To study the IBM PC, it is easiest to first break it down to small parts. Six sub-systems form a pipeline: Inputs, CPU, RAM, Disk, Video, and Audio.

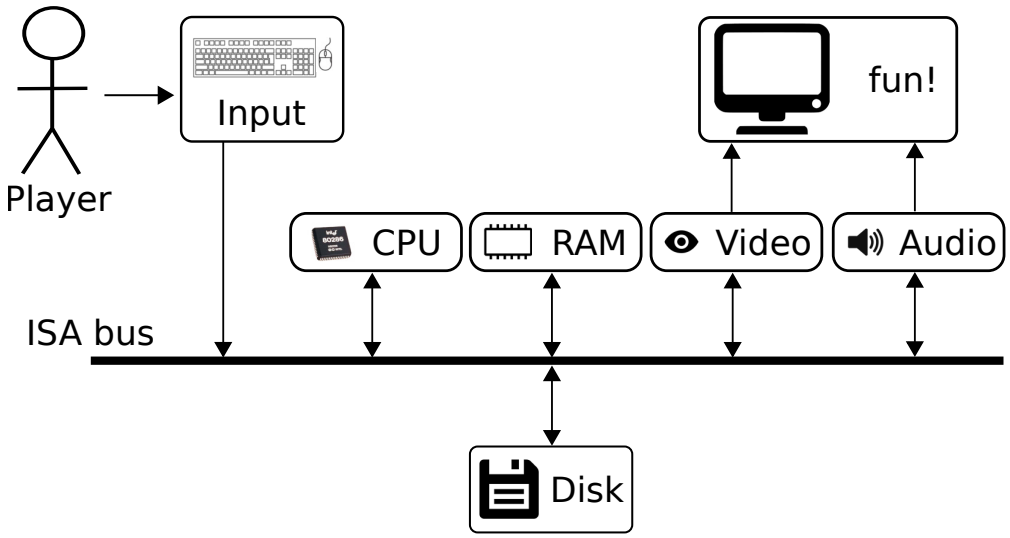


Figure 1: Hardware pipeline.

A lot of friction was present since manufacturers had not embraced the gaming industry yet. Parts quality varied from bad, terrible, to downright impossible to deal with.

Stage	Quality
RAM	Bearable
Video	Impossible
Audio	Very Poor
Inputs	Ok
CPU	Good
Disk	Ok

Figure 2: Component quality for a game engine.

0.1 CPU: Central Processing Unit

In 1989 around 15% of the households owned a computer[1]. The performance of these machines was so overwhelmingly determined by the CPU that a PC was referred to not by its brand or GPU[2], but by the main chip inside. If a PC had an Intel 8088 or equivalent, it was called a "XT". If it had an Intel 80286, it was a "286" or "AT".

0.1.1 Overview

Intel released the 8086 in 1979, which was the first microchip of the successful x86 family line. One year later, in 1979, it released the 8088 which was a variant of the 8086. The main difference between the two is that there are only eight data lines for the external data bus in the 8088 instead of the 8086's 16 lines. However, because it retained the full 16-bit internal registers and the 20-bit address bus, the 8088 ran 16-bit software and was capable of addressing a full 1MB of RAM. IBM chose the 8088 over the 8086 for its original PC/XT, because Intel offered a better price for the former and could supply more units.

In 1982 Intel released the 80286 microchip. A typical 8088 chip was running at 4.77Mhz, where the 80286 was running at 6Mhz and later popular versions at 12-16Mhz. The 80286 was employed for the IBM PC/AT, introduced in 1984, and then widely used in most PC/AT compatible computers until the early 1990s[3]. It would be the last, fastest 16-bit PC processor Intel made. It's successor, the Intel 80386, was a true 32-bit processor with a 32-bit data bus. Although Commander Keen was 100% compatible with the IBM-PC, it was hardly playable on a 8088 CPU. A 80286 or higher was recommended to play the game.

[1] https://www.statista.com/statistics/184685/percentage-of-households-with-computer-in-the-united-states-since-1984/

[2] There was no GPU yet. The term was coined by Nvidia in 1999, who marketed the GeForce 256 as "the world's first GPU", or Graphics Processing Unit.

[3] By the end of 1988, Intel estimates there were around 15 million 286-based PCs in use worldwide.

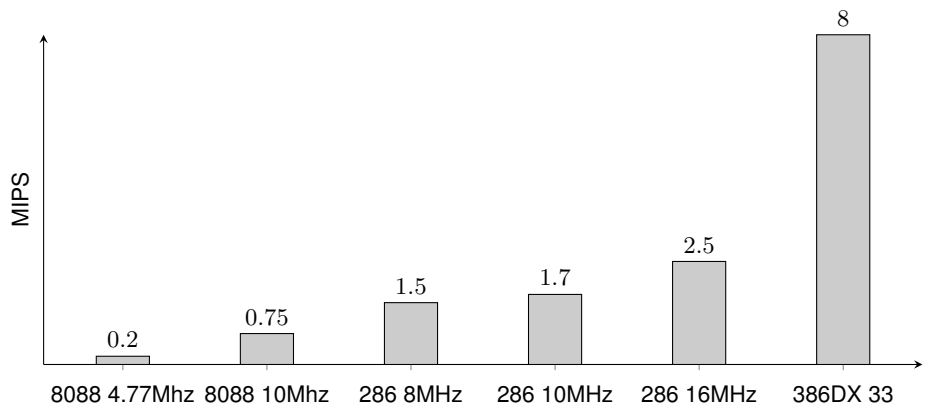


Figure 3: Comparison[4]of CPUs with MIPS

Trivia : A modern processor such as the Intel Core i7 3.33 GHz operates at close to 180,000 MIPS.

0.1.2 The Intel 80286

The Intel 80286 chip, first introduced in 1982, is the CPU behind the original IBM PC AT (Advanced Technology). Other computer makers manufactured what came to be known as IBM clones, with many of these manufacturers calling their systems AT-compatible or AT-class computers.

When IBM developed the AT, it selected the 286 as the basis for the new system because the chip provided compatibility with the 8088 used in the PC and the XT. Therefore, software written for those chips should run on the 286. The 286 chip is many times faster than the 8088 used in the XT, and at the time it offered a major performance boost to PCs used in businesses. The processing speed, or throughput, of the original AT (which ran at 6MHz) is five times greater than that of the PC running at 4.77MHz. The 286 computers are faster than their predecessors for several reasons. The main reason is that 286 processors are much more efficient in executing instructions. An average instruction takes 12 clock cycles on the 8088, but takes an average of only 4.5 cycles on the 286 processor. Additionally, the 286 chip can handle up to 16 bits of data at a time through an external data bus twice the size of the 8088.

[4]Roy Longbottom's PC Benchmark Collection: http://www.roylongbottom.org.uk/mips.htm.

The 286 chip has two modes of operation: real mode and protected mode. The two modes are distinct enough to make the 286 resemble two chips in one. In real mode, a 286 acts essentially the same as an 8086 chip and is fully compatible with the 8086 and 8088. In the protected mode of operation, the 286 was truly something new. In this mode, a program designed to take advantage of the chip's capabilities has access to 1GB of memory (including virtual memory). The 286 chip, however, can address only 16MB of hardware memory. A significant failing of the 286 chip is that it cannot switch from protected mode to real mode without a hardware reset (a warm reboot) of the system (It can, however, switch from real mode to protected mode without a reset).

Trivia : Gordon Letwin of Microsoft found a way to switch back from protected to real mode, using a "triple fault"[5] to soft reset the 286 CPU into real mode. However, it could take nearly 1 second, making switching from protected to real not feasible to be done often.

While the 8088 used a 3.0μm process, the 20286 used a 1.5μm process. The smaller process and increased surface (from 33mm^2 to 49mm^2) allowed Intel to pack 134,000 transistors on a 286 chip versus 29,000 on a 8088 chip.

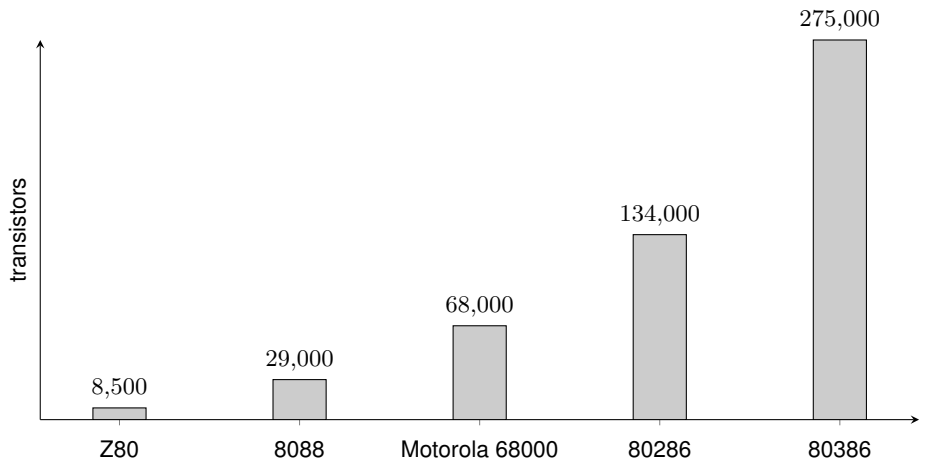
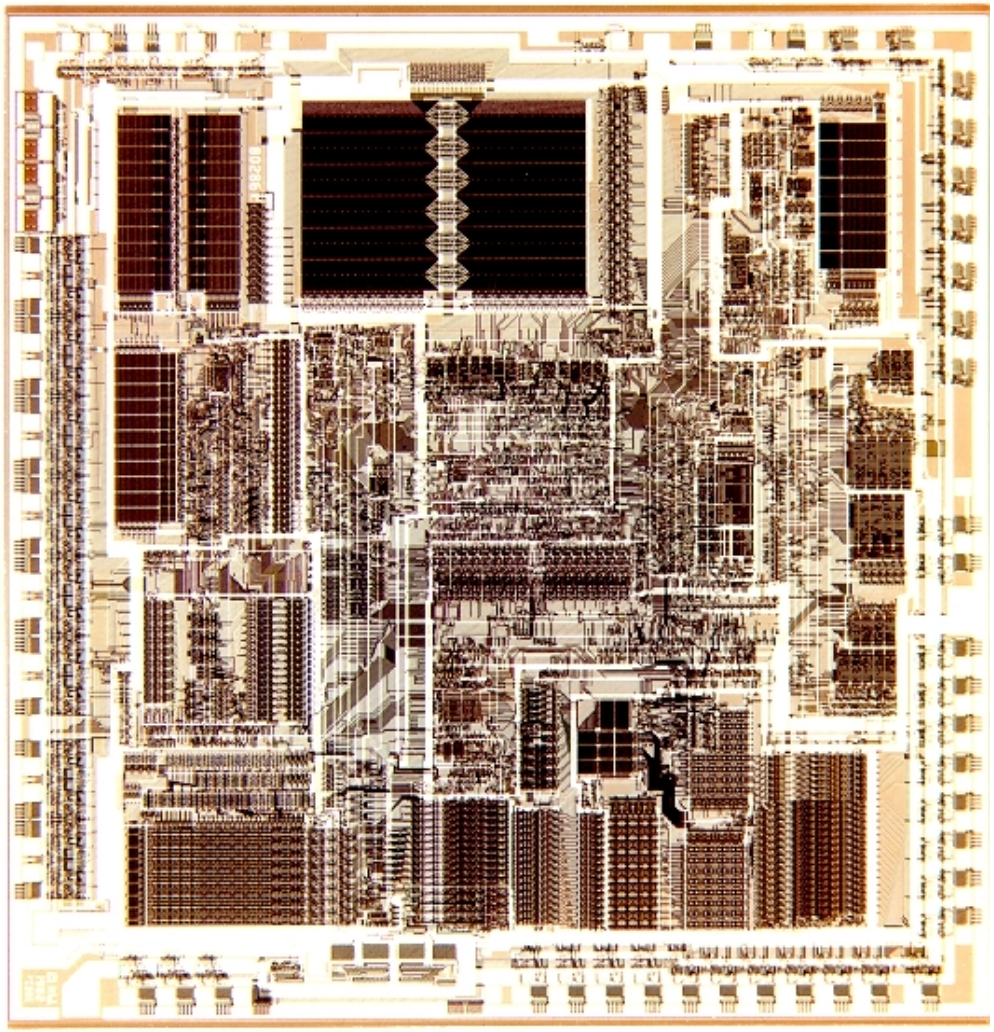


Figure 4: Comparison of CPUs with # of transistors.

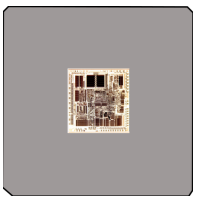
Trivia : Fast forward to today, the AMD Ryzen 9 7950X contains over 13 billion transistors. We've come a long way.

If you are holding a physical 9.25"x7.5" copy of this book, the CPU packaging is 25x25 mm square and the die is 7x7 mm, at 1:1 scale.

[5] https://en.wikipedia.org/wiki/Triple_fault.




A80286-8
L9409047
©1982,1985



Despite the apparent complexity, the 80286 can be summarized by four functional units and a three-stage instruction pipeline.

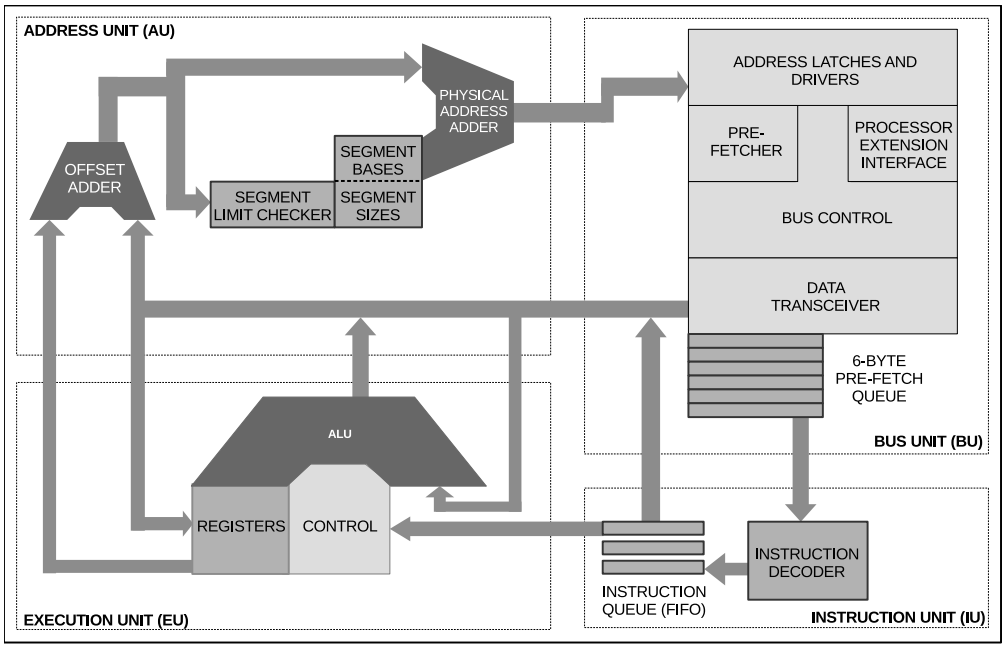
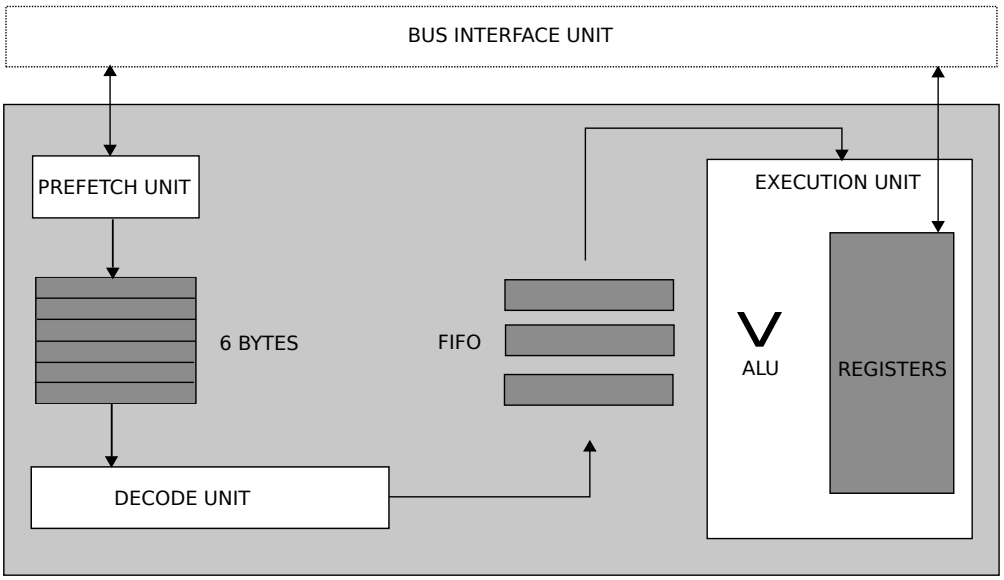


Figure 5: Internal block diagram of the 80286 processor

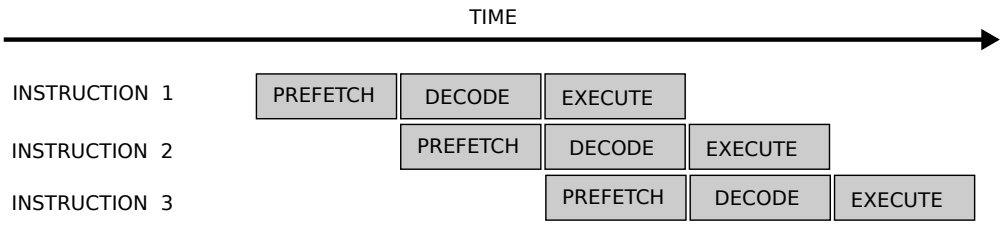
The four functional units can be described by

- **address unit (AU)** is used to determine the physical addresses of instructions and operands which are stored in memory. The address lines derived by AU can be used to address different peripheral devices such as memory and I/O devices.
- **bus unit (BU)** interfaces the 80286 with memory and I/O devices. The bus unit is used to fetch instruction bytes from the memory and stores them in the prefetch queue.
- **instruction unit (UI)** receives instructions from the prefetch queue and an instruction decoder decodes them one by one. The decoded instructions are latched onto a decoded instruction queue.
- **execution unit (EU)** is responsible for executing the instructions received from the decoded instruction queue. The execution unit consists of the register bank, arithmetic and logic unit (ALU) and control block. The ALU is the core of the EU and perform all the arithmetic and logical operations.

The performance increase of the 80286 over the 8088 was mainly to the fact that address calculations (such as segment+offset) were less expensive. They were performed by the dedicated address unit in the 80286, while the older 8088 had to do effective address computation using its ALU, consuming several extra clock cycles in many cases.



The three units in the execution group form a three stage pipeline: Prefetch, Decode, and Execute. The Prefetch Unit wakes up when the Execution unit is performing but not using the bus and fetches instructions in a 6-byte queue. The prefetcher is linear and cannot predict the result of a branch. As a result, a jump (`JMP`) instruction triggers a flush of the entire pipeline. Instructions go down the pipeline and are decoded by the Decode Unit: the result of the decode operation is stored in a three-element FIFO where it is picked up by the Execution Unit.



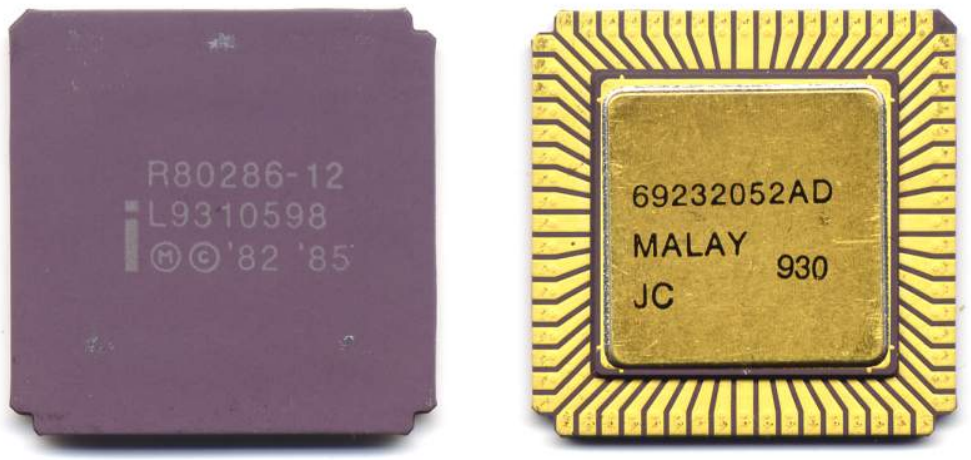


Figure 6: The Intel 286, 7mm by 7mm packing 134,000 transistors

From a programming perspective, a 286 CPU can be summarized by the following elements:

- Arithmetic Logic Unit performing `add`, `sub`, `mul` et cetera.
- 14 registers:
  - 16-bit General Purpose Registers: AX, BX, CX, DX
  - 16-bit Index Registers: SI, DI, BP, SP
  - 16-bit Segment Registers: CS, DS, ES, SS
  - 16-bit Status and Control Register
  - 16-bit Program Counter: IP
- A 24-bit address bus for up to 16MiB of flat addressable RAM
- Memory Management Unit

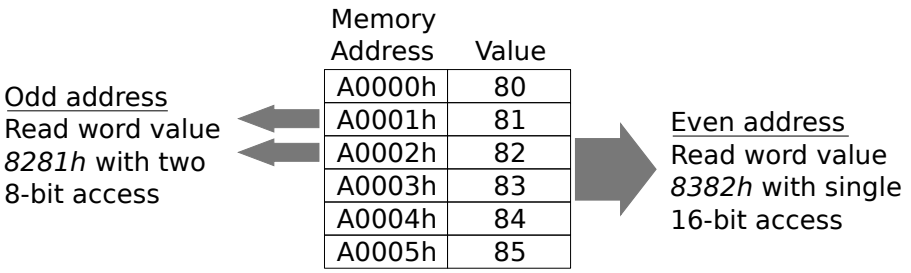
Despite its pipeline design, the 286 cannot do an operation in less than two cycles. Even a simple `ADD reg, reg` or `INC reg` takes two clocks. This is due to the absence of a SRAM on-chip cache and a slow decoding unit. Also have a look at multiplications which cost 24 cycles. So as a game developer you really want to avoid many multiplications during game runtime.

Instruction type	Clocks
`ADD reg8, reg8`	2
`INC reg8`	2
`IMUL reg16, reg16`	24
`IDIV reg16, reg16`	28
`MOV [reg16], reg16`	5
`OUT [reg16], reg16`	3
`IN [reg16], reg16`	5

Figure 7: 286 instruction costs[6].

0.1.3 16-bit Data alignment

Compared to the Intel 8088, the 286 CPU contained a 16-bit external data bus where the 8088 only had a 8-bit bus. Thanks to its 16-bit bus, the 286 can access and write word-sized memory variables just as fast as byte-sized variables. There's a catch, however: That's only true for word-sized variables that start at even memory addresses. When the 286 is asked to perform a word-sized access starting at an odd memory address, it actually performs two separate accesses, each of which fetches 1 byte, just as the 8088 does for all word-sized accesses. In other words, the effective capacity of the 286's external data bus is halved when a word-sized access to an odd address is performed[7].



The way to deal with the data alignment cycle-eater is straightforward: Don't perform word-sized accesses to odd addresses on the 286 if you can help it. This is not an issue for small memory operations, but it will harm performance when copying large memory blocks.

[6] Intel 80286 programmer's reference manual - 1987.
[7] See Michael Abrash's Graphics Programming Black Book Special Edition, chapter 11.

0.2 RAM

The first CPUs in the Intel x86 family were designed in 1976. It was introduced at a time when the largest register in a CPU was only 16-bits long which meant it could address only 65,536 bytes (64 KiB[8]) of memory, directly. For example, the Apple II and the Commodore 64 both shipped with 64KiB, which was enough to write and run amazing things.

But everyone was hungry for a way to run much larger programs. Rather than create a CPU with larger register sizes, the designers at Intel decided to keep the 16-bit registers for their new 8086 and 8088 CPU and added a different way to access more memory: They expanded the instruction set, so programs could tell the CPU to group two 16-bit registers together whenever they needed to refer to an absolute memory location beyond 64 KiB.

0.2.1 Memory addressing

If the designers had allowed the CPU to combine two registers into a high and low pair of 32-bits, it could have referenced up to 4 GiB of memory in a linear fashion. Keep in mind, however, this was at a time when many never dreamed we'd need a PC with more than 640 KiB of memory for user applications and data[9]. So, instead of dealing with whatever problems a linear addressing scheme of 32-bits would have produced, they created the Segment:Offset scheme which allows a CPU to effectively address about 1 MiB[10] of memory. The Segment:Offset schema combines two 16-bit registers, one designating a segment and the other an offset within that segment.

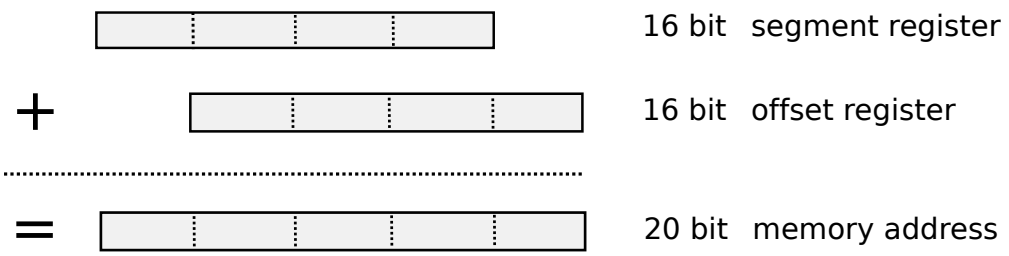


Figure 8: How registers are combined to address memory.

[8] This book uses IEC notation where KiB is 2^{10} and KB is 10^3.

[9] We've often heard that Bill Gates said something to the effect: "640K of memory should be enough for anyone.". Though many of us no longer believe he ever said those exact words, he did, however, during a video interview with David Allison in 1993 for the National Museum of American History, Smithsonian Institution, say: "I laid out memory so the bottom 640KiB was general purpose RAM and the upper 384KiB I reserved for video and ROM, and things like that."

[10] This book uses IEC notation where MiB is 2^{20} and MB is 10^6.

To support this architecture, the CPU keeps track of four different segments. Each of these segments has its own purpose and segment register:

- CS segment register, where the machine instructions resides.
- DS segment register, where the data resides.
- SS segment register, where the stack resides.
- ES segment register, which is used as extra data segment.

In C-language a memory address can be accessed directly using pointers. A pointer is a variable that stores the memory address of another variable as its value. There are two kinds of pointers: "near" and "far".

A near pointer refers to a function or data object that is within the default segment. It is 16 bits long and contains an offset into the current DS data segment if it's a data pointer, or into the current CS code segment if it's a function pointer. A far pointer could refer to a function or data that is in a different segment than the current, default segment. It is 32 bits long and contains a segment and offset, which identifies the location where the code or data is stored.

Accessing code or data with a near pointer is much faster than using a far pointer. When you use a near pointer, the program only needs to locate the code or data through the offset (or index) register. However, when using a far pointer, the program must first find the segment and then locate the code or data within that segment. For faster execution, one should use as many near pointers as possible. The drawback of using only near pointers is that they limit the program or data to 64KiB of memory.

It's important to note that a far pointer increments only the offset, not the segment. If you iterate over a data array larger than 64KiB, there will be no automatic overflow handling, meaning you can only address up to 64KiB of memory.

```
char far *p = (char far *)0xA000FFFF;
p++;
printf("%04X:%04X\n", FP_SEG(p), FP_OFF(p));
```

Will output:

```
A000:0000
```

To work with memory beyond this limit, you can use another type of pointer called a "huge" pointer, which allows pointer arithmetic to function correctly beyond the 64KiB boundary.

```
char huge *p = (char huge *)0xA000FFFF;
p++;
printf("%04X:%04X\n", FP_SEG(p), FP_OFF(p));
```

Will output the address:

```
B000:0000
```

The huge pointer is based on the absolute (or lineair) 20-bit memory location and Segment:Offset normalized address. The absolute memory address can be calculated by

```
Absolute memory address = (Segment * 0x10h) + Offset
```

For example the absolute address of `A000:002F` is `A0000h + 002Fh = A002Fh`. By confining the offset to just the hexadecimal values `0h` through `Fh`, we have a unique way to reference all Segment:Offset memory pair locations. This results in the normalized address `A002:000F`. A huge pointer is normalized when pointer arithmetic is performed on it.

```
# include <stdio .h>
# include <dos.h>
int main (void){
  char huge *p = MK_FP (0xA000 , 0xFFFF );
  p--;
  p++;
  printf("%04X:%04X\n", FP_SEG(p), FP_OFF(p));
}
```

Will output:

```
AFFF:000F
```

Trivia : Since the normalized form will always have three leading zero bytes in its offset, programmers often write it with just the digit that counts: `AFFF:F`

A `huge` reference is much slower than the `far` reference as it comes with additional overhead to update the segment and address normalization after every arithmetic manipulation. So, most programmers avoided the `huge` pointer, unless really needed.

0.2.2 80286 Real and Protected mode

By 1986, hardware had gotten cheaper and Intel made a departure from the old architecture with its 286. This new CPU could be put in what is called "protected mode" featuring a 24-bit-wide address bus for up to 16 MiB of flat RAM protectable with a MMU[11]. To make sure old programs could still run, the 286 processor could be put in "real mode" which replicates how the Intel 8086 and 8088 operated: 16-bit registers, 20-bit address bus giving 1MiB addressable RAM with segmented addressing.

For compatibility reasons all PCs have to start in real mode. You may assume that programmers of the late 80s promptly switched the CPU to protected mode to unleash the full potential of the machines and ditch the 20-year-old real mode. It would have worked out if the operating system had been able to run in protected mode. However, in the name of backward compatibility, Microsoft's DOS could only handle real mode which effectively locked developers into 16-bit programming.

With protected mode unavailable, 1990 developers programmed like it was 1976: with a 20-bit-wide address bus offering only 1MiB of addressable RAM. Regardless how much memory was installed on the machine, only 1MiB could be addressed. The memory layout for the real mode is as follows:

- From 00000h to 003FFh : the Interrupt Vector Table.
- From 00400h to 004FFh : BIOS data.
- From 00500h to 005FFh : command.com+io.sys.
- From 00600h to 9FFFFh : Usable by a program (about 620KiB in the best case).
- From A0000h to FFFFFh : UMA (Upper Memory Area): Reserved to BIOS ROM, video card and sound card mapped I/O.

Out of the original 1024KiB, only 640KiB (called Conventional Memory) was accessible to a program. 384KiB was reserved for the UMA and every single driver installed (`.SYS` and `.COM`) took away from the remaining 640KiB.

[11] Memory Management Unit

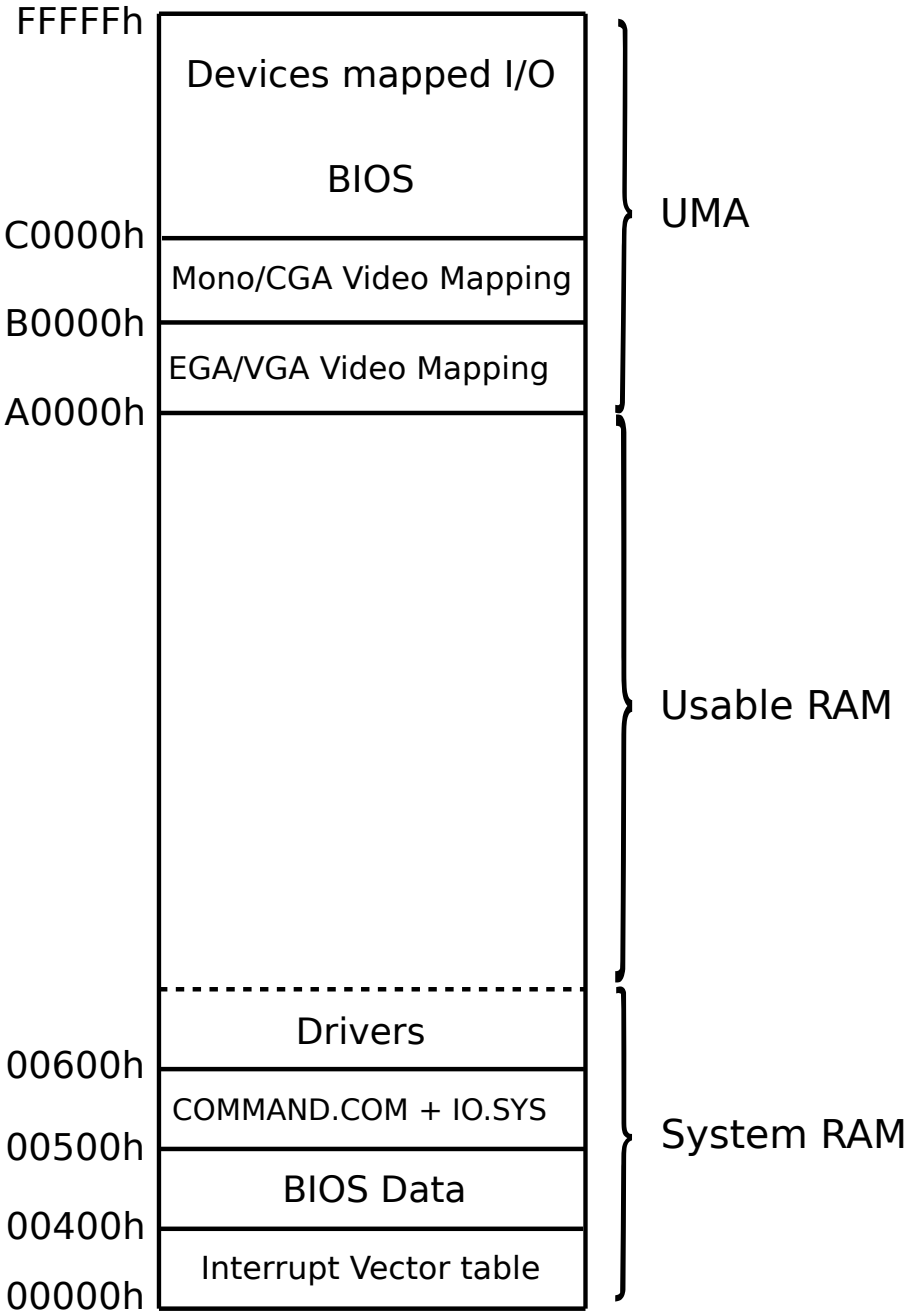


Figure 9: First 1MiB of RAM layout.

0.2.3 Real mode: Memory models

When a program is compiled and executed, the operating system allocates a chunk of memory to the program. This memory is divided into different segments:

- **Code section**: Stores the program executable. When you compile a C program, the compiler converts your code into assembly instructions that the CPU executes.
- **Data section**: Stores initialized and uninitialized global and static variables.
- **Stack section**: Memory used for local variables and data inside functions. The stack grows downwards, towards lower memory addresses.
- **Heap section**: Memory that is dynamically allocated using the malloc() function. The heap typically grows upwards, meaning it expands towards larger memory addresses.

The x86 real mode provides different memory segment layouts, called "memory models". Each memory model determines how segments are organized and defines whether the default pointer type for functions and data is near or far. A near pointer automatically associates with one of the segment registers. Six memory models[12] exist, each offering trade-offs between minimum system requirements, maximizing code efficiency, and accessing available memory.

	Default pointer type		**Size**		
Model	**Code**	**Data**	**Code**	**Data**	**Definition**
Tiny	near	near	<64KiB		CS=DS=SS
Small	near	near	<64KiB	<64KiB	DS=SS
Medium	far	near	>64KiB	<64KiB	DS=SS, multiple code segments
Compact	near	far	<64KiB	>64KiB	single code segment, multiple data segments
Large	far	far	>64KiB	>64KiB	multiple code and data segments
Huge	far	far	>64KiB	>64KiB	multiple code and (global) data segments

[12] See Borland C++ 3.1 Programmer's guide, section DOS Memory management.

The smallest is the "tiny memory model", where all three segment registers (CS, DS, SS) start at the same memory location. The first part of memory is used for code instructions, followed by the data section. The heap begins directly after the data section, and the stack starts at the opposite end of the segment, growing downward. Both the heap and stack can dynamically grow or shrink during program execution. If they continue to grow, they may eventually collide, causing a system or application crash.

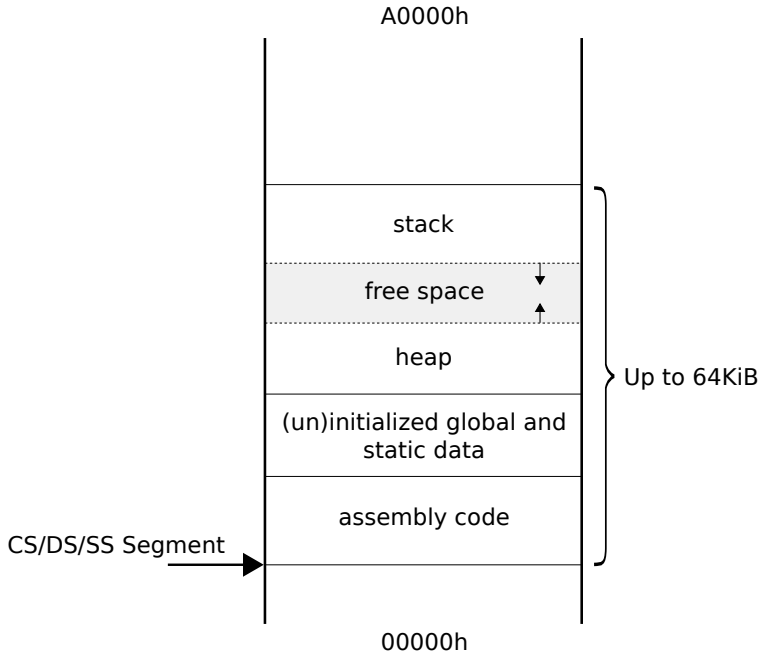


Figure 10: Tiny memory model.

All functions and data are accessed using near pointers, meaning the segment registers are set once and never changed during execution. However, this limits the program to a maximum of 64KiB, similar to programming on a 16-bit system.

Trivia : The tiny memory model was required by programs that ended with the .COM extension, and it existed for backward compatibility with CP/M operating system. CP/M ran on the 8080 processor which supported a maximum of 64KB of memory.

In the "small memory model", instructions and data are separated, each having its own 64KiB segment. The code segment can store up to 64KiB of instructions, while the global data, stack, and heap share a separate 64KiB segment. Code execution is efficient since both functions and data are accessed using offset registers (near pointers) only.

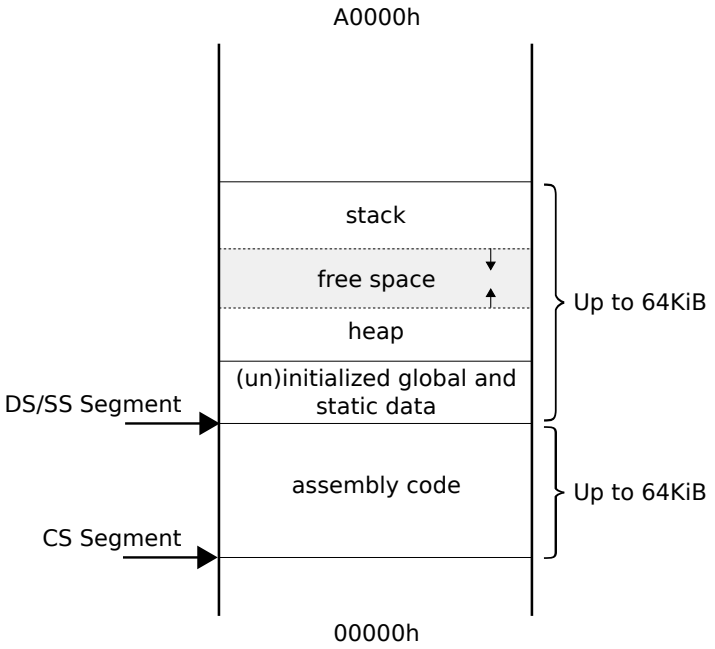


Figure 11: Small memory model, code and data each have 64KiB.

The "medium memory model" is ideal for programs with a large amount of code but minimal data. Many computer games fell into this category, since they had a lot of game logic, but not a lot of (global) game state data. In this model, each C source file has its own code segment, allowing up to 64KiB per file. A table manages all code segment references, with the CS register pointing to one segment at a time. Each function is a far pointer by default, requiring both the CS segment and IP offset register to be updated. While this model supports larger codebases, it comes at the cost of slower execution due to the far pointers.

Trivia : When compiling a source file, its code cannot exceed 64KiB, as it must fit inside one code segment. If the file is too large, the program must be broken into smaller source files and compiled separately.

The "compact memory model" is the opposite of the medium memory model, allowing more than 64KiB of data while restricting function code to 64KiB. The "large memory model" supports both code and data larger than 64KiB but requires far pointers for both, leading to slower execution. The "huge memory model" removes the 64KiB limit on global data, allowing more flexibility, but also incurs a performance penalty. A detailed layout and explanation of each memory model is described in Appendix **??**.

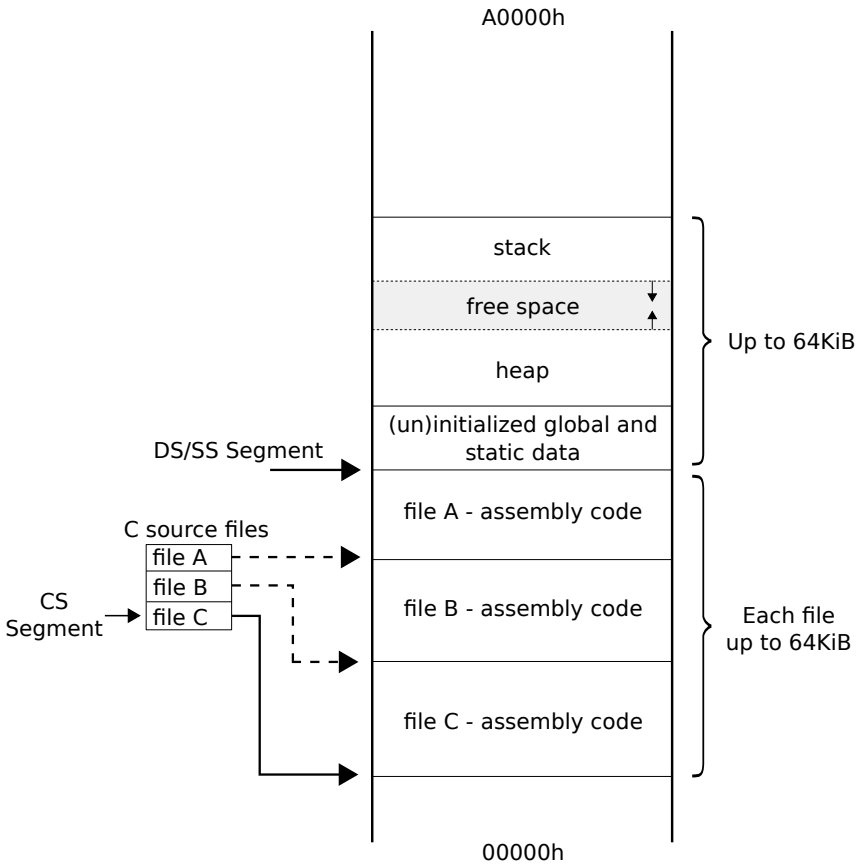


Figure 12: Medium memory model, code can be larger than 64KiB.

0.2.4 Mixed Model Programming

In the medium memory model, the data segment remains limited to 64KiB. For most games, this is enough to store game state variables. However, the size of the heap is far too small to store game data such as graphics and game levels. One option is to use the large or huge memory models, but these come with the downside of slower data access due to the use of far pointers for all data.

Fortunately, there is a way to access additional memory using the medium model. A large portion of unallocated memory is available between the stack and the high address

A0000h. This memory, known as the "far heap", can be allocated using the `farmalloc()` function. This technique, called mixed model programming, combines the advantages of one of the six standard memory models with custom near and far pointer allocation. In the case of the medium memory model, it allows the program to benefit from near pointer efficiency for global data and the stack, while also providing sufficient memory for dynamically allocated assets like graphics and levels.

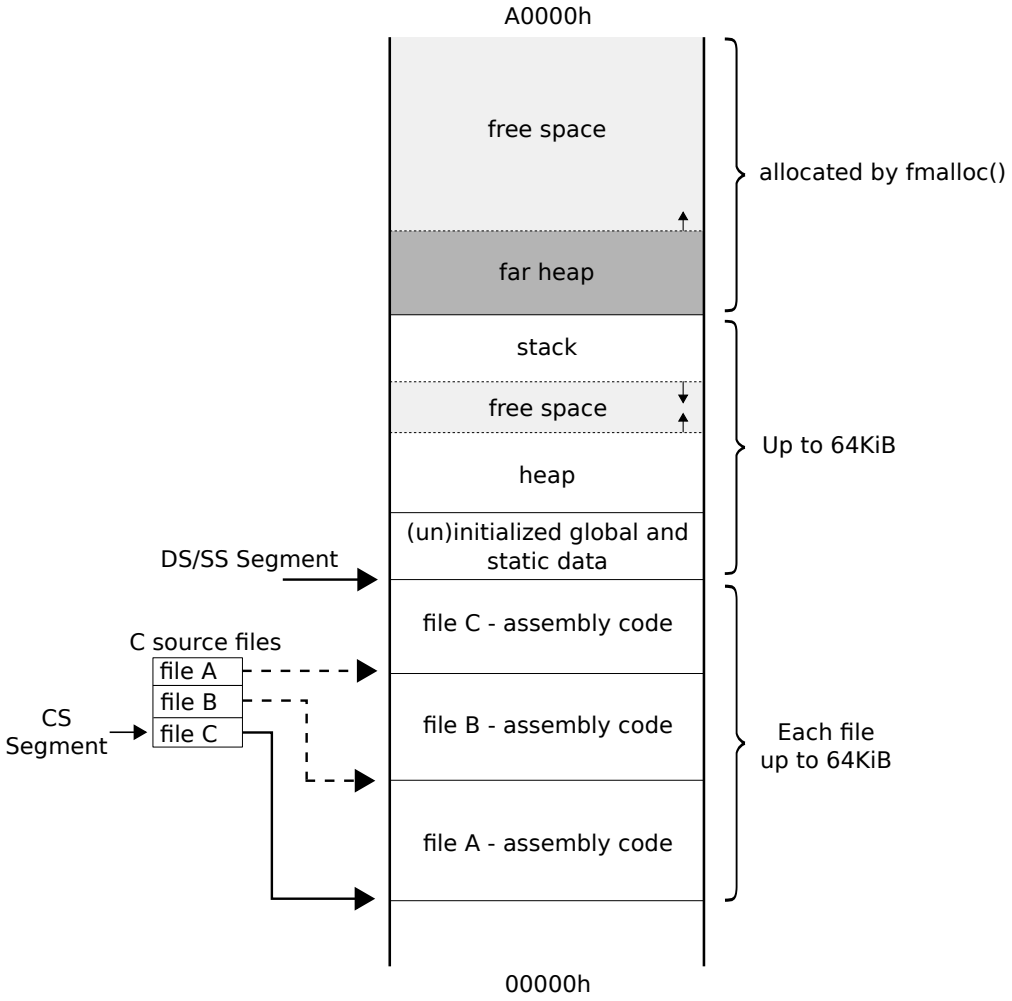


Figure 13: Medium memory model with far heap memory.

0.3 Video

PCs were connected to CRT monitors: big, heavy, small diagonal, cathode ray-based, curved-surface screens. Most had a 14" diagonal with a 4:3 aspect ratio.

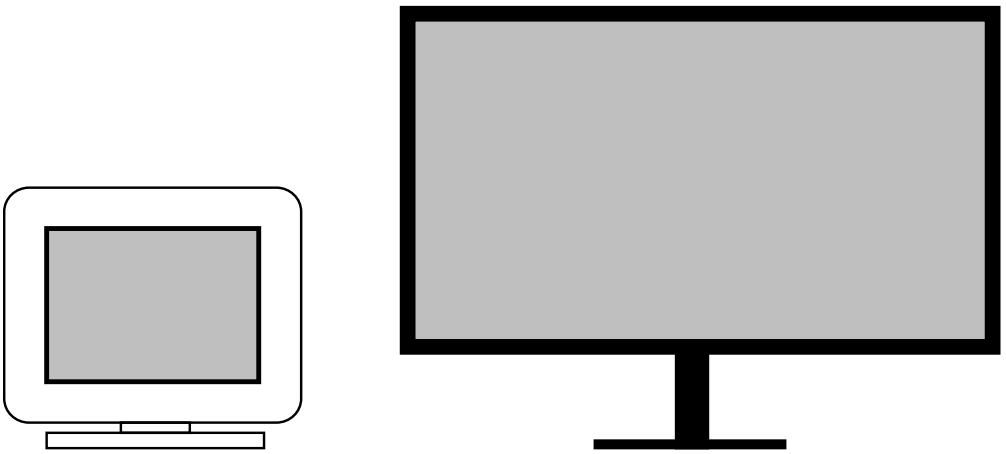


Figure 14: CRT (left) vs LCD (right)

To give you an idea of the size and resolution, figure 14 shows a comparison between a 14" CRT from 1990 (capable of a resolution of 640x200) and a 30" Apple Cinema Display from 2014 (capable of a resolution of 2560x1600).

Trivia : Despite their difference of capabilities, both monitors are the same weight: 27.5 pounds (12 kg).

0.3.1 CRT Monitor

All standard PC monitors use a raster-scan display to create the image. In a raster-scan display, the position of the electron beam is continually sweeping across the surface of the tube. The tube's surface is coated with phosphors that glow when struck by electrons (and for a short time thereafter), and, of course, the beam may be turned on in order to light a phosphor or off to leave it black.

The electron beam scans the phosphor-coated screen from left to right and top to bottom. The period during which the beams return to the left is known as the horizontal retrace. During most of the retrace, the guns must be turned off to prevent writing in the active display area (the area which contains the actual character and/or graphics data); this is

known as horizontal blanking.

The area immediately surrounding the display area, in which the beam may be turned on during the retrace interval, is called the overscan or border. The active display area is the portion of the screen that contains characters and/or graphics. These components of the scan are shown in simplified form in Figure 15.

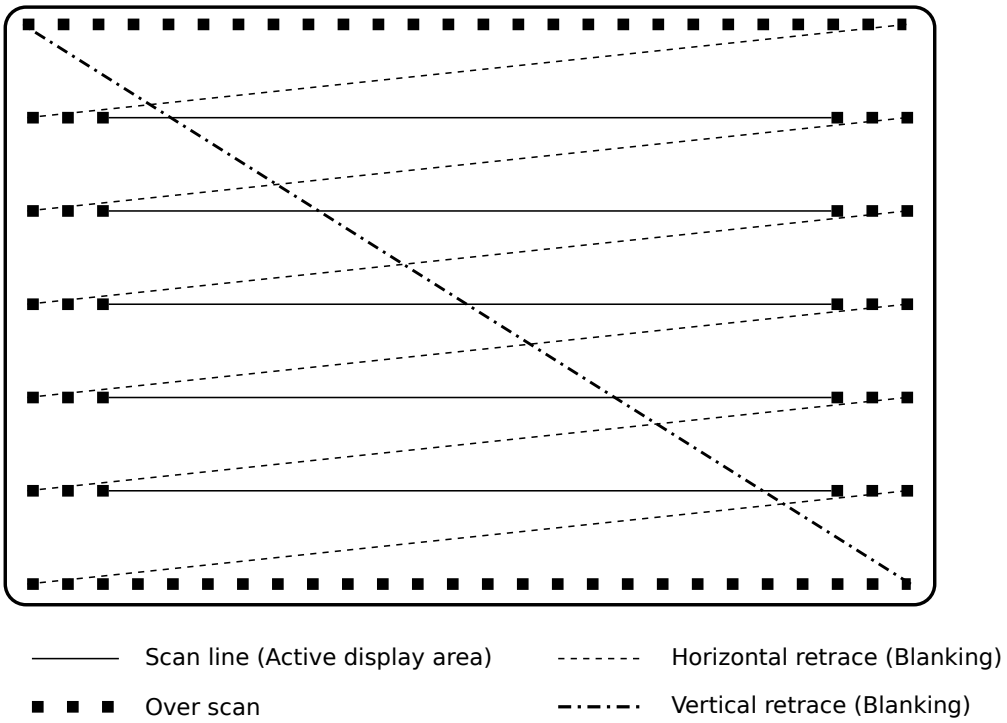


Figure 15: Simplified CRT monitor scan.

After a horizontal scan has been completed, the beam is moved to the next line during the horizontal retrace. This sequence continues until the last line, at which point the vertical retrace begins. The vertical retrace is similar to the horizontal retrace; the electron beam may be enabled through a small overscan area and then turned off (vertical blanking) as the beam returns to the top left comer of the screen.

If the vertical refresh is too slow, the display will flicker. Most people can detect flicker when the refresh rate drops below 60 Hz, and thus most displays use vertical refresh frequencies of about 60Hz (EGA) to 70Hz (VGA).

0.3.2 History of Video Adapters

The Monochrome Display Adapter (MDA) was released in 1981 with the IBM PC 5150. It offered two colors, allowing 80 columns by 25 lines of text. While not great, it was standard on every PC. Many other systems followed over the years, each of them preserving backward compatibility.

Name	Year Released	Memory	Max Resolution
MDA (**M**onochrome **D**isplay **A**dapter)	1981	4KiB	80x25[13]
Hercules	1982	64KiB	720x348
CGA (**C**olor **G**raphics **A**dapter)	1981	16KiB	640x200
EGA (**E**nhanced **G**raphics **A**dapter)	1985	64KiB	640x350
VGA (**V**ideo **G**raphics **A**rray)	1987	256KiB	640x480

Figure 16: Video interface history.

Each iteration added new features and by 1990 the predominant graphic system was EGA, although the VGA system was rapidly becoming the new standard. All video cards installed on PCs had to follow the standard set by IBM. The universality of that system was a double-edged sword. While developers had to program for only one graphic system, there was no escaping its shortcomings.

0.3.3 Introduction of EGA Video Card

IBM introduced the Enhanced Graphics Adapter (EGA) in 1985 as the successor of CGA. The standard card was shipped with only 64KiB video memory, but it had the option to expand the memory using the onboard graphics memory expansion card. Figure 18 shows the original IBM EGA card, a clunky beast full of discrete components. The memory consists out of TMS4416 RAM, a common memory chip for (home) computers around that period. Each chip contains 16KiB of 4-bits memory, so one needs two chips to end up having 16KiB of 8-bit memory and eight chips for 64KiB of 8-bit memory.

Trivia : Texas Instruments introduced the first 16KiB by 4-bits as TMS4416 in 1980[14]. Still it took until 1983-84 until they became widely available and lower priced than four TMS4116 chips (16KiB by 1-bit). However, at that time 64 KiB RAM was the way to go for new designs. Computers with only 16 KiB as base memory - and that's where TMS4416 would have been a cost saver - were already on the way out.

[13] Text mode only.
[14] https://pdf1.alldatasheet.com/datasheet-pdf/view/103706/TI/TMS4416.html

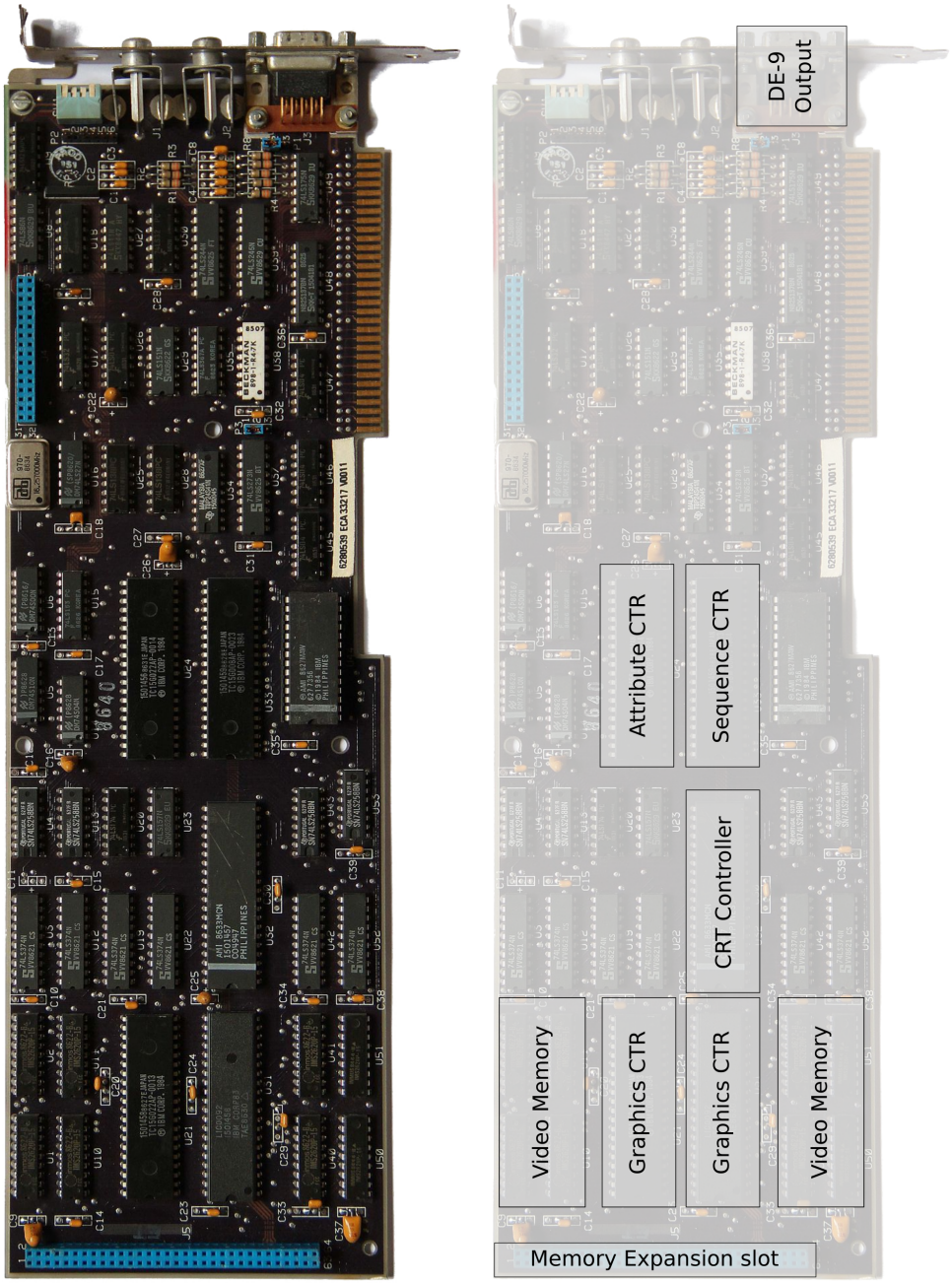


Figure 17: Original IBM EGA card

To add additional video memory to the IBM EGA card a Graphics Memory Expansion Card could be purchased. By default only the bottom row of memory was populated with chips, expanding the total EGA video memory to 128KiB. The expansion card provided DIP (dual in-line package) sockets for further memory expansion. Populating the DIP sockets with a Graphics Memory Module Kit adds two additional rows of 64KiB, bringing the EGA memory to its maximum of 256KiB.

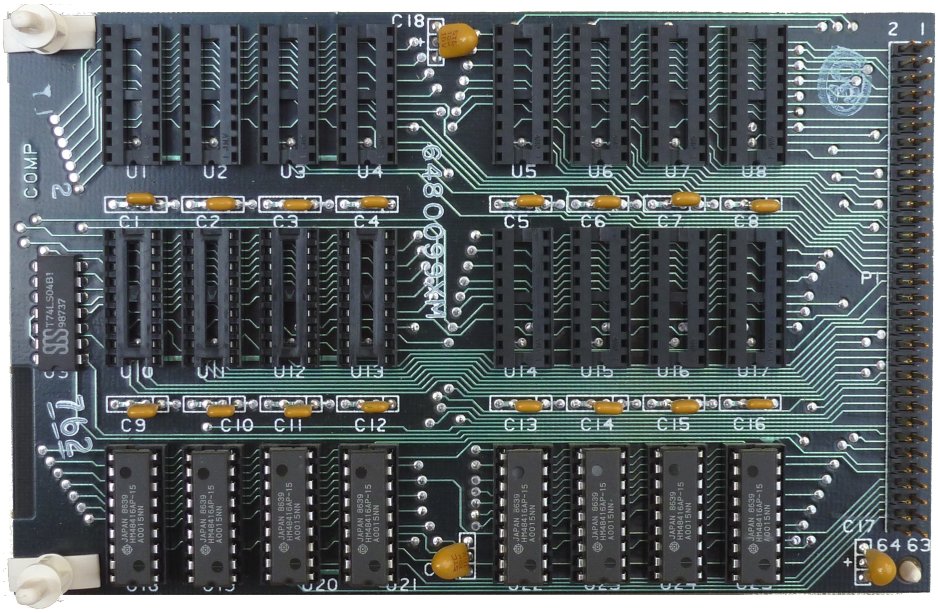
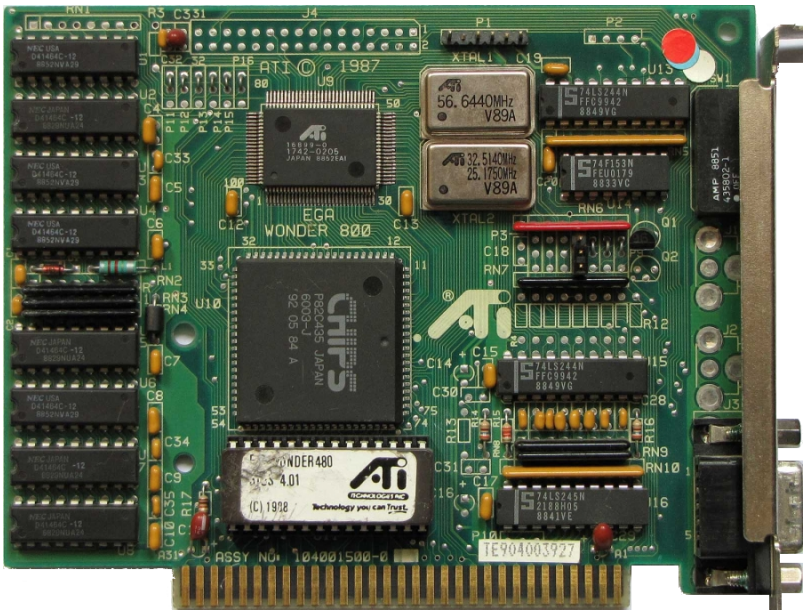


Figure 18: EGA Graphics Memory Expansion Card, bottom row populated with chips.

The EGA clones that started coming along in 1986-87 were based on integrated chipsets, and the vast majority of them came with the maximum of 256KiB on board. When Commander Keen came out, the headcount of EGA cards with less than 256KiB would've been practically negligible[15].

The next page shows an ATI EGA Wonder 800 (top) and Paradise AutoSwitch EGA 350 (bottom), both are 8-bit ISA. The eight chips on the left of the card form the VRAM where the framebuffers are stored.

[15]PC Tech Journal Oct 1986 (page 82-83) and PC Tech Journal Nov 1986 (page 148-149)


ATI © 1987
56.640MHz
V89A
32.514OMHz
25.175OMHz
V89A
EGA
WONDER 800
WONDER480
(C)1988
Technology you can Trust.
ASSY NO 104001500-0
1E904003927


PARADISE
PEGA1A
38304 8841
015567190502
Paradise Basic EGA Card
ASSY 28250 REV 1
FCC ID:DBM8FL24LMYRP AEGA1A
MADE IN HONG KONG
R2TP 553645

0.3.4 EGA Architecture

EGA can be summarized as three major systems made of input, storage, and output:

- The Graphic Controller and Sequence Controller controlling how EGA RAM is accessed (the CPU-VRAM interface)
- The framebuffer (the VRAM) made of four memory banks with 64KiB (rather than one bank of 256KiB).
- The CRT Controller and the Attribute Controller taking care of converting the palette-indexed framebuffer to RGB and then to digital TTL[16] signal for display

Trivia : In the 1980's integrated video DACs[17] were expensive and difficult to embed into custom chips. Most home computers with RGB output used TTL for digital output. With the introduction of VGA the DAC became the standard.

The most surprising part of the architecture is obviously the framebuffer. Why have four small fragmented banks instead of one big linear one?

The main reason was RAM latency and the need for minimum bandwidth. A CRT running at 60Hz and displaying 640x350 in 16 colors needs a pixel every $\frac{1}{640*350*60} = 74$ nanosecond. At this resolution, one pixel is encoded with 4 bits. Each nibble is translated to a RGB color via the TTL. So that means it requires one byte every 148 nano-seconds.

Unfortunately, RAM access latency was 200ns - not nearly fast enough[18] to refresh the screen at 60hz, so the TTL would starve. If latency could not be reduced, the throughput could still be improved by reading from four banks at a time. Reading in parallel gave an amortized RAM latency of 200/4 = 50ns, which was fast enough.

Keep in mind that this architecture reduced the penalty of read operations, but plotting a pixel in the framebuffer with a write operation was still slow. Writing to the VRAM as little as possible was crucial to maintaining a decent framerate.

[16] Transistor Transistor Logic
[17] Digital to Analog Converter
[18] Computer Graphics: Principles and Practice 2nd Edition, page 168.

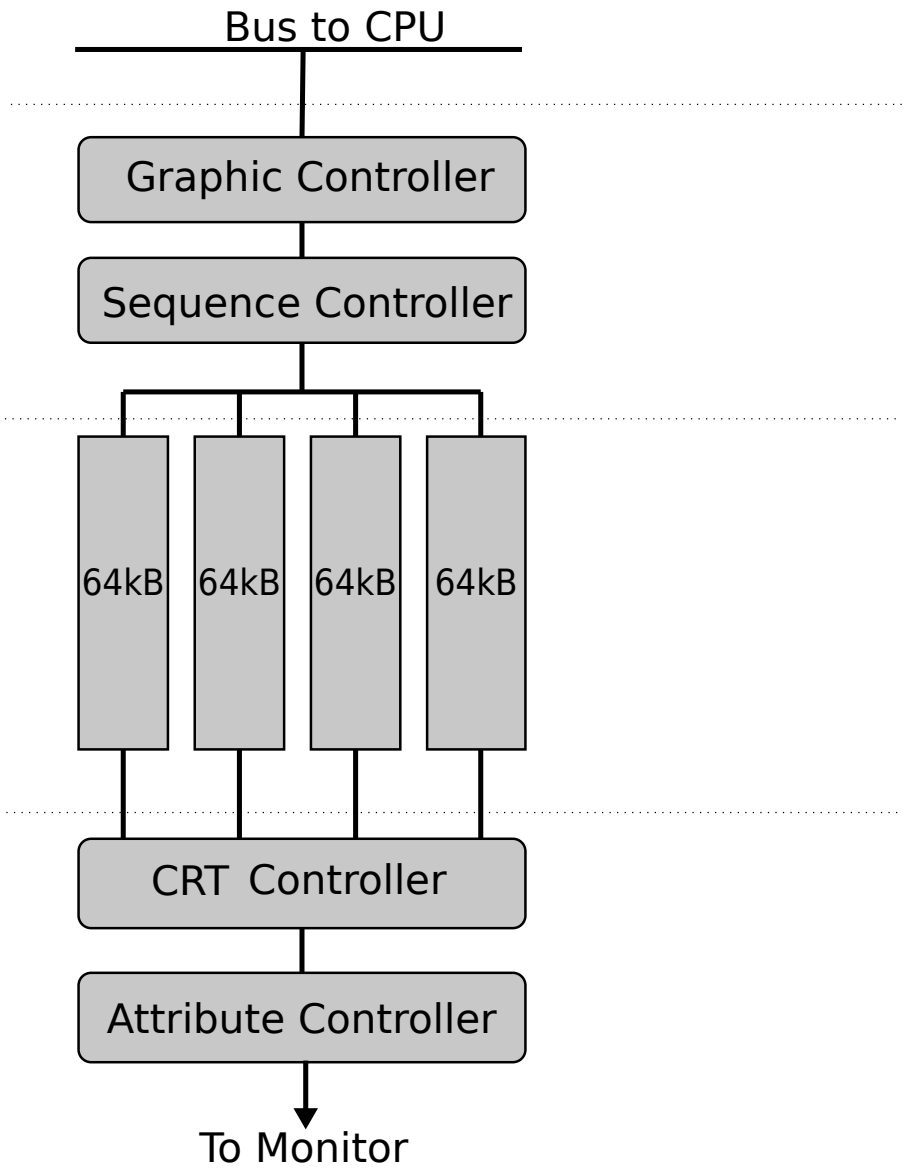


Figure 19: EGA Architecture.

0.3.5 EGA Planar Madness

Four memory banks grant enough throughput to reach high resolutions at 60Hz. This type of architecture is called "planar". Each plane is like a black-and-white image that stores

information about a single color. For EGA there are 4 planes, where combining one bit from each plane results in a color index. This color index is then via the attribute controller translated into a color to the screen. This layout is better explained with a drawing.

To write the color of the first pixel, a developer has to write the first bit of the first byte in plane 0, the second in plane 1, the third in plane 2 and the fourth in plane 3. The CRT Controller then reads 4 bytes at a time (one from each plane) and converts them via the Attribute Controller into 8 pixels on screen.

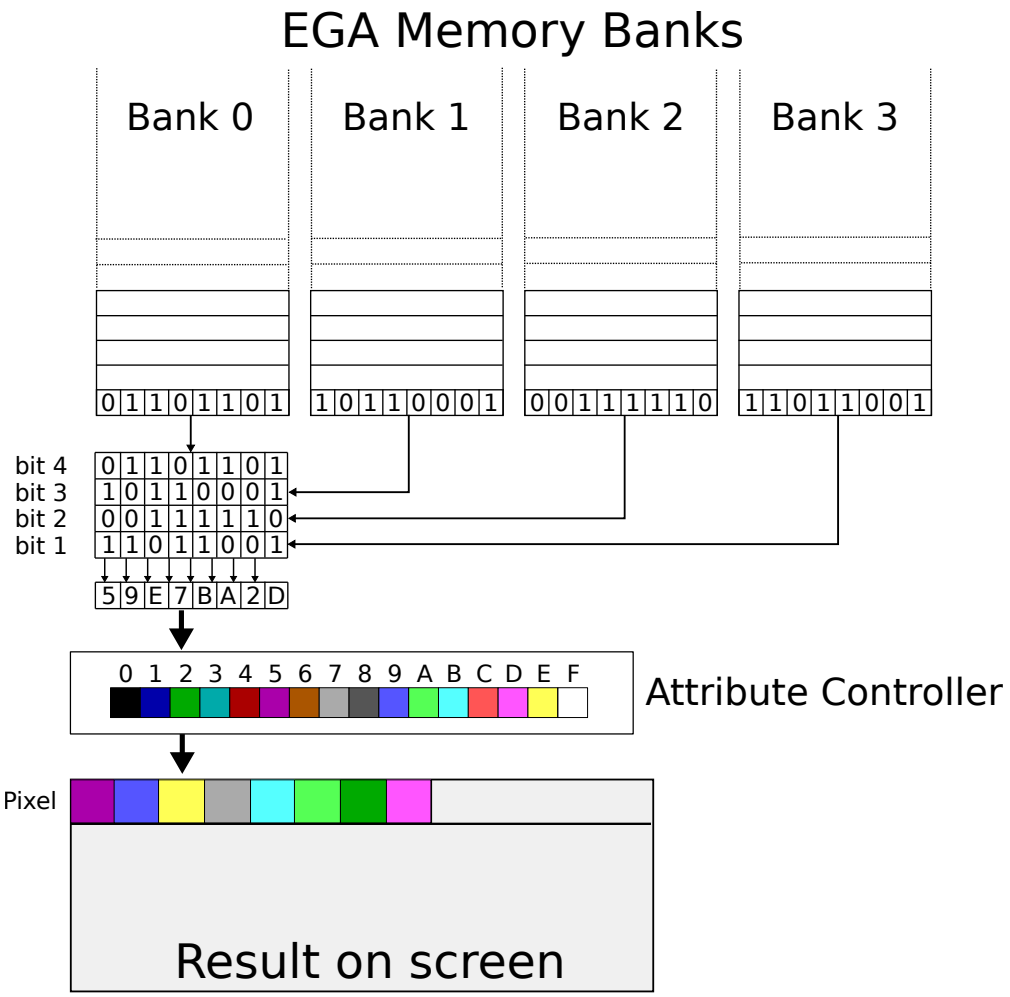


Figure 20: EGA mode 0Dh, how bank layout appears on screen.

0.3.6 EGA Modes

In order to configure this mess of planes and the controllers, 50 poorly documented internal registers must be set. Needless to say few programmers dove into the internals of the EGA.

Figure 19, which described the architecture, was actually deceptively simplified. Figure 21 shows how IBM's reference documentation explained the EGA. The maze of wire showcases well the actual complexity of the system.

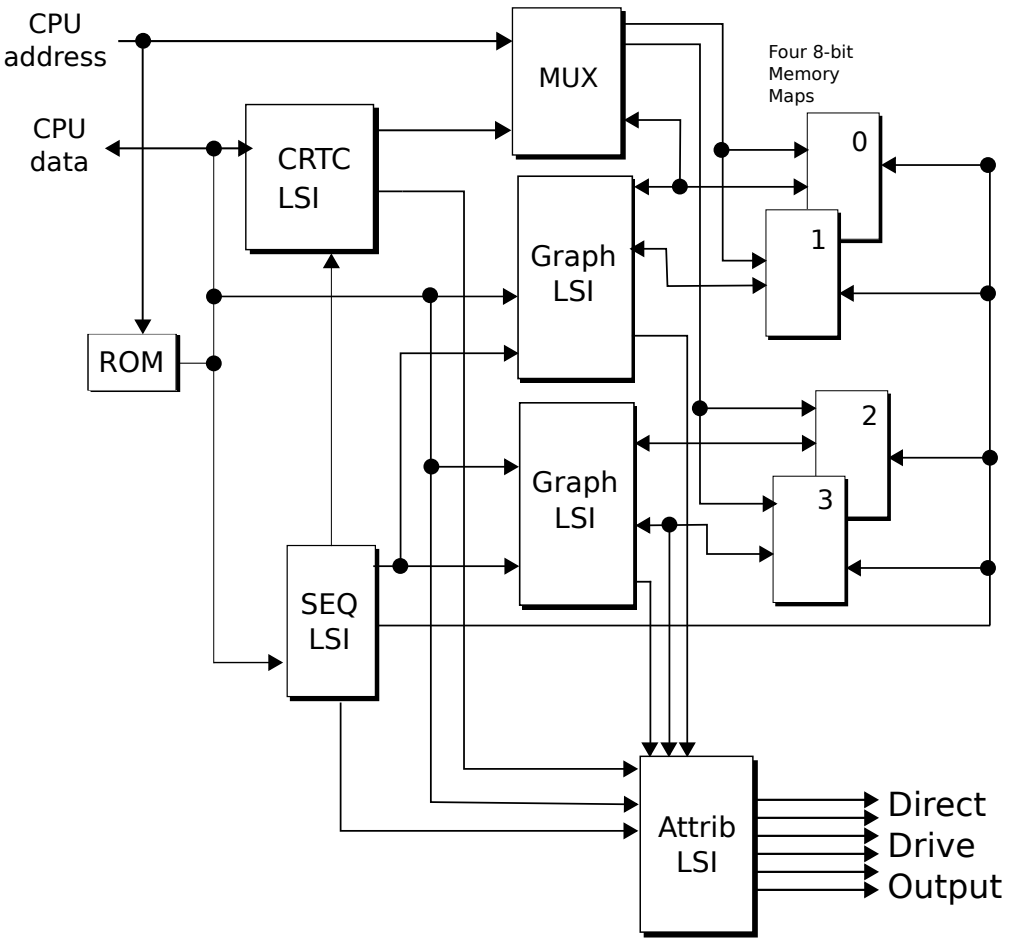


Figure 21: IBM's EGA Documentation.

To compensate for the complexity, IBM provided a routine to initialize all the registers via one BIOS call. One mode can be selected out of 12 available with an associated resolution, number of colors, and memory layout. The BIOS can be called to configure the EGA as follows:

Mode	Type	Format	Colors	RAM Mapping	Hz
00h	text	40x25	16 (monochrome)	B8000h	60
01h	text	40x25	16	B8000h	60
02h	text	80x25	16 (monochrome)	B8000h	60
03h	text	80x25	16	B8000h	60
04h	CGA Graphics	320x200	4	B8000h	60
05h	CGA Graphics	320x200	4 (monochrome)	B8000h	60
06h	CGA Graphics	640x200	2	B8000h	60
07h	MDA text	9x14	3 (monochrome)	B0000h	60
0Dh	EGA graphic	320x200	16	A0000h	60
0Eh	EGA graphic	640x200	16	A0000h	60
0Fh	EGA graphic	640x350	3	A0000h	60
10h	EGA graphic	640x350	16	A0000h	60

Figure 22: EGA Modes available.

Trivia : The Modes `08h-0Ah` are reserved for PCjr (or Tandy Graphics Adapter) graphics modes, which offered 160x200 with 16 colors, 320x200 with 16 colors and 640x200 with 4 colors. Modes `0Bh` and `0Ch` are reserved for internal EGA BIOS.

To setup the EGA in Mode `0Dh` using the BIOS is incredibly easy. It can be done with only two instructions:

```
_AX = 0xd ; AH=0 (Change video mode), AL=0Dh (Mode)
geninterrupt (0x10) ; Generate Video BIOS interrupt
```

The `geninterrupt (0x10)` instruction is a software interrupt caught by the BIOS routine in charge of graphic setup. It looks up the `ax` register, which can be set in the Borland Compiler by `_AX`, to setup all EGA registers with the corresponding mode.

0.3.7 EGA Programming: Memory Mapping

To write to the VRAM, the RAM's 1MiB address space maps 64KiB starting as indicated in figure 22. In mode `0Dh` for example, the VRAM is mapped from `A0000h` to `AFFFFh`. One of the first questions to come to mind is "How can I access 256KiB of RAM with only 64KiB

of address space?" The answer is "bank switching" as summarized in figure 23. Write and Read operations are routed based on a mask register indicating which bank should be read or written to.

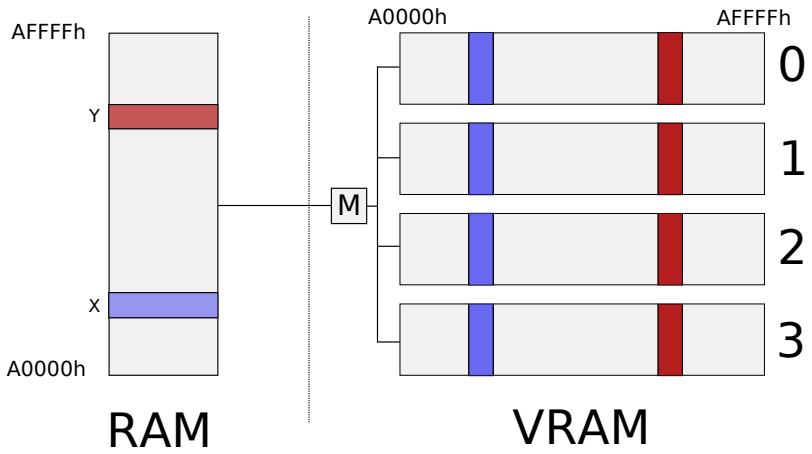


Figure 23: Mapping PC RAM to EGA VRAM banks.

0.3.8 EGA Color Palette

The EGA CRTC does not expect RGB values to generate pixels. Instead it is based on an index-based color palette system. Each pixel is a 4-bit index number, assigned to a color from the Attribute Controller. The default color palette are all 16 CGA colors, but it allows substitution of each of these colors with any one from a total of 64 colors.

When calculating the intended value in the 64-color EGA palette, the 6-bit number of the intended entry is of the form "rgbRGB" where a lowercase letter is the least significant bit of the channel intensity ($\frac{1}{3}$ color intensity) and an uppercase letter is the most significant bit of intensity ($\frac{2}{3}$ color intensity). The more intensity, the brighter the color is. For example, `02h` will produce green, `10h` will produce dim green and `12h` will produce bright green. Each of the 16 color indexes could be reassigned to one color from the "rgbRGB" palette.

	00h	01h	02h	03h	04h	05h	06h	07h	08h	09h	0Ah	0Bh	0Ch	0Dh	0Eh	0Fh
00h																
10h																
20h																
30h																

Figure 24: EGA "rgbRGB" color palette (64 values from 00h to 3Fh)

However, standard EGA monitors did not support use of the extended color palette in 200-line modes. Both CGA and EGA cards use a female nine-pin D-subminiature (DE-9) connector for output.

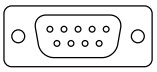


Figure 25: DE-9 video output connector.

The EGA monitor could only distinct EGA and CGA cards based on the Vertical Sync signal, which is either 200- or 350-line mode. If the Vertical Sync is 350-line mode, the monitor switched to Mode 2 operations which supported the extended rgbRGB-color information[19]. But in the 200-line mode, the monitor cannot distinguish between being connected to a CGA or an EGA card.

The CGA color output is based on the form "RGBI", where the 'I' stands for Intensity and adds brightness to the RGB color. Compared to CGA, EGA redefines some pins of the DE-9 connector to carry the extended rgbRGB-color information. If the monitor were connected to a CGA card, these pins would not carry valid color information, and the screen might be garbled if the monitor were to interpret them as such.

Pin	**Mode 2: EGA mode (rgbRGB)**	**Mode 1: CGA mode (RGBI)**
1	Ground	Shield Ground
2	Secondary Red (Intensity)	Signal Ground
3	Primary Red	Red
4	Primary Green	Green
5	Primary Blue	Blue
6	Secondary Green (Intensity)	Intensity
7	Secondary Blue (Intensity)	Reserved
8	Horizontal Sync	Horizontal Sync
9	Vertical Sync	Vertical Sync

Figure 26: EGA and CGA DE-9 connector pin signals.

[19] IBM Enhanced Color Display documentation.

Suppose one assigns the color brown (rgbRGB is `010100b`) to one of the color indexes, the resulting color on the CGA pin assignment is light red; The secondary green pin ("r" in rgbRGB) is mapped to the Intensity pin in CGA mode, which results to the color red with intensity and not the expected brown color.

For this reason, EGA monitors will use the CGA pin assignment (mode 1) in 200-line modes so the monitor can also be used with a CGA card and vice versa. Therefore, the EGA card is fully backwards compatible with a standard CGA monitor. Thereby it is able to show all 16 CGA (RGBI-)colors simultaneously, instead of only 4 colors when using a CGA card.

Index Number	Color	rgbRGB	RGBI
00h	Black	000000b	0000b
01h	Blue	000001b	0010b
02h	Green	000010b	0100b
03h	Cyan	000011b	0110b
04h	Red	000100b	1000b
05h	Magenta	000101b	1010b
06h	Brown	010100b	1100b
07h	Light grey	000111b	1110b
08h	Dark grey	111000b	0001b
09h	Bright blue	111001b	0011b
0Ah	Bright green	111010b	0101b
0Bh	Bright cyan	111011b	0111b
0Ch	Bright red	111100b	1001b
0Dh	Bright magenta	111101b	1011b
0Eh	Yellow	111110b	1101b
0Fh	White	111111b	1111b

Figure 27: Default EGA 16-color palette

0.3.9 Double-Buffering in EGA

Fundamentally, a core goal in rendering is for each frame displayed on the monitor to present a single, coherent image.

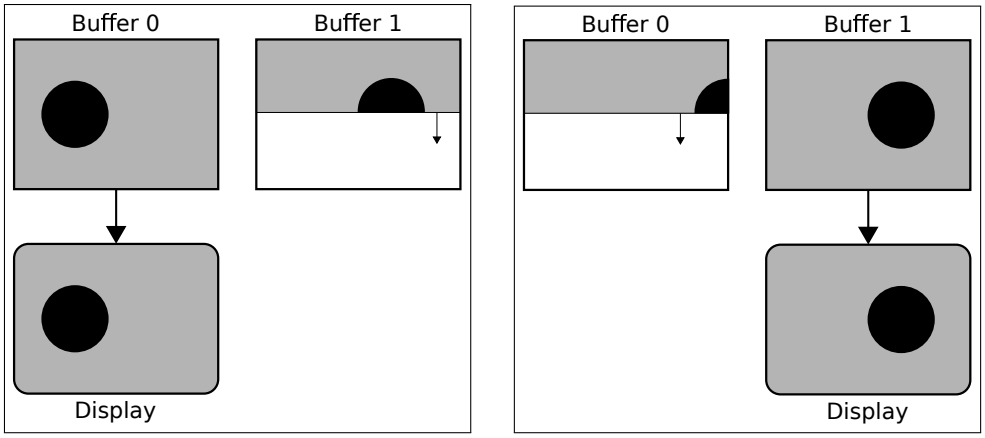
On the oldest hardware, there often wasn't enough memory to hold a full screen image, and so instead of drawing a screen image to VRAM, you needed to specify colors for each scanline individually, while the monitor was in the process of drawing that line. For example, on the Atari 2600, developers had just 76 machine instruction cycles to determine the color of each pixel in a scanline before the television started rendering it. This process repeated for every subsequent scanline until the entire frame was drawn.

Unlike early consoles, PC video cards contain dedicated VRAM, which the monitor reads from 60 times per second. Writing to VRAM, however, is controlled by the CPU. To avoid graphical artifacts, rendering must be timed carefully. If drawing a new frame takes too long, the scanline will "lap" the drawing process, resulting in a partially drawn frame. Conversely, if drawing gets ahead of the scanline, the new frame may overwrite part of the previous frame before it is fully displayed. This results in "tearing", where the top half of the screen displays one frame while the bottom half shows another.

Avoiding tearing with a single buffer is nearly impossible because writing to VRAM is significantly slower than reading from it. The scanline inevitably overtakes the rendering process, causing tearing.

Trivia : The original IBM CGA card could not simultaneously read from and write to VRAM. Because the video memory was not dual-ported, when both the CPU and the video card needed access to the same byte of VRAM, the CPU took priority. This resulted in the card reading a random value, causing a visual artifact known as "snow" on the screen. Later CGA clones could simultaneous read/write access to VRAM and did not have the same issue anymore.

A solution to tearing is double buffering, which introduces a second framebuffer in video memory. With two buffers, the CPU can draw the next frame in an off-screen framebuffer while the current one is being scanned to the display. Once drawing is complete, the buffers are swapped, ensuring a seamless transition between frames without tearing.



(a) Display buffer 0, rendering buffer 1.

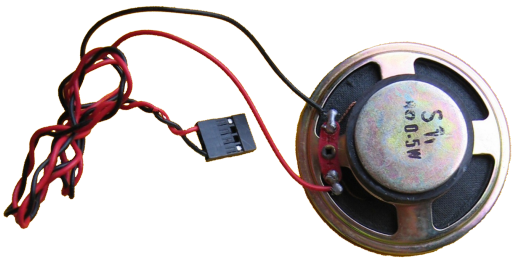
(b) Display buffer 1, rendering buffer 0.

Figure 28: Double-Buffering.

A full-screen image at 320 x 200 pixels with 16 colors requires 320 x 200 x 4 bits = 32 KB of VRAM (spread over four planes, each 8 KB). Given that EGA cards typically have 256 KB of VRAM, there is more than enough memory to support multiple buffers.

0.4 Audio

For the first 5-6 years of the IBM PC and its compatibles, their audio output came from nothing more than a simple loudspeaker with a tone generator. For business, this was acceptable - even preferable, since a PC in an office environment really shouldn't be a distraction to others! The loudspeaker, commonly known as a "PC Speaker", was capable of generating a square wave via 2 levels of output.



0.4.1 History of Sound Cards

The introduction of real game music and sounds on the PC started with Sierra back in 1988. They prepared to change all this by creating games that contained serious, high quality musical compositions drawing on add-on hardware. *King's Quest IV* was the first commercially released game for IBM PC compatibles to support sound cards. In addition to the familiar PC speaker and Tandy 1000, it could utilize

- Roland MT-32
- IBM Music Feature Card
- AdLib

Sierra struck a deal with two companies, Roland and AdLib, where Sierra would also become a reseller for these soundcards.

The Roland MT-32 was the higher end of these music devices. In today's terminology, it would be labeled a "Wavetable Synthesizer". A wavetable synthesizer usually implies that real instrument sounds are recorded into the hardware of the device. This device can then manipulate them to play them back at the various notes you need. The MT-32 had the ability to manipulate parts of its built in sounds using something called "Linear Arithmetic (LA)" synthesis. It was a very good device that can rival even today's sound cards. To connect the MT-32 to a PC required, what Roland called an MPU-401[20], in one of the PC's expansion slots. Sierra sold The MT-32 with a necessary MPU-401 interface for $550. The high price prevented it from dominating the end-user market of gaming.

Figure 29: Roland MT-32 synthesizer box.

The IBM Music Feature Card was launched in March 1987 as a collaboration between IBM and Yamaha. Essentially the Music Feature Card was a synthesizer installed on an 8-bit

[20]Midi Processing Unit-401

expansion card[21]. The Music Feature Card had 8 FM voices, controllable via 4 frequency operators. It came with over 300 high-quality synthesized instruments on-board, and it was actually possible to have two Music Feature Cards in a single PC to get 16 voices. With a tag price of $495 it was just like the Roland MT-32 an expensive card, and its audience was primarily business users.

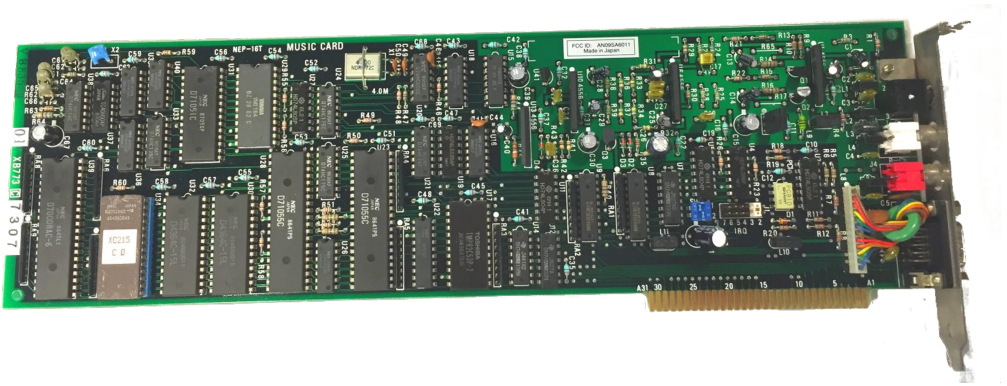


Figure 30: IBM Music Feature Card.

0.4.2 AdLib

AdLib was the other company, beside Roland, that struck a deal with Sierra as a reseller. The company was founded in 1987 by Martin Prevel, a former professor of music from Quebec. The AdLib soundcard used a technology called FM synthesis. The technology is based on the idea of generating superimposing waveforms to create a sound. This technology was much less expensive than Roland's Wavetable technology.

The AdLib card was built around the Yamaha YM3812, also known as the OPL2 chip, and could produce either 9 sound channels or 6 sound channels plus 5 hit instruments simultaneously using Frequency Modulation (FM). Ideally, if you have enough generators and can fine tune the waveforms well enough, you can create a realistic sound. However, to reach this ideal, you need lots of skilled people, lots of money for equipment, and lots of time to develop. Thus, FM synthesis sounded very artificial. Still, this was a great improvement over the PC Speaker. With a price tag of $219.99, it was much cheaper than the Roland MT-32 and IBM Music Feature Card, and soon ruled at the top of the early PC sound card market.

[21] Roland released the LAPC-I in 1989, which basically was similar to the Music Feature Card: a MT-32-compatible Roland sythesizer with a MPU-401 unit, integrated onto one single full-length 8-bit ISA card.

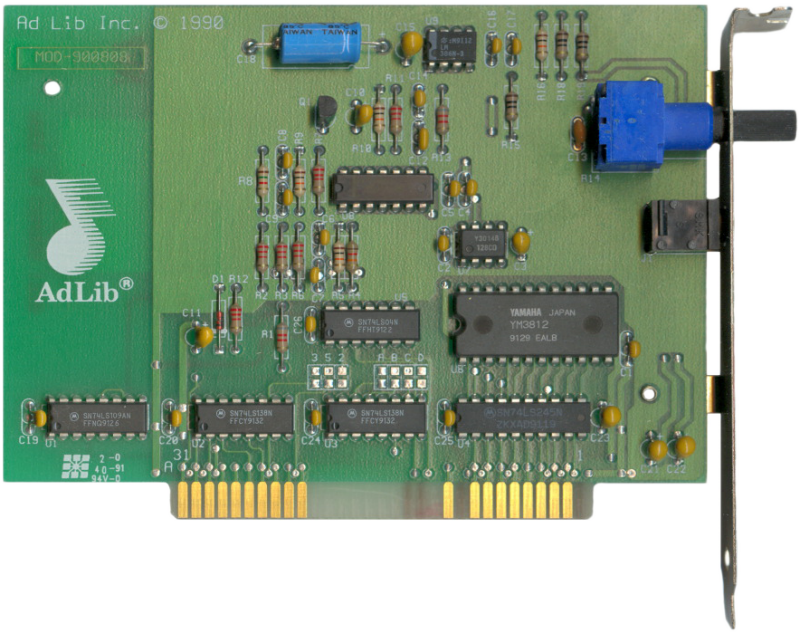


Figure 31: An AdLib sound card. Notice the big YM3812 chip.

The AdLib card dominated the PC market for almost three years. In 1989, Creative Labs released the competing SoundBlaster, which quickly dominated over AdLib. To compete with SoundBlaster, AdLib planned a new 12-bit stereo sound card called AdLib Gold. Sadly, due to AdLib's dependence on Yamaha who suffered long delays introducing their latest multimedia chipset, their new product, AdLib Gold PC-1000, was never to see the light of day under AdLib's management. Unable to remain solvent, AdLib closed its doors on 1st May 1992.

0.4.3 SoundBlaster

Creative Labs, the company behind the SoundBlaster, didn't enter the sound card industry until 1987 with the introduction of their Creative Music System (C/MS). From inception in 1981, they were a computer repair shop in Singapore. Creative Music System, or "Game Blaster" as it was renamed a year later, was a FM synthesizer card similar to AdLib. Based on the Philips SAA1099 chip, which was essentially a square-wave generator, it sounded much like twelve simultaneous PC speakers would have, except for each channel having amplitude control. The card did not sell well.

The original Sound Blaster v1.0 and v1.5 were released in 1989 as the successor to their

Game Blaster. Not only was it equipped with the OPL2 chip, providing 100% compatibility with AdLib music playback, but it was also technologically superior with a DSP[22] allowing PCM playback (digitized sounds) at 8 bits per sample and up to 22.05kHz sampling rate. The card also came with a DA-15 port allowing joystick connection. Most importantly, the SoundBlaster was $90 cheaper than the AdLib.

Figure 32: A SoundBlaster 1.5, model CT1320B

Figure 32 is the SoundBlaster model CT1320B. Notice the OPL2 chip (labeled FM1312) and the CT1321 DSP on the middle top. In the middle of the card are the two CMS-301 chips, to ensure backwards compatibility with the Creative Music System. The CT1320B model (Sound Blaster 1.5) was a cost-cutting measure. Having recognized that C/MS was unpopular, they replaced the two C/MS chips with sockets. You could still purchase the C/MS chips for $29.95 if you wished and install them into these sockets.

Trivia : Creative Labs boasts in their own advert AdLib compatibility and even uses an image of AdLib Inc's product[23]! On the other hand they sued every company that tried to market a 'Sound Blaster compatible' product for using the name of their product.

[22] An Intel MCS-51 "Digital Sound Processor", not "Digital Signal Processor".
[23] Advert p20 in Compute!, April 1990.

Figure 33: Sound Blaster advertisement with Adlib compatibility.

0.5 Floppy Disk Drive

In the time before the internet, a floppy disk was the main medium to share and distribute software and data. The original XT systems were equipped with 5¼-inch floppy disk with a capacity of 360Kb. In 1984, IBM introduced with its PC AT the 1.2 MB dual-sided 5¼-inch floppy disk, but it never became very popular. IBM started using the 720 KB double density 3½-inch floppy disk in 1986 and the 1.44 MB high-density version in 1987. The advantages of the 3½-inch disk were its higher capacity, its smaller physical size, and its rigid case which provided better protection from dirt and other environmental risks. By the mid-1990s, 5¼-inch drives had virtually disappeared, as the 3½-inch disk became the predominant floppy disk.

Trivia : An USB stick of 128GB contains more than 91K high-density 3½-inch (1.44MB) floppy disks.

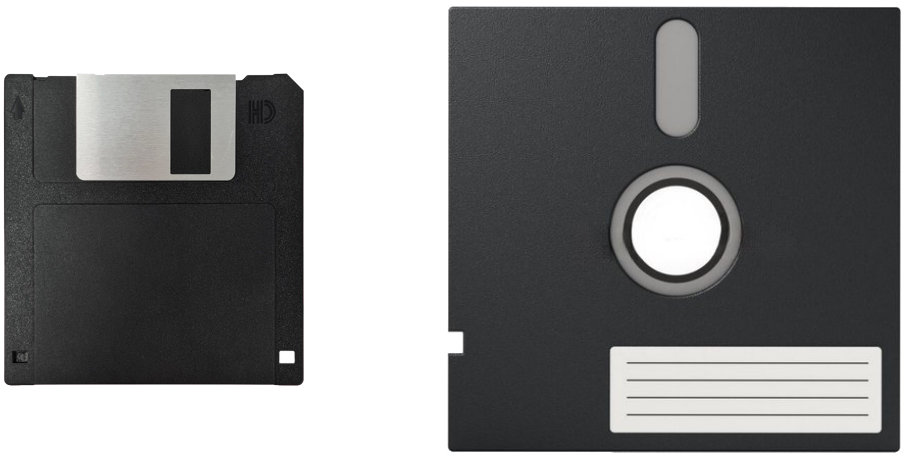


Figure 34: 3½-inch and 5¼-inch floppy disk.

A floppy disk is essentially a very flexible piece (hence the term floppy disk) of plastic coated on both sides in a magnetic material. This "disk" of plastic is contained within a protective envelope or hard plastic case, which is then inserted into the drive and automatically locked onto a spindle. It is then rotated at a constant speed, 360 rpm for standard PC floppy drives. A head assembly consisting of two magnetic read/write heads, one in contact with the upper surface and one in contact with the lower surface of the disk, may be moved in discrete steps across the disk and read the data from the disk.

The data on a floppy disk is stored in concentric circular tracks divided into arc-shaped sectors. The amount of data a disk can store is determined by the number of tracks and

sectors and the density of the recorded information (single and double density). Near the center, there is a small hole called the index hole, which marks the start of a sector.

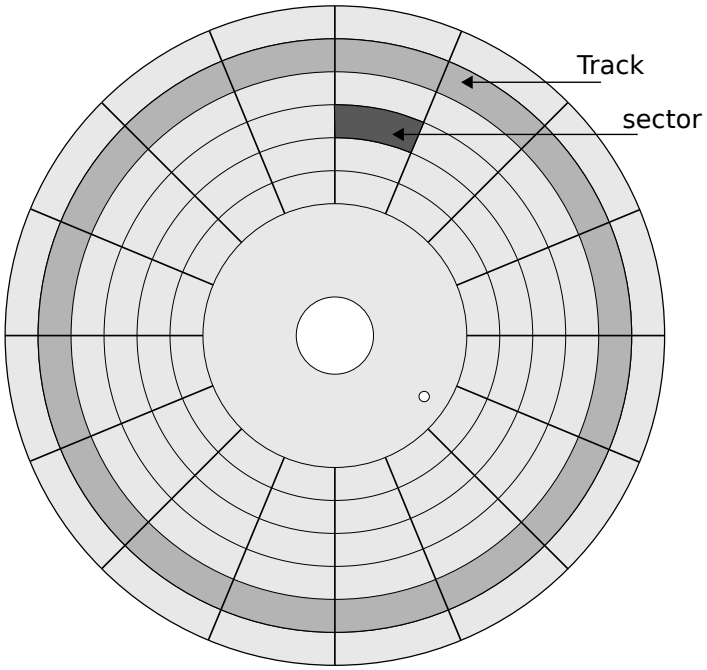


Figure 35: Floppy disk with tracks, sectors and the index hole.

When the floppy disk drive is powered up, the read/write heads move to the target track, starting from track 0 (the starting track on a floppy disk). Once the sensor reaches the target track, the computer is ready to retrieve or write data onto the floppy disk. To select a specific sector, the drive must wait for that sector to pass under the head. The drive determines which sector it is on by waiting for the index hole to be under the head. Since the rotation speed is kept constant, the time when the next sector is under the head is known. Now, the head can read or write to the specific sector/track combination on the disk.

The floppy disk is controlled via the Floppy Disk Controller (FDC), and a typical read operation from the floppy disk contains the following steps:

- Turn the disk motor on. When you turn a floppy drive motor on, it takes quite a few milliseconds to "spin up", to reach the (stabilized) speed needed for data transfer.
- Perform seek operation, which moves the head to the correct location for reading the data.

- Read the data from the floppy disk and store the data via the FDC to RAM memory.
- Turn the disk motor off.

The controller waited a few seconds before turning off the motor. The reason to leave the motor on for a few seconds is that the controller may not know if there is a queue of sector reads or writes that are going to be executed next. If there are going to be more drive accesses immediately, they won't need to wait for the motor to spin up again.

0.6 Keyboard

The original IBM PC and XT keyboards featured 83 keys. After receiving feedback from users frustrated by the layout, IBM introduced the 84-key PC AT keyboard, which included a rearranged layout and the addition of the "Sys Req" key. It also introduced 3 status LEDs for Caps Lock, Num Lock and Scroll Lock.

Figure 36: IBM AT Keyboard.

This design was further updated in 1986 with the release of the IBM Model M. It officially became the IBM PC standard in 1987 with the introduction of the IBM Personal System/2 (PS/2). The function keys were moved to the top, F11 and F12 were added, and the total number of keys increased to 101. The Model M was widely adopted and are still being used today.

Figure 37: IBM Keyboard model M (PS/2).

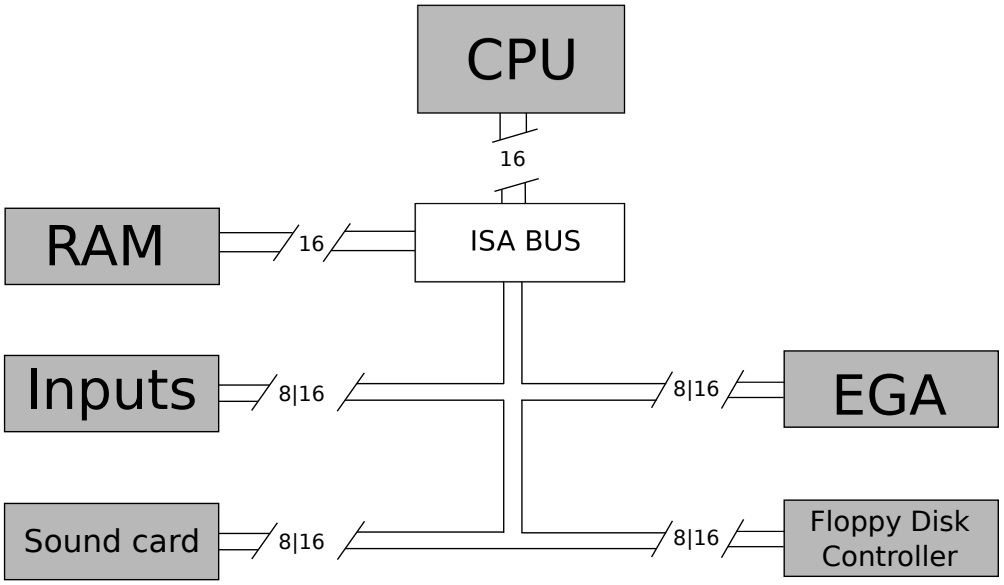
At the time, keyboards were connected either via PS/2 or AT ports.

Figure 38: PS/2 (left) and AT (right) keyboard connector.

0.7 Bus

Although developers had no control over them, it is still worth mentioning how these components were connected to each other.

The ISA[24] bus connects the CPU to all devices, including RAM. It was almost 10 years old in 1990 but still used universally in PCs. The data path to the RAM is 16 bits wide for 286 machines. It runs at the same frequency as the CPU.



The rest of the bus connecting to everything that is not the RAM can be either:

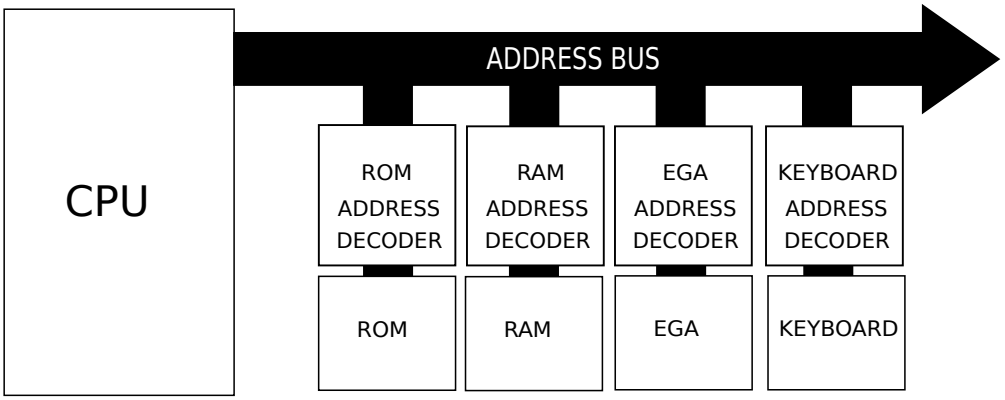
- 8 bits wide at 4.77 MHz for 19.1 Mbit/s
- 16 bits wide at 8.33MHz for 66.7 Mbit/s[25].

It is also backward compatible and an 8-bit ISA card can be plugged into a 16-bit ISA bus.

Trivia : On ISA all devices are connected to the bus at all times and listen on the bus address lane. Each device features an "address decoder" to detect if it should reply to a bus request. This is how the EGA RAM is "mapped" in RAM. The EGA card "address decoder" filters out everything that is not within `A0000h` and `AFFFFh`. Accordingly, the RAM

[24] Industry Standard Architecture.
[25] https://en.wikipedia.org/wiki/List_of_device_bit_rates .

disregards any request that is within the range [A0000h - AFFFFh].



0.8 Summary

To say a PC was difficult to program for games would be an understatement. It was a nightmare. The CPU was good at doing the wrong thing, the best graphic interface didn't allow double buffering, and the memory model only allowed 1 MiB with an address composed of two separate 16-bit registers. Last, but not least, the default sound system could only produce square waves.

Yet despite all these unfavorable conditions, teams of developers gathered to tame the beast and unleash its power to gamers. One of these called themselves *Ideas From the Deep*.