapped in a bank. Sometimes but just two functions could be interesting different functions for retrieving or setting parts of the information (the registers, the status, the memory map, the bank setting) and not the whole context.

The Execute() function is the one which actually starts the emulation of the CPU. Before being able of calling to Execute() the CPU core must have been initializated, the CPU context created and the CPU reseted. The Execute() receives at least one parameter, the number of cycles to execute before stopping the emulation. When the CPU emulators has executed (more or less) that number of cycles the function returns if the emulation was successfully. For example if an illegal opcode was found the emulation should stop and the Execute() function return an illegal opcode error.

The InterruptSignal() is used to signal hardware interrupts (IRQs) to the emulator core. If the interrupt system that the emulated CPU uses needs extra parameters (for example the interrupt number) for performing a signal, those parameters are passed through this function. It the CPU must return something to the device (for example if the interrupt was accepted) then this function must return that information.

The Stop() function could be an option so that the CPU emulator could be stopped while it is running. Since we are mainly working with sequential (monothreaded) emulators this only can happen when it is being executed a memory or IO function handler. May be is not a very useful function but in some cases could be needed.

Those are the basic functions we could found or implement for a standard CPU core but more could be added. For example functions for retrieving the only special parts of the CPU context. One which could be useful could be retrieve the number of executed cycles. The number passed to the Execute() function is hard to be the real number of executed cycles because the instruction cycle granularity can make hard to execute that exact number of cycles. And some instructions take different number of cycles to execute (old CISC CPUs). To perform an accurate timing emulation after the Execute() function the emulator should get the real number of executed cycles and add to the global count and so adjust the next cycle slice for the CPU core.

The CPU emulator could also add facilities for debugging, for example a step by step execution, or breakpoints, register reads and writes. There could be functions to handle all those functionalities.

```
void i8080Init();
void i8080Reset();
void i8080SendIntSignal();
void i8080SetIntAddress(UINT16 address);
void i8080SendNMISignal();
void i8080SetNMIAddress(UINT16 address);
void i8080SetContext(pI8080CONTEXT context);
void i8080GetContext(pI8080CONTEXT context);
UINT32 i8080Exec(UINT32 cyclesExec);
```

Figure 22. i8080 emulator interface.

Appendix A: Space Invaders/i8080 tutorial.

In the Appendix A there is a tutorial which explains step by step the implementation of an emulator for the old ‘76 Space Invaders arcade machine. This computer uses a Intel 8080A CPU (or a clone) and a basic graphic and sound hardware. The tutorial is centred in the implementation of a i8080 in C. In the tutorial it can be found a larger explanation of the process of implementation of a basic CPU interpreter emulator.

2. Assembly emulators.

Emulate another CPU using a high level language is a very expensive task. Some times happen that the emulated machine is too powerful to be emulated at the same speed in our target machine using a high level language. Then an assembly CPU core could be a good solution.

Assembly CPU emulators can improve greatly the performance of the emulator. There are two reason for this improvement: the first reason is because a well trained assembly programmer can exploit the capabilities of our target CPU ISA (the instruction set) better than a compiler (now there are good compilers, but not so good). The second reason is because any high level language hides many of the special features of the ISA (for example flags). Some of these features can be very useful in the emulation of another CPU. As everyone could see the thing that better emulates how a CPU work is another CPU.

A couple of examples which show the problem that the high level languages have while emulating CPUs are the registers and the flags. In an assembler core the programmer can assign all or some of the emulated registers to real registers in the target CPU. This can improve a lot the emulator because it is reduced the access to memory for loading the emulated registers. Some languages (C) can assign registers to variables but this also uses to be against the compiler optimizer and the generated code could be worst.

The problem with calculation of the flags is even more impressive. The high level language are independent of the used CPU, but flags (or condition codes) is one of the more depending features that can be in a CPU. Therefore flags are always hidden by the language. The problem is that in a CPU emulator we need them. If we want to emulate flags and our target CPU has flags which are similar to the emulated flags we could use them. We said in a previous section how expensive the flag calculation was. Using the own target CPU flags to emulate the flags of the emulated CPU reduces a lot the number of instructions needed, sometimes even to one or zero instructions (zero if we use the status word to emulate the emulated status word, one if we have to store them).

Portability vs performance.

Although an assembly core seems to be very good there are some limitations we have to know. An assembly core is not portable, it can be only used in the computer and the OS for which it was programmed or perhaps, ignoring the differences between similar computers and OSes, it can only be used in those computer/OSes which use or have the same CPU.

An emulator implemented in a high level language can be more easily ported to any computer/OS/system which has a compiler for the language. The emulator has, of course, to be implemented with portability in mind. It can not use features which are not standard in the different versions of the compiler (for example with C it should be used the ANSI C standard). It must be also take into account the possible differences between the different systems where it can be used. For example in C an integer (int) has the size in bits of the CPU word size. It could be than one of our target machines would be a 32-bit machine and the other a 64-bit machine, if we use directly 'int' we could have problems because the real size of the data would be different in the two machines. If the CPU emulated was a 64 bit CPU and we are implementing its registers as int variables, in the 64-bit target machine we will not have any problems. But when we would try to run the CPU emulator in the 32-bit machine it will not work.

This means that an emulator programmed in a high level language CAN be portable to any system but it must me implemented correctly. This special implementation can reduce even more the final performance of the emulator too. On the other hand an assembler core is not directly portable for any system, but it could be portable to any system with the same CPU. For example a x86 core could run in a Windows system, a Linux system or a BEOS system.

The decision to implement or not an emulator in assembly is related with our objectives and our resources. If the target machine is not enough powerful for emulating the CPU at 100% the speed with an emulator programmed in C (for example) but we know that with an assembler core it could work, that will be our choice. If we want that our emulator could be executed in different systems which use different CPUs and the performance was not a big issue, an emulator implemented using a high level language would be our choice.

An emulator programmed in a high level language is easier to implement, it needs less time for writing and testing it, and less knowledge from the programmer. On the other hand an assembly core needs a good knowledge about the target machine (and of course about the emulated CPU, but this also happens with the high level language emulator), the testing is more difficult and requires more time. An assembly emulator is more expensive in terms of the work, a high level language code emulator is more expensive in terms of the final performance (and thus the requirements of the target machine).

In fact, with the home PCs becoming more and more powerful every day performance is not an issue any more. Most of the old systems we want to emulate (8 bit and 16 bit consoles and arcade machines) can be easily emulated in any modem PC (Pentium III 500 MHz for example) with any need of put special effort in obtaining a good performance. Assembler emulators, and other techniques used to enhance the performance of the CPU emulator, are useful, however, for implementing emulators for the modern 32 bit computers and for porting to emulators to more limited platforms like handhelds.

Another question is if you want to implement a 'good' emulator or not then perhaps you will prefer to implement it using assembler. It is not the same an emulator which can run the old Pacman in an i486 than an emulator that needs a P-III 500 for running it. It is also important to note that most of the emulators are not going to be ported to other systems and therefore the portability is not an important issue. Try to obtain a good performance is always an issue for a good programmer.

A way to avoid the problem of the portability of the assembly CPU emulators is to try to implement and use automatic tools for generate the emulator. Tools for automatic generation of dissassemblers and assemblers already exist [1]. There are also generic binary translators which use description files for the CPUs [2]. A tool that could generate the full, or most of it, code for a CPU emulator from a description would not be harder to implement than the other two tools. Perhaps the emulator will not be as optimized as a hand coded emulator but it would permit an easy way of implementing assembler CPU emulators. In another iteration a programmer could patch and improve the generated code.

Such a tool should have two parts: a front-end and a backend. The front-end would deal with the emulated CPU and the backend with the target CPU. Description files for both the emulated (source) and the target CPU would be needed. These descriptions could be reused later for other emulated or target CPUs. Of course we only want to implement a CPU emulator for a specific target machine building such a tool would be useless.

Similarities between CPUs.

One of the reasons because a well written assembler CPU emulator can largely out perform an emulator implemented in any high level language (C could be the reference for the fastest and most low level high level language) are the similarities between the two CPUs involved.

If the two CPUs share a lot their characteristics the assembler core will be even faster. If there are a lot of different perhaps the improvement will be minimum. We can see this with two examples.

All the Intel family of processor is very similar (the fact is that the x86 family has kept binary compatibility for generations). For example all the i8080, i8085 and Z80 instructions are directly translated to the x86 instruction set. The flags are almost the same and can be directly emulated with the x86 flags. Even the Z80 registers can be mapped to x86 registers so many instructions can be implemented without accessing the memory. This makes that a Z80 emulator written in x86 assembler could be more than 10 times faster than the best Z80 emulator written in C [3] could.

The Motorola 68K family is also very similar to the x86 family (although it has more registers) and it is also a good target for a x86 assembler emulator. Both the Z80 and the M68000 are the more used CPUs in the old consoles and arcade machines so they are chances to reuse the cores.

The opposite example could be to implement an emulator for a RISC CPU, for example Alpha (Compaq/Digital) or MIPS (SGI). The instruction sets of these CPUs are enough simple so a high level language could emulate them without any problem. It happens both with these CPUs as the emulated CPU or the target CPU. If it is being emulated, the CPU has not features which are hard to implement in a hard level language (like flags). If it is the target CPU then the compiler will be already using all the features of the CPU for the emulation. A good assembler programmer could implement a better CPU core but the gain is less than in the previous example.

The difference between the two CPUs affects to the performance of the emulator in both cases. If the emulated machine has a feature that is hard to implement in the target CPU it will be hard to implement both in assembler and in a high level language. An assembler emulator just adds the possibility to use CPU features which are hidden by the compiler but which are useful for implementing an emulator. If such features does not exist it can not be used. MIPS for example does not implement flags or condition codes so the emulation of the flags will be as hard in assembler as in C.

Register usage.

Another of the advantages of implementing an emulator using assembly language is that you can control how the target CPU registers are used. A good compiler uses to analyze the code and generate a good register allocation (the assignment between variables and physical registers) for the most common applications. The problem is that a CPU emulator is not a common application. Most of the time the CPU emulator is jumping all around the code (while emulating each instruction) and the compiler would be very good to perform a good register allocation.

The compiler does not know either what is doing the code. However we already have this knowledge and we can use it to improve the emulation. We know that the variables which are more likely to be used are the emulated. That means that a good approach would be to assign physical registers to emulated registers. Or if the number of registers in the target CPU is limited, the most frequently used registers. Only with this optimization the performance of the emulation is largely increased because the access to memory is reduced and the emulation of the instructions simplified.

There are some high level languages that have directives to assign variables to registers (for example C). These directives can be used but they use to go against the compiler optimizer and the generated code could be worst.

Another specific benefit that can be obtained from using directly the registers is that some CPUs permit that a register could be used either in different sizes. For example x86 permit to access the register either as 8-bit, 16-bit and 32-bit registers. This can be useful for emulating other CPUs which have the same capability (Z80, M68000) or CPUs which have registers of a smaller data size (for example 8-bit registers).

Optimizations.

Most of the optimizations are related, as we have already said, with instructions which are similar in both the emulated and the target CPU but can not be directly used with a high level language. That includes flag and condition code calculations, data conversions (big endian to little endian, sign extensions and zero extensions, etc) and complex or special instructions.

We will see now some examples.

We already said that one of the most expensive tasks that a CPU emulator can perform is the flag (or condition code) calculation. There are two different reasons for this cost: the abstraction level of a high level language which hides the CPU flags, and some CPUs which does not implement flags or its implementation is very different from the emulated flags. If the reason was the first one an assembly core will be useful because we will be able to use the CPU flags. For the second reason an assembly implementation perhaps could be faster but the profit will be minimal.

Two examples of how the flag calculation can be improved are Z80 and 68K emulation in x86 CPUs. All three CPUs share more or less the same flags: zero flag, carry flag, parity flag, sign flag, overflow flag and similar. Most of these flags (for example carry flag) are hard to calculate using a high level language but as they work in a very similar way in all the CPUs with a few assembly instructions can be emulated. Even more, the same x86 status word (which carries the flags) can be used for storing, retrieve and restore the emulated flags. We will see even more about how helpful this feature is in the binary translation chapter.

The first example is the translation of a Z80 add instruction to x86. We could see the C and the assembly implementation. It can be easily seen why the assembly version is a lot of faster. The operation is only performed once and it needs only a few instructions to get the correct flags.

Z80 ADD flag calculation in C.

```
/*  Calculate 8-bit add carry  */
#define CalcFlagC(value1, value2) tC.flagC =
          ((((UINT16) value1 +  (UINT16) value2) & 0x100)?1:0);

/*  Calculate 4-bit add carry  */
#define CalcFlagAc(value1, value2) tC.flagAc =
          ((((value1 & 0x0f) +  (value2 & 0x0f)) & 0x10)?1:0);

/*  Calculate Z, P and S flags  */
#define CalcFlagZPS(value) tC.flagZPS = ZPSTable[value];



/*  Macro for Add instructions  */
#define Add(value) {                              \
          CalcFlagC(tC.AF.b.h, value) \
          CalcFlagAc(tC.AF.b.h, value)       \
          tC.AF.b.h = tC.AF.b.h + value;      \
          CalcFlagZPS(tC.AF.b.h)  \
}
```

Z80 ADD flag calculation in x86 asm (Neil Bradley's MZ80)

```
          sahf
          add     al, ch
          lahf
          seto    dl
          and     ah, 0fbh        ; Knock out parity/overflow
          shl     dl, 2
          or              ah, dl
          and     ah, 0fdh ; No N!
```

Figure 23. Z80 to x86 flag calculation. C and ASM versions.

The second example is the same but with a 68K subx instruction.

```
sbb  ebx, edx
 mov  dl, ah          /* keep temporary copy of old flags in DL */
lahf\n
setc byte [x]
seto al
jnz  short .z         /* if non-zero, cleared */
and  dl, 0x40         /* otherwise, unchanged */
and  ah, 0xbf         /* (get rid of new unwanted Z) */
or   ah, dl           /* OR in the old, unchanged Z flag */
.z:
```

Figure 24. m68000 SUBX instruction in x86 (Bart's Gen68K).

Another example in the x86 architecture of an assembly instruction which can be used for improving the emulator performance is 'bswap' (for 32 bits) and 'xcgh' (for 16 bits). We already introduced the problem of the byte ordering in memory (little endian and big endian) and the cost it has the data format conversion. Those two instructions could help to perform this conversion. The instruction BSWAP exchanges the high order 16 bits with the low order 16 bits of the register. At the same time the two bytes in each 16-bit subword are swapped. The XCHG instruction can be used to swap the low and high order bytes of a 16-bit register. So they can be used to perform a fast data format conversion for 32-bit and 16-bit data.

A data format conversion using common operations (and, or, shifts) is a lot of more expensive that use those instructions. In other architectures other specific instructions can be used to speed up the conversion. This instructions show how useful can be to have access to some of the specific instructions of the CPU which are hidden by the high level language abstractions.

As an example of a ‘complex’ and non-standard instruction we will see the instructions used in the Z80 and x86 for BCD adjust. BCD is binary coded decimal, a decimal is coded in hexadecimal format and adjustment instructions are used to avoid forbidden digits (‘a’ to ‘f’). This kind of data was used some decades ago for performing decimal calculations. Implementing such instruction is expensive using a high level language or requieres a large precalculated table. But a single x86 instruction can do the entire job.

This is just a simple example of the implementation of complex instructions using either a high level language or assembly can be improved if the source and target CPU have instructions similar.

Another good point in an assembly emulator is that it is easier to control the flow of execution than in high level language where most of the control is performed by the compiler. In most architectures (mainly x86) function calls are expensive and must be avoided. In assembly the developer has more freedom to code the internal flow of execution of the emulator.

One of the best enhancements which can be done to an interpreter emulator is to speed up the fetch-decode process.   We will see other ways to increase the performance of fetch-decode loops but in this case we will talk about inlining the fetch-decode at the end of each instruction. This could me implemented in a high level language but it needs special features (like labels and goto instructions) which can not be found in all the compilers.
An assembly program and therefore an assembly emulator gives more freedom to the programmer to write its code in the way it could improve the performance. We have already seen a lot of example. The fetch-decode loop we told about in the first sections of this chapter is a good target for this flexibility. An assembly coded fetch-decode loop eases the task of implementation and helps to improve the performance. You don’t have to trust in the compiler implementation of switch statements. And it is a clearer implementation than function tables or label based jump. In assembler is just a load from memory (the table with the address for the emulated instructions) and an indirect jump.

But this is just the basic way it can be emulated. A trained assembly programmer can find many others which can be more suited to the specific emulator. In [25] “emu-mech” we can find almost a dozen different forms of implementing it. We will talk about some of those forms (for example threaded interpreters in the next section).

```
        mov     edi, [cyclesRemaining]
        xor     edx, edx
        sub     edi, byte 15
        js      near noMoreExec
        mov     dl, byte [esi]      ; Get our next instruction
        inc     esi                 ; Increment PC
        jmp     dword [z80regular+edx*4]
```

(from Neil Bradley’s MZ80 Z80 emulator.)

Figure 25.   Timing update and instruction decode inlined at the end instruction implemention.

One the most useful implementations is to inline the fetch-decode at the end of the emulated instruction. This avoids the jump back to the main loop. So now an emulated instruction will have its own implementation, and after it has been executed it will read the next opcode, look at the decode table and jump to the code emulating the next instruction. The fetch and decode process can be reduced in number of instructions if some assumptions are made. For example size of each instruction emulation could be fixed to avoid reads from a table.

Code emitters vs assembly macros.

Write an emulator in assembler is a hard task, it is easy to have errors and they use to be hard to debug. Another problem is that most of the code is replicated a lot of times and therefore is hard to track an error and change all the places where this code is replicated. It happens because most of the instructions share a lot of the code.

For improving the performance of the emulator, the code for a type of instruction is replicated to avoid any decoding in the instruction code. For example we could use just a single routine for all the move register to register instructions, but such function will need to decode each time the source and target register. If we want to avoid this decoding we could write a version for each combination of registers. The full decoding is then performed in the fetch-decode loop as we saw in a previous section (using a jump table for example). All those functions share a lot of code but just the destination and source register.

```
MovRegtoReg(rd, rs)
{
        context.rd = context.rs;
}

MovRAtoRB:
        mov ra, rb
```

Figure 26. A single function for each instruction type or multiple functions for each instruction type (example mov rd, rs).

The way this replication of code and functions can be managed is using macros or code emitters. Most assemblers implement macros (in C macros are implemented with the directive #define). A macro is just an identifier that, when it is found in the code, it is replaced by the associated value (macro definition), piece of code or whatever in a first pass of the assembly process. A macro also can be used with parameters. We can write a macro for each type of instruction we will have, and then use the macro, with some parameters to code the different versions of the instruction. This way the code is only written once and can be modified easily.

```
macro movRegToReg(rd, rs)
        mov rd, rs
end macro

OPC1:
        movRegToReg(RA, RB)

 OPC2:
        movRegToReg(RC, RD)
```

Figure 27. Using macros for expanding generic instruction implementation.

Code emitters are a bit different. A code emitter, rather than use the capabilities of the assembler, uses a high level language program to output the full listing of the assembler emulator. Each generic instruction (for example a mov) has a high level function which receives parameters about the specific instruction (for example source, target and operand size) and writes in a file the assembler function which implement the specific instruction. The full emitter is a collection of functions for each generic instruction, more functions for writing the glue code of the CPU emulator and the main functions which control how the output assembly file is generated.

Then the process of building the CPU core has two phases: first the code emitter is written (or modified if we are already in the write-test-debug development cycle), then it is compiled with the high level compiler and an executable is produced. In the second phase the emitter executable is run with proper parameters and the output assembly file with the asm core is assembled and linked to the emulator (or the test program). Then if the CPU core is already finished the emulator can be used or continue the development of the other parts of the machine, or if the CPU core is already on development can be debugged.

A core emitter has some benefits over a core programmed in assembly just using macros. For example can be easily retargeted for different assemblers for the same architecture (for example in x86 there are different conventions: AT&T in Linux and GNU assemblers, Intel convention; different assemblers use different directives: NASM, TASM, MASM) just adding options and a bit more of code to the emitter functions. It has also a lot more flexibility to change the generated core. It can be added easily options which modify the output core, for example different calling conventions (stack, register) or how it will work the emulator (implement bankswitching or not). It could be changed the accuracy of the emulator for improving speed for example just passing a parameter to the emitter.

Basically a core emitter helps to admit different modifications or versions of the final assembly core. The high level language also helps to have a better control, more structured and easier to understand implementation of the assembly emulator. Most of the assemblers do not have as many funcionalities as a high level language (just basic macro and conditional directives).

```
EmitMovRegToReg(rd, rs)
{
        emit(“mov %s, %s¨, TargetReg[rd],
        TargetReg[rs]);
}
```

Figure 28. Using an code emiter for generating multiple versions.

Forcing even more the idea of a core emitter we could have core emitters which could produce assembly emulators for different CPUs or architectures. The control structure, which would control how the assembly file is generated, could be fully reused, but most of the generic instruction emitters should be reimplemented. Therefore the work saving is reduced but it could be useful.

Another approach could be to have a code emitter which would use an ISA definition file describing the output architecture. The generic instruction emitters would use a kind of intermediate representation which would be translated to the final assembly output using the information in the ISA definition file. And a final step would be to have an automatic tool which from two ISA definitions could generate interpreters from one CPU to the other without further help by the programmer. [1] But this is an advanced topic which goes out of the purpose of this document.

3. Threaded code.

The concept of threaded code for implementing interpreters was introduced in 1972 by James R. Bell [4] although some previous works were made around Forth language in early years. Forth is an interpreted language which is very related with threaded code interpreters.

The idea of threaded interpreters is related with the fetch and decode phase of the emulation. We have already said in the previous sections that this task is expensive to the emulation of the CPU, furthermore because it is not doing useful work (the real work is to execute the instructions). We have already seen various ways of implementing the decode: switch tables that need a good compiler for producing a good implementation, and functions tables which introduce more overhead and need from global variables. Some compilers also provide other features like labels and goto statements. If the compiler has the ability of optimize tail calls (basically non-returning calls which keep calling other functions) then this feature can also be used.

We also talked about moving the fetch-decode to the epilogue of the instruction emulation to avoid a jump back to the main loop (or a return in the case a function table). All those features can be still used with threaded interpreters but the main idea around threaded code is that, because of the temporal locality of the code (the same piece of code is run more than once), the decoding of an instruction has only to be done once, not every time the instruction is executed. Threaded code means code which is run through a list or array of code address (pointers to functions for example) rather than sequentially. Those addresses in a threaded interpreter are the decoded address for the emulated instructions. The purpose of threaded code is to cache the decode for next executions of the same piece of code.

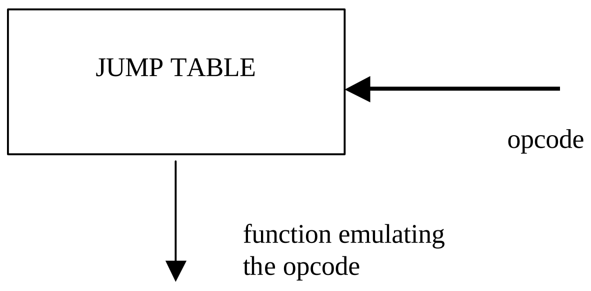
We will see how it works and how can be implemented either in some high level language (most of them are rather limited to provide a fast implementation) and with assembler interpreters. At last we will some commercial and academic products which use this technique.

Basic concept behind threaded code.

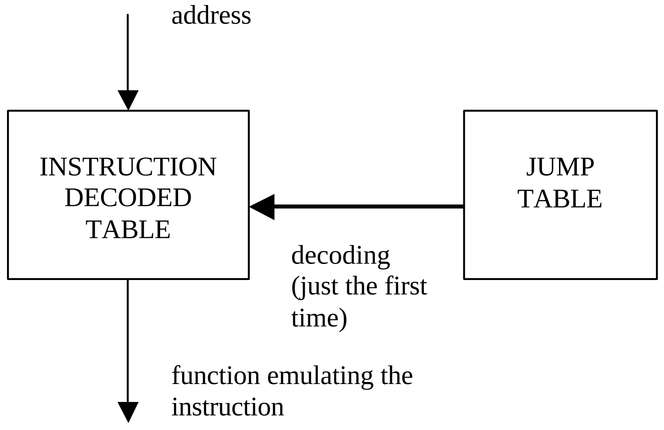
The idea behind threaded code interpreters is to separate the fetch and the decode of the instructions. The fetch still has to be done each time an instruction is executed but the decode will be performed only once, the first time the instruction is fetched. This could happen either in static time (when the code is loaded it is decoded before begin the emulation) or in dynamic time (the code is decoded only when it is executed, the first time it is fetched).

What means decoding the native code in this case? In a normal interpreter emulator the opcodes are read and either using a jump table or a switch statement a piece of code (the piece of code which emulates the opcode instruction) is selected and a jump is performed to it. So there are three steps: first read the opcode in the memory address pointed by the PC (that is the fetch phase), then obtain the address of the function which emulates the instruction and finally a jump to this function. A threaded code interpreter deals with the second step: how to obtain the address of the emulated instruction code.

In a typical interpreter this step means to use the opcode as an index in a table which contain code addresses. That means that in the best case (when the decode can be done in a single step) we are performing two memory access (one for the fetch and one for the address of the emulated instruction) for beginning the emulation of an instruction. The threaded code interpreter what does is to store the jump address for each instruction in a separate table. This table will be addressed by the same emulated PC. Now the fetch and the decode phases are performed at the same time and the process reduced to two steps: first a jump address is read from the table and then a jump to this address is performed.



Normal decoding using a function table (each time a instruction is executed).


address
INSTRUCTION DECODED TABLE
JUMP TABLE
decoding (just the first time)
function emulating the instruction

Decoding in threaded code.

Figure 29. Normal emulation vs Threaded code emulation.

Of course this is just in the general case. In a first stage this entry in the table has been filled with the address of the function which emulates the instruction in the equivalent address of the emulated memory. This process of 'translation' from the emulated code to the threaded code works in the same manner a typical interpreter but without the phase of emulation of the instruction. The translation can be performed either in static time or in dynamic time.

A static translation means that we already know what code will be executed before the start of the emulation. We know the start address and the end address of the code so we begin fetching opcodes from the start address. For each read opcode we will obtain the address of the function which emulates the instruction. Then instead of jumping to this address we store it in a translation table or threaded code table. This table has an entry for each address in the emulated memory which contains an instruction so it could be used the emulated PC directly as the pointer.

```
for(PC = ENTRYPOINT; PC < MAXPC; PC++)
{
        decodedTable[PC] = JumpTable[readMemory[PC]];
}
```

Decode phase.

```
while(!end)
{
        (void *) (decodedTable[PC++]) ();
}
```

Execution phase.

Figure 30. Static threaded code emulator.

A dynamic translation is a kind of mixture between a normal interpreter and a threaded code interpreter. The translation table is being built as the instructions are being emulated. When a given instruction was not already translated into threaded code the emulator will work as a normal interpreter. The opcode will be read from the emulated memory, the jump table with the emulated instruction functions will be accessed and a jump will be performed. The difference is that now the jump address is not thrown away after the jump but it is stored in the translation table for the next time the instruction is executed.

If the instruction was already executed it will just read the translation table at the entry pointed by the emulated PC and do the jump. There are different manners of implementing this kind of dynamic translation. A good one is to write a function which perform the execution as a normal interpreter (as we have said in the previous paragraph). Then initialize the translation table writing in each entry the address of this function. The fetch-decode code will be the same as in the static approach and each time a non initialized entry is accessed (an instruction which has not been executed before) the translation to thread code is performed.

```
while(!end)
{
        if (decodeTable[PC]  == NULL)
        {
                decodeTable[PC] = JumpTable[readMemory[PC]];
        }

        (*decodedTable[PC++]) ();
}
```

Figure 31. Dynamic threaded code emulator.

The dynamic approach is easy to implement as we already have seen. It can reduce a bit the performance of the emulation because the first time a full decoding must be performed. However it is very useful to avoid some problems that the static approach has. These problems also happen with static binary translation so we will talk further in the next chapter about binary translation. Just to point them, those problems include the problem of separate what is code and what is data, self modifying code and dynamic generation of code.

As we already said in the assembly emulators section, a good optimization to the CPU emulator is to put at the epilogue of each emulated instruction function the code for fetching and decoding the next instruction. This saves the jump back to the main fetch-decode loop and reduces the problems with misprediction of indirect jumps in modern CPUs.

```
jmp decodeTable[edi*4]    /*  emulated PC in edi  */
```

At the end of the instruction emulation the PC is used for accessing the table with the decoded instructions and it is performed directly the jump without returning to the main loop.

Figure 32. Example of NEXT in asm.

Threaded code emulators are always a good alternative to normal interpreters coded in a high level language. The performance gain can be great. Threaded code can be implemented with a high level language (although with some problems) as we will see so its is also a good alternative to assembly code emulators if portability is an issue. In case of assembly emulators in front to assembly threaded code emulators the performance gain will be bigger if the decoding is harder. ISAs which are easy to decode would not have a lot of gain in front of an assembly emulator.

Types of threaded interpreters.

The basic approach we have already talk about in the previous section is called ‘direct threaded code’. It is the simpler and faster type of threaded code but has some limitations. Most of the opcodes have fields with additional information for executing the instruction, for example literals (constants). In most of the cases this information will be easily obtained from the emulated memory by the instruction emulation function. In other cases, though, it will be necessary to fully decode the opcode and store the separate fields in the translation table. That is what is called indirect threaded code.

In this case each opcode is translated into a structure which contains the address of the function to jump to and additional data which is used by the function to fulfil the emulation of the instruction. It could be also implemented with direct threaded code coding a translation table of such structures rather than a table of jump address. In the case of indirect threaded code the translation table is an array of pointers to those structures with decoded information. This implies that an additional level of indirection is added and therefore an additional memory load.

The emulated PC is still the pointer to the translation table. The address in the given entry of the table is read. This address is stored to be used later by the emulated instruction function. From this address is read the address of the function and then the jump is performed.

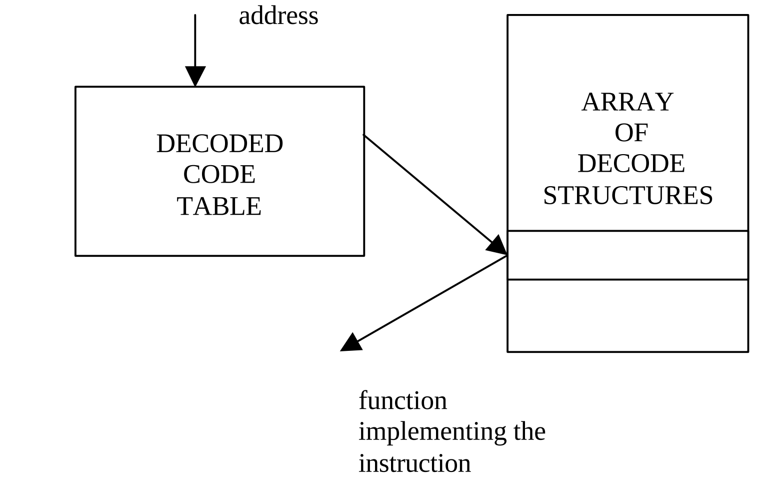

address
DECODED CODE TABLE
ARRAY OF DECODE STRUCTURES
function implementing the instruction

Figure 33. Indirect threaded code.

Token threaded code is another alternative which is intended for threaded code portability. In this case the code to be emulated is translated into an intermediate representation. The translation table contains tokens which are identificators for instructions in the intermediate representation. The process now begins reading the token from the translation table, then the token is used for retrieving the address of the function which implements the intermediate instruction and the jump is performed. This technique is only useful for portability issues. For example if we want to build different frontends for emulating different CPUs keeping the main core of the emulation (or backend). The front-end just translate the emulated code to an intermediate representation and then this intermediate representation is used in a threaded code interpreter. The backend could be reused for all the different emulated CPUs. [5]

Threaded code can be used to implement simple optimizations of the emulated code. The way of implementing these optimizations is expanding or grouping the emulated opcodes. A complex instruction could be expanded in the decode phase into many simpler functions. We must take into account now that the number of 'instructions' in the translation table has been expanded when dealing with offsets and relative jumps. This could increase because of the fetching of instructions but could ease the implementation of the emulator in some cases.

The opposite optimization would be to group simpler emulated opcodes into a single macro opcode which would emulate the functionality of the single opcodes. It could be functions emulating very frequent combinations of instructions, when those instructions are found together in the code the are fetched at the same time. Then the translation table entry will be filled with the address for the combined implementation of those instructions. The benefit of grouping instructions can be because of two reasons. The original instructions were too simpler and a grouped form in the target CPU or high-level language exits (it could help the compiler to produce better code). The other reason (perhaps the more important) is because we are eliminating the overhead of the fetch-decode after each instruction.

Further expansions of the idea of grouping opcodes, for example create dynamically macro opcodes for basic blocks, is a first step into the world of the binary translator. In fact way a threaded code interpreter fetches and executes emulated code can be directly applied to binary translation (as we will see). In the case of binary translation, like in the case of opcode grouping, only some of the instructions (the ones which are basic block entry points) have entries in the translation table.

Threaded code is also very used in emulators which translate from one or various source CPUs to one or multiple target CPUs using intermediate representations. An intermediate representation using threaded code is used to permit help in the portability of such emulator. And it is a fast implementation.

Implementation of threaded code.

The problem with threaded code is that it is difficult to implement high level language if they do not offer some special features. C is the language which has been more used for implementing threaded code and some implementations (GNU C) offer useful features for implementing it (labels and goto directives). We will se some ways it can be implemented in C a threaded code interpreter. The problem does not exist using assembly and it is really easy to implement an assembly threaded code emulator.

First the high level implementations. These implementations have been obtained from [6]. The implementations we will see will be: GNU C's labels as values, continuation-passing style, switch threading and call threading,

```
typedef void *Inst;

Inst *ip;             /*  you should use a local variable for this  */
#define NEXT goto **ip++;
```

GNU C's Labels as Values.

```
typedef void (* Inst) ();

void inst1(Inst *ip,  /*  other regs  */)
{
        ...
        (*ip) (ip + 1,  /*  others registers  */);
}
```

Continuation-passing style.

```
typedef enum { add /*  ... */ } Inst;

void engine()
{
        static Inst progam[] = {inst 1 /* ...  */};

        Inst *ip;

        for(;;)
                switch (*ip++) {
                case inst1:
                        ...
                        break;
                ...
                }
}
```

Switch Threading.

```
typedef void (* Inst) ();

Inst *ip;

void inst1()
{
...
}

void engine()
{
        for(;;)
                (*ip++)();
}
```

Call Threading.

Figure 34. High level implementations of threaded code.

The best implementation using C is to use labels and goto directives. But not all C compilers support them. GNU C supports it and is almost sure it will exist a port of the compiler for every architecture out there. We store in a table the labels or the function address of the instruction emulation. Then fetch is just a goto instruction through a translation table entry.

Another implementation is to use a continuation-passing style, where each emulated instruction function keeps calling the function of the next instruction to be executed. This implementation needs a compiler which can optimize tail calls into jumps. But most of the compilers do not support this optimization. Tail call is a function call at the end of the function call which could be optimized as a direct jump to avoid the overhead of a function return.

The next implementation is switch-threaded code which is near to the idea of token threaded code. A special set of token or values is used as 'address' for the functions. These tokens are stored in the translation table and used as cases in the switch statement.

The last implementation is call threading. The translation table is just a table of function addresses. Each fetch is a call to the function in the entry of the table pointed by the emulated PC. It has some overhead due to the function call overhead and the fact that the code for emulating the instructions is in separated functions. The emulated registers should be used efficiently by all the functions so the compiler should admit optimizations of global variables.

In assembly the best implementation is just a modification of the fetch-decode code that it is added at the end the implementation of each emulated instruction. In this case the emulated PC is used as an index in the translation table and then an indexed jump is performed to the address in the pointed entry.

Related works.

Threaded code and threaded code interpreters have been very used by many years. It is easy to find documents, source code and commercial products that use it. The first reference is J.R. Bell document introducing the technique but exists a larger library of references which can be useful. Searching for Forth, threaded code or virtual machines in the web is a good way of obtaining a lot of information. More actual research around the topic can be found in studies about the implementation of Java virtual machines. One remarkable is [7].

About commercial products it could be found a large number. One we will talk about later is Ardi's Executor a PC emulator for the MAC. This emulator is both a dynamic binary translator and a threaded code interpreter. It uses an intermediate representation which is executed as thread code for those architectures where the dynamic translator backend is not implemented, aiming so to a greater portability of the product. Ardi's Executor is one of the main references about commercial emulators.

4. Advanced questions about interpreters.

In this section we introduce some other topics related with CPU interpreters. We will talk about some general questions about all the types of interpreters emulators and about more advance topics which are out of the scope of this document. Additional references and resources for further searching about interpreter emulators will be also pointed.

Interpreters for simulators.

Most of the academic resources which can be obtained about CPU interpreter emulators is related with simulators. Although the nomenclature is a bit confusing some times, and the term emulator and simulator are used and abused for the same products sometimes, we will consider them as different uses of similar techniques.

The CPU and full architecture simulators are in fact emulators, and thus they use many of the techniques we are talking in this document, but their purpose is different. The purpose of a simulator is to model the internal behaviour of a computer (or a CPU, or just any other hardware device) to retrieve information about how it works in race situations. The emulators, our definition of emulators, are used to implement in a target computer another computer or architecture as a virtual machine, so all the original programs from the emulated computer could be used transparently in the target computer.

The main differences between a simulator and the kind of emulator we are studying are derived from this difference in the purpose. In a simulator the performance is not the primary issue, it is accuracy. A simulator should be as accurate as possible (always inside the specific use the simulator will have) and performance losses in front of accuracy are permitted. In an emulator performance is many times the main issue and, although it is not always desired, performance has preference over accuracy. So in an emulator the accuracy could be limited in some situations.

It should also be taken into account that although the desire in an emulator is the most precise reproduction of the emulated machine it just needs to 'seem' similar to the final user. That is the user will not need to know, or care about, if the emulator is running a very precise emulation of the internal hardware. The user just needs to feel that the system is emulated in the most similar way. The feeling comes from the emulator output: the sound, the graphics, and the speed of the emulation. Those are the main aspects to care in an emulator. Internal precision could, and will, be sacrificed to provide external precision.

Simulators use to emulate the hardware in a more low level than emulators do. An emulator just need a more or less (depending of the emulated system) cycle accurate emulation of the CPU and, usually, less precise of the other devices. The level of accuracy in a simulator depends in the kind of information is wanted to be obtained about the simulated system. For example in some cases it could be just cycle accurate, in accurate in the level of electric signal and some times just accurate in the number of executed instructions.

Another important difference in simulators is that they gather information while performing the emulation. As we will see in the next point, in some situation we will also want that our interpreter emulators would gather some information. But the main difference is the larger range and number of information a simulator is gathering. In an interpreter with profiling the emulation is the main purpose, in a simulator the main purpose is the information.

Most of the resources which can be found around simulators are useful for emulation, but we have to take into account that their scope is a bit different. The documents which talk about how to speed up the simulation and introduce techniques for achieve this purpose are the most useful (Shade [8], Bedichek, etc.).

First pass interpreter and profiler.

In some cases the interpreter emulator will be just the first step in the process of emulation. Many of the more modern and successfully emulators implemented using binary translation use interpreters emulators as a first pass emulation. The interpreter is used in the first executions of the emulated code and only the pieces of codes which are frequently executed are translated to the target machine code. For enhancing the translation the interpreter has to gather diverse information about how it is executed the code. The minimum profiling includes the frequency of execution of piece of code, the frequency of each branch and the creation of basic blocks.

This approach of using an interpreter for the emulation of the CPU in the first pass is used in many commercial products. Transmeta Code Morpher software which provides x86 compatibility using a software layer over the CPU uses it. The interpreter is used for profiling and the execution of the first pass, if a piece of code is spotted to be very frequently executed it is translated to native code and optimized. IBM's DAISY dynamic binary translator for VLIW CPUs (now it translates PowerPC and AS/390) uses a similar approach. Digital's (now Compaq) FX!32 is a static translator for x86 NT applications to Alpha NT. It performs code translations in background using the information gathered by a fast interpreter for all those portions of the code which are not already translated. Java Hot Spot only performs translation from Java bytecode to the native machine code for those blocks of the code more frequently executed, therefore it also uses a Java interpreter for the other parts of the code.

CPU emulators are also used in static translators for avoiding some problems. We will talk about those problems in the next chapter but we can point them here: self-modifying code or the difficulty of separating code and data. The problem is that many times is hard, or impossible, for a static translator to discover, and translate, all the possible code which could be executed in a given program. One solution is to add a fallback interpreter to the runtime of the translated binary. Whenever a piece of code, and address, is found that it has not been translated the interpreter is called to perform the emulation of these piece of code.

Advanced: Inlining and Pipelined interpreters for VLIW CPUs.

Just for introducing a couple of advanced topics related with CPU interpreters we will see two papers [9] [10].

"Optimizing direct threaded code by selective inlining" is a paper from the INRIA by Ian Piamarta and Fabio Ricardi which talks about a technique for speeding up virtual machine interpreters. The idea is based in threaded interpreters and in the idea of opcode grouping we talked in the thread code section. The paper discusses that a static approach for building macro opcodes (groups of opcodes which are emulated all together using the function) is not really correct because there are two many possible combinations. It is impossible with static grouping of opcodes to achieve good performance for all the possible programs to be executed.

They propose a dynamic approach, the macro opcodes are discovered at runtime and the functions for implementing them are generated on the fly (a kind of dynamic generation of code). The technique maintains the portability because it creates the implementation of the macro opcodes copying the code of the simple opcodes and patching them. It is a kind of a mixture between a threaded code interpreter (the execution basis is still a threaded code interpreter but which dynamically creates the opcodes to emulate) and 'portable' binary translator.

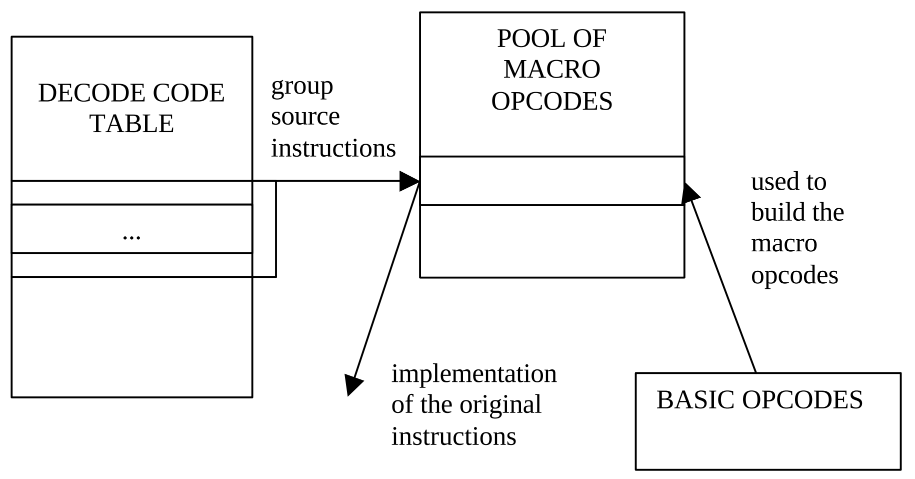
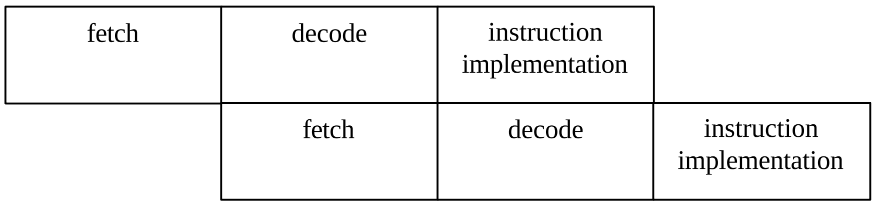

DECODE CODE TABLE
...
group source instructions
POOL OF MACRO OPCODES
used to build the macro opcodes
implementation of the original instructions
BASIC OPCODES

Figure 35. Threaded code and macroopcodes.

Another advanced aspect related with interpreters is the related with the new VLIW (Very Long Instruction Word) and ILP (Instruction Level Parallelism) architectures and CPUs. Part of the industry and the research are now going from implicit ILP (superscalar, reordering, and speculation) to explicit ILP. The task of scheduling the instructions will be now in the side of the compiler and not in the side of the control logic of the processor. In the field of interpreter (we will also talk about the aspects related with binary translation) this implies some difficulties. The paper "Pipelined Java Virtual Machine Interpreters" by Jan Hoogerbrugge and Lex Augusteijn from the Philips Research Laboratories introduces the concept of pipelined interpreters.

The paper discusses about an efficient way of implementing a Java Virtual Machine interpreter in a VLIW CPU. For obtaining a good performance in a VLIW you have to fed the CPU with as many full slots as possible (in a VLIW each 'instruction' tells the CPU to execute –initiate- a number of instruction at a time, each of this instructions is a slot, see IA-64 documents for a reference). It is a hard task to choose which instructions to schedule at a time, reordering and speculation must be implemented to achieve this objective. The compiler should have the responsibility of doing this task. But with an interpreter the functions are small and the code is jumping from one part to the other all the time. There are not good loops for optimization either.

In the paper they propose to pipeline the different phases of the execution of an emulated instruction. They divide the execution of an instruction into fetch and increment (the PC), decode and execution and jump for the next instruction. It is a three stage pipeline. Each instruction is performing the increment, fetch and decode for the next opcodes to be executed, and so filling slows in the VLIW instruction. The paper continues with the explanation of the technique and shows some results of the test they have performed.



Instruction emulation pipelining.

```
f1 = f2; f2 = f3; f3 = fetch();
d1 = d2; d2 = d3; d3 = decode(f3);
/*  Instruction emulation  */
```

Figure 36. Interpreter pipelining;

Integrated CPU cores.

We have talked about CPU cores which are implemented and work separated from the emulation of the other devices. The only communication between the CPU and the other devices is through the memory and IO maps, which at the time have a standard interface and are provide from the outside of the core. Although this is the normal manner that emulators are implemented in some cases it could be interesting to use another approach.

As we will talk in the next point many times we will reuse an already coded (and testes, that is very important) emulator core. This makes that we will have to adapt our emulation to the way the core works. In the previous sections we have talked about cores which would be for general purpose. But if we are implementing our own core as well as the rest of the emulation we could design the core to speed up the emulation, getting advantage of the specific features of our emulated machine. For example we could inline the most accessed memory handlers to avoid the function call overhead. We could also put additional code to synchronize and speed up the emulation with other device. For example wait loops for a trigger in another device could be implemented to skip them and directly call that device.

This topic could become very specific to every target so we will not talk further. But it is interesting, if we are implementing our own core to take this point into account. If we have the freedom of doing whatever we want with the CPU core the best is to take advantage of it. There are potentially many points for optimizing from the limitation of an already coded (a normally general purpose) CPU core. Of course this does not take from the fact that the core we are using or implementing must fit the minimum requirements of the emulated machine. For example it would not be a good idea to use a core which does not support bankswitching (the address space is organized in banks which can page different regions of the real memory, larger than the address space) because the performance loss due to the implementation outside of the core would be too big.

Using library CPU cores.

We have already said in the previous point that often CPU cores and emulators are implemented separately. CPU emulation is a hard task which needs from a large amount of time for coding and testing. Obtaining a fully working CPU core which works (most of the times) correctly can mean months of work (depending of the complexity of the emulated CPU). That is the reason because of in the freeware emuscene (where most of the emulators are being developed by old computer and video game computers fans) it is very usual to use open source CPU cores.

If in fact, if the main interest is just to emulate a system and open source CPU cores can be found for the CPUs used in the system it could be interesting to use those CPU cores. There are various advantages. Those cores have been already tested with many emulators and therefore they use to be very accurate and implement all the features (hidden or not) of the CPU. They use also to be very optimized and many of them are implemented in assembly, so it will be a good performance boost. Most of those open source emulators have years of testing and developing, that means that implementing an equivalent core will require quite a time. The main advantage is of course the great time reduction for the development of the emulator.

In the other hand using an open source emulator core implies to use the conventions defined by the core. The emulator will have to be implemented in the way the core works. The free cores uses to be implemented to be general CPUs, therefore, unless, a risky modification is implemented in the open source core some of the particularities of the system could not be used to improve the performance.

Another interesting use of the free CPU cores is to use them as reference for our own CPU core. The best manner for testing a CPU emulator is to run it hand by hand with the real CPU and look for the differences. This would need some kind of development board with the emulated CPU and many times is impossible to do. A good alternative is to compare the execution of our CPU emulator with another, trusted, CPU emulator.

The interface from one free core to another differs but it uses to be similar to the one explained in the first section of the chapter. Because, as we said, the CPU core is the center of the emulator the emulation will be driven by the way the free core works.

Related work.

There are many sources in the emuscene which introduce the concept around interpreted CPU emulation, as well with many open source CPU cores which can be used to learn how they are coded. Some of the information is this chapter comes from those sources.

About CPU interpreters and general emulation can be found various resources in the web:

- Dan Boris Emulation How To.
- Marat Fayzullin How To.
- Zilmar's Emubook.
- Arcade Emulation How To by various emulator authors.
- Emu-Mechanism is an old discussion about emulation around the Amiga.
- other similar documents and sources.

Opensource CPU cores can be found at:

- M.A.M.E / M.E.S.S. emulation projects.
- Neil Bradley offers various CPU cores written in assembly.
- Nell Corlet's StarScream, an assembly 68K emulator.
- Bart Tryz...., and assembly 68K emulator.

It can also be found some academic papers about CPU emulation searching for Java and virtual machine in web searchers and academically specialized searchers.

The information about threaded code interpreters was found in web page [6] about this topic. It can be found more information searching for Bell's "Threaded Code" article, 'Threaded code' or Java and virtual machine. Another threaded code interpreter is Bedichek's g88k simulator.

Chapter 4. CPU emulation: Binary Translation.

1. Introduction to Binary Translation.

We have just seen how an interpreter CPU emulator can be implemented. An interpreter CPU emulator works in a similar manner to how a real CPU works: data if fetched from the memories, this data is decoded and the instruction is executed. We have seen how to speed up the emulation of the CPU, either using assembly coded emulators or caching the decode data (threaded code emulators). But the process of emulating a CPU is still very expensive.

There is still a lot of overhead due to the fact that we are emulating instructions one by one (we also talked about using macro opcodes in threaded code interpreters). Also because we are emulating most of the characteristics of the CPU, some of them which are not required for our purpose, or not all the time (for example flags). In fact the purpose of the CPU emulator is to run some piece of code from the emulated CPU in our target CPU. Both CPUs are able to execute code, but the problem is that the languages in which this code is represented are different. Could it be possible of executing the code of the emulated CPU in the target CPU without having to build a 'virtual CPU'? The answer is binary translation.

Binary translation.

Binary translation means to translate from a binary of a source target machine or CPU into a binary for a target machine or CPU. A binary uses to mean a full program or executable (but it could be any piece of code that could be executed) stored in machine language code.

The instructions from the source CPU are transformed into instructions for the target CPU. This kind of CPU emulation differs from the interpreter approach. In an interpreter we are building software which tries to mimic the behaviour of a CPU. A binary translator does not mimic (all) the behaviour of the CPU but performs a transformation in the code which is going to be emulated. We will already have all or most of the CPU status information we could find in an interpreter emulator.

One of the differences with an interpreter emulator is that the state of the CPU is not kept all the time in the CPU context. Most of the time this state will be implicit (in the target CPU registers, in the flow of execution) and only in some checkpoints the CPU state will be equivalent to the emulated CPU. In a binary translator we will try to just do the minimum possible which maintains a 'workable' state for the emulated CPU.

In a binary translator the emulation can be divided into two phases: the translation phase and the execution phase. In the translation phase the code from the source CPU is translated into code for the target CPU. In the execution phase the translated code is executed in the target CPU. Both phases are very different and can be studied separately.

The translation phase resembles to the work of a compiler. In a compiler (or in an assembler) we have a program or algorithm written in a given language (a high level language or assembly language). This program must be converted into another language: machine language. In a binary translator we have a piece of code in a given machine language which must be converted into a different machine language. Of course there are a lot of differences between a compiler and a binary translation because the inputs are very different. In a compiler the input is represented with text (ASCII) and resembles a human language (more or less). This input must be lexically, syntactically and semantically parsed and analysed. In a binary translator the input is represented binary (bits) and just have to be decoded.

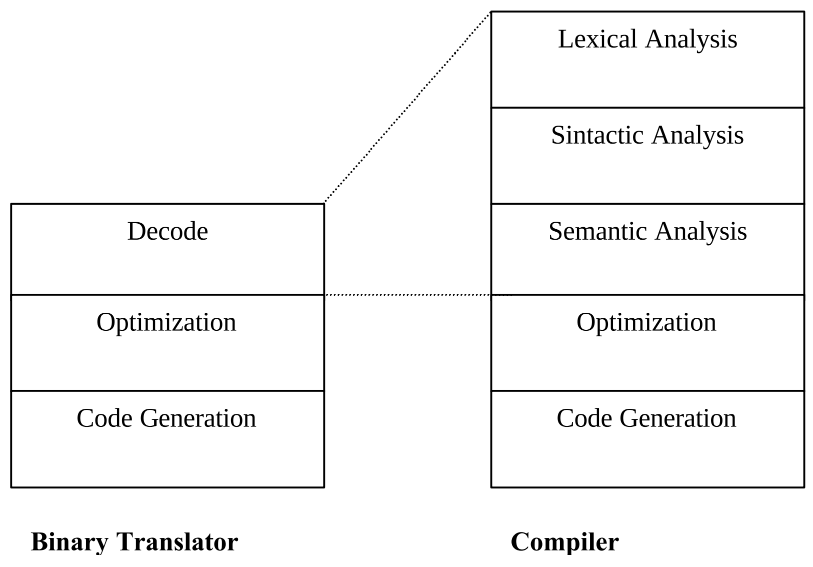


Figure 37. Binary translator and compiler phases.

Therefore the first main difference between compilers and translators is the first phase: the decoding for a translator and the parse and analysis in a compiler. In fact we should be happy, decoding is far easier than parsing and analysing a text. In a binary translator the first phase is decoding and just means to get the binary code and represent it in a more workable form, mainly some kind of decoded structure, which will be more useful for the next phases.

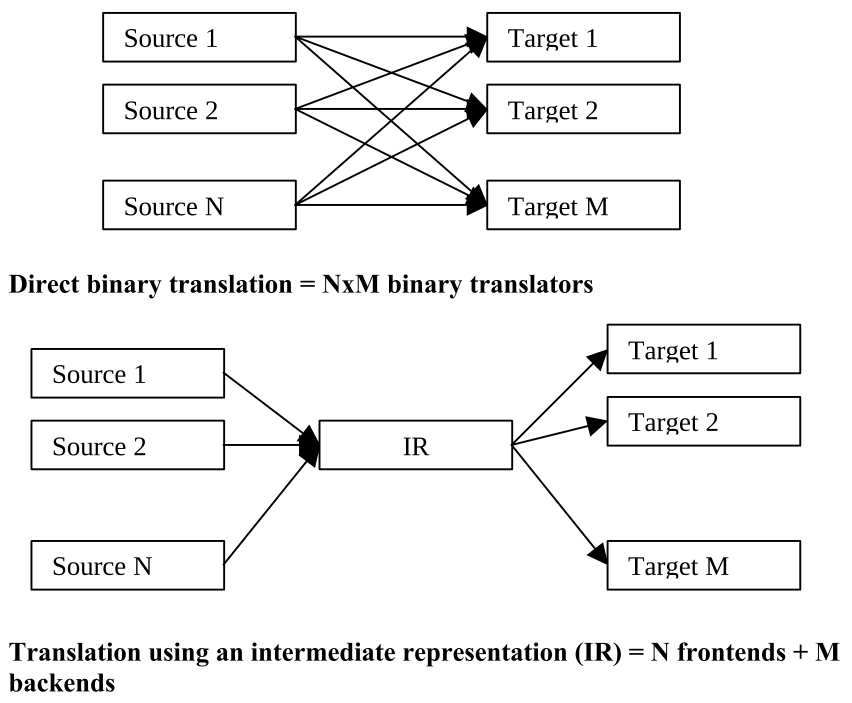

Source 1
Source 2
Source N
Target 1
Target 2
Target M
Direct binary translation = NxM binary translators
Source 1
Source 2
Source N
IR
Target 1
Target 2
Target M
Translation using an intermediate representation (IR) = N frontends + M backends

Figure 38. Binary translation with or without IR.

In a compiler the program is converted first to an intermediate representation (IR). The IR is the output of the first phases (parsing and analysis) of the compiler and the input for the next. There are different forms of intermediate representations (register transfer language or RTL, tree based IRs, etc) which are more or less useful for different purposes. The purpose of an IR in a compiler is to ease the transformations that the original code will suffer (transformations which use to be called optimizations) in next phases and to help with the portability of the compiler. An IR breaks the NxN graph of languages to machine code into a 2xN graph of languages to IR, IR to machine code.

In a translator some kind of intermediate representation will also exist. This representation could be just a structure with the decoded information and some additional fields (the one we could be output of the decode phase) or a full IR like the one used in a compiler. The choice of what kind of representation will be used must be done depending of the type of binary translator we are implementing. A full IR is useful for optimization but it is more expensive in time, in a static translator the time is not the main problem so a full IR would be the choice. In a dynamic translator the time for the translation is limited, then a more limited IR would be used, it will be depend upon the kind of optimizations we will perform how limited (or extended) the IR will be.

The choice of an IR is also related with the portability of the binary translator. In some cases if the translator is going to be implemented for more than one source/target machine it will be interesting to divide the process of translation in three phases: a fronted which deals with the source code, a common part which works with the IR (transformations/optimizations), and a backend or code generator which deals with the target code. The common part work with the same IR and for each source/target machine a new frontend or backend is implemented (the same it happened in a compiler). An IR can also be a virtual machine language which could be used as an interpreted code for those machines in which the frontend is not implemented (Ardi’s Executor [5]).

The IR (or whatever we have) is used in the next phase. This phase and the next are very similar in both a compiler and a translator. In this phase the code suffers from some transformations. These transformations use to be called optimizations. The main difference between a translator and a compiler is the number and type of such optimizations. It also differs between the different types of binary translators. In a static translator and a compiler it can be applied time consuming optimizations which use graphs and extended analysis of the code. In a dynamic translator de number and type of optimization is limited to the time we want to spend optimizing the code. A dynamic translator could apply different levels of optimizations to different blocks of code depending upon how often these blocks of code are executed. There are also optimizations (like function inlining and reordering) which are more likely intended for be applied in dynamic time than static.

Although the optimizations from a compiler can be used directly in a translator (they work in the same kind of information), limited by the time factor in the case of a dynamic binary translator, it could also be taken into account the difference between the code which comes from the first step of the compilation of a high level language and the code which comes from a real program. This code could have been programmed in assembly or have been already optimized because is the output of a high level compiler which performs optimizations. Sometimes this information can be applied to the translator to avoid some optimizations.

The optimizations performed in this phase are general optimizations, around the functionality of the code. In the code generation phase it could be performed architecture specific optimizations. Before passing to the code generation phase (or at the same time) it is performed the register allocation. The IR uses to work with virtual registers or temporal variables. These virtual registers must be assigned to real registers in the target CPU. This task is critical for getting a good code but also implies a lot of overhead (the basic algorithm is a graph colouring algorithm). In the cases that the time is a limited factor an algorithm which trades between time and quality will be needed (linear scan register allocation [11]).

The last phase in both a compiler and a translator is the code generation phase. In fact this phase should be almost identical in both. It must translate from the internal representation to the target CPU code. There are different manners of implementing this phase and depends because of the kind of IR used: matching trees, one to one translation, etc. At the same time there are performed architecture specific optimizations, taking into account the features of the target CPU: alignment, special instructions, the best ordering for the execution. After this phase the translated is stored for later use by the runtime of the binary translator.

The execution phase is performed by the runtime of the emulator. The equivalent in the compiler world would be to execute the compiled binary in the final system (usually through some kind of command shell). This phase differs for static and dynamic binary translators. In a static translator is just a loop which keeps running blocks of translated code, keeping the state, checking events, stopping the execution when it is needed and providing a background for the execution of the translated code. In a dynamic translator there is a loop which looks if the next block of code to execute has been translated or not and calls the translator. When the next block is already translated it is called and executed. It works in a similar way than a threaded code interpreter but in this case it is executing groups of instructions rather than just one instruction at a time.

Binary translation has the main advantage of a good performance when correctly implemented. We are avoiding many calculations which are not useful for the purpose of the emulation. The decoding is just performed once and the overhead due to the execution instruction by instruction is avoided. Now large blocks of source code are executed as target code in a single shot. Optimizations can also be performed which reduce the amount of executed code, for example flags are only calculated when they are really needed, many of the memory access can be directly translated avoiding to use all the memory map.

It has disadvantages too. One of the main problems is self-modifying code and dynamic generation of code (DGC). Self-modifying code and DCG have a great impact in binary translation because of the necessity of detecting them and providing a way to handle them. In a static binary translator this problem is hard to solve, most of times it needs a fallback interpreter or some kind of hack to the emulated program. In a dynamic binary translator self-modifying code and DCG can be handled but at the expensive cost of implementing detection and retranslating mechanisms. In a static translator it is also hard to deal with the distinction between data and code in the program, mainly because of the indirect jumps, mechanism for trying to solve this problem must be also provided.

The other main disadvantage of binary translation is the development time. Build a binary translator requires more efforts, more time and it is harder than implementing a basic interpreter emulator or a threaded code interpreter. The payback is that the binary translator will emulate the same source at around an order of magnitude faster (x10). However it depends in the source CPU and the target CPU differences and the way it has been implemented both the binary translator and the interpreter emulator.

Another limitation in binary translation is the portability. The binary translator works for a source CPU and a target CPU. The main problem is that it is generating code for a given CPU, it is the same we talked about assembly interpreters. In an interpreter emulator implemented in a high level language the emulator will run just one source CPU but it could be used in many different target CPUs. In fact the binary translator could be designed to be easily retargeted to many target (and source machines), for example UQBT [2]. This can be done at the cost of some performance lost in some of the phases of the emulation.

Types of binary translation.

There are basically two approach to binary translation as we already have seen: static and dynamic. A static binary translator is more or less something similar to a 'compiler' in which the input is a machine language. A dynamic binary translation could be a mixture between a dynamic recompiler and dynamic generator of code or optimizer. They share a lot of the main characteristics but they also have important differences.

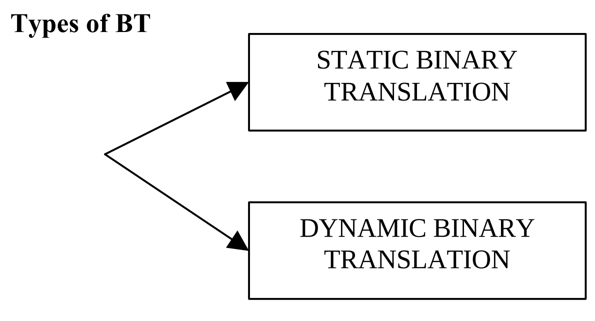


Figure 39. Types of binary translation.

The main difference between static and dynamic binary translation is the time. The time when the translation is performed and the time it can be spent in the translation. In a static translator the translation process is performed before the start of the execution of the program or even the emulator. Static translation has been used mainly to port applications from an old system to a new one without having to recompile the sources. This makes that the framework of a static translator is a program which receives and input program or binary from the source machine and generates an equivalent binary for the target machine. Then this translated binary could be executed alone in the target machine or could be executed using a loader which provides this a background for it. All the code of the source binary is translated and only once. After the translation phase of the process no more translation are performed and the output binary can be executed as many times as we want.

In a dynamic translator the translation is performed "on the fly" (or just in time, how it is called in the Java world), when the source code is being emulated. The translation of a block of code from the source machine is only performed when this code is first executed or when it has been already emulated (through an interpreter with profiling) a number of times. Next executions of the same block of code are implemented as a jump to the translation of this block. The translated blocks are stored for further use, although in some case those blocks could be erased from the translation cache to save memory. The translation is performed every time the binary is loaded in the emulator and executed.

If we compare this behaviour with a static binary translator we can se that it is a partial translation. A static translator must perform a full translation. A dynamic translator does not keeps any information nor translation after the end of the emulation therefore the translation must be performed every time the same program is emulated. A static translator most of times produces final results. The output binary is the already the full emulated program. In some cases though (FX!32 [12]) an incremental translation (as in a dynamic translator) can be performed. Then, however, the translation is kept for the next execution in some kind of database.

Static translators which do not keep the translation can also be implemented. They would work first translating the program to be emulated and executing it after it. It must be taken into account then the time spent in the translation. In a full static translator the time is not an issue and hours code analysis are allowed. In such a limited static translator the translation time should be limited to that reasonable for a binary load (for example from disk, with some initializations) process, that is, from a few seconds to one or two minutes.

Static translators are also more suited for performing any kind of expensive optimization through the code, including global optimizations, than a dynamic translator which works more in the scheme of a by block based translation. In the other hand dynamic translator are able to perform run-time optimizations, optimizations which can only be performed knowing how the code it is being executed (reordering of the blocks to avoid jumps, inlining). In the case of a static translator the optimizations are just limited by the amount of information about the code to emulate the translator can obtain. Using some kind of profiler to provide information to the static translator in a first phase would help to produce a better final code. In a dynamic translator the optimizations are limited by the time.

The reason because the idea of a dynamic translator works, and it is faster than an interpreter emulator is because most of the execution time in a program is spent in a few blocks of code (loops). It is the same idea with memory caches, most of the data and the code are reused, it exists locality in time and in address. In a dynamic translator that means that the time spent in the translation of a block must be compensated by the speed up produced because of the translation. That is, in a global view, the translated blocks must be faster enough and be executed enough time to overcome the overhead due to the translation.

From this basic classification there are other minor classification which depend in the manner the translation is performed or how the translator work. For example, as we already said, there are different ways of performing the translation either using or not an intermediate representation. The number of optimizations is diverse. The translation, in a dynamic translator, can be performed in the first pass or only after a number of passes (just translate – optimize – those blocks of code which are more frequently executed). This means also that there could be standalone translator or combinations of translators and interpreters/profilers. This happens in both static and dynamic translator for different reasons.

A combination between a static and a dynamic translator can also exist. It could be a dynamic translator which tries to translate as much code as possible at the start up, and only when new code is found it is used the its dynamic capability. It could be also used for performing a lazy translation of most of the code at load time, and retranslate the more executed blocks later trying to optimize the translation.

Another classification could be if the translator is able to deal with self-modifying code or any other form or new code which must be translated on the fly. Even a dynamic translator would not be at first able to support self-modifying code. The self-modifying code must be detected first and that it is an expensive task which it should be avoided if possible. Even more, self-modifying code mean retranslating and if it is very frequent the overhead of the translation could kill any chance of getting a good performance.

Basic concepts.

The basic concepts of binary translation come either from the compiler world or from the emulation (interpreters world). Since the translation, optimization and code generation phase in a binary translator are equivalent to the same phases is a compiler the same terminology and techniques can be applied. In a dynamic binary translator, though, the range of those techniques is limited to the amount of time it can be spent in the translation of a block.

We have already talked about intermediate representations. An IR is a kind of representation of a code or algorithm. There are many kinds of IRs, some are based in the compiler theory, like tree based IRs, other are more nearer to machine language. The IR is as an intermediate step between the source code and the target code, either for portability or for aiding in the process of optimization.

In the compiler theory the basic unit for code optimization is the basic block. A basic block is a sequence of instructions which only have one entry point (the first instruction) and one exit point (the last instruction). The idea is that such a piece of code can be modified, while maintaining its algorithmic meaning, without affecting the rest of the code. Therefore many of the optimizations (local) are basic block based. Further optimizations (global optimizations) are performed between basic blocks, working with their interrelations.

The basic algorithm for building the list of basic blocks of a function (or any other piece of code with an entry point) is based in searching the basic block leaders. A basic block leader is the entry point/first instruction of the basic block. The algorithm for finding the leaders is:

1) The first instruction of the function (entry point) is a leader.
2) The target instruction of a jump instruction is a leader.
3) The instruction after a jump instruction is a leader.

After the leaders have been found the basic blocks are built between a leader and the next instruction before the next leader in the list of leaders.

In static binary translation basic blocks will be the basic translation units. In dynamic binary translation they could be or could not be the translation unit. At run-time is hard to determine if a given block of code is really a basic block (in the future a jump instruction could jump to middle of the block). With profiling and multiple passes it could be built an arrangement of blocks which could be trusted to be basic blocks. We will keep, though, the idea of basic block while translating dynamically. Another question is that in dynamic translation sometimes is better to perform instruction by instruction translation or translation of blocks bigger than a basic block, for example for loop optimization. In a static compiler this last kind optimizations are performed in a second phase of optimization between the basic blocks.

Examples.

Although in the last years the interest in binary translation is growing this technique is not new. The main interest in our days is to dynamic binary translation to provide a software layer over the hardware layer to provide ISA compatibility for old CPUs in new ones. However in the past it was static binary translation the main interest.

Static translation was successfully used in the transition from CISC CPUs to RISC CPUs as an alternative to source code recompilation while porting applications from the old system to the new one. Digital (DEC) was one of the companies which made more efforts in this direction providing with successful binary translators from the old VAX to the new Alpha architecture. Digital (later bought by Compaq) also developed a MIPS to Alpha translator and a x86 NT to Alpha NT static translator with profiling: FX!32. We will talk later about FX!32 because it introduces interesting concepts.

In the academic world there have been also a lot of research on the topic. From the old MIMIC translator, Shadow and the more successful Shade for system simulation[8]. Also the Embra [13] dynamic binary translator which was used in the SimOS framework. The main interest in binary translation research in 80's and 90's seem to be related to improve the performance of simulators. There are also some efforts about code portability around static translators. Nowadays projects include Dixie and UQBT [2].

In last years the interest has moved to a dynamic approach. The release of the Transmeta Crusoe [14] VLIW CPU which uses a layer of software called Code Morpher to execute x86 code shows the new trend. Transmeta uses a software approach for x86 compatibility which differs from the hardware approach used before (Intel Pentium, Pentium PRO, II, III and IV; AMD and others). Transmeta mixes interpreting, dynamic translation, a long range of optimizations (hot spot optimization) and hardware solutions to speed up the translation.

In the same line of Transmeta's Code Morpher IBM has been developing DAISY [15] a dynamic binary translator from PowerPC and AS700 to a generic VLIW architecture. DAISY shares most of the techniques with Code Morpher but it is open source and can be freely downloaded and modified. IBM tries to exploit the explicit ILP feature of the VLIW CPUs to provide faster translations. IBM has been researching in similar topics for Java to VLIW translation.

Other modern projects which can be found are the dynamic optimizer Dynamo [16] which tries to perform dynamic optimizations for a HP CPU. HP is also researching in a HP to IA-64 translator [17]. Another project about dynamic binary translation is UQDBT the dynamic version of the UQBT project. Other researches and products can be found around the execution of Java bytecode, for example Java Hot Spots and similar dynamic compilers for Java bytecode.

Other commercial products which are related are Virtual PC (a PC emulator for MAC) and VGS (a PlaySation emulator for PC and MAC) from Connectix. Ardi's Executor is a MAC emulator for the PC. Apple provided different emulators, for supporting old applications, (DR Emulator) for the MAC when they changed from the Motorola 68K architecture to the PowerPC architecture (Motorola/IBM) [18].

In the emulation scene (emuscene) binary translation has been used to permit the emulation of the more modern systems: PSX, Nintendo64. The more used technique is dynamic binary translation which is called 'dynamic recompilation' or 'dynarec'. Another technique used in the emulation of the more modern consoles is High Level Emulation (HLE) which tries to perform the emulation at the library/system call level rather than at the hardware level. All the static binary translators provide an OS API emulation (UNIX in most of cases).

2. Static Binary Translation.

In this section we will introduce the basic concepts around static binary translation. Static binary is not very useful for the machines we are our main targets for emulation (videoconsoles, video arcade machines) because of its lack of flexibility and other reasons we will see later. It is hard to find static binary translator for those systems so most of the information will come from the corporation and academical world. Static translation can still useful for learning the basics of binary translation and in some situations, with some modifications, could be a good alternative to dynamic translation. This section will just talk about the basic points and it will not go in a deep study of this topic.

Basic Algorithm.

A real static binary translator is more nearer to a framework or a tool set than a single program. This is even truer in multitargetable binary translators like Dixie and UQBT where the process of implementing support for a new architecture is automated. The process of translating a source binary from a given system to another system requires most of time some kind of feedback. The multipass approach, using profiling for gathering useful information will lead to a better translation and to find hidden pieces of code. We will see a basic algorithm which shows how a static binary translator should work. Many of the phases of this algorithms are optional and can be modified or implemented in a different manner in each specific static translator.

```
translate all
        search basic block (+ profiling)
        for all basic blocks
                translate to IR
                optimize block
                store block
        global optimizations
        for all basic blocks
                generate target code
                store code

executeall/executeN
```

Figure 40. Static translation algorithm.

A static binary translator has two separate phases: the translation and the execution. They can be studied separately. The first phase, translation, is the more interesting to study and the harder to implement. The execution phase just has to deal with the connection between the CPU emulation (more exactly code emulation) and the other parts of the emulator (graphic hardware, sound hardware, OS, etc.).

The first step in the process of translation is to obtain as much information as possible. The basic information we will need (but of course the code) is the list of basic blocks in the code we are going to translate. This list can be obtained at the same time the source code is being decoded and translated to the IR, or it can be done separately, either using a profiler to build it in a first execution or just performing a partial decoding (jumps, branch and call instructions). If the static translator has mechanism for profiling the source binary before the translation this information will be also very useful (for example frequency of the taken branch, register usage, etc.) in the optimization phase.

After this basic information has been obtained (or at the same time) the source code is decoded and translated to the IR we will use for the next steps. The decoding can be performed in any of the ways we saw in the chapter about interpreters. Now the time is not a limitation so multiple switch level and conditional statements can be used freely to perform an easier decode. The decoded data will be organized in basic blocks for the optimization phase.

The optimization are modifications in the code (in this phase it will be in the form of an intermediate representation) that, while maintaining the same behaviour, minimise the number of instructions or the time spent in their execution. As it happens in a compiler the optimization has two levels: local optimizations and global optimizations. Local optimizations are performed in each basic block. Global optimizations are performed between basic blocks. The local optimizations can be performed at the same time decoding it is being performed, as soon as each basic block is found. It can be also performed after all the code has been decoded to the IR and grouped in basic blocks. Global optimizations are performed after the local optimizations changing basic block orderings and modifying their structures (mainly optimizations with blocks).

After the code has been optimized it must be translated to the machine code of the target CPU. This phase is called code generation. Before the code is going to be generated the virtual registers (or temporal variables) used in the IR have to be translated to physical registers and addresses in the target CPU. This phase is called register allocation. Register allocation algorithms are based in graph colouring algorithms. The code generation phase can be implemented in diverse forms, as we will see. For example with a tree-matching algorithm: branches and subtrees of the IR (which is represented as a tree/wood) are translated to target instructions. At the same time can be implemented optimizations which are architecture independent. The translated blocks are stored somewhere and output to a file at the end of the translation.

Later when the user would want to execute the translated binary it will execute it directly (if the run-time support program is included in the output binary) or it will execute it through a special loader. This loader or the run-time support code will perform all the additional tasks needed to run the binary in the target machine. It will translate the source system calls into the target system calls, it will emulate the hardware devices, the interrupts and exceptions and the timing. In some cases it will also include a fallback interpreter for non-translated code (non-detected indirect jumps, self-modifying code, dynamic code generation).

Since most of the machines we want to emulate (videogames computers) can be considered as real time machines we will have to reflect this characteristic in the translation. In an interpreter CPU emulator this is not a problem because as we saw just adding a cycle counter and update at each instruction is enough to control the emulated CPU time. In the case of a translation we will have to add checkpoints where the timing will be updated. The run-time will be the part of the static translator which will have to control that the emulated time is the same that the real time. It will go into an idle state if the emulation is too fast in the target machine. For information about real time translators check TIBBIT papers [19].

Intermediate Representations.

The intermediate representation (IR) has the purpose of helping in the optimization process and code generation. Some kind of IR is always needed because the source code in binary format is not the best representation with which perform modifications and translate to the target machine code. The difference is the extension and type of the IR used.

A simple IR could be a decoded structure which would contain all the information of the source code binary representation in an easier to access format. It could also contain fields which would help in the optimization and code generation phase.

From this simple IR it can be developed other IRs which are more nearer to the idea of intermediate representation used in compiler theory. In compiler theory different kind of IRs are used, some more similar to mathematical representations of the algorithm (tree based IRs), other more nearer to the final machine code, for example like RTL (Register Transfer Language). For a more intensive study of the IRs used in compilers use the reference books about compilers, for example “The Dragon Book” by Aho and Ullman [20].

Another approach which could be used in translation is to use a virtual ISA as intermediate representation. This have the benefit that it could be designed a fast virtual machine which could perform a fast emulation of this virtual ISA. The static translator could be retargetable. For those target machines which would not have a backend (code generator) implemented or just partially implemented (not all the source code blocks or instructions could be translated) the translator will output code translated to this virtual ISA. Then this virtual machine code will be interpreted with a fast portable threaded code interpreter. This approach is used in Dixie retargetable static translator [21] and in Ardi’s Executor dynamic translator [5].

The decision about using one or another IR depends upon what kind of translation we will perform. As we have said if we want a portable static translator the virtual ISA approach could be the more interesting. A virtual ISA has also the benefit that, if it is well designed, it can store more information about the original code than the usual IRs.

This is one of the points in the decision between a kind of IR and another. There are two points where the IR has to be strong: the optimization phase and the code generation phase. For the optimization phase it needs to help in the process of code modification, to find potential points for optimization and to help exploit these optimizations. For the code generation it has to help in the process of choosing the target CPU instructions. It has also to keep enough information from the source code to perform the better possible translation between the two CPUs (for example if they both have common complex or non-standard, or non-RISCy, instructions).

Local and global optimizations.

Code optimizations is the second point which makes translation a lot of faster than interpretation. The first one was the overhead due to the multiple decoding (which could be reduced with a threaded code approach) and the instruction by instruction based emulation. With optimization, most of time very simple ones, the emulated code can be even faster.

The main point in optimizing the source code is to avoid some calculations which are not needed for the final result of the code. One question arises here. In a normal compiler the optimizations are performed to the conversion of a high-level language code into an intermediate representation. This conversion or compiling is performed without any intention of generating good code (at the first step). Therefore the optimization opportunities come from two points in a compiler: the first step of compilation which produce suboptimal code and the level of optimization in the high language level code.

In a static translator the first one is limited to the overhead due to the translation from the source code to the IR. In some cases this overhead will be greater and in other cases smaller. Something similar will happen in the IR to target code translation as it happens in a real compiler too. These ultimate optimizations are called architecture specific optimizations.

The next optimization option for a translator comes from the optimization level of the source code. This level can be diverse, sometimes we will found very optimized binaries and other times non-optimized ones, so the gain here is very variable.

The main optimization point is some calculations which are performed by the emulated CPU which are not used. This kind of calculation used to be side effects or specific to the behaviour of the source CPU which either the programmer or the compiler were not intended to use in the program. Those calculation can not be avoided in the real CPU because that it is the way it work but it would be useless to translate them in the target CPU. The main example of those calculations is flag and condition code calculations. As we have already said flags are very hard to calculate in some architectures. The interesting is that although many instructions modify those flags it is less common that those flags are read and used. The translation will be performed so only the used flags will be calculated.

The typical basic block based optimizations from the compiler theory can be also applied in this case, static translation, because the time is not a problem. These optimizations include: common subexpression elimination, dead code elimination, temporal variable renaming, instruction swap, algebraic optimizations, etc. More information about these optimizations can be found in any book about compiler theory [20].

All those optimizations we have talked about are local optimizations which are performed in basic blocks. After this first phase of optimization it can be performed another which will perform global optimizations. Global optimizations are based in the relations between basic blocks. The main interest of those optimizations is to speed up loops and therefore optimizations like loop unrolling or to keep the invariant out of the loop are applied.

The only limitation of such optimizations in a static compiler, because the time is not a problem, is the fact that it must be maintained some kind of time control upon the translation. And there must be points where the translation must stop and return to the run-time. In those places the state of the emulated CPU must be restored and must reside in the CPU context structure.

Global optimizations could use profiling information to reorder the translated code to avoid jumps and get a better use of the code cache. These kinds of optimizations use to be more useful in dynamic time because the behaviour of a program can change but they can also be useful in static translation.

Code Generation.

After all the modifications to the code are performed and all the possible optimizations implemented it is time to begin the translation to the target machine. This phase is also very important to create a good translation. In this phase the final output code will be created. The translations should be enough fast and perform a good use of the registers and the instruction set of the target CPU.

The first step in the process of code generation is to allocate real registers and or memory variables for the temporal variables or virtual registers used in the IR representation. In an IR it is common to work as an infinite number of registers were available. Although as our source is already a machine code language with physical register allocated we could keep this information in the conversion to the IR code. This could reduce a number of optimizations which could be done in machines with more registers than the emulated machine.

There are different approach and the number of registers in the emulated and target CPUs have an impact on them. It will be different in the emulated CPU has less registers than the target CPU (for example in many of the CISC to RISC translations). In this case a static allocation where each emulated register has an equivalent target register is the best. The other target machine register would be used as scratch registers. In the opposite case, when the number of registers in the target machine is more reduced than in the emulated machine is a bit hard. Some mechanism for doing a good use of the few target registers should be used.

In this case, and when the number of register is similar in both the emulated and target CPUs, the choice could be a register allocation algorithm with perhaps some static allocation. In a static translator the time is not a problem so a slow register allocation algorithm can be used in any case. Some of the emulated register use to be allocated statically because they are frequently used (for example the PC or the SP) but the other could be allocated or not in register as they were being used in each basic block. A good approach used by DEC translators [22] is to allocate statically half of the registers (the more special registers and the more frequently used, for call passing arguments for example) and allocate dynamically the others.

There are different algorithms for register allocation. They can be found in books and papers related with compiler theory. The basis of the register allocation algorithm is a graph colouring algorithm. There is also a phase of spilling, when the number of registers needed in the basic block is larger than the number of available registers some of the already allocated registers must be deallocated. Good register allocators are expensive in time.

After the register are allocated, and some times at the same time. The translation to the target CPU code is started. The process is performed with pattern matching techniques. One or more IR instructions are translated to one or more target CPU instructions. The matching can be performed in different ways and the type of IR used determines the algorithm. A tree based IR would use a tree based matching algorithm. In a compiler theory book can be found example of code generation algorithms.

The basic algorithm for translate from a source representation to a target representation is a 1 to N translation. One source instruction is translated to one or more target instructions. This can be used with the IR but it has some problems. The main problem is that the IR has to be very similar to the target CPU to produce good code and therefore most of the instructions could be translated one to one instruction. In an IR which hides most of the work of a real CPU multiple IR instructions should be translated to a single target CPU instruction (for example the typical tree matching algorithms in compilers).

After the first translation is performed to the target code, usually just to an expanded or decoded representation, it can be performed another optimizations. These optimizations are architecture related and try to take advantage of special characteristics of the target CPU which are not present in the IR code. One of the types of optimization which can be performed is peephole optimization. This just stores the last generated instructions in a buffer and tries to find sequences of instructions which could be coded in faster way in the target CPU. Other optimizations are related to the way the jumps work (absolute or relative jumps, delay slots, penalties), memory alignment question, reordering and scheduling of the instruction in superscalar CPUs and others. If it was being used a decoded representation a last step translates this representation to binary code for the target machine.

Run-time.

The run-time is the code which provides an environment where the translated code can run in the target machine as it was the source machine. The translation just performs the part of the emulation related with the CPU, more exactly with the emulation of the program code to be emulated. This does not include all the other hardware which is present in a computer. In most static translators it does not implies either the emulation of the base and support code: libraries and operating system. The run-time must provide mechanism for emulating the hardware devices, if they are accessed by the emulated program, and the OS and libraries if they have not been translated too.

There are difference between the kind of programs and machines which are being emulated using static translation. Most of the examples which can be found perform just a translation of user level code. Then the system calls, some library calls (shared libraries) and the possible access to hardware devices are tracked and converted by the run-time to the equivalent service in the target machine. For example in Digital's FX!32 (x86/NT to Alpha/NT) the system calls, accessed through DLLs (shared libraries), are captured. Then it is executed a function which translate the arguments and results to the target argument format. These functions are called 'jackets' in the translator. In most Unix to Unix translators a similar mechanism is implemented for emulating the system calls. In this kind of translators the emulation of the hardware is unnecessary because all the access to the hardware is performed through the OS.

In the case that the emulated machine does not have an OS, we want to translate the OS or the user level programs access the hardware a more complex run-time must be implemented. In those cases this part of the run-time will be similar to the code of a normal emulator: a loop which executes N emulated instructions and then emulates the other hardware devices. The access to the hardware (IO ports or memory mapped IO) are emulated through memory handlers which could be inlined in the translation (that is another optimization which can be performed with translation) or performing memory access through a function. This also implies that the translation will have to control the number of emulated instructions or cycles.

In the case that the emulation must maintain the same time behaviour that in the real machine we will talk about real time systems. Most of the static translators are not implemented for real time systems and their primary goal is to execute the emulated code and programs as fast as possible. TIBBIT [19] introduces the concept of real-time systems static translation. For maintaining the time the translation must be divided by checkpoints where it is counted the emulated CPU spent time (number of cycles). If the number of cycles reaches a given counter the translation jumps back to the run-time main loop which decides either to emulate other hardware, other CPUs or other code translations. After the run-time main loops perform those entire periodic task is time to check if it must return to the translated code. It checks if the real time spent in the target machine is the same that the emulated time. If not it waits until the time has been reached.

The frequency of the time checkpoints and the slot of time (number of cycles) to spent in translated code will be different in each emulated machine. A more exact emulation of the time means a more expensive emulation. There is a lot of overhead due to the additional code for time checking and the transition from translated code to the run-time loop. At a checkpoint exit the state of the translation will be saved in the CPU context structure. This also limits the scope of the optimizations in the translated code which can only be performed between two checkpoints. It must be found the exact point where the accuracy of the time is enough to perform a correct emulation and the performance loses is the less.

The run-time will have also to deal with the accuracy of emulation of interrupts and exceptions. The problem with translation and interrupts and exceptions is that the interrupt could happen in the middle of a translated block where the equivalent stated in the real machine could not be recovered. There are different mechanisms for avoiding these situations but are expensive. Commit and rollback mechanism and a fallback interpreter are the mechanism used for deal with this problem. If such a level of accuracy is not needed it is better to avoid the implementation of this mechanism.

The translation of the binary is organized in blocks of code. We will try that all the translated blocks will connect each other, when a translated blocks ends it jumps directly to the next translated block. The another option would be to return back to the run-time main loop, perform a search in a table for the next emulated address and jump to the translated block. In some situations it is needed, though, to return to the main loop for performing that search for the next translated block or for executing non translated code. Indirect jumps, although the mechanism could be inlined in the block translation, will return to the main loop. In the case of an indirect jump if a translated block is not found the run-time will either exit and output an error and profiling information or will start an interpreter which will emulate this part of code which is not translated (the original code must be keep somewhere for being used in those cases). The same mechanism works to deal with self-modifying code.

Implementations.

The problem with static translation is that it does not have enough flexibility for emulating most of the machines which are interesting in the emulation scene (videogame computers). One of the problem is the fact those machines need an accurate emulation of the time, they does not use to have an OS to support the user programs, and the other hardware must be emulated as well.

Some of our target machines, the personal microcomputers, could be implemented using some kind of static binary translation if the emulated programs were working over an OS (sometimes they do not use the provided OS). But most of the videogame consoles does not provide a good framework for implementing a static translator. Other problems with using static translation with those machines are the legality of keeping modified (translated) copies of the games binaries. The fact that it could exist hundred or thousand of different games, and to keep a translated copy will be wasteful (when the original copy could work with another emulators). The static translation uses to need some kind of feedback by the person performing the translation and most of users would not have enough knowledge level.

In any cases most of those problems could be solved, but as there is a more flexible approach (dynamic binary translation) static translation has not been very used. Some Nintendo 64 emulators use some kind of static translation combined with interpreters and library and OS emulation (High Level Emulation or HLE). It can be found some implementations which use static translation or a modified form, translation to a high level language and recompilation (decompilation), for emulating specific arcade machine games.

A good alternative for static translation in videogame emulation would be a load time translator which does not keep the translation after the emulation ends. The emulation is performed when the game binary is loaded and then it is emulated through the execution of this translation. When the emulation is finished the translation is erased. This also works maintaining a small database with hints and profiling information which would help to perform a correct and faster emulation. This approach is a bit similar to the used in some Nintendo 64 emulators.

In the emulation of arcade machine games it could be also interesting to use static translation. The number of binaries is very reduced, less than a hundred in any case, and it is easier to discover the settings (profiling information) for each of the games to be emulated. There is also a good separation between code data and the graphic and sound data (separate ROMs and even boards). And in each time there is a single program being emulated.

Most of the information about real static translators can be found from academic researches and commercial products. Static translator has been used as good alternative to source code recompilation for porting old applications to new architectures. The first example of using static translation was in the 1987 when source HP3000 programs were translated to the new HP architecture.

Later it has been used by Tandem (1991)[23], Digital (DEC) and other companies. Good sources for information are the open papers about Digital static translators. DEC implemented two static translators for translating from their old VAX architecture, from MIPS (R3000) and from SPARC architecture to the new Alpha architecture (1992, VEST and mx) [22]. Those translators work at a user level and provide a run-time which emulates the VAX and ULTRIX original OSes. The run-time also provides a fallback interpreter for those situations where there is a problem with running a translation or the translation does not exist.

Another interesting static translator implemented by DEC is FX!32. FX!32 is a framework which executes transparently x86 code for the Windows NT OS over an Alpha machine running NT. This emulator combines a fast interpreter which performs profiling at run-time with a background static translator which translate the profiled code and stores it in a database. The system attaches to the NT system and detects when a

TIBBIT [19] is a static translator which tries to emulate a real-time system. It is useful about how the time must be implemented in a static translator and the accuracy level and overhead it has.

Other interesting projects are UQBT and Dixie [2] [21] which are tool sets for implementing retargetable static translators. UQBT uses a more nearer to a compiler approach and the IR is a RTL. Dixie uses a virtual machine approach. The IR is a virtual machine code (a bit similar to the Alpha ISA), 128-bit opcode length which supports big and little endian operations, any data size (8, 16, 32 and 64 bits) and vector instructions. In a first step the source code is translated to the virtual machine language. This translation can be executed through a virtual machine (Dixie VM). At a second stage the Dixie code is translated to the target machine code and executed through the same DVM. If some blocks of code are not translated to target machine code the DVM will execute the Dixie code. Both static translator frameworks provide special compilers, languages and tools for describing the source and target machines: instruction sets, characteristics of the CPU (endiannes, data size, registers), the OSes, the binary formats.

3. Dynamic Binary Translation

The alternative to static binary translation is dynamic binary translation. Although in fact dynamic binary translation can be considered slower than the static translation approach it has many advantages and it is very interesting for the kind of machines we want to emulate. Dynamic binary translation is slower (theoretically) than the static approach for two reasons. The first reason is that the translation is performed at run-time so to the time spent emulating the code we have to add the time translating it. This implies that the translation must be performed as fast as possible. It also means that the benefit from executing translated code must overcome, by far if possible, the time spent translating code. The second reason derives from the first, as the time spent in the translation must be the less possible most of the optimizations from a static translator can not be performed. This reduces the quality and performance of the translated code and thus of the emulation. In the other hand there are some optimizations, as code reordering and inlining, which can be performed easily and with better information and result at run-time.

But dynamic translation has important advantages over static approach. Static translation works better for standalone binaries or programs which are executed in a machine using a full featured OS. Those binaries use (although it is not mandatory) to have a structured format which differentiates between data and code. Those binaries do not access the hardware but use system call instead. Basically static translation works well for user level programs. The system level is emulated through the translation of the OS system calls to the target OS or hardware. And most of time they usually do not use nasty programming techniques as self-modifying code or dynamic generation of code. Those are the main problems with normal static translation: it just runs user level code, it needs a good separation between data and code and does not support easily self-modifying code or dynamic generation of code. Dynamic binary translation solves all those problems.

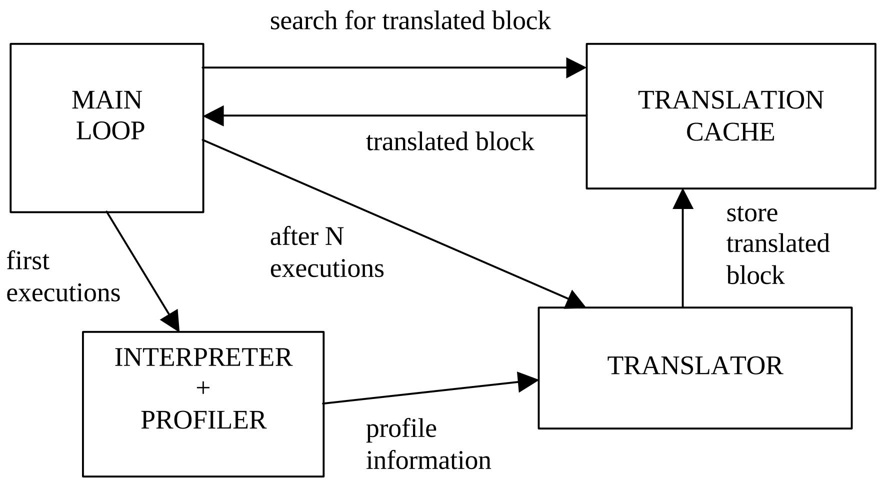
Dynamic binary translation can easily emulate all the code in a given machine, from BIOS code and OS code to user level code. It works like an interpreter emulating the code as it is being found and, in fact, in many cases it works with an interpreter for the first emulation pass. Only the code which actually executed is translated (in some cases just the code which is frequently executed), in a static translator all the code must be already translated at run-time (or it should use a fallback interpreter for the new code). This helps to handle easily indirect jumps which can be a problem in static translation. It also means that any kind of dynamic generation of code, code load and self-modifying code can be handle. But it will have the cost to detect this new code, throw away the previous translation and start a new translation.

Dynamic binary translation can be implemented so it provides a software layer which hides the target machine and shows the interface of the emulated machine. This way the emulated code can be executed transparently. This is the kind of emulators we are interested. The videogames machines and home microcomputers which are our main target does not have an OS or it is very limited. The program or game accesses directly the hardware and many times they use self-modifying code. Dynamic binary translation (or dynamic recompilation, dynarec in short, as it is known in the emuscene) is also a best option over static translation because it is not easy to translate the thousand of different programs which exist.

Basic algorithm of dynamic binary translation.

The basic algorithm for dynamic binary translation resembles the standard interpreter (or maybe more nearly the threaded code interpreter) algorithm. It just adds a phase which perform the translation of new code and a phase where the translated code is executed. There are two basic ways a dynamic translator can work: as a standalone dynamic translator or as an interpreter with a dynamic translation and optimization phase. We will see the two algorithms.

The algorithm for the interpreter with a dynamic translator is as follows:



Dynamic binary translator with interpreter + profiler.

```
while(!end)
{
   translatedCode = ReadTranslationCache(PC);
   if (translatedCode != NULL)
   {
      blockInfo = ReadBlockInfoTable(PC);
      if (blockInfo == NULL)
      {
         blockInfo = CreateNewBlockInfo();
      }

      blockInfo.executions++;

      if (blockInfo.executions > MAXEXECUTIONS)
      {
         translatedCode = translate(blockInfo)
         (*translatedCode)();
      }
     else
      {
         runInterpreter(PC, blockInfo);
      }
   }
   else
   {
      (*translatedCode)();
   }
}
```

Figure 41. Dynamic binary translation algorithm (interpreter-profiler based).

Just the code which is frequently executed is translated. The rest of the code is executed through an interpreter. The algorithms start searching a translation of the code in the address pointed by the emulated PC in the translation cache (a structure which keeps the translated blocks of code). With the translation cache exists a structure which will associate the original address of the translated blocks with its address in the target machine. Another characteristic of this structure is that it must be accessed as fast as possible because the transition between blocks must be the fastest possible. It is usually implemented using different kinds of hash tables. Some times, if the address range where it can be found code is small enough, the structure performs a direct map one to one for each code address for its equivalent (if it exists) translated code address.

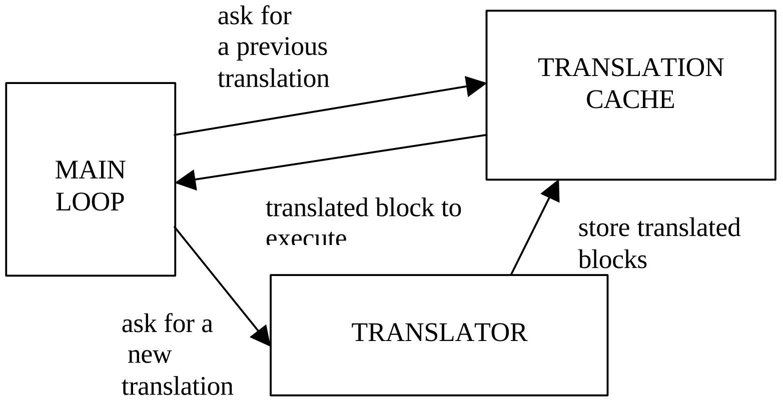
If a translation for the actual PC address is found it is executed. If there is not a translation for the actual address this code will be executed through the interpreter. The first step before emulating this code is to determine if this it is the first execution of the code. If it is the first a new structure is allocated to store information about the block of code which starts in this address. If there was a previous execution this structure is retrieved. This structure will store profiling information about the block of code. This information includes the entry point address, all the possible exit address, the number of previous executions of the block and any other profiling information that the interpreter would be gathering. Before the interpreter is called the counter of the number of executions is incremented.

If the counter reaches a limit value the translator is called and the block is translated to target code and then this translation is executed. If the counter is below the limit of executions for performing the translation the code is just executed through the interpreter. The interpreter can be implemented with any of the techniques which were discussed in chapter 3. The idea is that the interpreter will just run glue code, code which changes frequently (self-modifying code or dynamic code generation), any code which is rarely executed. The code which is frequently executed (loops) is translated and the execution through the interpreter avoided. The interpreter must also get profiling information. The main information is about the execution flow changes (jumps, function calls and returns) which are needed to build the blocks (and create basic blocks) and keep the frequency of execution of each piece of code. Another information could be used to improve the translation optimizing the translated block to the observed behaviour of the emulated code.

The translator will perform the translation to a form of intermediate representations, any optimizations that the dynamic translator implements and it will generate target machine code. This code will be stored in the translation cache. The translation cache could be implemented to keep for ever (the time of the emulation) the translation or it could have implemented a garbage mechanism which would throw away non used translations. Such a mechanism is wasteful and keeps some optimizations from being used so most of the times should be avoided. If the system could be loading new code frequently (an emulated computer which runs for a long time) this mechanism should be provided, for single program emulation is not necessary.

The translated code and the interpreter will be the responsible of keeping a correct emulated CPU state before returning the main loop. They will also have to update correctly the cycle (time) counters if the dynamic translator must be time limited.

The algorithm for a pure dynamic binary translator is simpler:



Pure dynamic binary translator.

```
while(!end)
{
        translatedCode = ReadTranslationCache(PC);
        if (translatedCode != NULL)
        {
                translatedCode = translate(PC);
        }

        (*translatedCode) ();
}
```

Figure 42. Dynamic binary translation algorithm (pure).

The different functions are equivalents to the ones in a dynamic translator with a first pass interpreter, but now the interpreter is gone and all the emulation is performed through translation. This approach has advantages and has disadvantages. The interpreter saves the dynamic translator from translating code which is not going to be used again. This can be very useful if the code is abusing of self-modifying code because each time the code is modified it must be retranslated. The interpreter also provides useful information for the translation like the path followed by the code. The advantage that a pure dynamic translator has is that it reduces the time of development because it does not need and attached interpreter. In a pure dynamic translator the number of optimizations performed to the translated code will be smaller because the translation must be faster.

This makes pure dynamic translators more suited for emulators with a low level of complexity and less ambitious. The final performance of a pure dynamic translator and a dynamic translator with interpreter could change in each case, but it is expected that a well implemented dynamic translator with interpreter and a full set of optimizations will be faster. Pure dynamic translators are very suited for the kind of projects we study in this documents (microcomputers and videogames computers). We want to emulate those systems as fast as possible but the emulator project must be kept in a reasonable level of difficulty because most of times it will be implemented by just one or two programmers. The fact that most of the work of the emulated machine is performed by external hardware to the CPU (sound and video hardware) makes also that emulation of the CPU does not have to get all the effort in the emulation. Most of times, if the CPU is well implemented, the more expensive part in the emulation will be the video and sound part. A full featured dynamic translator with interpreter, profiling and large background optimizations would have little impact in the final performance and would increase largely the development process.

In the pure dynamic translator the first step is to search a translation for the block of source code which begins at the address pointed by the emulated PC. If a translation is found it is executed directly and the translated code will continue in execution, usually branching directly from one block of translated code to the next, until it reaches the time limit (if the translation must be time driven) or an address which can be resolved by the translation (indirect jumps) is found or there is not translation for the code in this address. If a translation for the code does not exists it is performed the translation at this time and it is executed. The translation uses to be performed as fast as possible with minor optimizations or no optimizations, for example just redundant flag calculation elimination and similar peephole optimizations.

In any case a pure dynamic translator could be implemented with all the functionality of a combined dynamic translator-optimizer with an interpreter. For example the first translation could be performed as fast as possible (just code template generation for example) with some instructions added for performing profiling. Further executions of this fast translated block would gather information about the execution and if this block is executed a number of times it could be retranslated through an optimizer. There would be different levels of optimizations in the translations. All the executed code would be translated but only the more frequently executed blocks would be translated with optimizations. The benefit is that it could faster than an interpreter though of the translation cost of all the code and there is just a way to execute the code which reduces the complexity. Debug a dynamic translator and an interpreter is harder than just debugging the dynamic translator.

From this two basic algorithms there are different possible implementations of a dynamic translator. From the more basic dynamic translator which could be implemented extending an interpreter or a threaded code interpreter into a translator through the generation of translation using templates in an instruction by instruction basis. To the full dynamic translators which are a first layer of software for running all the other software over this architecture (Transmeta Code Morpher and IBM's Daisy) using all kinds of optimizations, different levels of translation, interpreters and profilers. The main differences are in the use of an interpreter, the range of optimizations performed and way the target code is generated. Between those two approaches, template code generation and a full dynamic translator, exists an alternative of direct translation between source and target code using two layers (source code layer and target code layer) and decoded structures, rather than an IR, implementing some basic optimizations (flag suppression and peephole) which is very suited for the kind of project we are studying. This approach produce better translated code than a template code generator but it does not have the extreme complexity of a full dynamic translator. It is also a very suited approach for produce time accurate translations. [24].

Translation cache.

The translation cache is the structure where the translated blocks are stored. Basically is just a memory region reserved at the startup of the dynamic translator. This structure can grown up or not, it depends in the characteristics of the machine emulated and the dynamic translator. It can also provide mechanism for freeing translated blocks or to flush the entire cache. It depends if the way the dynamic translator handles self-modifying code or what it does when the translation cache is full. Flushing single translated blocks is hard to implement if the translated blocks use a technique to optimize the execution which links directly one translated block with its next translation blocks to be executed. This avoids the overhead of returning to the main translator block after each translated block is executed. Garbage collection in the translation cache thus, should be maintained to the minimum. In special cases which has to be detected, for example when an emulated program finish and a new one is loaded, it will performed a controlled flush of all or a part of the translation cache. In some architectures self-modifying code and dynamic code generation can be detected through memory cache flush instruction.

About the storage the translation cache just has to keep the translated block correctly aligned for a good cache performance and in some cases try to reorder the translated blocks to enhance the locality of the code.

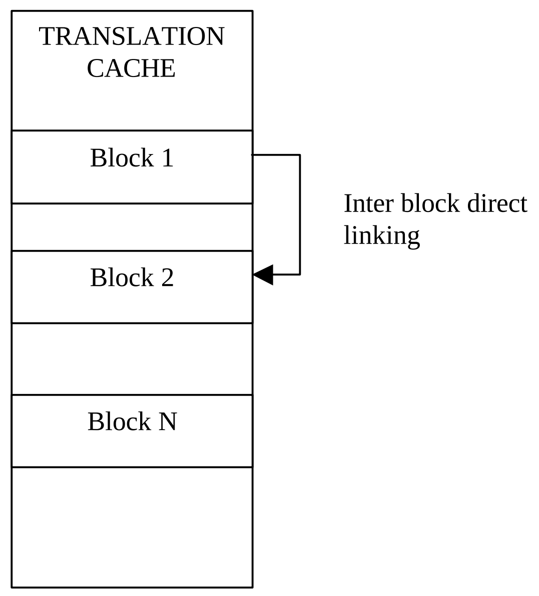


Figure 43. Translation cache.

The other function related with the translation cache is the mechanism to link the source code addresses with the translated block addresses. This is usually implemented using some kind of hash table. This hash table is used by the translator main loop to know if exists a translated block starting in the actual source address (the address in the PC). If it exists the hash table returns the start address for the translated block and the main loop can jump to this code. This address translation table can be implemented in different ways but it has to be as fast as possible for reducing the time spent in the translator main loop. One implementation could be a one to one table, each source address with his translated code address. This implementation is useful for machines with a limited address space, or when the region of the memory where the code can be located is known and limited. It is also implemented for instruction based translators, where each source instruction has its separated translation.

Interpreter and Profiler.

The interpreter, in those dynamic translators which make use of a first pass interpreter, can be implemented with any of the techniques studied in the chapter 3. Usually, as a dynamic translator that combines translation with interpretation is already a complex projects, it is a very fast interpreter and it could have been implemented before the translator to provide a basis from where to execute emulated code. Later as the translator was being tested and expanded the time spent in the interpreter would be decaying. An interpreter emulator is faster to implement and to test so it can be a good basis for the in-development translator.

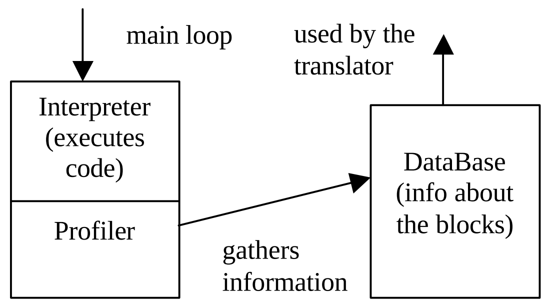


Figure 44. Interpreter + Profiler.

The interpreter has various purposes. It executes code which is not already translated until it reaches a number of executions and the translator is triggered to translate that block. This helps to reduce the impact of the translation overhead in the full emulation time because just the frequently executed blocks of source code are translated and optimized. It also reduces the impact of self-modifying code and dynamic generation of code because if this blocks of code change frequently or they are not very used the translator work is not affected. An interpreter can be fully responsible of the entire problem around self-modifying code and dynamic generation of code easing the work of the translation cache manager. It also executes those parts of the code or instructions which are never translated by dynamic translator. It is possible that some complex instructions or instructions which can not be found frequently from the source CPU are not implemented in the translator and the interpreter is used to execute them.

Finally the interpreter works as an execution profiler gathering information about the execution of the source code. The main information which is responsible of obtaining is the execution flow changes. It traces the entire jump, branches, procedure calls and returns building the information about the basic blocks of the source code. This information will be used later by the translator and optimizer for building the translated code. This and other informations which can be obtained can be used for implementing run-time or dynamic optimization, optimizations which depend in the real working of the source code. Such optimization include reordering of the code for reduce the overhead of the jump instructions. Additional information like register usage, flag usage and frequency of access to memory can be used to exploit other optimizations.

Translation Unit.

One important to take into account in a dynamic binary translator is the translation unit. In a static translator the translation unit uses to be the basic block which could be extended to a full function or all the binary when the global optimizations are performed. But working with basic blocks in a static translator is possible because it can spent as many time as needed optimizing each block and later optimizing between blocks (usually inside functions). In a dynamic binary translator the time for the translation and optimizations is limited. Some static translators could work with source instructions as translate unit but it would have just a few optimizations. It could be useful perhaps as a form of predecoding ,as in threaded code interpreter, performed at the startup.

In a dynamic translators using the source instruction as the translation unit the optimizations which can be performed are very limited. It can avoid redundant flag calculation, perform some peephole optimizations, some memory and IO access inlining and other simpler features. Such a dynamic translator could have been implemented using template code generation at would be just a bit faster than a threaded code interpreter would. Another reason for an instruction by instruction translation could be the necessity of a very accurate timing. In some old machines with slow CPUs (8-bit CPUs) the programs need a cycle exact timing. In most of them which use this timing to access the video and sound hardware any important change in the timing means a mess in the image and sound output. In those cases it is needed a time check after each source instruction (or small groups of instructions) and the translation must be performed instruction by instruction. In any other cases the instruction has to be avoided as the translation unit. It is only useful for timing and fast developments of interpreter derived translators.

The basic block as the translation unit has some problems in dynamic binary translation. In a static translator all the code is available at the translation time and the list of basic blocks can be built with more or less confidence and ease (there is the problem of indirect jumps though). In a dynamic translator the code is translated as it is being found or executed and it is hard to ensure if it is a real basic block or a future jump instruction will jump in the middle of the block. Although translators which use the interpreter profiler could ensure a bit more the '*basicity*' (how much chances of being a real basic block) of a block the problem is still there. This problem could be solved with multiple translations of the source code blocks or with retranslations. The second should be avoided if it has too much overhead. The first solution means that the blocks will be translated as they are found. If in the future there is a jump in the middle of the translated block and, therefore, a translation for this address does not exists a new translated block is built. The last half of the original source block will have two translations.

Another problem related with basic blocks is that although it is the basic unit for optimizations in a compiler and in a static translator it is not very suited for dynamic translation. A dynamic translator must perform the translation as fast as possible and most of the optimizations are not performed. The main point is to optimize the execution of loops so the translated code could be looping inside itself and it would not need to return to the main loop or jump to another block. That means that a basic block is too small for dynamic translation. In a complex dynamic translator with different levels of optimizations in the translation basic block could be used for local optimizations like in a static translator.

Another dimension of the translation unit is how much code is translated each time the translator is called with a given address. Translate just one instruction would be silly because the entire overhead would be in the call and return from the translation functions. Translate a basic block is not a good option either because most of the basic blocks are very small. The translation could end at the first jump but it would be still too small. The idea is to translate as much code as possible each time. In a dynamic translator using profiling that means all the code which is reachable from the start address which has triggered the translator because it has been executed enough times. In a dynamic translator without interpreter profiling that means all the code which is reachable from the start address through any jump which can be solved. In some cases it will be a limit (for example a maximum number of instructions, 1M, or code size 4MB, this depends in the amout of memory available) to avoid to translate too much code.

In a dynamic binary translator with an interpreter profiler the code which must be translated and optimized and some times retranslated and reoptimized is already know so there is no need of a limit. In this case the optimizations and the manner it is performed the translation depends upon the time which is given for the translator, and this time can be relatively large if a good optimization of this section of code is needed. In a dynamic translator which translates as soon as it reaches de code time is always more limited. Most of time the differentiation in basic blocks for the optimization phase is not performed and the source code is just parsed into decoded structures for the source code which suffer some modifications (optimizations) and are translated to decoded structures for the target code. Finally the target code is generated from this structures with some target dependant optimizations.

Optimizations.

There are various levels of optimization. In a template code generator for example there are almost no optimizations. Perhaps it could be detected if the flags generated for the current instruction are going to be erased by the next instructions without being used, then a template without flag calculation could be emitted for this instruction. The number of possible optimizations increases as the complexity of the translator increases.

The IR chosen for the dynamic translator affects to the optimizations which are going to be performed. In the case of a template based code generation it does not uses to exist any kind of IR so the optimizations could be just performed over the flow of instructions being translated. Another approach would be to use decoded structures for both the source and target CPUs. First the source code is decoded into a list of structures which the decoded information of the source code. At this level can be applied source dependant transformations and optimizations. Then this decoded list of instructions is translated to a decoded list of target instructions, basically in a one by one approach, a source instruction becomes one or more target instructions. Some optimizations, most marked in the previous phase can be performed in this process. Then the decoded list of target instructions can be modified with some target dependant optimizations and the final translation is emitted.

Other dynamic translators could use a virtual machine or code which could be suitable for implementing a fast threaded code interpreter which could be used for add portability to the system. Ardi’s Executor uses this approach. Finally a full IR of some form, a tree based IR or a three address code (or register transfer language) IR. This IRs can be used for implementing any kind of optimizations but they are expensive in the decoding, the optimizations and the code generation.

The common optimizations which can be found in a compiler or in a static translator can only be implemented in those complex dynamic binary translator which perform profiling and retranslating for obtaining a better code as it is being more frequently executed. The same happens with the common global optimizations with loops and inside the scope of a function. The optimizations have to be performed as fast as possible because it is a time which is added to the emulation. This level of optimizations is only affordable for blocks code which are a lot of used and use to be only implemented in large and complex dynamic translator environments like Transmeta Code Morpher or IBM Daisy.

In any case the optimizations which are basically implemented in dynamic translation, in any kind of implementation, and which are the most useful and fast to perform are: redundant condition code calculation suppression, memory access inlining and various kind of peephole optimizations to do a better use of the target ISA. As we said in previous chapter the calculation of flags could be very expensive, for a dynamic translator it is easy to parse the code to be translated and check the usage of the flags. Only the flags which are really used are really generated and target code generated for calculating them. With the memory access could happen the same, those address which can be determined in translation time does not need to go through the memory handler list but just access the proper handler or memory buffer. The peephole optimizations depend on the source and target ISA and try to take advantage of features that the source CPU does not have and the target has. For example it could be detected a 16-bit addition in an 8-bit CPU without 16-bit addition using two instructions and be translated in the target CPU into a single instruction.

In a dynamic translator there is a level of optimization which is not present in a compiler or in a static translator: the dynamic optimizations. There are a many references about dynamic recompilation and compilation at run-time which are used to improve the code with run-time information about the real behaviour of the code. Such techniques can be applied in a dynamic translator that implements a powerful translator and a profiler. Such optimizations use to be associated with the changes in the execution flow and tries to reorder the code to reduce jumps, reduce jumps overhead (for example mispredictions if it is applicable to the target CPU) and improve the locality of the translated code. HP's Dynamo implements some of this techniques over the same HP architecture (it is not dynamic translation but just dynamic optimization).

Register Allocation.

Register allocation is a critical part of the translation, keeping as many of the source registers into target registers as much time as possible is the main objective. This reduces the memory access and the number of instructions in the translation. We have again the two approaches: dynamic allocation and static allocation. A good option if a static allocation is not possible is a mixture between static and dynamic allocation. Dynamic allocation in a dynamic translator has two problems: it is expensive and increases the translation time and it has the problem of having to spill all the allocated registers after each translated block end. If for each translated block the register are allocated separately a mechanism for maintaining an equivalent status (registers) between translated block is needed. The easier solution would be to store all the used source registers in the CPU context structure at the end of the translated block. Then the next translated block would start allocation registers from scratch and reading them from the CPU context structure. This is very wasteful in the number of memory access and it could only be useful if the translated blocks are large. Another solution would be a structure for passing info between blocks about the actual register allocation but it would be harder to implement.

The static solution is the best possible between the allocation is kept between blocks and the number of memory access for accessing the registers is minimized. Only when the dynamic translator must exit the source registers are stored again in the emulated CPU context. The static approach needs that the number of registers in the target CPU would be larger than the number of registers in the source CPU. If that is true we are in the best case, for example emulation 8-bit CPUs or x86 over a RISC CPU with 32 registers. Sometimes the number of registers is very similar between the target and the source CPU, this happens when translating similar architectures like two RISC CPUs. Some of the target register must be used as scratch register and for other purposes. Then a good approach is to statically assign the more frequent source registers to target registers and perform intrablock register allocation for the rest of source registers.

When the target register are less than the target registers we have a problem and we will have to use memory for storing the source registers. If some source registers are very used it would be a good idea to keep them in some of the target registers and all the others be accessed through memory or temporal assignments to target registers.

Code Generation.

The code generation can be implemented in different ways. The pattern matching approach we talked about the static translators would be wasteful and only useful if the dynamic translator is using a full featured IR (many times in the form of a tree to use tree matching algorithms). Complex dynamic translators with background optimizations and with different levels of translation could use this approach.

As we have talked in other sections a simple implementation for dynamic binary translation is to use template base code generation. This just reads the source code, parsing perhaps a block instructions for finding some hints (flag usage), and emits for each source instruction a preassembled block of target code which emulates this instruction. This preassembled blocks are emitted one next to the other and at the end of the translated block it is performed a patching phase which writes the information about relative jumps and literal values. This is a limited form of dynamic translation which produces really bad code. The preassembled instructions are more or less the same code which can be found in an assembly interpreter, therefore the main benefit is that avoids the decoding overhead and the jump from one the functions which emulates one instruction to the next.

The last implementation is that the target instructions are really assembled by the translator but the translation is still performed more or less in a one to N basis. One source instruction is translated to one to N target instructions. The best it to have use two lists of decoded instructions. One of the list will be a list with the source instructions in a decode format. In this form the source instructions can be analyzed and some changes and hints can be added for the next phase. Then the decoded source instructions are translated in the one to N basis to decoded target instructions. This list of decoded target instructions is not target code still so more modifications can be performed in this phase. At last the decoded target instructions are sent through a code emitter which uses the decode information to assemble real target instructions.

The format of a translated block can be very different but usually it starts with a prologue, continues with the translated block body and can end with an epilogue. In the prologue it can be performed initializations for the translated block, for example the emulated registers can be loaded into the target registers. It can also be used to check if the emulated code has changed with a checksum for detecting self-modifying code. It is not the best way to detect self-modifying code but sometimes is an expensive but useful solution. It can also check the time. If the remaining number of cycles to spent in the emulation of the CPU is smaller than the number of cycles it would take the translated block the translated block will return to the translator main loop. The time checking can be also implemented in the epilogue, the difference is that in this case the CPU emulation will stop when the number of cycles to emulate has been exceeded. The epilogue serves also to restore the state between the different translated blocks and the translator main loop. It can also implement a direct jump to the next translated block to be executed. This avoids a jump back to the translator main block after the execution of each block. The body of the translated block contains the translation of the source block code to the target code. Each translated block can also have additional information attached to it and not just the code of the translation. It could be added counters for profiling with what frequency is being executed. In a translated block which ends with an indirect jumps the last jump address could be stored to try a fast jump to the next translated block without returning to the translator main loop.

Implementations.

Many implementations of dynamic translators can be found. The first academic researches start in the 90. Shade and Embra [8] [13] are dynamic binary translator which are used for profiling or in frameworks which are designed for profiling and simulation (Embra is part of the SimOS framework). Shade introduces the basic techniques around a fast dynamic translation at a low cost. Embra continues the work of Shade and adds the emulation of the source CPU MMU and cache. Although the last one is not very useful for emulation in some systems it will be interesting to emulate the MMU of the CPU.

At the same time different dynamic binary translator were developed for commercial use. Apple extended the Motorola 68000 emulator, which used an interpreter, and implemented a full dynamic translator for execution old Mac applications [18]. This emulator used avoided the problem of the self-modifying code tracking the use of the cache flush instruction which was needed in the 68K systems with cache implemented. Other interesting products is Ardi’s Executor [5] dynamic translator which emulates a 68K based Mac over a x86 architecture. This emulator uses a virtual internal representation which can be emulated through a threaded for systems which does not have the translator layer. Executor translates the 68K code to x86 code. Other products are SoftPC and Virtual PC.

Lately two commercial emulators of the Sony Play Station have stepped into the market. Bleem from Bleem co. fails to achieve an accurate emulation of the PSX and the number of games which are playable is limited, but it is a very fast emulator and uses Direct3D to enhance the PSX graphics. Connectix has developed Virtual Game Station for the Mac, later ported to the PC. VGS emulates very accurately the PSX and most of the games work perfectly but it is a bit slow and does not uses Direct 3D emulating all the PSX 3D hardware through software rutines. Both use dynamic translation for the emulation of the PSX MIPS.

In the time when most of the CISC architectures were abandoned and replaced by RISC CPUs the trend of the binary translation was to use static translators as an alternative to port applications from the old CISC architectures to the new RISC architectures. The different Digital static translators are a proof of this trend. Today there is a new change in the architecture of the CPU. The RISC superscalar, with implicit instruction parallelism and instruction reordering seems to be replaced, by part of the industry, with VLIW (Very Long Instruction Words) with explicit parallelism. Now the effort of obtaining a good ILP (instruction level parallelism) is going to move from the CPU to the compiler. The intention now is that this new implementation of the CPUs does not affect to the existent code. The idea is to add a software layer which will hide the real architecture of the CPU in the system and show a virtual architecture which will maintain the compatibility with the old code. Now the dynamic translator is a part of the system which in a level between the hardware and the OS. Implementations of this new trend are Transmeta Crusoe processor and Code Morpher software and IBM Daisy dynamic binary translator.

Transmeta is a company which has designed a VLIW CPU which uses a software layer, Code Morpher, to hide the internal VLIW architecture and execute transparently x86 code. The Code Morpher software is a dynamic translator which uses an interpreter, profiling and different levels of optimizations for executing the x86 code in the VLIW processor. The Crusoe processor implements features which helps to enhance the performance of the dynamic translator like register shadowing, commit and rollback instructions for precise exception emulation. The Crusoe software resided in ROM and it is loaded in a separated and protected region of the system memory at the startup before starting the execution of the OS.

Daisy is an IBM research project which is open source and can be downloaded by everyone. It is basically the same idea than Transmeta Crusoe and Code Morpher. The Daisy project just implements a simulator of a VLIW processor and the dynamic translator which executes any code from the source machine over this processor. The VLIW implements also features to help the translation. Daisy implements PowerPC and S390 translators. The main effort in this project is to achieve a good usage of the ILP facilities of the VLIW processor. Thus complicate algorithms for reordering an implementing a good instruction scheduling are being developed in the project.

Other project of interest is UQDBT which is the dynamic version of the multitarget and multisource UQBT static translator. It is in development. HP has developed Aries a dynamic translator from HP PA-RISC architecture to the IA-64 VLIW architecture. HP also developed Dynamo dynamic optimizer which tries to dynamically optimize the HP PA-RISC code. Dynamo uses a fast interpreter emulator of the same native ISA to profile the paths followed in the execution of the code. After a number of executions and a trace of the execution flow has been built it performs a ‘translation’ or regeneration of the native code which tries to reduce the jump overhead and improve the code locality. The results of such dynamic optimizer are limited but it could be useful as a backend for a real dynamic translator.

In the emulation scene it can be found diverse implementations of dynamic binary translator (called dynarec in this ambit). Most of the emulators for the more modern machines use this approach (PSX, N64) or similar ones. There are various open source emulators which use dynarec (or DBT). Usually is just a template code generator but in some cases it could feature a full dynamic translator, for example in Basilisk and UAE JIT Motorola 68K emulators.

It can also be found some last year university projects related with dynamic binary translation, for example Julian Brown’s ARM dynamic binary translator Armphetamine [26]. David Sharp is working in a similar project about ARM translation. [27]

Chapter 5. Memory, Interrupts and Timing.

In this chapter we will see some concepts that we have already seen in the chapters about CPU emulation. We will study them more in depth and in a separated chapter because they are very important for performing a good emulation. The first topic we will talk about will be memory emulation. As we will see a fast emulation of the memory access is very important for a good performance. We will see why a plain access to a memory buffer is not enough for the emulation of our target machines. We will see advanced topics about the emulation of the memory as how to emulate a MMU or how to use the target CPU MMU for memory emulation. In the second point we will talk a bit more about the emulation of interrupts and exceptions. Interrupts are related with the communication between devices and the CPU, exceptions are just related with the CPU. At the third point we will talk about the importance of an exact emulation of the time in out target machines and how it can be implemented in the CPU emulation.

1. Memory emulation.

The memory address space of a real machine is not like the virtual address space in a process running over an OS. It is not a plain area of memory which can be accessed with freedom. There are special regions which map real physical memory, others are mappings for memory from other devices (video memory for example) or mappings of device registers. Some areas of the memory space contain a type of memory than the others, for example a region for ROM and a region for RAM. Some machines implement complex mechanism for mapping the real physical memory to a given memory address, for example bankswitching and MMUs (Memory Management Units). This behaviour of the memory must be emulated if we want to run any kind of applications in our emulators.

Something important to know is how the machine we are goint to emulate use memory. We could differentiate two kinds of machines here. Most of the old cartridge based (ROM based) videogame consoles and arcade machines used ROM based memory for the all the application code and data. There were then two separated regions of the memory space: the ROM space which is only read and the RAM space can be written. The ROM space must be write protected because some games tried to write in (because of errors in the code or for performing antipiracy checking) so the memory emulation must implement this protection. The RAM could be accessed freely. Some of the memory regions could be mirrors of other memory regions so they were mapping the same physical memory. This happened because not all the bus address lines were used to decode which device or memory chip was to be accessed. There could be large regions of the memory space which are not used. This regions should return a default value (0x00 or 0xff) when read and be unaffected by writes. Some regions could map video or sound memory devices and other registers from the hardware devices.

The other kind of machines is microcomputers and the more modern CD based consoles. Their memory address space is more similar to a standard machine. They have a lot of more RAM and only BIOS and internal OS are stored in ROM format. The RAM is now needed to load the data and code from the CD-ROM, disk or tape support. The other parts of the memory address space are very similar with mapped device registers or memory and non used regions.

In the ROM based systems the cartridge or ROM cards can be used directly as memory by the machine. ROM uses to be very slow compared with standard dynamic RAM (DRAM) or even more with static RAM (SRAM) but at the time the ROM cartridge system was used CPUs were not so fast to have many problems with it. The code was executed from the ROMs as well as the data for sound and video. The small RAM memory was used for generation or modifications of the sound and graphic data and for the game variables. In some cases some cases RAM used to execute small pieces of code faster than in the ROM memory.

The memory map of a computer is how the memory address space is organized between the different memory types and mapped devices. A memory map for a computer can be obtained from the official or non official documentation of the computer or using the computer schematics. The schematics show which chips are connected to which lines in the CPU. Knowing the usual types of decoders and memory chips it can be discovered the memory mapping in a system. Another method of discovering the memory mapping is to analyze the code which runs in the machine. In the code it can be seen which parts of the memory contain RAM or ROM, some times if they are device registers. It can also be differentiated the code and data ROMs. In video arcade machines the ROM for code and the ROM for data use to be separated and have different address spaces. They also can have ROM chips which can be only accessed by some of the CPUs in the system or a part of the hardware devices (sound ROMs, graphics ROMs). In a home videogame machine (console) both data and code share the same address and ROM chips.

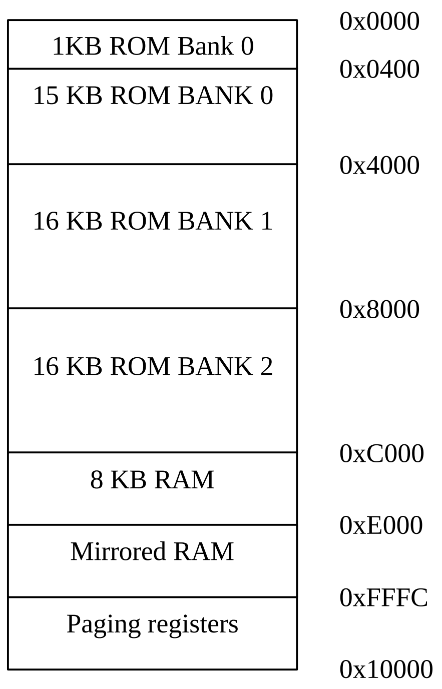


Figure 45. Master System memory map (banks and page registers, no memory mapped IO).

The memory address space can be used for both memory access (either system memory or memory from a device as video memory) and IO (input/output) access. This kind of IO access is called memory mapped IO. The devices registers are directly mapped in the memory address space through some kind of decoders which decide with some or all the bus address lines which device or memory chip must be accessed. Some computers (i8080, z80 and x86) have a separated address space which can be used for IO access. Each address in this address space can access a register in a device, the address space uses to be smaller (for example just 8-bit). Special instructions are used for accessing this address space (IN for reading from the port and OUT for writing to the port).

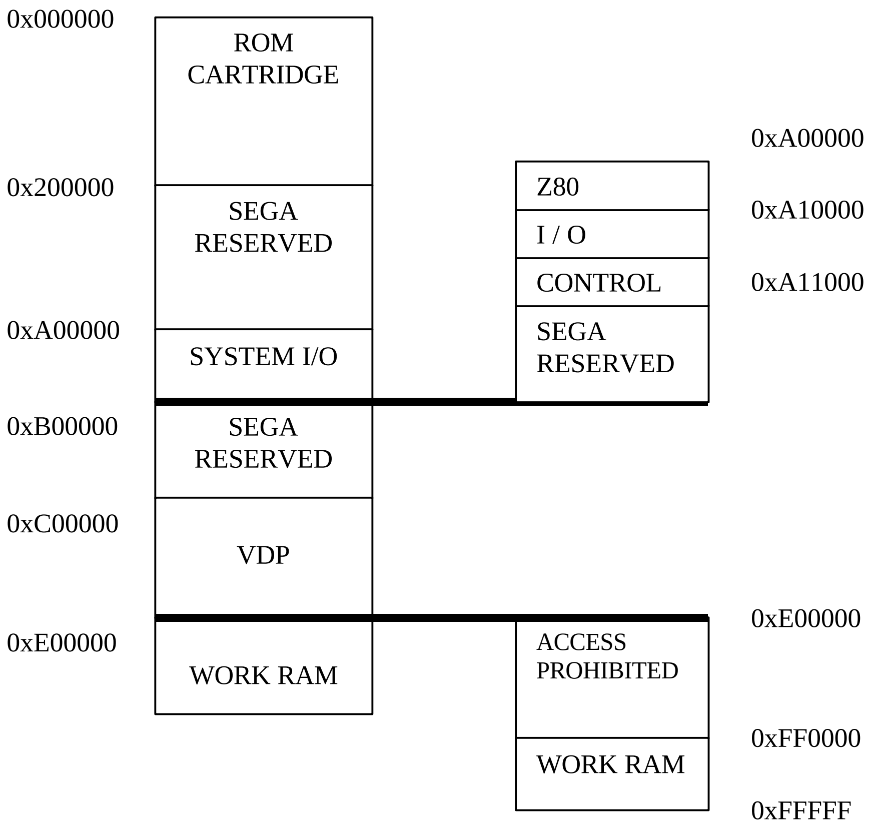


Figure 46. Genesis memory map (memory mapped IO).

Memory Map. Region List.

The basic memory map is a list of regions which must be accessed in a different manner. For each region exists a pointer to a memory buffer where the read or writes will be directed or a pointer to a function which will be called with an access to this address is found. There uses to be more than one list of those memory regions. Since it can be very different the behaviour in read and write access many times it will be implemented a separate list for read and writes. The data size of the access can be also important in some cases so sometimes we will found different list for the different data sizes. If a separated address space for IO exists it will have its own region lists for reading and writing and the different data sizes. Some CPU emulators include a separated region list for fetching (read code). However most of the CPU emulators try to do not read opcodes through the memory regions because it is very expensive, a direct read to the main memory buffer uses to be enough for fetching code. In some cases this could be different and special mechanism will be implemented.

```
struct z80PortRead ReadPorts[] =
{
        {0xbe, 0xbf, readVDP},
        {0x7e, 0x7f, readVHCount},
        {0xc0, 0xc1, readJoy},
        {0xdc, 0xdd, readJoy},
        {0x00, 0x00, readStartNationGG},
        {0x01, 0x05, readIOGG},
        {-1, -1, NULL}
};

struct z80PortWrite WritePorts[] =
{
        {0xbe, 0xbf, writeVDP},
        {0x7e, 0x7f, writePSG},
        {0x3e, 0x3f, writeNationBIOS},
        {0xde, 0xdf, unknownWrite},
        {0x01, 0x05, writeIOGG},
        {-1, -1, NULL}
};

struct MemoryReadByte ReadMemory[] =

{
        {-1, -1, NULL}
};


struct MemoryWriteByte WriteMemory[] =

{
        {0x0000, 0xbfff, romWriteProtect},
        {0xc000, 0xfffb, ramWrite},
        {0xfffc, 0xffff, writePageRegistersBS},
        {-1, -1, NULL}
};
```

Figure 47. Master System memory and IO region list handlers.

This is the structure of a memory region entry in C:

```
{0xa000, 0xc000, NULL, videoMemoryHand}
```

The first two fields are the limits of the region. When a CPU emulator is going to perform a memory access of any kind it will test the address in the memory region list for this kind of address. If a range in the list matches the address then the action associated with this region is performed. If a match does not exists it can be interpreted as it was an access to the normal main buffer (faster access) or that it is an access to an undefined address and a default value must be returned. The next fields in the entry have the information about which action must be performed for this memory region. In this case the fields are a pointer to a memory buffer and a pointer to a function. In the example the memory buffer address field is NULL or undefined so this field is ignored. The next field contains a pointer to a function which will be called for implementing the access.

The basic implementation of the memory emulation using region lists starts with a main buffer for all the emulated memory. This buffer will be accessed whenever and address and access type does not match an entry in the region list. This kind of access is the fastest (a memory access is directly emulated as a memory access in the target CPU). Then the basic algorithm is to put a loop in each memory access instruction which checks the access address for the region list sequentially. There are other implementations (hash tables) which are faster but if the number of regions is small enough they are not useful. After the loop there is a code which decides what kind of access must be performed, either an access to the main buffer, or to the associated buffer or to call the function handler.

The actions which have to be performed because of an access in some cases can be complex. This is the reason that, rather than implementing different types of general access in the main access emulation code, a function which acts as a memory handler is implemented for each region. The memory handlers have a standard interface. They receive the address, the data (if it is a write access), and the entry of region list. If it is a handler for a read access they return the read data. Those functions can perform any task. The most common is just access a special memory buffer for this region, or perform mirroring (the same physical memory is accessed through two regions, so any write to any of the regions must be performed in the two regions). If it is a mapped region of device memory or a device register the function handler will be implemented in the device side using the standard interface (function definition). Device register handlers use to trigger actions in the device emulation or to set internal variables for later use. Some access will be directly emulated as access to the hardware in the real machine.

```
UINT8 readByte(UINT32 address)
{
    struct readMemoryHandler *last;
    UINT32 handlerIndex;

    handlerIndex = 0;
    last = &tC.readMemoryHandler[handlerIndex++];

    while ((last->startAddress != -1) && (last->endAddress != -1)
    && (last ->memoryHandler != (void *)-1) && (last->pUserData != (void *)-1))
    {
        if ((address >= last->startAddress) && (address <= last->endAddress))
        {
            if (last->memoryHandler != 0)
                return (*last->memoryHandler)(address, last);
            else
                return ((UINT8 *) last->pUserData)[address];
        }
                        else
        last = &tC.readMemoryHandler[handlerIndex++];
    }

    return tC.mainMemory[address];
}
```

Figure 48. Read byte through a memory region list.

Memory Banking.

In early CPUs the size of the memory address space was small and in some cases it was not enough to access all the physical memory at a time. Systems to avoid this problem were developed. It uses to be a small hardware which can be found usually out of the CPU (sometimes also inside the CPU, it could be said that it derived into the standard MMU hardware in the modern CPUs). This hardware redirects the access in a given memory address to a physical memory block or another depending upon a special register. The basic mechanism is called banking or bankswitching. Some regions of the address space can map different physical memory regions. The regions in the address space are called banks, and they can map pages of physical memory. This system was very used in the early videogames consoles because the ROMs cartridges became larger than the available address space. Many times the banking hardware is present in the cartridge (Master System, NES) and it can also map IO registers from special devices in each cartridge (NES). The memory page mapped in each bank is selected through an IO register in a special memory address or IO address.

Rather than with region lists, which are slow, this can be implemented inlining the access to memory through an array of memory pointers. Each bank has an associated pointer. The emulator will control the memory buffers associated with each of the banks as the bank registers are accessed. A region list could still be used and the access through the banked memory would replace the access to the main memory buffer since code fetches also have to be performed through the bank emulation code. First the access emulation code will search in the region list, if the address is inside the range of a region the proper action will be performed. If not the access will be through the bank mechanism. First it will get the bank number in which the access will be performed (usually is just a logical right shift), with the bank number we will obtain the base address for the mapped page. Then from the original address it will be obtained the offset inside the bank and it will added to the base address, the result address will be used for the emulated access.

```
#define PageIDNBits 2
#define PAGEOFFSETMASK 0x3FFF
#define NUMBANKS 4

char *memoryBank[NUMBANKS];

void writeBank(int address, unsigned char data)
{
   memoryBank[address >> PageIDNBits][addres & PAGEOFFSETMASK] = data;
}
```

Four banks, 16-bit addresses, 16KB per bank.

Figure 49. Bank access emulation.

Other implementations.

We have already said that a sequential scan through the region list is expensive. In many cases is a good approach, further more if we are using a general purpose (and implemented by someone else) CPU core. The number of regions uses to be small, and in the case of reads, the more frequent memory access, sometimes the region list is empty. Write access uses to be slower but as they are less frequent the impact is reduced. In the case of memory mapped IO or in a separated IO space it must be taken into account than in a real machine IO access is also slower and less frequent than normal memory access, so the performance lost is minimal.

If there are many regions or we want to implement a faster memory access there are some alternatives to a region list. Hash tables can be used for example to perform a faster access for the list, but hash tables have the problem they search for exact values, not for values inside a region. Hash tables could be used with memory mapped IO registers for example which are single addressed, and in some cases they are mirrored in different addresses. If we are building the CPU core or we can change it we could, rather than parse a standard region list, inline directly the checks for the more frequent regions or address in the memory access emulation code.

An alternative to a list of regions could be a vector of regions. Like in the case of banking we could divide the memory address space in a number of regions. These regions would have all the same size and they would have associated information about how they should be accessed. For example each region could have a function pointer, the memory handler. Rather than a sequential scan, now the source address is divide in two parts: the region number and the offset inside the region. It works exactly as banking, but we are not necessarily emulating banking. The region number is used to retrieve the information about the region and the offset is used to perform the access inside this region. For example a region would have an associated memory handler and the offset (or the full source address) could be passed to it. The size of each region, the granularity of the regions, will depend upon how many different real regions are in the emulated address space, how fast we want to perform the access and how much memory is available for the emulation. In an extreme case we could have implemented and array of memory handler pointer for each memory address. This would expand the memory usage to more than (N + 1) the emulated machine memory address space (N the pointer size in bytes in the target CPU). This kind of access to memory could be implemented as nested regions, regions with many different handlers (for example regions with mapped IO register) would expand into region tables addressed by the offset.

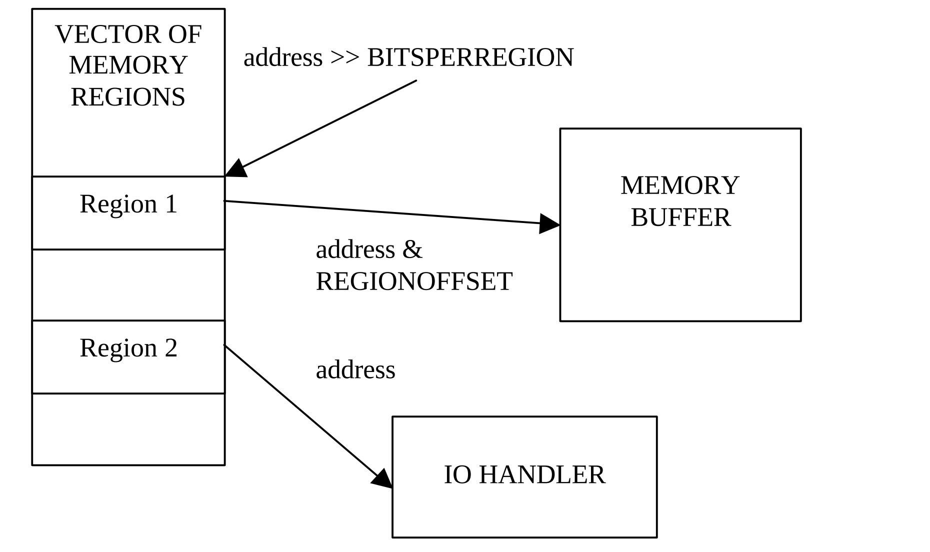


Figure 50. Vector based memory access.

Memory Management Unit (MMU).

The MMU is a special hardware which is present in all modern CPUs that controls how the memory is accessed by the operating system and the applications. The MMU implements different tasks and mechanisms. The MMU creates a virtual address space for all the applications and the OS, mapping freely physical memory pages into virtual address memory pages. The MMU is used to create its own virtual address space for all the processes and the OS, so their address space become separated and protected from each other. The MMU offers different levels of protection for the different address of the virtual space: read, write and execution protection. The MMU also is used to implement virtual memory through page fault exceptions. Whenever a protected page or not assigned page is accessed the MMU signals an exception to the CPU to deal with the problem. The MMU is a complex hardware with a complex behaviour which is very hard and expensive to emulate in software.

The MMU works with memory pages. A page is an aligned block of memory of a known a fixed size (usually 4KB or 8KB). The full memory address space is divided in pages. The MMU maps physical memory pages to virtual memory pages, it maps the real memory with an address in the CPU memory space. It also sets different properties and protections for each virtual page. A page usually can be read, write and execution protected, though many other flags are possible. Some MMUs (x86) implement callback mechanism through MMU entries. All this information is keep in a table of pages. Since the full table for all the possible virtual pages is very large only a few entries (the last accessed pages) are stored in the MMU. The structure which stores this entries is the TLB (Translation locaside buffer). The MMU sends exceptions to the CPU when it detects some events, for example if the a page was write protected and an instruction tried to perform a write to the page, or if the virtual page was undefined. When an access to a page which is not in the TLB the CPU is signalled and the OS (or the software which is running at privilege level) is the responsible of updating the TLB and performing the opportune actions. The MMU is used to implement virtual memory and swapping, moving the pages from memory to disk and vice versa.

As we can see the software emulation of this mechanism would be very hard because many checkings must be performed in each memory access. Some binary translators implement this (for simulation purpose of full OS emulation) mechanism but at a cost. Embra uses a software approach (mainly for simulation) which uses fast cache tables and hash tables to perform fast checks through the emulated

TLB. Another problem is that the MMU generate exceptions which must be very exact so they could be handled by the OS (the emulated OS). Most of our source machines for emulation do not implement MMU mechanism, just the more modern ones (DreamCast, Play Station 2) have this hardware implemented but often it is disabled.

In any case a MMU in our target machine (where the emulation will be performed) can be very useful for emulating in hardware some characteristics of the source machine memory. A MMU can be used too to emulate the MMU of the source CPU, Virtual PC and other similar emulators use the target CPU MMU to emulate the source CPU MMU. In this case the difficulty will to how to implement the differences between the two MMU models. The MMU is also useful emulating non MMU memory systems. Although the MMU has an expensive cost because of the signalling of exceptions which must go through the OS sometimes will be faster than a software implementation which checks every memory access. The use of the MMU for the emulation will be restricted by the interface and mechanisms that the target machine OS provides to the applications for using the MMU. Unix systems (Linux for example) provide a good mechanism through the mmap() call. Win9x facilities are very reduced in the general API, special mechanisms through virtual devices could be implemented to expand this facilities. Win 2000 provides a similar mechanism to the Unix mmap().

The MMU can be used for protecting regions of memory for read and write. In this case when a protected read or write is performed the MMU sends an exceptions and the emulator could handle as it was needed. For example a memory mapped IO register could be write protected. Rather than using a search through a region list before each memory write, it could be implemented as a direct write and the MMU would be the responsible of detecting the special cases. When the write to the protected emulated IO register is performed the MMU raises a memory exception, the exceptions goes through the OS and arrives to the emulator memory exception (or signal) handler. In this handler it is detected which address has been accessed. If it detects it is an IO register it performs the associated task for example calling the IO write handler for this register. It must taken into account how expensive is the overhead due to the exception and how expensive is a software check before each memory access. This will also change depending how frequently a special access is performed. The more frequent is a special memory access the more expensive it will be the MMU fault approach.

Another useful mechanism of the MMU is the physical page mapping system. This can be used for emulating banking (bandswitching). The banking mechanism is basically a primitive version of the MMU paging system. The emulator keeps the regions of the emulated address space which can be banked as a special region of memory in the target machine which can be paged. Each time the emulator detects that a bankswitch must be performed (a bank in the emulated machine is loaded with a different page) the emulator changes the physical page mapping for the associated page in the target machine memory. As in the previous case the MMU system will be better than a software approach (through an indirect access using an array of pointers) if the page switches are not very frequent. The MMU systems have an overhead due a call to the OS and the changes in the MMU, and TLB entry faults.

The last interesting implementation we will talk about is the detection of self-modifying code. Although in most of the modern systems (but for example x86) self-modifying code can be through flush cache instructions in the early computers which cache was not used this is not possible. The presence and detection of self-modifying code is very expensive for threaded code and binary translation (not for normal interpreters). Write protecting the emulated memory pages which have been already translated could be a good mechanism for detecting self-modifying code. Other solution is software checks in each memory access or checksums at the start of each translated block in binary translation. As in the previous examples the frequency it happens a self-modifying code access is critical.

Some special combinations of binary translation and specialized CPUs for binary translation can use protections in the memory space associated with the emulated memory space to help the translation (Daisy, Transmeta Crusoe/Code Morpher 14]). For example the MMU could have an additional bit to mark if a page in the emulated memory is already translated or not. If an execution is tried in a non translated page the translator is called and it can perform the translation. The target MMU can also be used for separate the memory spaces for the emulator and translator and the emulated memory space.

Endianness and CPU data size.

Another problem related with the memory is how the data is stored in the memory and how it is accessed by the CPU and the applications.

There is no problem accessing bytes because all the memories are addressed and accessed in a byte basis. The problem is how multibyte data is stored and retrieved. The different CPUs implement two different orderings of multi byte words in memory: little endian and big endian. In little endian the less significant bytes of the word are stored in the low memory addresses, the more significant bytes of the word are stored in high memory addresses. Big endian CPUs store the bytes in inverse order, in low addresses the more significant bytes and in high addresses the less significant bytes.

A	B	C	D

Big Endian. Low order bytes are stored first.

D	C	B	A

Little Endian. High order bytes are stored first.

Original word

0x12345678

Big Endian (stored in memory)

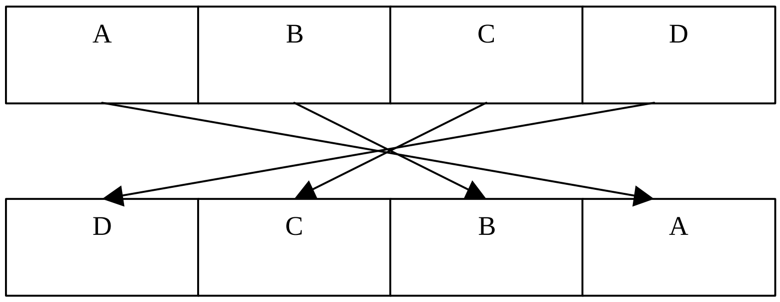
0x12 0x34 0x56 0x78

Little Endian (stored in memory)

0x78 0x56 0x34 0x12

Figure 51. Endiannes.

If the emulated CPU and the target CPU have the same endiannes there is no problem. Only when the target CPU and source CPU have different endianness a mechanism for translating from a data format to the other must be provided. The conversion of the data can be performed statically, at startup when the code and data is loaded. Or it can be performed after each multibyte word access to the emulated memory. The problem is that the manner the target CPU reads and writes the data is different to the emulated CPU and therefore the data that is stored in the registers for operating with is invalid. The conversion after (or before for a write) each access is very expensive. The conversion of a big endian word to little endian word means many rotations, shift and logical operations (masks and ors). Some CPUs will have special instructions or combinations of instructions which makes this conversion faster. If the conversion is performed enough fast it could be a good implementation. This is not usually the best option.



Input value: in. Output value: out.

```
out = 0;
for (i = 0; i < BYTESPERWORD; i++)
{
    out = out << 8;
    out = in & 0xff;
    in = in >> 8;
}
```

Figure 52. Endianness conversion algorithm (basic)

Some CPUs have the ability to change the endianness mode. The problem is that in many cases the endianness mode is set at startup of the machine when the OS is loaded and can not be changed. However if the emulator can change the endian mode of the target machine to match with the emulated machine mode the problem will disappear.

The alternative to a conversion for each memory access is to try to perform the conversion when the data and code are loaded into the emulated memory. The time spent in this conversion will not be important because the time of loading the data is always larger by far. The problem is that a static conversion does not solve all the problems. The conversion depends in the byte size of the words. The endianness conversion must be performed for a determinate word size: 16 bits (2 bytes), 32 bits (4 bytes) or 64 bits (8 bytes) for example. This means that any other access in a different data size from the conversion data size will still be wrong. There are solutions to this problem. For example we could have a copy of the emulated memory converted in all the possible data sizes (this is limited by the CPU data size access modes), but then the writes will be very expensive (a write must be performed in each of the version of the emulated memory), the memory usage would be dramatically increased and the memory cache performance could be reduced.

Although the data converted for a given data size can not be accessed directly in a different data size small changes made this access faster. For example we have an 8-bit CPU which is big endian. Our target CPU is 32-bit little endian. The emulated CPU can perform 8-bit and 16-bit memory access and the target 8, 16 and 32 bit memory access. If we convert the emulated memory from 16-bit big endian to

16-bit little endian we will have problems in 8-bit access. But the byte which must be read in an access to an emulated address can be found at the target CPU address with the less significant byte complemented. This can complement can be performed with just one instruction which is far better than a little endian to big endian conversion. The same happens with all the other data sizes, for all data sizes which are smaller than the data size in which the conversion was performed a similar translation of the address can be performed. The only limitation is that the address must be aligned with the access data size.

Endiannes mode	Access size	Access Mode
word ordering	byte (8 bits)	address xor 1
	word (16 bits)	address (direct)
	dword (32 bits)	xchg words (lo, hi)
	qword (64 bits)	xchg word + dwords
dword ordering	byte	address xor 3
	word	address xor 2
	dword	address (direct)
	qword	xchg dwords
qword ordering	byte	address xor 7
	word	address xor 6
	dword	address xor 4
	qword	address (direct)

This table explains how can be accessed (reads and writes) different word sizes in buffers swapped (endian mode changed) in different word orderings.

Figure 53. Access modes swapped memory buffers.

The size of the word and the access modes of the target and emulated CPU can be very important too. When using a high level language it must be taken into account the size in bits of this data type, furthermore if the emulator is going to be ported. The types for the register and size fixed variables of the emulator will have to be defined in an independent way of the architecture where it will be compiled the source code. Problems related with the size of operations and operands in the CPU emulation are the flag calculations and some operations which are sign dependant. Sometimes it will be needed to sign extend o zero extend the source data from the emulated machine in a larger data format in the target machine for a correct result.

2. Interrupts and exceptions.

Exceptions and interrupts are mechanisms that the CPUs implement to stop the current flow of execution and start the execution of special pieces of code (interrupt and exceptions handlers) when a special event happens. In some cases the exact emulation of the moment (in which instruction or cycle) an interrupt or exception happens is very important for a correct emulation. The CPU state that the interrupt or exception handler can see is also important because in some cases (binary translation) the state in a given PC or instruction could not be the exact state in the real machine.

Interrupts.

Interrupts are a mechanism for communication between the CPU and the hardware devices attached to the bus. The CPU provides special lines which are used to signal events (interrupts) from the devices to the CPU. This kind of interrupts is also known as IRQ or hardware interrupts (to differentiate from the software interrupt which are just a special form of procedure call). The CPU can provide with one to any number (the number uses to be small) of interrupt lines, each of them can be used by a different device. There are also control lines which tell the devices when interrupts are disabled (the CPU is not replying to the signals) and to acknowledge a received interrupt. It uses to exist an order of priority between interrupts.

The more priority interrupt, when the interrupt handler for this instruction is being executed, will not be interrupted by any other interrupt from any other device. Less priority interrupts could be interrupted by a more priority interrupts from another device. The NMI (Non Maskable Interrupt) is the more priority interrupt level, which can not be interrupted by any other, and never can be disabled. The CPUs have instructions for handling the priorities of interrupts and to disable and enable different levels of interrupts.

The emulation of the interrupt mechanism is first implemented in the CPU emulator. All the context information related with instruction is stored and maintained by the CPU core. The special instructions for dealing with interrupts are implemented in the core. The core must provide an interface for the emulated devices out of the code to signal interrupts. This kind of functions just tell CPU core that an interrupt has been signalled and with which level of priority or type of interrupt (and any other parameter which could be needed). The function just checks if the interrupts are enabled and push to the stack the current CPU context (it can be the PC and the flags or any other registers, it is CPU dependant), then it "jumps" to the interrupt handler address. The jump is performed changing the emulated PC register. When the CPU core starts again the emulation of the CPU it will start at the interrupt handler address. The interrupt handler address can be a fixed address or can be vector directed, when to the interrupt is associated a fixed location of memory (for example the first bytes in the address space) that contains the address of the interrupt handler.

The interrupts are used by different devices: keyboard, disks, graphic and sound hardware. In the case of the machines we want to emulate we found two different classes. The videogame consoles and arcade machines use to implement only a few interrupts related with the video hardware and sometimes with the sound hardware. The video hardware will signal vertical synchronization interrupts after the full screen has been drawn and horizontal interrupts after a given number of scanlines (displayed lines) have been drawn. The sound interrupts are signalled when the playback of a sample buffer has ended. Both types of interrupts are very important to be emulated accurately because the graphical and sound output of the emulator could get messed. The explanation is that the code for the emulated machine is very dependent in the hardware and sensible to the timings. The gamepad input is performed using polling to the hardware rather than with interrupts (in keyboards) in most of the cases.

In the home microcomputers exist the typical computer interrupts for the keyboard, the disk and the communication ports and all the other devices which work better using interrupts than polling.

In most of the videogames system the graphic interrupts are used for the main timing of the game. The vertical retrace interrupt indicates when a frame start and when a frame ends. Between the end of a frame and the start of the next the graphic memory can be accessed without corruption in the display so it is when the game code updates the video information for the next frame. In most of those systems the video interrupt is the only interrupt and the only reliable form of timing but a precise counting of the executed CPU cycles. Video interrupts use to be at 60Hz (NTSC TVs/Monitors) or 50Hz (PAL TVs/Monitors). The time of the game is synchronized with this signal, this has an interesting effect in some games which can be directly played in either PAL or NTSC versions of the machine. In the PAL system the same game runs slower than in a NTSC system.

This behaviour of the videogame system has a deep impact in the structure of the emulator. Since the video interrupt is the main periodic event in the emulator the emulation is performed around it. The CPU emulates the number of cycles which corresponds at the time of an interrupt interval (or the time of a frame) then it emulates the graphic hardware, outputs the frame to the screen and emulates the sound. Then it generates the video interrupt and returns to the CPU emulation. This is the general algorithm which is modified in some systems because of the existence of other events: for example a more accurate sound emulation or an H-Int (horizontal interrupt) which implies a line by line of the frame emulation of the machine. The time slice in the CPU emulation is critical for the performance of the emulation, a small CPU slice means a bigger overhead because of the entry and exit code for the CPU core.

Exceptions.

Exceptions are events which happen inside the CPU. They use to be errors in the execution of the code. Typical exceptions are the divide by zero exception, unaligned memory access exception, invalid opcode or one of the more important page fault exception (we talked about it in the memory section) generated by the MMU hardware.

The emulation of the exceptions is performed in the CPU core implementing checks in the functions which emulate the different instructions. For example the divide by zero exception will be generated by a check in the divide instruction. The MMU fault exception is generated in a faulty access to memory. The invalid opcode exception will be generated by a non-matching opcode or invalid data in an instruction byte stream (multibyte variable length instructions). In an interpreter emulator the emulation of exceptions just adds the additional overhead of the checking, this overhead can be very big in some cases (MMU exception). In most of cases the exceptions are not needed for the correct emulation, for example an invalid opcode will not be needed in the emulation of a game because theoretically the program is already working. We will try to avoid any useless emulation of the exceptions.

In a binary translator exceptions have additional problems and it is even greater the necessity of avoiding its emulation. The problem with an exception is that it is produced by a precise instruction and the exception rutine handler will need the exact CPU context at the point when it happened the exception for solving the problem correctly. Standard binary translation hides some on the state of the emulated CPU. In a binary translator the state of the emulated CPU only is needed to be equivalent to the real CPU when the CPU emulation stops. In fact after each translated block end the CPU should be the correct for the next translated block to start. Binary translators have to implement special mechanism for accurate exceptions (a memory exception can not happen in a different instruction).

There are different techniques some of them using special hardware (Crusoe and Daisy). The main idea is to rollback the state of the CPU to a secure point. From this point the emulation is performed step by step in a more accurate way, for example using an interpreter rather than binary translation. The way such checkpoints are implemented, how is committed the state of the CPU and how the CPU state is restored has many alternatives. In any case the emulation of an exception is always expensive, but if possible it is better to avoid an instruction by instruction check.

In some cases it could be possible to use the same target CPU exceptions for implementing the emulated CPU exceptions. Most OS provide a mechanism for redirecting exceptions (signals in Unix) to user handlers. Such exception handlers should differentiate exceptions produced by the real code for the target CPU and exceptions from the emulated code form the source CPU.

3. Timing.

When the CPUs were slower (only a few MHz) the timing used to be very important. Further more when the program was written in hand assembly. Many programs coded for this early CPUs (the 8-bit and 16-bit era) relied in a precise count of the number of cycles consumed by a given piece of code to run correctly. With those CPUs it was quite easy to calculate the number of cycles a piece of code would take. Today with hiperpipelined superscalar CPUs with specular reordering an exact timing of the code is almost an impossible task.

There are two main reasons to maintain an exact timing in the emulation of the code. In those early machines with slow CPUs programmers tried to use all the power they could get from the hardware. They programmed taking into account how and when each device should be accessed and coded programs which keep performing calculations until the exact moment of the access was enabled. That was to avoid expensive wait loops. The hardware device could only be accessed in some moments, for example the video hardware (video memory and video registers) could only be accessed when it was not drawing the image to the screen. In some cases the hardware provided mechanism to inform when this event would happen (vsync interrupt, line interrupts) but in some cases they did not exist. In other cases the hardware had to be accessed in a precise instant to perform the correct task. For example accessing a sound generator to produce voices (in a hardware which is not intended for that), in this cases the timer which is used for produce the sound is the same code and the number of cycles it is spending.

There are different accuracy levels for emulated timing. In most of cases an instruction accuracy level (the timing is updated after each emulated instruction) is the most accurate level needed. Sometimes though a cycle accurate model it is needed (the different phases of the CPU emulation are counted with all the bus cycles) but this is just needed in some very old machines (old Atari videoconsoles) or in simulation (using an emulator for profiling information about an existent or non-existent machine running some programs). In most modern machines such a precise timing is not needed and as we will see in most binary translator use block based timing (the cycle count is updated after each translated block execution).

In an interpreter implement accurate timing is easy if you have the information about how many cycles spends each instruction. After (or before) the emulation of each instruction a counter of the executed cycles (or the cycles remaining to stop the emulation) is updated. The CPU core keeps running until it exceeds (or before it arrives) the number of CPU cycles it was told to execute. This is the first part of the timing control.

In binary translation it could be used a similar approach emitting cycle update and check code for each translated code. This approach makes the translation quite slow and should be avoided but in many cases (old 8-bit and some 16-bit machines) it is needed for a correct emulation of the machine. In some cases the solution would be to mark some regions of the emulated code as time intensive code and just perform a very accurate time translation for that regions. This can be done because the time accurate code uses to be reduced and it just performs some special hardware accesses (video and sound hardware). The emulator programmer could study the code of the emulated game and pass hints to the binary translator. This can be easily applied with arcade machines because there are a limited number of games, but with videoconsoles or computers is a harder task.

Binary translators will try to reduce the overhead of the cycle count in the translation. In most modern CPUs just providing a check and update of the cycle counter after (or before) each translated block it is enough for providing the needed accuracy for the emulated machine. If the translated blocks are very large another approach could be used emitting time check code after a fixed number of ‘translated cycles’. Then each translated block would have more than one time check. For a paper about timing in binary translation and real time system binary translation it can be checked TIBBIT (Time Insensitive Binary to Binary Translation) [19].

The CPU emulation code (either an interpreter or a binary translator) counts the number of cycles executed since the last call to the ‘executeCPU’ function and when the count arrives to a limit (usually passed as an argument to this function) it stops and returns to the emulation main loop. This number of cycles (the CPU time slice) is the time (in CPU cycles) it takes to happen the next event in the emulated machine. These events are interrupts, emulated hardware updates and output of the emulated sound or video to the target machine sound and video. Usually the events are regular (fixed time interrupts, for example the 60/50Hz vsync interrupt) but they could also be non-regular events (for example disk interrupts).

The emulation main loop starts with the execution of the starting CPU time slice, then it emulates the event (updates hardware, signals interrupts) and calculates the CPU time slice for the next event. Then the main loop jumps back and executes CPU the time slice. If there are more than one CPU in the system each CPU is emulated sequentially in a proportional (to the relative speed in MHz of the CPU) number of cycles. With multi CPU machines it is even more important to find the correct time slice for the emulation because they could be losing too much times in wait loops for CPU synchronization (they use memory and IO ports for inter CPU communication). This time slice is the second dimension of time accuracy and it is usually even more important than an exact count of the executed cycles.

The more current events which happen in the emulated machine and should stop the CPU emulation are interrupts, usually video interrupts (we will see the more common are vertical interrupt after each frame generation and horizontal interrupt after each line generation) and in some cases sound interrupts. All the videoconsoles and arcade machine uses to use the video vsync interrupt (60/50 Hz) as the main timing of the machine. In many cases stopping at this event is enough but if the emulated hardware needs more frequent update, for example if the sound or video hardware is accessed many times in the middle of the frame, it is needed a smaller time slice. Usually, as the video generator works in a line by line basis and the video settings can be changed in the middle of the frame, the time slice uses is the time of generating a video line. This uses to be a rather small number of CPU cycles (in a Master System with a 3.58 MHz Z80 is 220 cycles). This implies a great overhead because of the function call overhead and the restart and end of the CPU emulation.

The last step or dimension of the time emulation is the synchronization with the real (target machine) time. All the modern computers provide mechanism for knowing (with more or less accuracy) the time between two events. Using internal clocks, CPU internal register (RDTSC in Pentium CPUs is a 64-bit register which counts the passed CPU cycles since the CPU reset) or the video vsync signal it can be know the real time that a piece of code spends in execution. In an emulator after a given emulated time it must be performed a check for the real time spent. This check is implemented in a frame based time slice. The check is performed after the time of the emulation of a full frame (the generation of the full image screen) or in multiples of this time.

The time emulation in the emulated CPU level is for emulating correctly the emulated machine hardware and generates correct emulated output. It is something related with how it works internally the emulated machine. The check between the real time and the emulated time is related with the emulator user (the emulator would be still working perfectly and accurately without this check) which it has to feel the same time behaviour than in the original machine. A videogame running two to ten times the speed (or even worst at a variable speed) of its speed in the original system is unplayable and it can be said it is incorrectly emulated. This is an important difference between videogame based computer emulators and the academical or commercial computer emulators (simulators and others), which are intended to run as fast a possible the emulated code in the target machine. Video game computer emulation could be then understood as emulation of real time systems in this aspect. Because it is the emulation of the time for the user the accuracy level needed is less, usually just a 2 o 3 frame based check is enough. It just has to make the user feel it is the same timing of the real machine.

This means that sometimes the emulator will go faster than the original machine, it will stop a while (until the time excedent is exhausted) and it will start again to run. This will happen enough fast and frequently so the user will not notice it. In some cases the observed timing is so important that we will reduce the accuracy of the emulation for implementing a better timing. It is as bad game which run faster than the original as a game that runs slower. In videogame emulation there some tasks which can be skipped, reducing the accuracy of the emulation but keeping a near to the original feeling of the game. For example we are usually emulating the generation of 60 video frames per second, when in true the human eye can just notice a 25 o 30 frames animation and with even less still has a good feeling of animation (some cartoon animations work at 6 to 15 frames per second). As this emulation is very expensive, but it is needed for generating some video effects, it can be a main target to reduce the emulator execution time. It is a common task to provide a frame skipper which enables or disables the emulation of the video as the emulator is running enough fast or too slow than the real machine. This is also an important mechanism which has to be implemented if the emulator could run slower than the original machine in some of the target machines. Something similar could be applied to sound generation reducing the sampling rate for example.

4. Others.

There are a few other topics related with CPU emulation which could be interesting to introduce.

High Level Emulation

The first working emulator of the Nintendo 64 UltraHLE (by that time it was almost the more powerful video console available) introduced the concept of 'High Level Emulation'. This concept is not new but it was not being used in any of the previous emulators or in the same manner. High Level Emulation tries to stop the emulation of the original hardware and the source code as soon as possible (in this case at the library level) and convert it to target CPU functions and hardware.

UltraHLE tries to detect the entry points in the ROM code for the functions which perform the access to the video hardware and sound. It puts a break at these entry points and when the functions are called the emulator gets the raw arguments passed to these functions and uses them to implement the same or similar behaviour using target native functions and primitives. In this case it would be a translation from the N64 3D API to the 3DFX Glide API (a subset of the Open GL 3D API). This increases the performance of the emulator since the hardware of the emulated machine has not to be directly emulated and the code of those functions has not to be emulated. It just converts data for one API to another API and uses the API in the target machine to emulate the emulated machine hardware.

In fact this idea has been working since a lot of early in the world of the commercial binary translators. Most of the static binary translators work with user level applications converting the original OS system calls into target machine OS system calls. Since in most cases those machines work with UNIX OS the translation of the OS services is quite easy. The conversion process is easy and in fact it is speeding up the emulation because executing already compiled target code is faster than emulation source OS level code. The emulator or binary translator detects system calls (they are usually called through special instruction or in a fixed range of address) while emulating the source code. Then it calls a function wrapper which converts the arguments from the emulated machine OS format to the target machine OS format. At last it calls the native target OS service (if exists) or it implements it with a function. The results are converted again, this time from the target machine format to the emulated machine format.

Not only with OS system calls this idea can be also applied with any kind of library for it which could be found a standard API and the function entry points. FX!32 a x86 to Alpha NT static binary translator implements covers or wrappers for a number of standard DLL (shared libraries in Window world). This permits to use native target libraries in the emulation process rather than emulating the original libraries improving the performance.

In the world of the videogame consoles, arcade machines and home microcomputer this approach is rather limited. In all the early systems no real OS, library or API was used and most of times games developers implemented themselves their own libraries for accessing the hardware. Therefore in this case the emulation of the hardware is needed. In some cases this approach can be used to implement some BIOS services (not very used at all though) and primitive OS systems without stopping from implementing a full hardware emulation. For example in original IBM PC and compatibles BIOS and MS-DOS services were usually by-passed and a direct access to the hardware implemented (video hardware most of time). This implies that although the emulation could be intended for user level applications the hardware should still to be implemented.

Now in more modern videoconsoles which have large development kits provided by the console manufacturers this approach could have more interest. New systems are beginning to include more extended OSes and standard APIs (Dreamcast Windows CE) but they usually are not used, game developers choosing for nearer to the hardware approaches and libraries. In any case as the case of the N64 it shows in some cases this approach can be useful. With the N64 it is show that the compatibility of this implementation is limited by the capability to find the same functions in the game code. The number of games fully working in N64 emulators using HLE is smaller than the ones could be emulated with a full hardware emulation approach.

Float Point Emulation

Float point instructions are just another part of the modern CPU instruction sets. The emulation of those instructions should be easy as they theoretically all the CPUs implement more or less equivalent functionalities in their float point cores. This is not all true though. In any case most of our target machines will not need float point emulation.

Float point hardware can be found in some old machines in external hardware to the CPU and in early vector or 3D graphic systems. In fact the implementation of those special chips would be quite easy as their interface and characteristics are known (if not it is another problem, a reverse engineering problem). They just receive some input values (in memory or in special registers), perform a calculation and output the result somewhere.

In the case of the emulation of CPU FP instructions it is just a matter about how they emulated FP instruction work and how the target FP instruction work. Any binary translation would be faster than just interpreted emulation but as the FP instruction are already expensive the overhead of an interpreter is lesser.

The two main problems with FP emulation are the encoding format of the FP value and the precision of the calculations. Most modern CPUs use the IEEE 754 standard FP encoding format and therefore it should not be any problem with the emulation. But it uses to be small differences and problems. Old hardware performing FP calculations can or not use IEEE standards therefore some kind of conversion or software implementation of the calculations will be needed. Precision is a more common problem, different CPUs have different FP register sizes (x86 for examples has 80-bit FP registers, while the common RISC CPUs use 64-bit), some operations use internally a larger data size (Power combined instructions for example). The problems with precision are really important when the target applications are scientist calculations or similar. In our case an exact precision will be rare to be needed so the problem can be ignored. Small precision difference in our target system could mean small differences, for example, in 3D scenarios but without much quality loss.

Vector instruction emulation.

In the last times CPUs are implementing limited vectorial capabilities (x86 different versions of MMX instructions for example). We will not found this kind of hardware in our target systems for emulation (but perhaps some specialized hardware in some cases) unless we start to try to emulate old scientist kind vectorial machines (not really very fun). Taking advantage of vectorial capabilities (integer or float point) of the target machine for improving the execution of the emulated code (for example in binary translation) uses to be too hard. It is more a field for advanced research in vectorizing than emulation. Those instructions, in the target machine, could be really useful though for emulating other parts of the emulated machine, speeding up video and sound emulation.

Lately the new Sony PS2 video console implements a MIPS core with powerful vectorial coprocessors. Time to come it will be interesting to see how the x86 vectorial instructions could be efficiently used for emulating those coprocessors. PS2 emulation is though far from a reality at this time.

Chapter 6. Graphic Emulation.

1. Graphics in computers.

In all user oriented computers the graphic system is very important. As we all know vision is the fastest (greatest bandwidth) way humans can obtain information. This makes the visual systems the best for the computer to output information to the user. The input of information from the user to the computer is just a different think, computers can hardly see (and understand) anything yet. It can be LEDs or other kind of light signals in a panel, or it can be paper printed information, or the most common now, a CRT (Cathodic Ray Tube) monitor and the more modern LCD (Light Colored Diode) displays. All is graphic information. The more useful is of course CRT or LCD graphic changing monitors.

CRT monitors are based in an electron gun which line by line draws in a phosphor screen an image. In a monochrome system just one type of phosphor exists for each dot in the screen. In a color system there are three (red, green and blue) types of phosphor together. The electron shotgun is signalled to which types of phosphor shot in each dot and with which intensity building the displayed color in the dot. The phosphor just maintains the color or for a small time, this means that the electronic beam must redraw the full screen after a time. In fact as soon as the electron gun draws the line at the bottom of the screen it starts again with the first on the top. The full screen image is called frame, and the time to draw a frame is called Vertical Retrace. The time for a line is called Horizontal Retrace. The end of a frame is used as synchronization information with the video hardware and it is signalled with an interrupt and flags in a status register. It is called either Vertical Sync (VSync) or Vertical Interrupt (VInt). Some systems have also the ability of generating interrupts after each line is drawn, they are called Line Interrupts or Horizontal Interrupts.

The refresh of the full screen is performed at fixed frequency. The most common for early monitors and TVs is 50 Hz or 60 Hz (60 Hz is the NTSC system used in USA and Japan, 50 Hz is the PAL system used in most Europe). Videogame computers come in the two flavours: PAL and NTSC systems. Many games detect the version of the computer where they are being executed and adjust their internal timings. The VInt is the main (most times the only) time synchronization. This frame interval which is the most important in the time handling of the videogames will be also our main time synchronization event with the user in our emulator (we have already discussed this topic in the Timing section). The emulation of the hardware will be driven around of the generation of frames.

If graphics are important in any user oriented computer they are even more important in videogame based computers (which are our main targets for emulation). In those systems, in the games, the graphic aspect is the main and almost unique important characteristic of the computer (there are also the sound but it is secondary, you could play without sound but hardly without image, not the way we understand videogames). This makes graphic hardware the most important part in a videogame based computer. The second place in importance uses to be between the sound and the CPU, most of time winning the sound hardware over the CPU. Some systems have quite low powerful CPUs (SNES uses for example a 1 MHz CPU) but with very heavy graphic and sound hardware. In others the difference between CPU and graphic/sound hardware is smaller (Mega Drive/Genesis). The graphic (and also the sound hardware as we will see) hardware is implemented to the take as many of the graphic generation effort from the CPU (relying it for control tasks and in some cases for calculations or implementation of visual effects). As it is a very specialized hardware it is designed to produce the better output with the technology it can be used at the time of the machine development.

There is a difference between arcade machines and video consoles. The video consoles use cheaper and less powerful hardware than the equivalent (in kind and age) arcade machines. Arcade machines implement more powerful hardware than the video consoles, many times enhancing or duplicating them (for example duplicate the number of CPUs or the graphic and sound chips, more memory, larger ROMs for the game data). This is because a console must be cheap to be sold to normal customers, while the number of arcade machine is limited and are only used in videogames centers. In the case of the home microcomputers and different kind of PCs (MACs, IBM PCs or whatever) there are many differences. Some systems are really intended for videogames playing (C64, MSX, Amiga) while other not. Some could be considered as video consoles with extended capabilities, others as real general purpose computers.

The graphic emulation in most of videogames computers will be more than the 50% of the emulation time, sometimes as much as the 80-90 % (more if the sound is not emulated or disabled for faster emulation). This means that a correct and efficient emulation of this hardware is even more important than the CPU emulation. This is not an easy task because of the amount of real work which this kind of hardware performs, and reproduce specialized hardware tasks in software is always far more expensive. The task performed are also quite complex and calculation expensive.

When one starts the emulation of those machines it starts with the CPU (if we are not using an already written CPU core, which happens many times) but until we do not arrive to the graphic emulation we are not in the real hard task of the emulation. This document could have been rather a discussion about graphic emulation (and sound) but the lack time and the large number of different graphic hardware makes this task almost impossible. This will serve as a mere introduction to the graphic emulation problem.

2. Types of graphic hardware.

All the graphic hardware which we will study (or mostly introduce) is designed for CRT based displays. Some system will translate the same behaviour to a LCD based display (handheld videoconsoles as the GameBoy and the GameGear) but the system works in the same way. So vsync signals, retrace times, frames and lines will be in our vocabulary.

Most CRT based graphic hardware is based in pixels. A pixel is a point in the screen, the smallest graphic unit which can be assigned reproduced. A pixel has coordinates, its position in the screen, and a color (in a monochrome display would be black –no color- or the display color: white, blue, green or whatever). All modern graphic systems and most of the old ones are based in pixels. The games provide information about the pixels that the graphic will have to show in the screen (TV or monitor). This graphic information can be managed in different ways: framebuffers, tiles, sprites and tile maps, 3D directives and textures. But at the end all the information is relative to pixels in the screen. The graphic system we will study in the next sections will be all pixel based. The pixel information is translated by the graphic hardware into signals to the CRT monitor or TV (or LCD display). For example in the case of a CRT display the pixel information will be translated to orders to the electron beam. In any case this last aspect of the graphic generation will not affect us because both our emulator and target machine will use pixel based hardware.

In the early days of videogame computers existed CRT displays and graphic hardware which controlled them that permitted a direct control of the electron beam. This graphic system is called vectorial graphics. Those vectorial graphic systems were not related with later systems which implement polygon, line and 3D drawing capabilities. The vector part of the systems is related with the way it was controlled the electron beam. Rather than providing information about each pixel in the displayed screen the vectorial systems provided directs orders to the electron beam. These orders were in form of drawing vectors: a start point for the beam to draw and an end point where to end the drawing.

The game code had to be drawing lines in the screen. As the screen phosphor loses the image in a few moments the game had to tell time to time to the graphic hardware to redraw the line. In a pixel based system is the same graphic hardware which automatically starts to redraw the image after the full display has been drawn. The game code has not to care about this problem. In a vector systems it would have to be implemented by the same game code. This system is hard to emulate in a pixel based emulator because it needs a very accurate timing emulation and the calculation of the fade times for each drawn line. In any case only a few old arcade machines used this system. Vector based CRT displays disappeared soon after and were replaced for nowadays CRT pixel based systems which began to be enough cheap to be used.

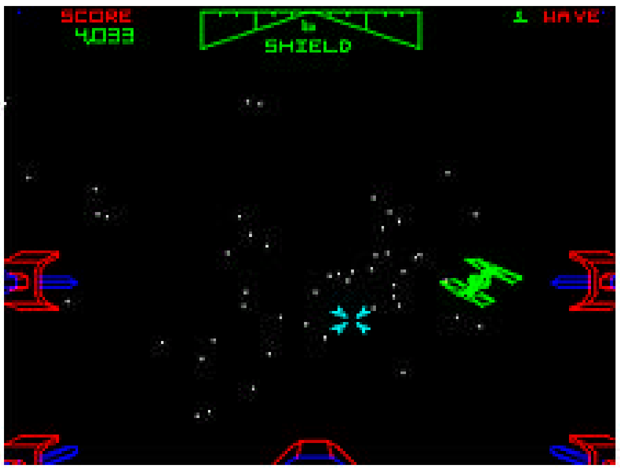


Figure 54. Screenshot of Star Wars vector game (Atari 1983) taken from M.A.M.E. emulator.

There are different pixel based graphic systems. The easier perhaps is the system that is directly based in a buffer or matrix with the color information for each pixel. This buffer is called frame buffer and contains the data of the image which will be shown in the display. Changes in the frame buffer are drawn in the next screen redraw of the changed pixel. Each system will be able of displaying a different number of colors for pixel. The size of the data unit for each pixel determines the number of colors per pixel. For example if each pixel is represented with a bit only two colors are possible. With 8 bit 256 colors can be displayed (counting the no color or 0 value as a different color), with 16-bit up to 32K colors. The color information can be stored in different formats: paletized or not paletized, RGB or YUV. The more common are palette based color data and RGB color data. A palette is a table which maps a color number or identifier with a real color. This real color is identified usually using a RGB data and the range of possible real colors is larger than the displayed colors (the palette range). In paletized systems the information about the color show is keep thus in a separate hardware table and the frame buffer contains identifiers of colors rather than real colors. RGB and YUV are formats of identifying real colors using physical components of the colors, for example the RGB format contains the information about the three basic components of any colors: Red, Blue and Green. These graphic systems can provide mechanism for moving data from and to the frame buffer in different ways (it is called bliting). Most modern home computers (PC or MAC) use this system for 2D graphics. We will talk a bit further about these systems in section 4.

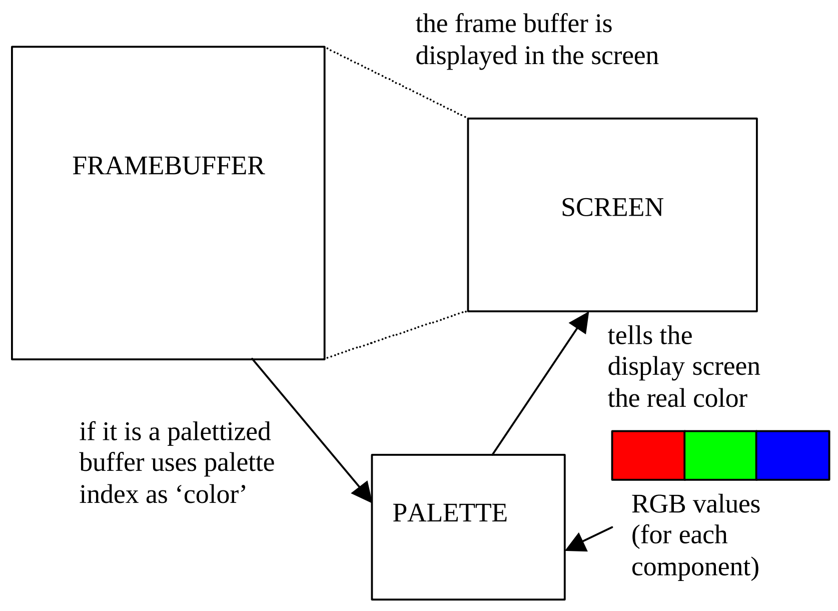

the frame buffer is displayed in the screen
FRAMEBUFFER
SCREEN
tells the display screen the real color
if it is a palettized buffer uses palette index as 'color'
PALETTE
RGB values (for each component)

Figure 55. Framebuffer and palette graphic system overview.

Most of the machines we want to emulate do not use this system for 2D graphics but another which involves a more complex and powerful graphic hardware. In a plain 2D graphic system based in framebuffers and bliting most of the cost of drawing is performed by the computer CPU. Those systems also need large video memory for storing the full frame buffer, and usually it is needed more than a single buffer (primary buffer and secondary buffers) for a more smooth display. In the 8-bit and 16-bit videogame era (80s and early 90s) both CPU power and memory were expensive. Providing a more complex but expensive graphic hardware (but not as expensive as the memory or a very powerful CPU) these problems could be avoided. We are talking about tile/sprite based graphic hardware. The intentions were to reuse as much of graphic data for reducing the needed video memory, to decrease the bandwidth between the video memory (in the graphic hardware) and the CPU (it uses to have a slower access time than CPU memory), reducing the amount of data that the CPU should change to change the display of each frame, and to reduce the amount of time the CPU had to spend with graphic generation implementing as many of the graphic algorithms using faster specialized hardware.

We will explain tile and sprite 2D graphic engines in the next section but in a few words these systems are based in dividing the image in square sections which could be moved and displayed in different parts of the screen at any moment. A same 'square' of an image can be displayed in many places in the screen, reducing therefore the used memory. These squares (they are called tiles) are stored in a region of the video memory. Another region of the video memory has a table (for tiles) or a list (for sprites) which tells the graphic hardware which tile or sprite will have to be draw in each part of the screen. A change of the displayed image can be implemented many times with just a change in the table of displayed tiles and sprites. This reduces a lot the memory bandwidth needed for a single change image and the CPU power needed. The difference between sprites and tiles is that tiles are displayed in fixed positions (the screen is divided in regular square pieces each one storing a tile) and sprites can be displayed anywhere. Some systems will implement both tiles and sprites, some just one of them. Tile and sprite based engines also provide other hardware features to ease the generation of animation like vertical and horizontal scrolling, graphics effects (zooming, tile horizontal and vertical flip, rotations, etc).

Finally in the last years (since late 90s) a new kind of graphic hardware has been introduced in home computers and videogame computers: 3D graphic systems. From the early versions which provided just reduced polygon and line drawing capabilities (some enhanced 16-bit game cartridges like Mega Drive Virtua Racing, or some SNES games for example) to nowadays impressive 3D graphics chips with a full featured hardware 3D engine with polygon drawing, textures, textures and graphics effects and geometry calculations integrated, they have become a common piece of computers.

A 3D graphic system is based in displaying in a 2D viewport a 3D image. Their are based in a 3D scenery formed with polygons. The polygons are projected in the 2D viewport using coordinate transformation matrix. Then if the system implements textures they are drawn over the project polygons. A texture is a pixel image (similar to a tile for example, but usually larger) which is assigned as the color or surface of the polygons. The emulation of 3D systems is either the implementation of the 3D mechanism or algorithms is software (which is quite expensive and hard) or translating 3D directives from a 3D graphic API to another. For example Sony PSX has a 3D graphic engine, Connectix VGS (Virtual Game Station) emulates this hardware in software, EPSXE (a Spanish freeware PSX emulator) implements it using either OpenGL, DirectX or Glide (3DFX cards API) directives.

Emulating 3D graphic systems is a very interesting programming task but explaining how 3D graphics are implemented and the different systems and APIs which can be found is out of the scope (and the time of this project) of this project. Therefore we will not study this systems. For emulating them it is needed a knowledge of the emulated and target 3D APIs (in the PC world the more common APIs are MS crappy Direc3D and SGI OpenGL). Basically, but perhaps the older less powerful graphic chips, all the systems work in a very similar way. In fact chips is in the new videogame consoles and arcade machines are equivalent or versions with a few changes of PC 3D graphic chips (Nec VR2 in DreamCast, NVidia chip in the X-Box).

3. Tiled based graphic engines emulation.

Tile/Sprite based graphic engines are the most common in 80s and 90s videogame computers. They were very popular because of their ability to produce good graphics and animations without too much CPU or memory usage. Different videogame computers implemented tiled engines in different manners. Arcade systems implemented powerful sprite generators with a lot of memory and pattern tables, videogame consoles implemented simpler tile/sprite engines and used less video memory. But all the different engines share the same basic principles. In this section we will introduce how it works one of such engines using as an example the Sega Master System VDP and the Sega Genesis (also called Mega Drive in Europe) VDP. For emulation both graphic and sound hardware the basic is to understand how the hardware works. After this a simple algorithm reproducing the hardware behaviour can be implemented. The next step would be to optimize this basic algorithm to perform a faster emulation. Both graphic and sound emulation has a lot of space for performance improving.

Sega Master System (and its equivalent handheld version GameGear) is an 8-bit videogame console release in the year 1982. It uses a Zilog Z80 at 3.579 MHz as main CPU, has 8 KB of work RAM and the games are packed in ROM cartridge from 16KB to 512KB in size. The hardware is very similar to the MSX computer which uses the same CPU. The graphic and sound hardware is a derived version MSX equivalent hardware. The original Texas Instrument TMS9918 was extended with a new video mode to provide a 32 colors background and 32 colors sprite layer. The SMS VDP (Video Display Processor) has a 16 KB video RAM (VRAM) which is accessed through IO ports by the Z80 CPU. It maintains some compatibility with the old TMS9918 video modes. Arcade machines using an equivalent hardware were also released (System-E, uses two SMS VDPs). The SMS VDP shows the very basics of a tile engine and it is a good example to start with.

The Sega Genesis is a 16-bit videogame console. It was releases at the 89/90. It uses a 7.6 MHz Motorola 68000 CPU as main processor and a Zilog Z80 at 3.58 as sound CPU and for SMS compatibility. It has 64KB of main RAM (68K) and 8KB of RAM for the sound CPU. It can handle ROM cartridges up to 4 - 5MB. The Genesis VDP is also derived from the SMS VDP (and the original TMS9918) but their capabilities are far better. It can handle two background layers with tile priorities, a sprite layer and window (which replaces the background layer). Up to 64 colors in screen from a palette of 512. It has 64 KB of VRAM which is also accessed through IO ports (in this case memory mapped IO, the 68K does not have separated IO ports). The VDP can also perform DMA (Direct Memory Access) operations from and to the VRAM (read, write and fill). The Genesis VDP is already a good example of the capabilities which can have a good tile engine. Although it does not has any additional sprite effects (zooms or rotations for example) it is really impressive. Other videogame consoles, and most of the 16-bit era arcade machines had more powerful graphic hardware but the basics are the same.

The SMS VDP.

We can start with the basics, let see how the SMS VDP works. The SMS screen is 256x192, this mean it is 256 horizontal pixels (columns) by 192 vertical pixels (lines) (some games use a special mode which increases the vertical resolution up to 224). In fact most of the games only use 248 horizontal columns because the first column is used for scrolling as we will see later. The SMS VDP draws a background layer of 8x8 tiles (the tiles are 8 horizontal pixels by 8 vertical pixels) and up to 64 sprites which can be either 8x8 or 8x16. Let see first the background layer.

The background layer.

The background layer can be described as a virtual screen of 256x224 pixels. This screen is divided in 8x8 cells which are called tiles, that means that it is formed by 32x28 tiles. The content of the background layer is defined by a table, called pattern table or name table, which contains information about what tiles are drawn in each position of the background layer and how they will be drawn. This table is 2KB in size, each entry being 16 bit. The information about each tile in the background layer is stored line by line. This table can be found by default at address 3800h of the VDP RAM. The pixel information about of each tile in fact is not stored in this table but it can be found in all the range of the VRAM. The entry in the name table defines a position in the VRAM which contains 32 bytes that define the pixel data of a pattern. This pattern (8x8 pixels) can be used either as tiles in the background layer or for drawing sprites. The same pattern can be used more than once in either the background or the sprite layer which can be very useful for reducing the amount of memory needed. Games usually store as much pattern info as possible in the VRAM so a full videogame stage graphics can be displayed with minimum access to the VRAM. Only when the graphics must really change the VRAM is refilled with new patterns. This permits a faster animation and frees the CPU for other tasks as sound generation, collision detection and sprite movement.

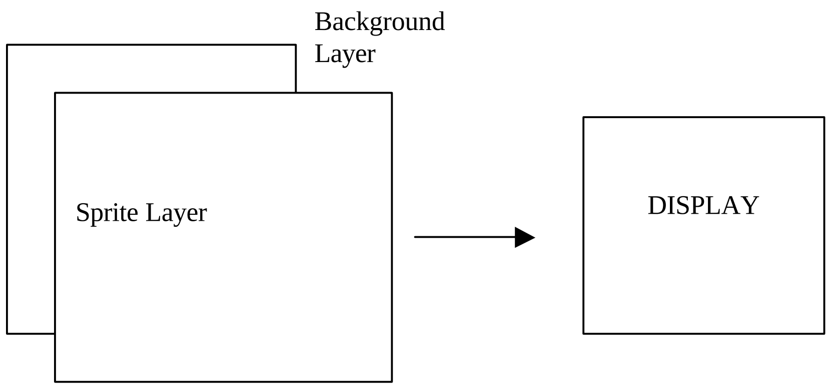


Figure 56. BG and Sprite layers.

The pattern data is stored line by line 4 bytes for each line. This means that it is 32-bit wide for each line, and 8 pixel sized resulting in a 4-bit color value for each pixel in the pattern. This 4-bit color value is an index in one of two 16 entry palettes (the palette table is called CRAM or Color RAM). Which of the two palettes is used when drawing the pattern is defined by the information in the name table or in the sprite list, this means that the same pattern could be used in the image with two different sets of colors at the same time. Each of the four bytes of a pattern line contains the information of the bit n (0 to 3) of the pixels in the line. This is important because this form of storing the color data is not useful for the graphic generation algorithm (but it was good for the hardware engine) and needs a decoding. This decoding can be implemented with a translation table and must be as fast as possible because the more frequent task in the graphic engine emulation will be read patterns and drawn them.

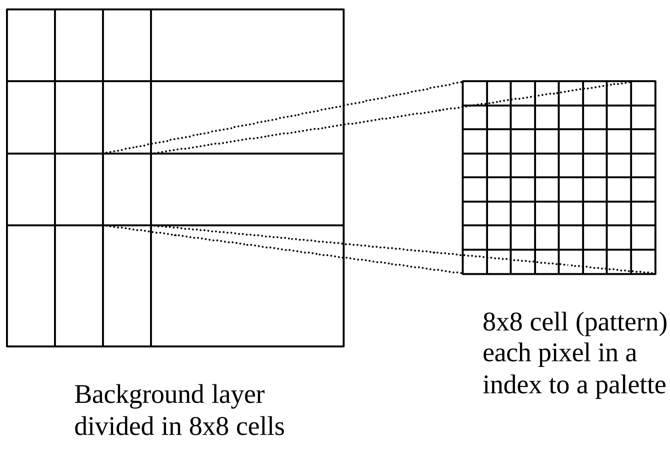


Figure 57. Background Layer.

The entries of the name table contains not only the pattern identifier (a 9-bit number defining 512 patterns, the pattern address in the VRAM is found multiplying it by 32, the number of bytes of each pattern) but also a priority bit and a palette selector bit (only the two 16 colors palettes) and vertical and horizontal flips. The flip bits are useful for using the same pattern in different positions and purposes in the image (for example a single pattern could be used as four corners in a rectangle). It is also useful because moving items based in the background (the moving items are usually implemented using sprites though) can use the same patterns for moving in one direction or the another.

The background layer has the ability of scrolling vertically and horizontal. Many videogames use scroll and having a hardware which implements this scrolls helps a lot. A scrolling image implies that the full screen should be redrawn in a framebuffer based engine (if the framebuffer does not support scroll either of course). This would be really expensive. A better solution is to implement scrolling in hardware. Two hardware registers in the VDP store the vertical a horizontal scroll value. The scroll value tells which line and which column will be the start point for the screen drawing. For example HScroll 8 and VScroll 120 means that the first point in the upper left corner of the screen will be the one which can be found in the position (8,120) of the background virtual screen. The line and the column wrap when they arrive to their limit. For example if you arrive at column 255 of the background line and your screen line has not been fully drawn you continue drawing the pixels from the column 0 of the background line.

For scroll to work the information about the tiles which will be shown in the next frames must be stored in the name table. While the screen is being drawn it is not possible to change the name table (it could produce a graphic mess or just the hardware does not allow this access), only in the small time slot between the VSync signal and the start of the next frame VRAM can be freely accessed. For a proper scrolling it must exists a hidden part of the virtual background which is not drawn on the screen. This part is where the image is updated for the next frames. The SMS VDP has 32 lines which are not shown meaning that up to 32 pixel vertical scroll steps can be performed before new information must be loaded in the name table. But it lacks a horizontal scroll buffer. This is solved with a bit in the VDP control registers which hides the first displayed column. More powerful tile engines have larger virtual background layers with far more scrolling buffer space.

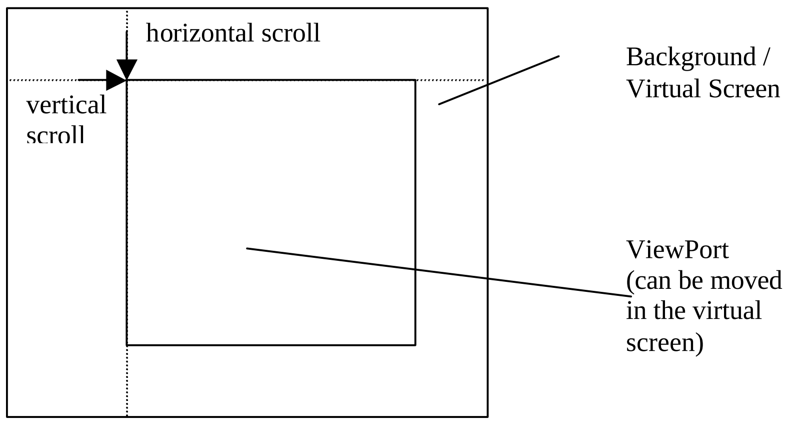

horizontal scroll
vertical scroll
Background / Virtual Screen
ViewPort (can be moved in the virtual screen)

Figure 58. Background and scrolling

This explains most of the part related with the background layer. In next sections we will explain how it is emulated. This will work as a basic description. It is still missing the information about the priorities.

The sprite layer.

Sprites can be considered as mobile tiles. They also use to be drawn over the background layer. Basically the background provides the scenery of the game and the sprites the actors and other moving elements. This does not happen in all the sprite based engines. In some of the more powerful sprite based arcade machines the background layer is not built with tiles but with sprites. The property which differentiates a tile and a sprite is that a tile is a fixed size square of pixels which is drawn in a fixed position of the screen. A sprite can, in some cases, have different sizes and can be displayed in any position of the screen or of the virtual screen. The tile information is stored in matrix table, the sprite information is stored in list with X and Y coordinates.

The SMS VDP can display up to 64 sprites and a maximum of 8 sprites in the same line (this could be because of a hardware time limitation, other systems permit as many sprites as can be displayed in a screen line). The VDP starts drawing from the first entry of the sprite list and ends either when 8 sprites are displayed in the same line, a sprite with an Y coordinate of 208 is found (marks end of the sprite list) or all the 64 sprites have been drawn. The sprite list or table can be found at address 3f00h of the VRAM (default position). Each entry contains the vertical and horizontal coordinates of the sprite in the virtual screen (it is usually used the same size of the background layer) and the identifier of pattern used by the sprite (only 256 patterns selectable) which works in the same way than in the tile table. The SMS VDP provides just basic sprite capabilities and it does not implement any additional effect applied to a single sprite. Other systems permit sprite zooming, may be rotations and sprite flips (as the SMS tiles have). In the VDP control registers it can be selected to zoom all sprites to 16x16 (not too useful) and to use 8x16 sprites, that is two patterns put one above the other. In this case the last bit of the pattern identifier is discarded accessing only even positions of the pattern data.

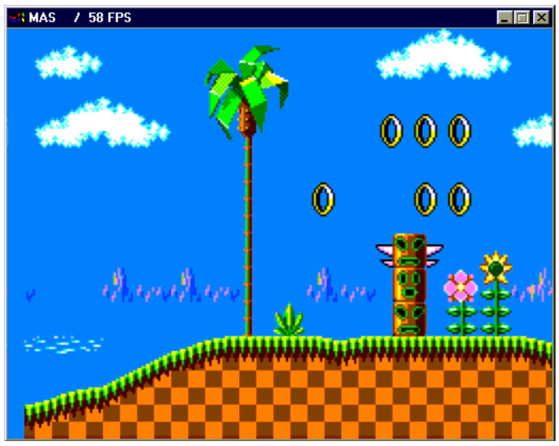
Handling priorities.

An another characteristic that sprites and tiles have is they have a transparent color. When this color is found in a pixel from a tile or a sprite in the point of the screen where this tile or sprite is drawn it is showed the color of a tile which is behind the tile or the sprite. This is used to produce different levels in the animation (for example with different scroll layers it can be simulated a feeling of a 3D scenary) and object animations. The objects in a game are not all square shaped but they can have any form. The sprites which form those objects are square shaped though. Using the transparent color in those pixels of the sprites which are not part of the object enables the animation of non square shaped objects. All this process is performed by the graphic hardware freeing the CPU from a task which would be really hard to implement because it needs comparisons and mask operations for each pixel of the screen. The more levels of sprites or tiles which can be drawn above one of the other the hardware it will be to emulate the video hardware in software. The transparent color uses to be coded using the '0' identifier.

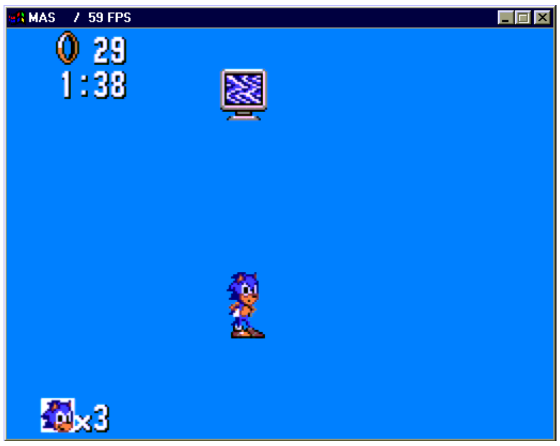
This is the basic behaviour. To this transparent color it is provided a priority policy for each tile layer and the sprites. The tiles and sprites are ordered from the bottom to the top. The pixels tiles or sprites in the bottom will only be showed if all the pixels of the tiles and sprites above it are transparent (use the transparent color). Usually there is a main ordering between the different layers and then there is a priority bit which indicates when the priorities should change (for example a background tree which has to be showed in front of a moving object).

The SMS VDP has only two layers: a background formed with tiles and a sprite layer where a list of sprites is drawn. By default the sprites are drawn over the background tiles but in the entries of name table which defines the background exists a priority bit. When this bit is enabled this tile is shown over the sprites. This means that when emulating the background layer we need to keep the priority of each pixel before starting to draw the sprites. Since a sprite can be displayed in any position of the screen it is also possible to find overlapping sprites. In this case the order of the sprites in the sprite table determines which is shown. The sprites are drawn from the first entry of the table to the last. Sprites from the start of the table are shown in front of sprites in the end of the sprite table.

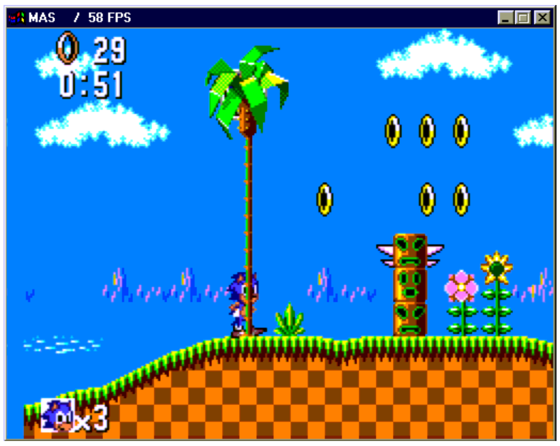
The SMS VDP is quite simple and the priorities are easy to implement. Other systems which a larger number of layers and sprites and with different levels of priorities in the same layer are really hard to emulate. The emulation of the tile and sprite priorities and transparency is perhaps the harder part in the emulation of a tile based engine.   It must be taken into account that each for each pixel in the screen the priorities of each of the pixels in the different layers and the sprites must be calculated, the color compared with the transparent color to determine which color must be drawn. This is one of the better places to try to improve the performance of the graphic emulation.



Sonic (Master System). Background Layer.



Sonic (Master System). Sprite Layer.



Sonic (Master System). Background and Sprite Layer. High priority background tiles (palm tree).

Figure 59. Background and sprite layer with priorities in the Sega Master System.

The palette.

As we already said the information stored in the pattern data (which is used to form the sprites and tiles) is not a color itself but a color identifier. In fact is an index into a table which codifies colors. This table is called CRAM (or Color RAM) and in the case of the SMS VDP has 32 entries (a maximum of 32 colors at the same time in the screen, theoretically) and each entry is 6 bit. The color is in RGB format, 2 bits for each color component. RGB is a form of describing a color based in the properties of colors. All the colors can be discomposed in three basic color components: red, blue and green. A color can be defined by the intensity of each of these components. Colors with all the same intensity in all three components are grey tones. The black color is the one with all the components to 0 and the white color is the color with all the components to the maxim value (in this case 3). With 6-bit RGB values 64 different colors can be selected.

The GameGear palette uses 3 bits for each component, therefore it can select up to 512 colors, but still there are only 32 palette entries allowing to just 32 colors on screen. This is a common characteristic of the palettized graphic systems, it allows with a smaller number of bits per pixel to display a larger number of colors. It can not be shown more colors at the same time that the number of entries but it is easy and fast to change the displayed colors between a larger range of colors. In fact they could even be changed in the middle of the frame doubling the maximum number of displayed colors as we will see in the interrupt part.

The graphics systems based in palettes are also very efficient producing effects which need fast changes of colors of large, or all, blocks of the screen. While in a system based in raw RGB colors (or another equivalent format) to change the color all the pixels must be changed one by one in a palettized system the pixels does not change but just the entry or entries of the palette table. This is another hardware feature which enhances the graphic generation and reduces the CPU usage. Emulating it without a palettized graphic mode increases the overhead of the hardware emulation because the translation of the palette code to the real color must be performed by software in a pixel to pixel basis. The advantage of graphic modes based in RGB (or any raw color definition) values for the pixels is that a largest number of colors can be displayed at the same time, but also with a larger usage of video memory.

The 32 palette entries are treated as two separate tables each with 16 entries. Color is defined in the patterns as a 4 bit value therefore each tile or sprite can only use one of the 16 color palette tables. For tiles can select which palette to use with a bit in the entries of the background table. Sprites can only use one of the palettes which is specifically assigned for the sprites.

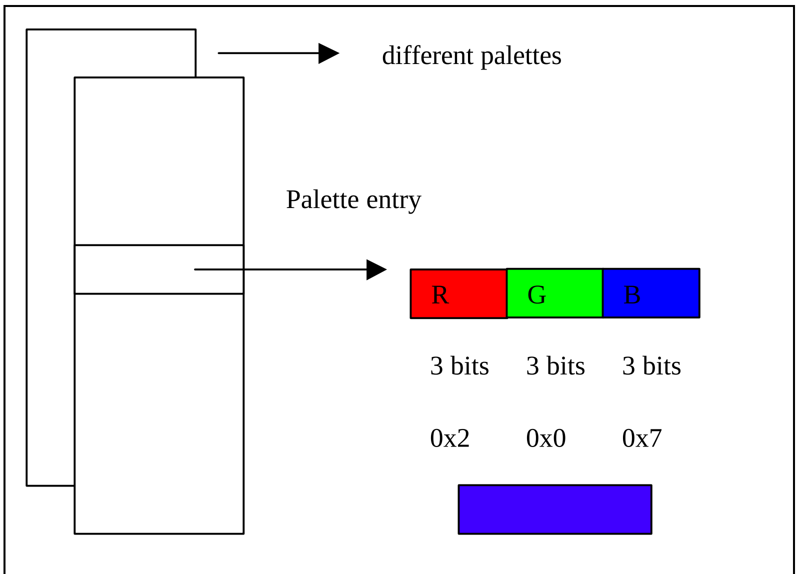


Figure 60. Palette overview.

Vertical and horizontal interrupts.

With taking into account diverse control registers which select different displays modes and specific and not very important features in the SMS VDP remains just one important topic to explain about graphics in the SMS. This topic is the graphic interrupts.

As we already said in previous chapters and sections the videogame computers used the graphic interrupt systems to control the timing of the games (sometimes they used to the sound). This timing is produced with the vertical interrupt (VInt also called vertical sync or VSync). The VInt is produced by the SMS VDP hardware (and in fact it works the same in all other systems, it can be considered a standard) after the VDP finishes drawing the last line of the screen. The interrupt signal (there is also a bit copy in the VDP status word) is being signalled in each line (the time the beam takes until it starts to draw again the screen is usually counted as lines) until the CPU reads the status word or the starts again the frame draw. This signal serves as synchronization for the game and as a start point for the time when VDP VRAM and control registers are accessed and the modifications for the next frame are performed. Its emulation as we said is produced knowing the number of CPU cycles (for the CPU which is being used as reference in a multiCPU system) that the VDP takes to draw a full frame. VInt emulation is mandatory for implementing any emulator using this kind (or even any kind) of graphic system and to emulate the time of the games.

VInt is used as a global timing synchronization event and for updating the VDP for the next frames. There is still another interrupt which can be generated by the graphic hardware and it is implemented in many tile based systems. It is the horizontal interrupt (HInt or line interrupt sometimes). This interrupt is signalled after a number of lines of the screen are drawn. There is a counter register in the VDP that when HInts are enabled it is decremented after each screen line is drawn. When the counter reaches to zero the HInt is produced, and in the case of the SMS VDP the previous value restored and the counter starts again to decrease until the HInts are disabled. The HInts are only generated in the lines which are really drawn to the screen (without taking into account

As the VInt emulation is really needed for a correct or even any emulation of the computer the HInt can or can not be implemented. It will just change how accurately the graphic system is emulated. As we already have said as the graphic and sound hardware as so complex and expensive to emulate sometimes there are features which are not implemented because they would slow down the emulation a lot. HInts is one of those features. HInts are used to produce the so called raster effects. These effects are changes of the VDP graphic parameters in the middle of the frame, when the VDP is already drawing the image to the screen. This means that just a part (a number of lines from the start of the change) of the screen will be modified by this change.

Usually the graphic parameters (palette, scroll, tiles and sprites) are only changed in the time between frames and not in the middle of it, that is the use of the VInt. An important difference is that the time for performing changes after a HInt is very limited and even in many cases the same graphic hardware blocks most of the parameters. The number of parameters which can be changed in the middle of the frame depends in the freedom each graphic hardware permits. It also depends in how fast is the main CPU and the rest of the computer, for example it will be possible to perform more changes in the Genesis running a 16-bit CPU at 7.67 MHz than in a Master System running a 8-bit Z80 at 3.57 MHz.

The parameters which are the most changed in raster effects are the palette entries and the horizontal scroll registers (the vertical scroll is not changed because it would produce rather strange effects). The palette changes permit to use more colors in the screen that the number which can be showed by default. It is also used for produce fast changes of color (for example for implementing flashes of part of the screen). The horizontal scroll changes can be used for screen splits or even (in more powerful systems than the SMS) a different horizontal speed for different lines of the screen. This is useful for a greatest impression of a 3D scenary. Sprite changes and other modifications related with specifics of each system can be also implemented.

The emulation of the HInt is therefore only needed (in most of the cases) just for some graphic effects that some games implement. In fact in many videogame consoles just a limited number of the games (but they are usually the technically best games) use these features. Some of the games could, though, depend in some aspect of the HInt and do not run at all in the emulator without it, but this case is rare. This leaves the decision of implementing raster effects and HInts to the programmer. The decision also implies as we will see to change the way the screen image will be generated by the emulator. If HInt is emulated it is more likely that a line by line basis (or some mechanism for tracking the changes in the time) drawing of the screen will be needed. Without HInts and raster effects it is better to generate the full frame after the CPU time of a full frame has been emulated. In any case if we want the most exact emulation of our emulated machine HInts and raster effects, if they exist in that machine, should be implemented.

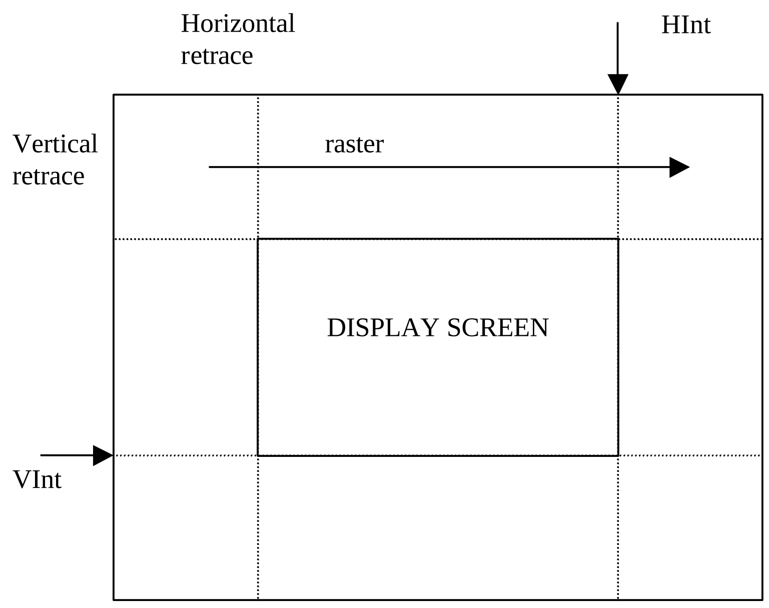


Figure 61. Vertical and horizontal retrace and signals.

Emulating the SMS VDP.

We know now the general structure of the SMS graphic hardware. A background layer built with tiles and a sprite layer. Both the sprites and tiles use the same 8x8 4-bit per pixel patterns stored in the VDP VRAM. The 4-bit pixel value is used as an index in a color table, the palette. There are two 16 entries palettes and each entry can select between up to 64 different colors. Now we have to implement in software the same or equivalent behaviour than the VDP hardware. Using the same inputs (VRAM contents: pattern data, background name table and sprite table; and the VDP control registers) we should be able to produce the same output, the screen image which displayed by the original SMS.

There are different ways of generating the image which will have to be displayed in each frame. We can replicate as much as possible the way the VDP really works drawing line by line. The VDP works in parallel with the CPU but us our emulator will be implemented sequentially we will have to interlace CPU emulation with graphic emulation. Usually the faster solution is to interlace at a full frame basis, emulate the CPU all the time of a frame (VInt event) and then generate the image from the data stored in this moment in the VDP VRAM and registers. This is the faster and more primitive implementation. For a more accurate emulation as we already said talking about HInts and raster effects it is needed to interlace the emulation in a line basis. The CPU emulator runs for the time that the VDP takes to generate a full line, then the graphic emulation generates the line and draws it to a backbuffer. At the end of the frame time the backbuffer is drawn in the target machine screen (blitted to the primary buffer). A line by line engine is quite more accurated because it does not miss any changes (if correctly timed the CPU emulation with the VDP) in the VDP but it is a lot slower than drawing the full frame at once.

An alternative is to track all changes produced in the VDP in all the frame time, identified with a time stamp to replicate the same behaviour of a line by line engine. This complicates the render rutines. Another step is to try to avoid unnecessary calculations, in this case try to only draw those parts of the screen which have actually changed. In most cases only a small portion of the screen changes from frame to frame, for example a sprite which has been moved or a tile which has been changed. The idea is rather than draw the entire frame every time the VDP emulation rutines watch for changes and stores them. At the time of the frame generation only the changes are applied to the image. This reduces a lot the graphic emulation time. This technique is called dirty rectangles, meaning that it tracks the portions of the image that has changed and only copy them to the screen. These portions are rectangles to ease the managing and the copy process. This kind of graphic engines are very fast but are a lot of harder of implement that a typical engine.

A slight modification of the dirty rectangles technique is generating at the same time of the modification of the data in the VDP the corresponding changes in the image. The image would be stored in a backbuffer and the changes would be applied as they would be performed. At the end of the frame the backbuffer (in fact just the part which changed) would be dumped to the screen.

This kind of engine is very fast if the number of changes in each frame is kept to small number. However the same characteristics of the tile based engines produces situations where a lot of changes in the screen are performed in just a moment. Palette changes can mean a redraw of the entire backbuffer image if the buffer uses RGB values for the pixels. Scroll also hurts to the engine because each scroll pass means that the full screen image is moved a position and must be redrawn. If the graphic emulator was keeping the full virtual screen drawn perhaps it would be just a matter of blitting (copying to the screen memory) the appropriate parts of the virtual screen to the target screen. If not the full frame should be recalculated. In any case is expensive and the performance would be the same, or even worst because of the checkings needed to keep a list of all the VDP changes, that a graphic emulator drawing all every time.

The process of the generation of a frame can be divided in three phases. First the pattern data is decoded to a useful form. As we said the 4 bytes which for a pattern line store each one of the bits of the pixel color (4 bits per pixel). This format for storing the color is not useful for us because we need to work with the color of each pixel. Then we have to retrieve to reencode the data of a pattern line in a useful format. A better format for using this data in a software renderer is that each pixel color was stored in a byte. We need therefore to read the four bytes of a line and extract the bits which form the color for each pixel. If this process was to be done every time a pattern is read (while generating the background layer or the sprite layer) the cost would be too expensive. Because of this conversion is better to be implemented when a VRAM write is performed in the case of the SMS VDP. The amount of memory for storing all the pattern data in a decoded format is not that large (64KB). It is also needed to implement this conversion as fast as possible, usually using tables to decode the information faster.

Leftmost pixel 0: bits 31 (msb of colour), 23, 15, 7 (lsb of colour)
1: bits 30, 22, 14, 6
...
Rightmost pixel 7: bits 24, 16, 8, 0

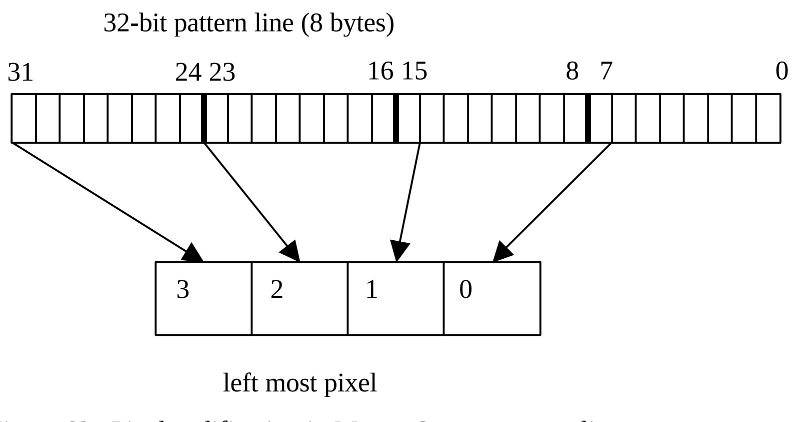


Figure 62. Pixel codification in Master System pattern lines.

This first phase can also be performed at the initialization of the emulator. Most arcade systems have separate ROM ICs for program code, graphic and sound data. Most of times these graphics ROMs are directly attached to the graphic hardware to avoid the need of perform so many memory transfers. This property can be used, knowing how the graphic data is stored in the ROMs, to decode the graphic data and prepare it for the emulation when the ROM file is loaded at initialization time. In videogame consoles this can not be done because it is not frequent to have a separation between the different types of data in the cartridge ROM.

The second phase is start with the generation of the graphics in the different layers. We will think as we were working line by line (although it could be also applied in a full frame generation context). In the case of the SMS VDP exists two layers: background and sprite layer. Usually it is better to use the default order of drawing of the VDP, in this case first the background is generated and then the sprite layer is drawn over it.

For generating the background layer first it is read the name table or tile table. For a given line **y** it is calculated using the horizontal and vertical scroll value and the line number which tile must be read. Then this entry of the tile table is read, this provide us with the base address for the pattern which is being to be shown in this tile. Using the modulus of the line number and the vertical scroll (and detecting if the tile must be vertical flipped) the proper bytes for the line are read from the pattern. This will be already decoded data so we will store it in a buffer which contains the generated background line. In the case of the generation of all the frame at once it would be better to read and draw the full tile at a time (draw in 8 lines by 8 lines basically). We need also to adjust the color according to the palette used (the tile layer can use both the tile and sprite palette) and store the priority for each pixel in the line.

In a frame generation basis it would be more efficient to generate the entire background layer (limited by the screen viewport) and later apply the sprites because reduces the number of the access to the sprite table. In the case of a line by line generator now it is searched in the sprite table sprites which should be draw in the line we are generating. It is used the y coordinate of the sprite, the line number and the sprite size (8x8 and 8x16 sprite sizes). When a sprite which matches the y coordinate is found it is read the appropriate pattern line of the sprite (the same way we did in the case of the tiles). Then it is time to calculate the transparency and priority of the sprite pixels and the pixels already drawn in the line. If the priority is low and the sprite pixel is not transparent (color value 0) the sprite color is stored for that pixel. If the priority was high and the stored color was not 0 the sprite color would not be used. It must be taken into account if a sprite is being drawn over another sprites, this means that the sprite drawing must also update the priority information.

The SMS VDP also has a bit which is set when overlapped non transparent sprites are found (two non transparent pixels of different sprites are drawn in the same position) which can be used in some cases for collision detection. This feature should be also emulated.

After the sprite and the background layer are generated we have a buffer storing the pixel colors of a line (or the full frame). In fact it is not still the color but a palette index. If our target computer uses a palettized video mode we just have to convert the SMS palette values to the target palette values (sometimes they can be directly the values, if not it could be added an offset if the first entries of the target palette have to be used for other tasks). The real color is emulated translating the writes to the SMS palette into equivalent writes in the target machine palette.

If the target video mode is not palettized the final phase must be performed. Emulating how it works a palettized graphic hardware each byte (if a palette value is stored in a byte) from the frame or line buffer is read, the value is used to index a table of colors and the obtained color (which will be already in RGB format) will be stored in the final framebuffer or in a secondary buffer. This task can be performed when the full frame has been drawn, in both line by line and full frame renderers. At the same time or after the frame buffer is converted to RGB (or any other color codification) it can be applied special effects to enhance the final result, for example filter for reducing the pixelation when using a video mode with a higher resolution than the original console video mode. Popular effects are bilinear and trilinear filters (some implemented now by the video cards in hardware), Eagle and 3XSAI which have the same purpose but with better algorithms. This filters are very CPU intensive (if not implement by the video card hardware) so this must be taken into count when implementing them. Usually they are an option rather than the default setting. Another popular effect is scanline which tries to keep a separation between the displayed lines to simulate the aspect of a TV in a computer monitor.

Finally the result buffer is blitted (copied) to the target machine screen, this can be more or less expensive depending in the target video hardware. The modern graphic cards implement the more common blitting operations by hardware and in an asynchronous way so the CPU can resume the emulation work as soon as the graphic hardware has been setup for the blit. We will talk a bit about this in a later section.

This is basically how it works a basic tile renderer. As we will see in the next section talking about the Genesis hardware this is just a small view of the tasks which must be performed. We can see that the more expensive part is the drawing of the background and calculation of the transparency and priorities between the different layers. Another part which has a big impact is the final conversion to the target machine graphic format and the final blitting. As the PC and other systems which are used to implement emulators grown in capacity it is more common to emulate the original video modes with larger resolutions and greater bit depths (the number of bits per pixels, determining the number of color). This means a large reduction of the performance due to the increased memory needed for storing the full frame, the time for generating it, apply the filter effects and blit it the video hardware which increases the memory bandwidth needed. In some cases the oldest systems will suffer from a large performance penalty if faster modes are not implemented.

A more advanced VDP: Genesis.

In this section we will see a more advanced example of a tile engine and we will see the additional problems we could found in the process of emulating the graphics. We will also talk about other features which can be found in the videogame computer graphic hardware.

The Genesis has a display screen of 320x224 pixels (NTSC) 320x240 (PAL). The Genesis VDP is an advanced version of the SMS VDP and keeps a backward compatibility mode with the SMS VDP to be able to use the SMS games in the Genesis. It can show up to 64 colors in screen (4 palettes of 16 colors each) from 512 selectable colors (9 bit RGB). The VRAM is 64KB in size. It has a CRAM for the palettes and a small vertical scroll RAM (the scroll can be performed in the full frame, tile by tile or even line by line).

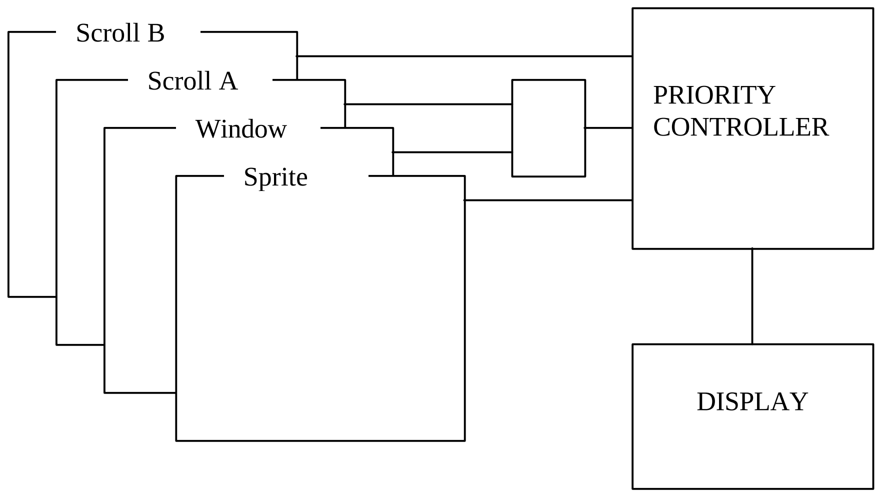


Figure 63. Genesis VDP overview.

The Genesis has two background layers, named A and B, and a sprite layer. A special layer called window can substitute the layer A if it is enabled. Both the tiles in the layer A, B and the sprites have a priority bit. The default ordering is sprites in front layer A and layer A in front of layer B. With the priority bits set the ordering is sprite high priority, layer A high prio, layer B high prio, sprite low prio, A low prio and B low prio. Both layer A and B are larger than the actual Genesis screen up to a 512x512 virtual screen. This allows a greatest scroll buffer than the SMS VDP. The name tables for both background layers are 8KB in size allowing for different virtual screen combinations: 32x32, 32x64, 64x64, 128x32, 32x128 and so, only with the limitation 8KB limitation. Each entry is 16 bits and contains the pattern identifier, a field for choosing which of the 4 palettes to use, the priority bit and vertical and horizontal flip bits. The pattern can be 8x8 or 8x16.

The Genesis can display 80 32x32 sprites. The sprite table contains the x and y coordinates of the sprite, the size of the sprite (how many horizontal and vertical 8x8 cells have), a priority bit, a palette field selector and vertical and horizontal flip bits. It also has a very common feature in sprite based engines, a link field which is used to create lists of sprites to be displayed. Rather than use a sequential order following the sprite table, the Genesis VDP draws the sprites starting from the entry 0 and following the content of the link field until a link entry with a 0 value is found. This is useful for having different lists of sprites to be displayed and to perform fast changes between them (it just needs to change the entry 0 link field). With two or more lists it can work in a similar manner than a secondary buffer in frame based graphic hardware. The Genesis VDP can draw as many sprite pixels by line as pixels in the line. The full screen could be formed with sprites, this is typical from the more advanced sprite based computers. The size of the sprite table is 640 bytes. Both the background layer tables and the sprite layer are in the VRAM with the pattern data.

Finally, the Genesis VDP can perform vertical scroll for the full screen or scroll independently groups of two cell columns for both background layers. The vertical scroll for each two cell columns is stored in the VSRAM which accessed separately from the VRAM. The horizontal scroll is stored in a 960 byte table in the VRAM. It can perform independent scroll at the line level and the cell level. It can of course scroll the full screen at a time. The last features are a window, which is defined with a smaller tile table which can be drawn in any position of the screen as a substitute of the layer A and a shadow highlight capability which is used to change the intensity of the displayed colors using the window and a list of rules.

The Genesis VDP also implements DMA access (Direct Memory Access) between the 68K memory and the VDP memory. The first reason is because the access to the memory VDP is performed through IO ports (in this case mapped in the 68K memory address space) the same than in the case of the SMS VDP. The problem is that although for the Z80 the IO access was fast enough it does not happen the same with the faster 68K. DMA is a technique used to move data from the memory to the hardware devices (and from the hardware devices to the main memory) with passing through the CPU. The bus between the hardware devices and the CPU with the memory is the same so while a DMA operation is being performed the CPU can access the memory (and in fact it is stopped because it can not fetch new opcodes).

The Genesis VDP can perform data writes and data read from the VDP memory and also fills (write the same byte in the range of the memory where it is performed the DMA access). This feature must be emulated because many games use it for moving data from the ROM cartridge to the VDP memory. The emulation of the DMA implies to emulate the data movement, which can be implemented directly as access to the emulated main and video memories, but also the emulation of the time it takes to perform this operations. As the DMA operations take time from the CPU it must be calculated the amount of CPU cycles it takes each of this operations and this number used in the calculation of the passed time. If this is not emulated there will be problems with games which are very time dependant.

If we take into account the two layers, the sprite layer with a larger number of sprites displayed in each frame and the scrolling features we can understand why it can be so expensive to emulate the graphic hardware. The Genesis VDP is rather powerful but other machines, like the NeoGeo, the Super Nintento or the Saturn implement even more powerful 2D features. The process of calculating the priorities and the mix of the different layers will take a good amount of the processing time. All the layer pixels must be calculated with a priority and later for each pixel it must be decided the color of which layer must be drawn in the screen. The line by line scroll increases the cost of deciding which pattern must be read each time.

The Genesis VDP does not show all the features which can have a 2D graphic engine. Other graphic systems can implement scaling (NeoGeo) and rotations in the sprites. The SNES has a special video mode (called Mode 7) which provide special scaling features which are useful to produce near to 3D 2D graphics. Those special graphics effects in tiles or sprites have to be implemented in hardware and increase a lot the amount of CPU cycles needed to emulate the graphics.

Until now we have talked about how can be implemented tile/sprite based engines using a framebuffer based graphic hardware, but it could be possible to use a hardware sprite engine to emulate another hardware sprite engine. In the PC world does not exist any videocard implementing such engines but there are consoles and home computers which have this feature. Lately many videogame consoles has started to have emulators. Modern 3D consoles does not implement any more tile engines (or not in the same way) but older ones still use them. For example the Saturn which has basic 3D capabilities has also one of the best 2D graphic hardware. A project called Stardust tries to emulate the Nintendo 16-bit console (SNES) using the 2D tile hardware of the Saturn (a Sega 32-bit console) [28]. The project is still in development but it is said to become open source when it will be finished.

The way a sprite based engine can be implemented using another sprite based engines depends a lot in the difference of the two systems. The efforts will be made to convert the pattern data from one format to the other and arrange the tile and sprite layers in an equivalent way. For example if the emulated graphic system use tile based background layers and the target graphic system only uses sprite layers the background tile tables will have to be translated into sprites (building the background with sprites) for the target system. The handling of the priorities can be harder to implement if the two systems are too different.

4. Plain 2D graphic generation.

In the videogame computer world we will not find many times this kind of hardware (but perhaps in very old systems, like the Space Invaders arcade machine which uses a monochrome framebuffer based graphic hardware) but we have to know about it because it will be graphic hardware we will find in our target machines. In fact this was true for the old 8-bit and 16-bit systems but modern 3D based videogame consoles and arcade machines use for 2D graphics this kind of graphic hardware. The reasons are diverse: memory and CPU are no more a problem so the reasons for using 2D tile based engines are gone, and the graphic chips used in modern videogame computers are the same or based in the same principles than the PC world graphic chips (3D graphics).

We will see basically how it works a framebuffer based graphic engine. The video ram is used now, as its first purpose, to store the frame buffer. The framebuffer is a matrix of pixels of the size of the displayed screen or larger if the graphic hardware implements virtual screens and scrolling. Each position of the matrix stores the color of the pixel. The color, as we already said in the previous section, can be implemented using palette indexes or directly using RGB values (or other color definition formats). If the frame buffer is palettized it will use less memory (palette based systems usually have from 4 to 256 different entries, or what is the same from 2-bit pixel depth to 8 bit pixel depth). RGB based framebuffers need a greater color depth (the minimum uses to be 15/16 bit depth) and therefore more video memory.

For implementing graphic animation with this kind of hardware the game code, the CPU, has to replace all portions of the frame buffer which has to change from new data from the main memory or, sometimes, from the video memory if there is an excess (there is more video memory that the used by the framebuffer, this is quite common now). The process of copying a block of data from the main memory or the video memory to the video memory is called blit. The blit can be performed by software using the CPU to move data from main memory to video memory or it can be performed by hardware. In this case is the own graphic hardware which perform the operation, usually as an asynchronous operation, allowing the CPU to perform other tasks. In the early systems (CGA to VGA and SVGA in PCs) this task was performed using software. In modern PC graphic cards hardware blit already exists. The blit can be even faster if it is perfumed from video memory to video memory. If the video memory is large enough to store the full frame buffer and additional graphic data this feature can be use to produce a faster animation.

There are diverse blit operations. The basic one is just copy a block of data with any changes directly in the video memory. However, as we already said while talking about 2D tile engines for animation it is needed to draw objects with transparent parts in front of an already drawn image. Those transparent blits must also be performed and can be supported by hardware or not (most modern graphic cards support it). The transparent color is called key color (DirectDraw API). The blits can be performed in all the framebuffer or just in part, but always of course in a rectangular area.

It also exists a vsync signal which tells if the hardware has finished the drawing of the screen. It is used both for time synchronization (although PCs have other synchronization systems, as hardware timers and the sound card) and as a start signal for updating the framebuffer for the next frame to be drawn. The framebuffer can be accessed and changed while it is being drawn by the graphic hardware but this can produce some graphic flicking. To avoid this flicking it is also used another technique. Usually exists a second buffer of the same (or larger) size than the frame buffer that is called secondary or backbuffer. The updates for the next are performed in this backbuffer and only when all the changes are already performed it is blitted the full (or the part which changed) buffer to the primary buffer. The primary buffer contains the image which will be displayed in the screen. This also has the advantage that the next frame can be freely built by the CPU as the primary buffer is being read for drawing the screen. It can be applied also with more than two buffers implementing a rotating list of buffers which are being displayed. Although using just a secondary buffer is the usual implementation lately it is starting to be implemented a three buffers scheme.

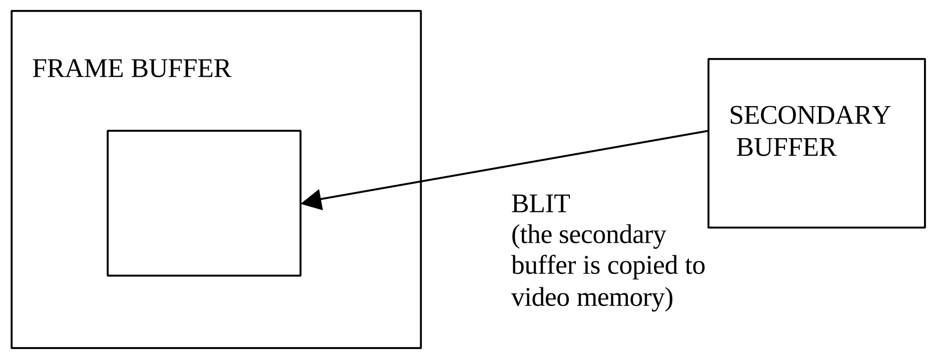


Figure 64. Frambuffer based system. Blit.

5. How to find information.

The information of the graphic hardware used by the videogame consoles can be really hard to find. It is usually specific hardware designed by or for the company which sells the computer, therefore the information about it uses to be protected. It is hard to find official specifications (the programming manual) for most of the videogames consoles. Even harder is for arcade machines because the development of arcade games is performed by the same company which assembles the hardware so the manuals never go out of the original company. For home computers is easy to find more information as many people usually use them for programming games and programs. For videogame consoles it can be found searching for demo groups (a demo is a small program which implement some graphic and sound effects) and the code of this demos and homemade videogames.

Another sources of information are open source emulators of the emulated machine. For example M.A.M.E. (Multi Arcade Machine Emulation) emulates hundreds of arcade machines from the early 80s to the late 90s. It is open source and has the source for dozens of CPUs, graphic and sound hardware. It is a good source for searching how it works a specific hardware.

If the machine we want to emulate has not been still emulated we can try searching for information about similar machines, they will share some of the characteristics. In any case most of the information for those machines without any good information source must be found using reverse engineering of the game code while developing the emulator. It is a hard task to figure how it works and how must be the displayed images just debugging the game code.

For information related with graphic programming it can be found a lot of information around the web and in books. For PCs the same DirectX SDK (software development kit) has a good manual and a lot of example about how it is programmed a graphic application. DOS based Allegro graphic libraries have also a lot of source open projects and tutorials. About tile engines implemented in software can be found information too searching for videogame programming and in old game and graphic programming forums, mail lists and newsgroups.

Chapter 7. Sound Emulation.

1. Sound in computers.

While some kind of graphic hardware can be found in almost any computer it is more rare to found sound hardware. Old PCs only implemented a basic internal speaker that produced simple sounds. Modern PCs now have full featured sound cards though. The reason is that modern PCs have become full multimedia computers (as the marketing department of the computer companies call them) with the capability of playing music or videogames. In any case the start of the sound hardware in computer (without taking into account professional sound oriented computers like sound synthesisers) was in the videogame consoles, home microcomputers and arcade machines.

The sound is very important in a game. The sound generated in a videogame can be divided in two types: sound effects like explosion, shot sounds and voices, and music which is played in the background while playing or in videogames intros like a movie soundtrack. There are many different sound devices which produce different kind of sounds. They show the evolution it had the sound hardware from the early arcade machines to the modern days.

The early sound hardware was designed with the same purpose as the graphic hardware, to reduce the amount of memory and CPU power needed to play sound and music. Later as the CPU gained in power and memory was cheaper the sound hardware changed. In the first days we can find just dedicated circuits which produce fixed sounds (for example the sounds in the Space Invaders arcade). Later we will see hardware which generate tones and FM synthetized sounds. This kind of sound hardware requires nearly no data movement, if compared with any graphic system, to produce a decent music and effects. At last sample based sound hardware with high sample rate was implemented in most of the machines, coupled in the last machines with CD sound track playing capabilities.

In this chapter we will introduce some of the different types of sound hardware which can be found in the machines we would want to emulate. We will point how it can be performed its emulation.

2. Types of sound hardware.

Basically there are two kinds of sound hardware: sample based sound hardware and wave based sound hardware. In the first the sound hardware receives sample data of the sound which must be played. In fact it is the simpler to emulate and the system which can be found in all the modern PCs. The second one is based in the generation by the own sound hardware of sound waves which can be more or less parameterised. In some cases, as the Space Invaders arcade and similar machines, the sound hardware plays fixed hardcoded sounds. More advanced sound hardware has the ability of produce different tones (play a wave with different frequencies) to produce music and different sound effects.

There are a few concepts which has to be explained around sound generation. Sound is produced by the vibration of the air (or any other material). This vibration can be represented as waves with a frequency and amplitude. Usually a sound is formed with the addition of many basic waves (sinusoidal wave). The human ear can detect waves in the frequencies between 40Hz to 24KHz. Below 40Hz it is detected as separated sounds and above 24Hz it isn't even detected (ultrasounds). The musical tones use to be around the 200 to 4000 Hz. The frequency is the number of phase changes in a second and it is detected by the human ear as the wave tone. The amplitude is the volume of the wave and it is measured in decibels (dB), the dB measures a logarithmic scale.

The way a sound is digitalized is to take samples of the amplitude in fixed intervals of time. This fixed intervals of times is called sample frequency. Since a basic wave is a phase change it is needed a minimum of two samples to produce any sound, this means that the higher frequency that could be sampled is the half of the sample frequency. For generating sound using digitalized, samples, data the same can be applied, the higher frequency will be the half of the sample frequency. For example with 44Hz sample frequency (CD quality sound) the higher frequency which can be recorded or generated is 22KHz.

Another important parameter with digitalized sound is range of each sample, the number of bits with which the amplitude of a sample is stored. The more bits, the largest the sample range, the better and more accurate sound it will be. It is better because it can store smaller phase changes. The usual sample bit sizes are 8 bits and 16 bits. The first, 8-bit sample data is used in low quality sound and it can store up to 256 amplitude steps. The second is CD quality and it can store 64K amplitude steps enough for an almost perfect sound reproduction. Both the sample frequency and the sample bit size have a big impact in the amount of memory (or any other form of storage) needed to keep a sample sound. As a simple example CD quality is 44.1 kHz 16 bits for sample (2 bytes) and 2 channels (stereo): 44100*2*2 = 176400 bytes = 172 KB to store a single second.

The amount of storage needed for good quality sampled sounds implied the necessity of implemented others ways of generating sound using less memory. The sound systems based in waves, either hardcoded or stored in buffers in memory, which could be modified in frequency, amplitude and others parameters to produce different sounds were introduced to help with this problem. The first sound systems implemented in either videogame computers or any other computers were just fixed beeps implemented using analogic circuits. There was a small fixed set of sounds which could be produced and the quality of the sounds was quite low. This kind of hardware can be found in the first arcade machines. The game code just had to set the bit corresponding to each of the sound that could be produced in a hardware register and the sound was played.

Later systems that synthetized sound were developed. Those systems work writing the frequency and the volume you want to play the wave to the sound hardware. The amount of memory needed is small, you just need the notes of the music you want to play or the frequencies of the sound you want to generate. The changes in the frequency use to be rare (less than 100 in a second) and therefore full songs can be stored in a few kilobytes. This is basically the system which is used in most 8-bit and 16-bit videoconsoles and arcade machines. There are different implementations of this kind of sound generation.

The basic implementation are the so called PSGs (Program Sound Generators) used in the 8-bit systems. They had a few sound channels each capable of producing sound at a different frequency and volume. They generated basic waves either with sinusoidal form or square form (a square wave can be represented as a list of 1s and 0s). Some also implement noise channels (fixed frequency, usually high, randomly generated amplitude) to be used as shot, explosion and percussion. Other chips had the ability of changing the envelope of the wave. That meant to pass the wave through a filter which could change the amplitude of the generated wave in function of the time. For example it could start at a high amplitude and be reducing the amplitude until the end of the period.

This is the more basic hardware that can be found. The problem with those systems is that the produced waves were too simple. Music instrument waves are based in harmonics, the frequency changes which are added to the base tone frequency of the note played by each instrument. To try to simulate the harmonics Frequency Modulation sound systems was implemented. This is based in the combination of two or more waves to produce a sound. It is not as the addition of waves produced by the mixing of various sound channels. In this case there is a wave which is called carrier that has the tone frequency and a modulator wave which modifies the phase of the carrier. The formula for the YM2413 (used in some models of the Master System and in MSX computer) that uses two waves is as follows:

$$FM = E \sin(w1t + I \sin w2t)$$

Two frequencies and two amplitudes are provided. There are a number of parameters which are used to set the amplitude and frequency of the carrier and modulator waves. A combination of those parameters is usually called an instrument because it is used to simulate the sound of a specific instrument (piano, guitar, ...). This kind of hardware also has extended envelope capabilities, meaning that the full envelope of the wave can be modified (attack and decay time, sustain and retain level) to produce a nearer to the real instrument sound. The number of sound channels is also increased.

FM sound hardware produces music and sound with a medium/high quality. It has some problems though in the generation of percussion sounds and sound of the explosion and shots kind. FM is one of the most used sound hardware in 16-bit videogame consoles and arcade machines.

Another approach to wave sound generation is MIDI sound. MIDI is a standard for producing music using simulated instruments. A MIDI file or song uses a set of instruments which are implemented by the MIDI hardware (they can be implemented in different ways: wave tables, hardware or software simulation). The song contains the notes, times and other parameters for each of the playing instruments. The hardware reads this parameters and generates the music. MIDI is not designed to be used for sound effects. PSG can generate enough good sound effects and FM has some ability with them too. MIDI is more designed for music. The quality of MIDI music, using a good hardware based system or a good software based one and a good instrument table, is really good. MIDI is implemented in PC videogames and in the PSX.

Those systems are the more frequent wave based sound generation systems we will find. In the next section we will introduce the emulation of the PSG and FM based systems. We will not talk about MIDI emulation.

The other general kind of sound hardware is the based in sampled sound. As we said previously this system needs a large storage for containing the full sounds and musics. We can find diverse implementations which differ in the capabilities and the storage medium used. For example modern videogame consoles (and some arcade machines) use CD-ROM readers. A CD-ROM can contain large amounts of data (650MBs) and it can store up to 74 minutes of high quality stereo sound. Either stored in sound tracks or in data tracks as files which are feed into a sample based sound generator voices, sound effects and full songs can be stored along with the game data. Most of the games store voices and music as sound tracks in the CD-ROM and sound effects are stored as files in the data track. The sound tracks are played directly from the CD using hardware between the CD-ROM reader and the sound hardware, the CPU just has to program the CD-ROM reader to play them. The sound data stored as files in the data tracks must be read into the main memory and sent using DMA or other mechanism to the sound hardware to be played.

Not only stored in CD-ROMs but in other formats sampled sounds and musics have been used in videogame computers. The Sega Genesis has a DAC channel (Digital to Analog Converter) which can output 8-bit 4KHz samples. This low quality sample based sound generator can be used to produce some sound and voices hard to produce with the FM hardware used for music and sound effects. The Sega Mega CD (which as add-on to the Sega Genesis and work as two computers working in parallel) has 8 PCM (Pulse Code Modulation) based channels and a link between the CD-ROM reader and the sound mixer to play sound tracks. The PCM channels can hold 16-bit 38KHz samples which is quite a good quality sound. There is a special memory region in the Mega CD 68K side which stores the samples for being played. Arcade machine store the sample data in ROMs cards attached to the sound hardware for example for the QSOUND ADPCM sound hardware of the CPS1 and CPS2 (Capcom Play System 1 and 2) based arcade games.

The sample based sound systems can also apply different effects and filters to the original sampled data. Those sound chips, sometimes called DSP (Digital Signal Processors) can perform mixing operations, fade-in and fade-out and many other filter effects. They can also be programmed to produce more effects. Using a sample based generation system any kind of sound hardware can be emulated in software (this will how we will be emulating our emulated machine sound hardware). The opposite is not true, most of the other sound systems are unable to emulate properly other sound systems or sample based sound systems. For example is common to use waves stored as samples to implement systems which work similar to the MIDI and FM system where a base instrument wave sample is modified in tone and amplitude to be used as a virtual music instrument. The Nintendo SNES use this kind of system and some of the games use special and more powerful sound chips inside the ROM cartridge.

In the next sections we will introduce a bit how it work a sample based system because we will find in some of our emulated machines and because it will be our main tool for emulate all the other systems.

3. Wave generator based sound hardware.

We will just introduce how a basic PSG. The Master System SN7649, works and can be emulated and provide a few comments about FM emulation, introducing the Master System YM2413 and the Genesis YM2612.

Basically there are two ways to emulate this type of sound hardware: use a sample based sound system and generate waves as near as possible to the produced by the original hardware or use a similar hardware, using the correct settings, to try to produce a similar effect. The first is the main option in most emulators being implemented today. The second was used when MS-DOS emulators were still the main option. The Sound Blaster 16 (basic PC sound hardware) and the Adlib (an even older PC sound standard) had similar hardware to those FM and PSG hardware. In fact the Adlib is more or less a PSG while the SB16 (and previous 8-bit version of this sound cards) uses a FM chip. This hardware is not supported the Win32 or DirectX APIs and as the modern PCs has enough storage space (memory and secondary storage) for storing large amount of sample sounds sample based sound generation is now the standard. Either from CD-ROM sound tracks or any other forms of sampled data, including largely compressed sampled based formats as the MP3 (Mpeg 1 layer 3).

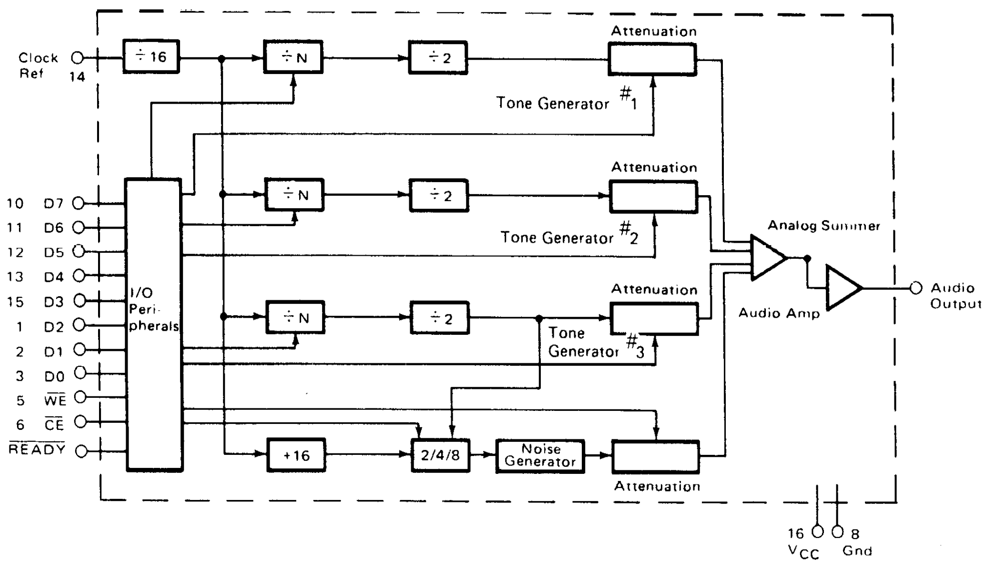
PSG emulation.

A PSG is just a sound chip which has a number of channels in which each of them can be programmed to sound in a given frequency and with a given volume. Some PSGs has other features as envelopes too. The wave which is produced can have any form, there are PSGs which generate sinusoidal waves, square waves and even triangular waves. The Master System PSG is a SN76489 running at 3.57 MHz. It has four channels, three are tone channels and the last is a noise channel. This is more or less the standard configuration. Different sound chips will have more or less channels. Usually machines which need a better sound use more than one of this chips to provide more sound channels. The SN76489 is the more basic that can be found in videogame consoles. It produces square waves and only can be configured the frequency, which can go from 122 Hz to 125 kHz (it uses a 1024 divider with a 125 kHz base frequency), and the volume which has 16 steps from no sound to loud volume. The fourth channel is a special channel with a feedback xor array to produce both periodic and white noise.

A square wave can be just implemented, and it is, using a decrement (or increment) register which changes the output of the channel from 0 to 1 or 1 to 0 each time it arrives to 0. The content of the decrement register for each channel is the value corresponding with frequency of the wave to be produced. In fact it is a divider to the base frequency (the chip frequency divided by 32). The output signal of each channel is attenuated with the given volume value and then the four channels are mixed. The noise channel can use three fixed frequencies (clock/16 = 7800Hz, clock/32 = 3900Hz and clock/64 = 1940Hz) or the frequency of the third channel (channel 2).

The way to emulate it is to generate samples for each of the channels, mix them and write them to the playback buffer that the sample based sound hardware will play. The calculated samples for each channels will be just 0/1 values (or –1 and 1 values if we use signed samples) that are multiplied by the volume. The volume value is an attenuation factor, that is, the high value, 15, means no sound and the lower, 0, the louder sound. Then the output sample for a channel is therefore calculated as MAX_VOLUME*(15 – channelVolume[i])*channelOutput[i]. For updating the output of a channel we will have a counter with the number of cycles since the last output change. This counter is decremented (or incremented) by the cycle step value passed each time the PSG emulator is called. The original value stored is the frequency divisor value that tells the frequency of the channel. When it arrives to 0 or below to 0 the output of the channel is changed and the value in the counter is restored, but keeping the overflow cycles (the negative cycles) to provide an accurate sound. If it is just wrapped to 0 (increment) or to the frequency value (decrement) there will be noise in the final output.

BLOCK DIAGRAM



BLOCK DIAGRAM DESCRIPTION

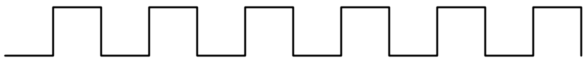
This device consists of three programmable tone generators, a programmable noise generator, a clock scaler, individual generator attenuators and an audio summer output buffer. The SN76489AN has a parallel 8 bit interface through which the microprocessor transfers the data which controls the audio output.

Figure 65. SN76489 Schematics.

This task is performed for the four channels. Then the samples of each channel must be mixed. If we are using signed samples the mix is just the addition of the four sample values. This addition must be performed with saturation, it can't produce and overflow, if the final value is greater than the maximum signed value for the sample size in bits the result must be capped to this maximum value. If not the sign of the wave in this step would change producing a totally different sound. It is good to prevent saturation from a first step just adjusting the different volume values (the range of the samples) for each channel so the addition will never overflow the sample bit size. If the samples are not signed then the process is a bit harder because it must be performed taking into account a center value and the saturation as well. A good solution could be to use signed sampled always and in a last step if the target hardware uses unsigned samples convert them to unsigned.

The noise channel is generated using the same algorithm but at the time of the output change it is used a xor value to produce random wave changes. The xor values are different for white and periodic noise. The white noise uses the same output from the channel as input to produce more random sound.

The three tone channels are used for music and for most sound effects. The noise channel is used for percussion effects (very bad quality) and for shots and explosions. The music and sound generated are very plain and of course there is no sense of different music instrument. There is also a great difference between a sinusoidal wave and a square wave. A square wave sounds sharper and it is easy to detect it, it is a very frequent sound in old videogame consoles and arcade machines.



Square based waves.

```
sample = 0;
/*  Generate sample data for the three standard channels  */
for(channel = 0; channel < 3; channel++)
{
   if (!channelEnabled[channel])
      continue;

   if (PSGchannelFreq[channel] != 0)
   {
      PSGchannelCount[channel] += cycleStep;
      if (PSGchannelCount[channel] >= PSGchannelFreq[channel])
      {
         changes = PSGchannelCount[channel] / PSGchannelFreq[channel];
         PSGchannelCount[channel] -= PSGchannelFreq[channel] * changes;
         if ((changes & 0x01) == 1)
            PSGchannelStat[channel] = ~PSGchannelStat[channel];
      }
   }
cSample = PSGchannelStat[channel];
cSample *= (INT16) ((15 - PSGchannelVol[channel]) << 8);
sample += cSample;
}

/*  Noise channel.   */
if ((PSGchannelVol[3] != 15) && channelEnabled[3])
{
   if (PSGchannelFreq[3] != 0)
   PSGchannelCount[3] += cycleStep;
   while ((PSGchannelCount[3] >= PSGchannelFreq[3])
            && (PSGchannelFreq[3] != 0))
   {
      PSGchannelCount[3] -= PSGchannelFreq[3];
      if (PSGnoiseRand & 0x01)
      {
         PSGchannelStat[3] = ~PSGchannelStat[3];
         PSGnoiseRand ^= PSGnoiseChannelFeedBack;
      }
      PSGnoiseRand >>= 1;
   }
 cSample = PSGchannelStat[3];
 cSample *= (INT16) ((15 - PSGchannelVol[3]) << 8);
 sample += cSample;
```

Figure 66. PSG sound generation algorithm (Normal channels and noise channel).

For produce sinusoidal and triangular waves the best is to use a table with precalculated base sample values which will be later modified with the volume value. The table will be indexed by the content of the counter register of each channel arranged to the range of the table.

The additional effects which can be introduced as different wave envelops can also be applied. A time counter for the envelope of each channel is used, when it reaches some values the volume will be modified with the form of the envelope. This kind of PSG is for example implemented in the original GameBoy. Other features can be found, for example stereo: the original Master System PSG has been modified in the GameGear (that it is a portable and a bit upgraded version of the Master System) so each channel can be directed to the left, right or both channels.

The samples are stored in a buffer and the sound hardware or the sound API will read and play them. About this part we will talk in the next section.

FM emulation.

FM sound generation is harder to emulate. As we already said it is based in the combination of multiple waves to produce harmonics similar to the produced by real music instruments. It is intended therefore for music generation. Without the using of harmonic it can be seen as a sinusoidal sound generator as the one seen above. The have more channels than more primitive PSGs and a lot of more parameters to be configured: base frequency, relative frequency of the modulator wave, amplitudes and complex envelopes. Some, as the Master System YM2413 have the settings for the instruments hardcoded in an internal ROM. Others, as the SB16 FM chip and the Genesis YM2612 use hardware registers to set the instrument values and the game is which has to know this values and set them. Therefore we can see there are two groups of settings: a group of settings which are related to the instrument to be emulated (relative frequency of the harmonic, number of waves – operators – combined, the envelope of the instrument) and a group of settings related with how the instrument will play (note, volume and duration).

Mathematically the combination of the two waves using FM, or frequency modulation can be described this way:

$$F = A \sin (wc\ t + I \sin (wm\ t))$$

F is the output sample for a channel, A is the amplitude of the carrier wave, wc is the angular frequency (expressed in radians) of the carrier wave, I is the amplitude of the modulator wave and wm the angular frequency of the modulator wave. In this case it is said that the final wave is produced combining two operators. An operator is a Asin(wt) group and can receive as phase input the output of another operator. The YM2413 can just combine two operators (carrier and modulator) or just use one but the YM2612 can combine up to four operators to form an instrument. The carrier wave produces the base wave with the frequency, the tone, of the sound or instrument we want to play. The modulator wave produces the harmonics that produce unique instruments. For building an instrument it is also configured the envelope of the instrument.

There are two kind of envelopes, or the YM2413 has two types. In any case they are the most common. A decaying envelope that when a note is played the instrument (virtual) does not keeps the amplitude but decays until it arrives to 0 and then mutes. Or sustained envelope when the note (also called key) is kept at a certain level until another key is played for this instrument or it is silenced. The parameter of the decaying envelope are attack rate (AR), decay rate (DR), Release Rate and Sustain Level. The decaying envelope increases at the attack rate until the maximum amplitude or volume, then decreases until the sustain level using the decay rate. Until this point the decay and sustain envelopes are the same. Then the decay envelope continues decreasing, using the release rate parameter, until it arrives to 0 or to the sustain point. In the sustained envelope the volume is maintained in the sustain level until the key off when it begins to decreases using release rate parameter. It can be seen in the envelope figure.

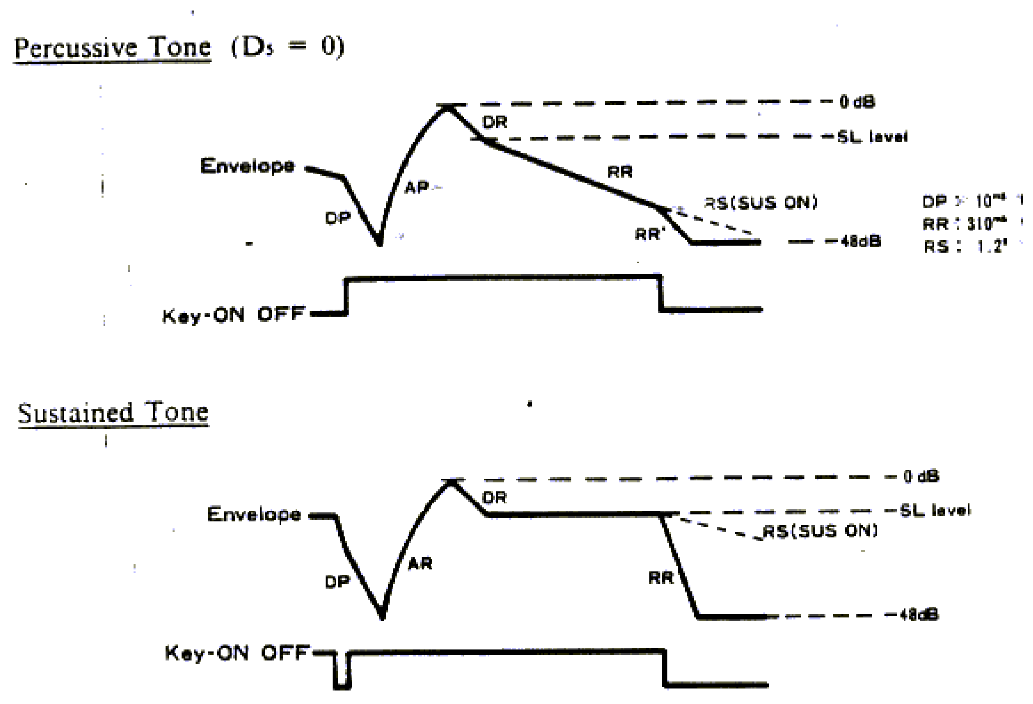


Figure 67. Envelopes for FM sound generation.

The emulation of the YM2413 can be viewed easily in three phases as it is described in the datasheet [29]. There are three blocks: the Envelope Generator (EG), the Phase Generator (PG) and the Operator (OP). The EG using the clock as input, the amplitude (volume) and envelope settings produce the output amplitude for the generated wave in a given point of the time. The PG generates the phase of the wave to generate using the frequency registers and the clock. The OP receives both the EG and PG outputs and additional settings from the register bank and produces the output wave. This is just a digital sample that it is converted to an analog signal with a DAC.

The EG must be emulated using counters which are updated with the cycle step and provide the appropriate amplitude parameter for the part of the envelope that it is being played. The PG is just a counter updated with the cycle step and adjusted with multipliers and dividers to provide an index into a table with a precalculated sinusoidal. The OP just must read the sinusoidal table and multiply the result with the value from the EG. This is performed for each operator. Two operators form a channel and in the YM2413 there are up to 15 channels.

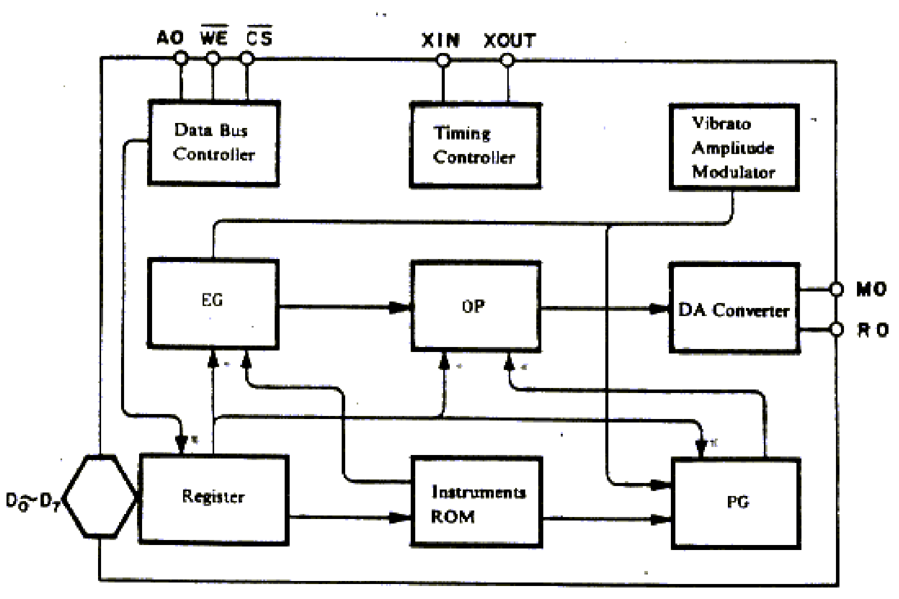


Figure 68. YM2413 Schematics (Envelope Generator, Operator and Phase Generator).

There are some variations for this basic algorithm, for example percussion instruments are hard to emulate using this kind of waves. They are emulated using only one operator and special settings as a feedback input (like the PSG noise channel) from the same operator. Other effects can also be implemented as a vibrato for the final output wave or different kinds of filters.

After each channel is generated the result are mixed as we said in the PSG section.

This is basically as it works the FM sound synthesis. It is harder and slower to emulate because of the large number of parameters and the larger number of channels. The fact that for each channel the same operations are performed twice, for the carrier and modulator wave and the additional overhead of the other sound effects applied at the end of the process. Another problem is to find the appropriate values and settings for the generation, how many time represents each release rate value and so, because it can affect in the final sound. In some cases it will also needed to find the appropriate values for the instruments if, as the YM2413, they are stored in an internal (non-readable) ROM. However this is not very common and in many other FM chips the instrument are provided by the program.

4. Sample based sound generation.

Sample based sound emulation is basically the easiest to implement. It must just to take into account a few things as the format of the sample values, the number of channels that must be mixed and the envelope and filters applied to the channels. The sound hardware and sound API that we will use for emulating other systems will be a sample based system. Therefore it just a question of correctly converting the sample data from the emulated system format to the target system format and then mixing the different channels correctly.

There are many systems which implement sample based sound hardware. The Genesis has a very low quality DAC channel as we already said. The Mega CD (or Sega CD) has 8 PCM channels with a good sound quality (38KHz). The samples are stored in a special area of the Mega CD memory which is organized in banks. For each channel it is selected the bank that will be played. The Sega CD also has a CD audio source which is mixed with the other sound fonts. Other systems also have similar systems, the CPS2 uses QSound ADPCM system with the samples stored in separated sound ROMs and the sound chip being controlled by a Z80 (the main CPU is a 68K).

The case of the emulation of a CD sound track is rather easy if the emulator works with original CD-ROMs. The target system CD-ROM and audio card just must be told to play the same sound track than in the emulated machine. CD sound tracks are a standard and work in the same way in all the systems. The emulation is just the conversion from the emulated machine command to the target machine command to play CD tracks. If rather than the original CD-ROM we are using a CD-ROM image (a copy in the hard disk of the full CD-ROM) we should emulate it through the sample based services of the target sound hardware. It will be just a matter of reading the raw data from the image and copy it (and convert it may be) to the playback buffer.

There are different formats for storing the samples as we already said. The samples can be in raw format, can be compressed or not, can be signet or not, use 8-bit samples or 16-bit samples, be in different sample frequencies and so. It is important to know what are sample characteristics of the emulated sound hardware and the sample capacities offered by the target sound hardware. For example the PC standard sound hardware can play from 8KHz to 44KHz samples, either 8-bit or 16-bit and stereo (two channels) or mono sound. The sample bit size, the sample frequency and if it is a mono or a stereo sample determines the size of the sound buffers needed. Stereo samples are usually stored coupled, two samples together one from each channel (right or left) which are played together.

There are different ways of compressing sound data (similarly what happens with graphic data), some are loosely forms of compression and other are not. For example MP3 compression format can decrease in an order of magnitude the size of the sound data but it is loosely and needs a lot of CPU power to decompress and compress the data. Other formats as ADPCM (adaptative PCM) are not loosely but reduce the amount of memory needed. ADPCM is based in store the changes between two samples rather than the samples themselves. This way the sample size can be reduced (or the quality increased). For example two 16-bit samples with a difference between them less than 127 can be stored as the first sample as a 16-bit value and the second as an 8-bit signed value which is the difference between the two original samples.

The way the samples are played is really easy. The sound sample data is stored in RAM or other form of memory (ROM could work). The sound chip is attached to the bus and can read the memory. When we want to play that sound the sound hardware is tell to play it from the given address (or it can be fixed if the sound hardware is attached to fixed memory addresses). The sound hardware begins to read data from this position and playing the sound. When the buffer is exhausted it raises a signal (an interrupt usually) to the CPU demanding for more data. The sound hardware can then stop playing data or automatically loop back to the start of the buffer. This later feature could either work for repetitive sounds or for reusing the same buffer. The program code could, with the proper synchronization, be filling the sound buffer from the start while the sound hardware is already playing ahead of it. When the sound hardware arrives to the end of the buffer the start will be already filled with proper data by the program code. The sound sample system is basically implemented in all the systems using a callback.

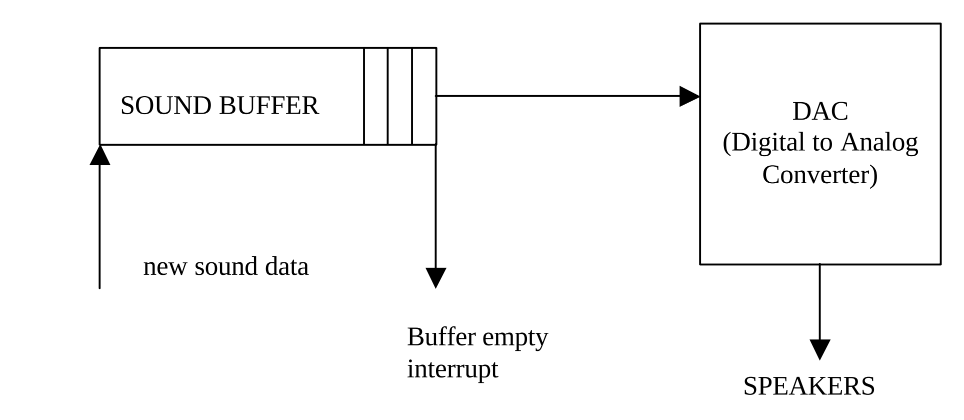


Figure 69. Sample based sound generation.

This is basically as it works such a system. Different sound APIs and sound hardware can have additional features as different output channels and hardware (or automatic) mixing, filter and volume controls, but with the basic features almost any sound system can be emulated. We could talk about different filters effects and other features that can be found in particular systems, but as it is easy to find information about sound generation (even easier than information about graphic programming) we will end this chapter here.

5. How to find information.

The information about how to produce sound and the way the sound hardware of the emulated machine works can be found more or less in the same places as the information about the graphics. Datasheets of the original hardware (if they are freely available), source code from demos and source code emulators of the machine. Some reverse engineering could work but only if you have a good knowledge about the kind of hardware you are emulating. It is hard to figure how something must sound without the original system for example.

Information about sound format, sound filter, sound effects and such topics is easily found searching the web for sound programming or similar words. The programming APIs use to have a lot of examples and explanations as well (DirectSound). And the basic sample system is really easy to use.

Chapter 8. Information, testing and legal.

1. The information. Research. Reverse Engineering.

One of the more important tasks around the process of emulating a computer is to gather as much information about how this machine works as possible. This is not an easy task. Sometimes it does not exist an official or complete documentation. Other times this documentation is private and it is kept in secret by the company that designed the machine. This last case is one of the more common as the videogame consoles and arcade machines are closed systems with heavy protection rules for the game developers who get official development kits. Sometimes just the company that designed the machine builds program for this machine. Although the task is hard there are a few spots to start with the task.

The more popular and the older a machine is the easier it will be to find information. An old systems means that a lot of people will have work with it already, investigating all the hidden features, maybe the protection rules will be no more in use or the company will not care about them. The official documentation can have been 'stolen' and become free. The same happens with popular systems but in a smaller amount of time. A popular system is one that many people are working with it, programming their games and programs and investigating how it works. It is easy that information or the full official documentation could go out to the public. One of the better sources of information for new systems with lack of information has always been the demo groups. A demo is just a small program with graphics and sound that tries to use the features of the used machine to produce an impact in the user or to push to the limit the capabilities of the machine. Before the people who tries to build their our home made games for a platform the demo coders will work with it.

The last resource is to try to use reverse engineering to discover how the emulated machine hardware works. This can be a hard task sometimes and requires a lot of experience and knowledge with similar hardware to the emulated one. The program code from the emulated code is scanned to try to discover the commands sent to the hardware. Then it must try to discover what those commands do. It is easy if you have access to the original machine and to tools to write programs for this machine. Getting the information from the original code and using it with the original hardware is a good way to find a lot of information. The reverse engineering work uses to be done as well with the own emulator integrating a debugger and disassembler inside it. The errors of the emulator and difference between the emulator output and the real machine output are followed until the correct settings are found. It a very expensive task that needs a lot of time and patience. Reverse engineering is the only legal way of emulating a machine which has protection rights of all its documentation and hardware.

2. Testing and debugging.

Other important task in the process of emulation is the testing and debugging of the emulator. This task is as or even harder than gather the information. In some cases, when reverse engineering has to be applied, the two tasks are performed at the same time. The different part of the emulator has to be tested and debugged, but each part needs special efforts in this process. It is not the same the testing of the CPU with thousands of different inputs (the instructions) than the graphic or sound hardware. There must be integration test and separate test. Usually just the CPU is tested alone (using small programs in the emulated assembly language) and the other parts are being developed, tested and debugged as part of the full emulator.

The CPU is perhaps where more errors can be hidden because of all the code needed to emulate all the CPU instructions. It is also faster to find easy errors (because the game or program just does not work) and harder for strange errors, because of some instructions or some inputs rarely being used. The advantage of the CPU is that it can be built a test system to perform systematic tests for all the CPU instructions. There are two basics ways to perform such systematic tests. One could generate with a tool small programs for testing a few instructions with random parameters. Then compare the results of the CPU core with the results of a real CPU that we will have access (in another computer for example). This is the best way because you are testing your core with the real CPU. The problem is that some of the CPUs used are very rare to be found in computers we could use easily and create a dedicated board for the testing is expensive. A cheaper and faster alternative is to test our new CPU core against an already implemented and trusted CPU core. We will know that the other core will have none or very few errors but it is not still the real CPU and sometimes we will end finding difference due to the different way the two cores can be implemented.

Other hardware can be more or less difficult to test. Basically all the other testing, and also most of the CPU core testing if not such tool is used, is performed debugging programs or games that does not work in the emulator. This is a very time consuming task and should be avoided trying to create better test techniques. For example it can be created or used small test and demo programs from the original machine. This can be already implemented or be created by the own emulator author. They could be applied in both the original machine and the emulator to compare the results. This approach would be easier because different from a common program or game we know what the test/demo is doing and we can test specific parts of the emulator. In any case all the other test, debug and reverse engineering process is performed through a classical debugger integrated with the emulator.

Graphics and sound are easy to find the big errors because you can sense them easily. If you know or have at hand the original machine with the same programs or games you can compare colors, the displayed image, the music and the sound. This makes easy to find that exists a problem but it is still hard to find the cause of this error. An accurate model of both emulated graphic and sound system is almost impossible unless we know the precise characteristics of the hardware. This is because it is hard without them (perhaps using an oscilloscope or image comparers) to notice small difference in sound and graphics. Other 'errors' are more subtle like the sense of the program or game speed. This is something it can be feel and that it can be related with a non very accurate emulate of the timing but it is hard to solve it.

The full reverse engineering, test and debug topic could take hundred of pages to be properly explained. The experience, a lot of knowledge, practice and being systematic are the better tools against the problems we will find. In fact none or only a few emulators are really perfects. It is hard to make work all the game or programs a machine can use because it is impossible to test them. And as some ROM cartridges in some system add additional hardware (even less documented) to the original emulated machine hardware the task becomes almost impossible (Genesis Virtua Racing or a few SNES games with special chips). It could be said, as it can be said for any software product, that the process of debugging never ends. Only systems with a limited number of features, a lot of information and a limited number of programs which can be all fully tested can be considered as fully emulated.

3. Legal and Commercial aspects of emulation

One important aspect that is related with emulators is if they are legal or not. Something is sure the creation of software duplicates of the games and programs which were stored in a different format (ROM cartridges, tapes, whatever) can only be considered legal if you own the original storage system. If not is just another part of the warez and software piracy problem this time related with old and many times no more sold systems. As the companies would not care about what happens with this systems the emulation and duplication of those systems and programs is a grey area but it is not dangerous.

However as emulation has been more and more developed systems which are still being sold (PlayStation, Nintendo 64, GameBoy Color) or even systems which have not been even commercialized yet (GameBoy Advance) have started to be emulated. This arises a new problem because the companies that produce the original hardware will be very beligerant with the use of illegal copies and with the own emulators which make the original hardware unnecessary. Sometimes the emulator can add features that enhance the original hardware (Bleem and BleemCast).

This has been even worst when the first commercial videoconsole emulators have started to push the market. Connectix Video Game Station (VGS) and Bleem/BleemCast are two PlayStation emulators for the Mac and the PC the first, and for the PC and the DreamCast the last. They have been sued by Sony (the owner of the PlayStation) many time but until the time Sony has lost all their legal attempt to stop them. The last news includes that Connectix has been bought by Sony and a commercial and legal war between Bleem and Sony about the newly released BleemCast.

As laws are involved here the same principles than for any reverse engineering case are applied. If the emulator is implemented without using any hardware or software part copyrighted or without using private information but rather public information and reverse engineered information it is fully legal.

What happens then and how it is used and abused by the users are another question. In any case software or hardware emulation, out and inside the world of the videogame computers has been used by ages. The Sega Genesis has hardware compatibility with the Master System. The SNES has a hardware device to play GameBoy games. The GameBoy Color and Advance keep backward compatibility with the original Gameboy. The CPUs, PC x86 CPUs, are being emulated as an attempt to break Intel monopoly both in hardware (Cyrix, AMD) and in software (Crusoe, Virtual PC). There are PC emulators for MAC and MAC emulators for PCs. The market and number of applications for emulation is still growing.

Chapter 9. Conclusion.

1. Initial objectives.

The project, as stated in the preliminar report, had two important parts: a document about how can be implemented a videogame computer emulator and the implementation of some one of such emulators using some of the techniques described.

At first it was intended a Sega Mega Drive (Genesis) emulator, but the complexity of the emulation of the M68000 process, the Mega Drive VDP (graphic hardware) and the sound hardware which uses a YM2612 FM synthetizer and a Z80 coprocessor made a first change in the objetives of the project. A more simple machine, the Sega Master System, which was the predecesor of the Mega Drive, was designed as main target. At the same time it was developed a Space Invaders as a main example of a basic tutorial about CPUemulation.

The documentation was intended to be written in english, and would talk about the more important techniques about CPU, graphic and sound emulation. Also about the general structure of the emulator and how the different parts work together.

The process of gathering enough information and learning about emulator programming was started very soon using free resources that could be found in the Web. Different lists about emulation programming were also used to contact with the people who was working in this topics. Most of the knowledge about programming emulators for videogame consoles or microcomputers can not be found in any document but has to be asked to the people who has been working with them.

2. Objectives acomplished.

The process of learning the basics was acomplished in a three or four months before the inscription of the project. In later phases a more deeply knowledge about some of the topics about emulator programming were obtained, mainly around graphic and sound emulation and dynamic binary translation. There were three differents sources of information. The first was general documents which could be found in the 'emuscene' (the small subset of the Web which is dedicated to programming and using videoconsole and old computers eulators). Those documents were written by different emulator authors (Marat Fayzullin, Dan Boris and others) and provided a vision of how the problem of emulation was being solved in the 'emuscene'. This first source includes also different open source emulator projects like M.A.M.E. (Multi Arcade Machine Emulator), which is a full library which provides components and descriptions to emulate thousands of old video arcade machines. Other open source emulators were also used as information sources. The second source were the people in the 'emuscene'. The mail list about emulation programming like MUL8 and Dynarec were very useful to contact people interested in the topic and learn a lot from them. Also programming forums like S8Dev forum about the Master System programming and emulator implementation. The first two sources provided a general vision about interpreter emulators and sound and graphic hardware emulation. The last source of information were commercial and academical research around emulation. The main aspects about which those sources talk are binary translation and interpreter design.

Using this sources the documentation was started. Althought the first draft were started and the initial steps the document about emulator programming was not finished to the actual form until the end of the project. The result is a document which provides a good framework to start learning about emulation programming. The main topics are mostly covered. The more extended part is about CPU emulation because in general emulators is the more important part. This is not enterely true for video console emulatos, which are our primary targets, because in those systems the graphic and sound system are as or even more important than the CPU. In any case graphic and sound emulation were also so covered in the main aspects. At last the document is missing additional information about the process of reverse engineering and debuging of the emulator. There is also missing a bit more of information about the implementation of other small devices, as timers, comunication chips and device controllers which can be also present in many machines. A bigger overview of the interconnection of the different parts, CPU, graphic, sound and miscelaneous devices, could be also interesting. The document was fully written from the start in english, as it was said in the first report. This aspect could be considered a mistake because the quality of the text has suffered a lot because of the lack of knowledge of the writer. In any case it was too late to change de language of the document and although harder to end it was funner becuase of it, because it was a good way to try to learn more about the language.

The Space Invaders tutorial was writting in a couple of month in spare time with the help of the members of the MUL8 mailing list, where it was intended to be distributed the tutorial. The tutorial is missing the last part about the Space Invaders hardware which is very simple (just a couple of sound circuits for providing simple sound, emulated using presampled waves, a simple monochrome framebuffer and two time interrupts used for synchornization). The emulator was implemented in Visual C for Win32 and using the DirectDraw API (inside the DirectX API version 7.0). Both the CPU and the hardware part were fully implemented by the author.

The Master System emulator was intended to test the techniques about graphic and sound programming. Because this and for providing a better performance an already implemented CPU core was used. This CPU core is MZ80's Neil Bradleys Z80 emulator programmed in both C and x86 ASM (using a code emitter). This core is very fast and reliable. It was missing, though, an important feature for Master System emulation, it didn't provide support for banked memory. It could be still implemented using slower techniques (as memory memcopies) but it was thought that to add bankswitching features to it would pay the effort. So it was done improving a lot the performance of the emulation. This was achieved in a couple of weeks of debugging (becuase of the difficulty of testing an assembly CPU core). To the Master System emulator it were added most of the graphic features and sound features implemented by the original hardware. The compatibility, the number of original games that work in the emulator, was being increased all the time, though still there are about a 30% of games with problems and a 10% which does not work at all. It was also implemented support the GameGear a handheld version of the Master System with additional features (smaller screen, more palette colours). The first version was finished in a month, later changes, debugging and implementation of other features were developed in a range of two or three months.

The Master System emulator was developed using Visual C and a graphic and sound library (multimedia library) named SDL. SDL is x86 multiplataform library (Unix, Beos and Windows systems) which provides an easier interface to the graphic and sound hardware. This was the reason, not the portability, which made to choose this library rather than using other interfaces. The implementation used was for Win32 which used the DirectX (v 5.0) for accessing to the graphic and sound hardware.

The first phases of the project, research, were started at the Spring 2000 course but the project was not really fully started until the Autumn 2000 and continued until its end this Spring 2001 course. The project was developed in half time day (because the author was working). The Space Invaders and Master System emulator were almost finished at the start of the Spring 2001 course. The remaining time period was used for finishing this document.

3. Further work.

Some objetives that could be interesting to achieve could have been to emulate the YM2413 chip used in some versions of the Master System. This chip is a FM synthesis hardware which is harder to implement that the basic PSG hardware of the Master System. The lack of time made impossible to end the implementation of the emulation of this device, which was stopped at the research and design phase because the document should be finished first.

It would be also interesting to implement a more complex system, like the Sega Mega Drive or the Sega Mega CD, but that would be out of the scope of the size of an universitary project.

One aspect of the emulation which has remained out of the project, although it has been one of the more developed in the research time and has been a source of fun in the free times, has been dynamic binary translation. The Dynarec mailing list has been used to continue the works around this topic although this part is not show in this project. The work in the reserch project Dixie in the UPC around static retargetable binary translation has been also interesting for this part of the emulation range of knowledges.

References

[1] New Jersey Machine Code (NJMC) toolkit. Norman Ramsey.
http://www.eecs.harvard.edu/~nr/toolkit

[2] UQBT a resourceable and retargetable binary translator. Cristina Cifuentes. Mike van Emmerik.
http://www.csee.uq.edu.au/~csmweb/uqbt.html

[3] Neil Bradley's MZ80 Z80 emulator. ftp://synthcom.com/pub/emulators/cpu/

[4] Threaded code, James R. Bell, ACM 1973.

[5] Ardi's Executor (Mac emulator for PC). http://www.ardi.com/executor/index.html

[6] Threaded code, http://www.complang.tuwien.ac.at/forth/threaded-code.html

[7] Some Efficient Architecture Simulation Techniques, Robert Bedichek.

[8] Shade: A Fast Instruction-Set Simulator for Execution Profiling. Bob Cmelik, David Keppel. ACM SIGMETRICS 1994.

[9] Pipelined Java Virtual Machine Interpreters. Jan Hoogerbrugge, Lex Augusteijin.

[10] Optimizing direct threaded code by selective inlining. Ian Piumarta, Fabio Riccardi. ACM Sigplan 98.

[11] Linear scan register allocation. Massimiliano Poletto. ACM 99.

[12] DIGITAL FX!32: Combining Emulation and Binary Translation. Raymond J. Hookway. Mark A. Herdeg. Digital Technical Journal, Vol 9. No. 1 1997.

[13] Embra: Fast and Flexible Machine Simulation. Emmett Witchel, Mendel Rosenblum. Sigmetrics 96 ACM.

[14] The technology behind Crusoe processors. Alexander Klaiber. 2000 Transmeta Company.
http://www.transmeta.com/

[15] DAISY: Dynamic Compilation for 100% Architectural Compatibility. Kemal Ebcioglu, Erik R. Altman. IBM Research Division. 1996.
DAISY Dynamic Binary Translation Software. Erik R.Altman, Kemal Ebcioglu. 2000.
http://www.research.ibm.com/daisy/

[16] Dynamo: A Transparent Dynamic Optimization System. Vasanth Bala, Evelyn Duesterwald, Sanjaeev Banerjia. PLDI 2000, ACM. www.hpl.hp.com/cambridge/projects/Dynamo

[17] PA-RISC to IA-64: Transparent Execution, No Recompilation. Cindy Zheng, Carol Thompson. IEEE 2000.

[18] The DR emulator. http://developer.apple.com/technotes/pt/pt_39.html

[19] Timing Insensitive Binary to Binary Translation of Real Time Systems. Bryce Cogswell, Zary Segall.

[20] Compilers: Principles, Techniques and Tools. Aho, Sethi and Ullman

[21] Dixie: A retargetable binary translator. Manel Fernandez, Roger Espasa.
http://research.ac.upc.es/dixie/

[22] Binary Translation. Richard L. Sites, Anton Chernoff, Matthew B.Kirk, Maurice P. Marks, and Scott G. Robinson. ACM 93, Vol. 36 No. 2

[23] Migrating a CISC Computer Family onto RISC via Object Code Translation. Kristy Andrews, Duane Sand. Tandem Computers Incorporated. ACM 92.

[24] Dynarec Site. Neil Bradley, Mike Konig, Neil Griffiths, Julian Brown, David Sharp, Victor Moya. http://www.dynarec.com

[25] “Emu-mech”. An old discussion about emulators implementation in Amiga computer. 1991. http://pds1.nchu.edu.tw/pub/x2ftp/console/sega/docs/emu-mech.txt

[26] Dynamically Recompiling ARM Emulator (Armphetamine). Julian Brown. http://www.dynarec.com/~jules

[27] TARMAC: A Dynamically Rrecompiling ARM Emulator. David Sharp. http://www.dcs.warwick.ac.uk/~csuix/project/

[28] StarDust. SNES emulator for Saturn. http://stardust.emuhq.com/

[29] YM2413 Datasheet. Yamaha Corporation. 96.

Bibliography

Binary Translation

Final Year Universitary Projects

"Dynamically Recompiling ARM Emulator", Julian Brown (May 1, 2000)
http://www.dynarec.com/~jules

"Generator: A Sega Genesis Emulator", James Ponder (1997-1998) **http://www.squish.net/generator/**

"TARMAC: A dynamically recompiling ARM emulator", David Sharp, 2001
http://www.dcs.warwick.ac.uk/~csuix/project/

Thesis

"A Robust Foundation Binary Translation of x86 Code", Liang Chuan Hsu, University of Illinois

Investigation Projects

"DAISY: Dynamic Compilation for 100% Architectural Compatibility", Kemal Ebcioglu, Erik R. Altman, IBM Research Division Yorktown Center [Computer Science RC 20538 08/05/96]

"DAISY Dynamic Binary Translation Software", Erik R. Altman, Kemal Ebcioglu, IBM T. J. Watson Research Center, 2000 **http://www.research.ibm.com/daisy**

"UQBT: A Resourceable and Retargetable Binary Translator", Cristina Cifuentes, Mike Van Emmaik (Queensland University), Norman Ramsey (Hardvard University), June 1999
http://archive.csee.uq.edu.au/~csmweb/uqbt.html

"UQBT: Adaptable Binary Translation at Low Cost", Cristina Cifuentes, Mike Van Emmerik, PACT'99

"Machine-Adaptable Dynamic Binary Translation", David Ung, Cristina Cifuentes (University of Queensland) 1999

"Binary Translation: Static, Dynamic, Retargetable?", Cristina Cifuentes (University of Queensland), Vishv Malhotra (University of Tasmania 1996

"The Design of a Resourceable and Retargetable Binary Translator", Cristina Cifuentes, Mike Van Emmerik (Queensland University), Norman Ramsey (Virgina University), 1999

"Transparent Dynamic Optimization", Vasanth Bala, Evelyn Duesterwald, Sanjeev Banerjia, HPL Cambridge, June 1997

"Dynamo: A Transparent Dynamic Optimization System", Vasanth Bala, Evelyn Duesterwald, Sanjeev Banerjia , HPL Cambridge, 2000 http://www.hpl.hp.com/cambridge/projects/Dynamo

"'Microprocessor' = Dynamic Optimization Software + CPU Hardware", HPL (DYNAMO)

"Dynamo as VM Accelerator", HPL (DYNAMO)

"Dixie: Instrumentador y Emulador de Binarios Unix" Vol I. Manel Fernandez Gomez, FIB, 1998.

Reports and Papers

"An Out-of-Order Execution Technique for Runtime Binary Translators", Bich C. Le (PDL-HP, leb@cup.hp.com 1998) [1008 ACM 1-58113-107-0/98/0010]

“Migrating a CISC Computer Family onto RISC via Object Code Translation”, Kristy Andrews, Duane Sand (Tandem Computers Inc. 1992) [1992 ACM O-89791-535-6/92/0010/0213]

“Optimizations and Oracle Parallelism with Dynamic Translation”, Kemal Ebciuglu, Erik R. Altman, Sumedh Sathaye, Michael Gschwind (IBM Yorktown Center, {kemel@watson.ibm.com , erik@watson.ibm.com , sathaye@watson.ibm.com , mikeg@watson.ibm.com 1999) [1072-4451/99 1999 IEEE]

“BOA: Targeting Multi-GHz with Binary Translation”, Erik Altman and others, IBM Research

“Dynamic and Transparent Binary Translation”, Michael Gschwind, Erik R. Altman, Sumedh Sathaye, Paul Ledak, David Appenzeller, PACT’99

“Binary Translation”, Richard L. Sites, Anton Chernoff, Matthew B. Kirk, Maurice P. Marks, Scott G. Robinson (Digital 1993) [ACM 1993, vol.36 no. 2]

“Binary Translation: A Short Tutorial”, Erik R. Altman, David Kaeli, Yaron Sheffer (PACT’99)

“Welcome to the opportunities of Binary Translation”, Erik R. Altman, David Kaeli, Yaron Seffer, PACT’99

“PA-RISC to IA-64: Transparent Execution, No Recompilation”, Cindy Zheng, Carol Thompson (HP March 2000) [0018-9162/00 IEEE March 2000]

“Embra: Fast and Flexible Machine Simulation”, Emmett Witchel, Mendel Rosenblum (MIT witchel@lcs.mit.edu, Standford mendel@cs.standford.edu 1996) [Sigmetrics’96]

“Shade: A Fast Instruction-Set Simulator for Execution Profiling”, Bob Cmelik, David Keppel (SUN rfc@eng.sun.com, UW pardo@cs.washington.edu 1994) [ACM SIGMETRICS 1994]

“Shade: A Fast Instruction-Set Simulator for Execution Profiling”, Bob Cmelik, David Keppel (SUN rfc@eng.sun.com, UW pardo@cs.washington.edu 1994) [LARGE REPORT]

“A Structuring Algorithm for Decompilation”, Cristina Cifuentes, Queensland Universtiy, August 1993

“The Impact of Copyright on the Development of Cutting Edge Binary Reverse Engineering Technology”, Cristina Cifuentes, U. Queensland

“A Transformational Approach to Binary Translation of Delayed Branches”, Norman Ramsey, Cristina Cifuentes, 1999

“Software Profiling for Hot Path Prediction: Less is More”, Evelyn Duesterwald, Vasanth Bala, HPL 2000

“A JAVA ILP Machine Based on Fast Dynamic Compilation”, Kemal Ebcioglu, Erik Altman, Erdem Hokene, IBM Yorktown Center, 1997

“Timing Insensitive Binary to Binary Translation of Real Time Systems”, Bryce Cogswell (CMU), Zary Segall (U. Oregon), 1994

<u>Commercial Products</u>

“DIGITAL FX!32: Combining Emulation and Binary Translation”, Raymond J. Hookway and Mark A.Herdeg, (DIGITAL Technical Journal, 28 August 1997)

“White Paper: How DIGITAL FX!32 works”, DIGITAL

"Executor Internals: How to Efficiently Run Mac Programs on PCs", Mathew J. Hostetter mat@ardi.com , Clifford T. Matthews cmt@ardi.com (Ardi 1996)

"Syn68k: ARDI's dynamically compiling 68LC040 emulator", Mat Hostetter mat@ardi.com (Ardi October 27 1995)

"The Technology Behind Crusoe Processor", Alexander Klaiber, Transmeta Corp., January 2000

"Combining hardware and software to provide an improved microprocessor" (Crusoe's Transmeta Patent), Robert F. Cmelik, et al. (US Patent and Trademark Offioce, February 29, 2000)

"Builiding the Virtual PC", Eric Traut, Core Technologies, Nov 1995

"The DR Emulator (Technote PT39)", Eric Traut, Apple, February 1995

"HW 28 – DR Emulator Caches", Apple, 8 April 1996

"Java Hot Spot Documentation"

"Opendoc and Java Beans", Gregg Williams

"What is Binary Translation (Freeport Express)", Richard Gorton, Digital 1995

<u>Other documents</u>

"DRFAQ: Dynamic Recompilation – Frequently Asked Questions", Michael König (M.I.K.e), mike@dynarec.com (www.dynarec.com/~mike/drfaq.html 29-8-20000).

"Embra Web Pages"

"Emulators and Emulation", Bill Haygood 1999

"Interview to Jeremey Chadwick"

Dynamic Code Generation

"VCODE: A Retargetable, Extensible, Very Fast Dynamic Code Generation System", Dawson R. Engler (MIT 1995) [ACM PACT'96]

"A VCODE Tutorial", Dawson R. Engler (MIT, April 28 1996)

"DCG: An Efficient, Retargetable Dynamic Code Generation System", Dawson R. Engler (MIT), Todd A. Proebsting (University of Arizona) 1993?

"Dynamo: A Staged Compiler Architecture for Dynamic Program Optimization", Mark Leone, R. Kent Dybvig (Indiana University 1997)

"Efficient Compilation and Profile-Driven Dynamic Recompilation in Scheme", Robert G. Burger, Indiana Universtity, March 97

"'C: A Language for High-Level, Efficient and Machine-Independant Dynamic Code Generation", Dawson R. Engler, Wilson C. Hsieh, M. Franskaashoek, MIT 1995 (ACM)

"tcc: A System for Fast, Flexible and High-Level Dynamic Code Generation", Massimiliano Poletto, Dawson R. Engler, M. Frans Keashoek, MIT 1996

"tcc: A Template-Based Compiler for 'C". Massimiliano Poletto, Dawson R. Engler, M. Frans Keashoek, MIT 1996

Emulation and Simulation

“Some Efficient Architecture Simulation Techniques”, Robert Bedichek, University of Washington, 1990

“SUPERSIM – A New Technique for Simlationof Programmable DSP Architectures”, C. Zivojnovic, S. Pees, Ch. Schläger, R. Weber, H. Meyr, Aachen University (Germany), 1995

“Optimizing direct threaded code by selective inlining”, Ian Piamurta, Fabio Riccardi, INRIA, ACM 1998

“Pipelined Java Virtual Machine Interpreters”, Jan Hoogerbrugge and Lex Augusteijin. Philips research labs.

“SimGen:Development of Efficient Instruction Set Simulators”, Fedrik Larsson, Peter Magnusson, Bengt Werner. 1997

“SimICS/sun4m: A virtual workstation”, Peter S. Magnusson, Fredrik Dahlgren and co.

General emulation

1964 Recompilation Engine Documentation. **http://1964.emulation64.com/**

An unconventional proposal: using the x86 architecture as ¨The obiquitous virtual standard architecture”, Jochen Liedtke, Nayeem Islam, Trent Jaeger, Vsevolod Panteleenko. IBM Research Center.

“Emu-mech”, Discussion in comp.sys.amiga.emulations about emulators in the Amiga computer, 1991.

“Arcade Emulation How To v 0.30”, Michael Adcock, Neil Bradley, Dave Spicers and a lot more. 1997.

“How do I write an emulator?” Daniel Boris, 1999.

“How to write an emulator” Marat Fayzullin.

Information about CPUs, consoles and emulated machines

“YM2143 Datasheet”, Yamaha Corp, 1996.

“Sega Master System Technical Information”, Richard Talbot-Watkins, 1998

“Sega CD programming FAQ”, Christian Schiller, 1998

“Sega Genesis VDP documentation”, Charles Mac Donald, 2001

“Sega Genesis Technical Notes”, Bart Trzynadlowski, 2000

“Genesis Technical Overview”, SEGA, Confidential official documentation.

“Master System Technical Document”, Jason Starr, 2001

“SMS/GG hardware notes” Charles Mac Donald.

“Genesis Hardware notes”, Charles Mac Donald.

“Everything you have always wanted to know about the Playstation but were afraid to ask”, Joshua Walker.

"8080 Datasheet", Intel
"Z80 Datasheet", Zilog

"M68000 Programmer's Reference Manual", Motorola

"Pentium and Pentium 2 Programmer's Manuals", Intel

Web pages

General information about console, computer and arcade emulatros.

www.vintagegaming.com

www.retrogames.com

www.emuhq.com

www.zophar.net

www.emulatronia.com

www.emumania.com

Binary translation

http://www.uq.edu.au/~csdung/index.html/

http://www.dcs.warwick.ac.uk/~csuix/project/

http://www.support.compaq.com/amt/

http://www.csee.uq.edu.au/~cristina/

http://www.research.ibm.com/daisy/

http://www.digital.com/DTJP01/DTJP01HM.HTM

http://www.ac.upc.es/dixie/

http://www-flash.stanford.edu/Embra/

http://www.xsim.com/bib/index.html

http://www.ifi.unizh.ch/richter/people/pilz/oct/

http://www.cs.washington.edu/research/compiler/papers.d/shade.html

http://www.csee.uq.edu.au/~csmweb/decompilation/

http://simos.stanford.edu/

http://www.digital.com/info/DTJ809/DTJ809SC.TXT

General emulation

http://cps2shock.retrogames.com

http://www.atarihq.com/danb/emulation.html http://zan.emuunlim.com/emul8/

http://www.jonshome.net/EPR/

http://www.classicgaming.com/epr/

http://etc.home.dhs.org/

http://www.emuhq.com/howtoemulation/

http://cgfm2.emuviews.com/

http://www.arcadedev.com/

http://smspower.speedhost.com/dev/forum/index.shtml

http://www.emuhq.com/stardust/

http://www.system16.com/

http://trzy.overclocked.org/

http://www.zsnes.com/

Acknoledgements

To my project director Agustin Fernandez for supporting me.

To the Dixie Team, Roger Espasa, Manel Fernandez and the others.

To the MUL8 mail list and the people there (Dan Boris, Neil Bradley, Kirion Wilkinson and all others) because it as a good start.

To the Dynarec mail list and web site, a place to learn even more about emulation. Neil Bradley, Mike Konig, Neil Griffiths, Bart Trzynadlowski, David Sharp, Julian Brown and all the others, the list is too large.

To Daniel Gracia for supporting me.

To the #emuladores IRC-Hispano channel and some of the guys who helped me there.

To the SMS8Dev group, because I learned even more about the Master System and emulation, Zoop, Mike and all the others.

To all the emulators authors, starting with Neil Bradley and the great Sardu.

To all the people who gets fun playing with old systems.

To Akira Toriyama for the better characters designs and Square for the best RPG ever made ;).

I don’t have a cat ... so I can’t thank him, sorry.

Appendix A. Space Invaders Emulator Tutorial.

For reducing the size of this document and because we thought that the tutorial was more suited to be with the implemented emulator and source the Space Invaders Emulator Tutorial could be found in the CD-ROM that will be delivered with this document.

Appendix B. CD-ROM contents.

The CD-ROM will contain the binaries and sources of the Sega Master System emulator and the Space Invaders emulator. It will also contain software copies of a small number of Master System Cartridge ROMs (games) to permit the test of the emulator. It will contain copies of the Space Invaders Arcache Machine ROMs and sample wave files of the sound produced by the machine. There will a file explaining the contents of the CD-ROM, the requeriments of the emulators and how must be used.

At last it will contain text files with the Space Invader Emulator Tutorial.

Appendix C. User manuals.

They will be found in the CD-ROM delivered with the document.