Arquitetura de Hardware

Capítulos 2.2 e 2.3 — RAM e Vídeo

Integrantes do Grupo

Felipe Mavigno Stein

Kaiky Viglioni Tavares Moura

Marco Antonio Faustini Pessoa

Nícolas Medeiros de Pinho

Sérgio Alexandre Gobbi Rebello

7 de abril de 2026

Sumário

1 RAM

Os primeiros processadores da família Intel x86 foram projetados em 1976. Em uma época em que a RAM era extremamente cara, os modelos 8080 e 8086 possuíam registradores de 16 bits com um barramento de endereços de 20 bits, capaz de endereçar 1 MiB de RAM. É difícil dimensionar o quanto 1 MiB de RAM representava nos anos 70; como referência, o Apple II e o Commodore 64 foram lançados com apenas 64 KiB — memória suficiente para criar e executar aplicações notáveis. Registradores de 16 bits e um barramento de 20 bits eram mais do que adequados, ainda que a programação fosse trabalhosa e exigisse a combinação de dois registradores para construir um ponteiro.

Em 1986, com o barateamento do hardware, a Intel rompeu com a arquitetura anterior ao lançar os modelos 286 e 386. Essas CPUs podiam operar no chamado *modo protegido*, que apresentava um barramento de endereços de 24 bits para até 16 MiB de RAM plana, protegida por uma MMU[1]. O 386 também trouxe registradores de 32 bits no modo protegido. Para garantir retrocompatibilidade, ambos os processadores podiam operar no *modo real*, que replicava o comportamento do 8080 e do 8086: registradores de 16 bits e barramento de 20 bits, resultando em 1 MiB de RAM endereçável com endereçamento segmentado.

Por razões de compatibilidade, todos os PCs precisam inicializar em modo real. Seria razoável supor que os programadores dos anos 90 migrariam imediatamente para o modo protegido para aproveitar todo o potencial das máquinas e abandonar o modo real de duas décadas. Porém, havia um obstáculo significativo: o sistema operacional MS-DOS, da Microsoft.

1.1 Limitações do DOS

A Microsoft prezava muito pela manutenção das aplicações em execução em seus sistemas operacionais. Como prioridade de negócio, a empresa era inflexível quanto à preservação da compatibilidade em cada nova versão do sistema. Como muitas aplicações foram escritas durante os anos 80 em máquinas que operavam apenas em modo real, o DOS 5.0 continuou funcionando dessa forma — e, consequentemente, suas rotinas e chamadas de sistema eram incompatíveis com o modo protegido.

Isso criava uma situação peculiar: o sistema operacional de fato entregue com cada máquina vendida impedia os programadores de utilizar o hardware em sua plena capacidade. Os desenvolvedores eram obrigados a ignorar todos os recursos de uma CPU de 1992 e utilizá-la como se fosse um Intel 8086 de 1976, extremamente acelerado. As restrições impostas eram:

- **ULA** (Unidade Lógica e Aritmética)

[1] Memory Management Unit — Unidade de Gerenciamento de Memória.

- **16 registradores:**
  - Registradores de uso geral de 16 bits: AX, BX, CX, DX
  - Registradores de índice de 16 bits: SI, DI, BP, SP
  - Contador de programa de 16 bits: IP
  - Registradores de segmento de 16 bits: CS, DS, ES, FS, GS, SS
  - Registrador de status de 16 bits
- Até 1 MiB de RAM

Curiosidade: Um ano antes, em 1991, um estudante da Universidade de Helsinki começou a trabalhar em um projeto pessoal — "nada grandioso", segundo ele mesmo — que, ao contrário do DOS, era capaz de utilizar a CPU em modo protegido e aproveitar a MMU e os registradores de 32 bits. Tornaria-se o pior pesadelo da Microsoft. Linus Torvalds acabava de iniciar o que viria a se tornar o Linux.

1.2 O Infame Modo Real: Limite de 1 MiB de RAM

Sem acesso ao modo protegido, os desenvolvedores de 1991 programavam como se estivessem em 1976: com um barramento de endereços de 20 bits que oferecia apenas 1 MiB de RAM endereçável. Independentemente da quantidade de memória instalada na máquina, apenas 1 MiB podia ser endereçado. Para agravar a situação, o endereçamento não podia utilizar os registradores de 32 bits disponíveis — era necessário combinar dois registradores de 16 bits. Um deles representava o segmento e o outro, o deslocamento dentro desse segmento. Daí o nome: *programação segmentada de 16 bits*.

O layout da memória era o seguinte:

- `00000h` a `003FFh`: Tabela de Vetores de Interrupção
- `00400h` a `004FFh`: Dados do BIOS
- `00500h` a `005FFh`: `command.com` + `io.sys`
- `00600h` a `9FFFFh`: Disponível para programas (cerca de 620 KiB no melhor caso)
- `A0000h` a `FFFFFh`: UMA (*Upper Memory Area*) — reservada para ROM do BIOS, placa de vídeo e E/S da placa de som

Dos 1024 KiB originais, apenas 640 KiB (denominados *Conventional Memory*) estavam acessíveis a um programa. Os 384 KiB restantes eram reservados para a UMA, e cada driver instalado (`.SYS` e `.COM`) reduzia ainda mais os 640 KiB disponíveis.

Curiosidade: Na França, os usuários precisavam carregar o driver `KEYBFR.SYS` para que as teclas do teclado AZERTY fossem mapeadas corretamente. O driver consumia 5 KiB da memória convencional. Desnecessário dizer que os franceses logo descobriram que o modo *god mode* era `IDDAD`[2].

1.3 O Infame Modo Real: Endereçamento Segmentado de 16 Bits

Com um barramento de endereços de 20 bits e registradores pequenos demais para conter um endereço completo (largura de 16 bits), a Intel precisou criar um sistema de endereçamento. A solução foi combinar dois registradores de 16 bits: um para designar o segmento e outro para o deslocamento dentro desse segmento.

Existem dois tipos de ponteiros: *near* e *far*. Um ponteiro *near* tem 16 bits e é considerado rápido porque pode ser utilizado diretamente, mas só permite saltos dentro do segmento de código atual. Um ponteiro *far* pode acessar qualquer endereço e permite saltos para qualquer lugar, porém é mais lento, pois o registrador de segmento de 16 bits precisa ser deslocado 4 bits à esquerda e combinado com o registrador de deslocamento de 16 bits para formar um endereço de 20 bits.

Isso pode não parecer tão problemático, mas na prática o endereçamento segmentado causa diversas complicações. A menos grave envolve a linguagem. Como o C foi criado em máquinas com memória linear, os fabricantes de compiladores para PC precisaram estendê-lo, dando origem às palavras-chave `near` e `far`. A macro `MK_FP` construía esses ponteiros, enquanto `FP_SEG`/`FP_OFF` acessavam seus componentes individualmente. A `libc` também era "diferente": `malloc` retornava um ponteiro *near* e, portanto, só podia alocar até 64 KiB. Para obter mais de 64 KiB, era necessário usar `farmalloc`.

O problema maior é que dois ponteiros que referenciam o mesmo endereço podiam falhar em um teste de igualdade. Nesse modelo, o 1 MiB de RAM é dividido em 65536 parágrafos pelo ponteiro de segmento. Um parágrafo tem 16 bytes, mas um deslocamento pode ser de até 65536 bytes, gerando diversas sobreposições. O exemplo a seguir ilustra esse comportamento:

Listing 1: Ponteiros far apontando para o mesmo endereço, mas falhando na comparação

```
#include <stdio.h>
#include <dos.h>

int main(int argc, char **argv) {
    void far *a = MK_FP(0x0000, 0x0120);
    void far *b = MK_FP(0x0010, 0x0020);
    void far *c = MK_FP(0x0012, 0x0000);
```

[2] O modo de invencibilidade no Doom é `IDDQD` em um teclado QWERTY, mas `IDDAD` em um teclado AZERTY sem o driver francês instalado.

```
8       printf("%d\n", a == b);
9       printf("%d\n", a == c);
10      printf("%d\n", b == c);
11  }
```

A saída do programa acima é `0`, `0`, `0` — mesmo que os três ponteiros apontem para o mesmo endereço de memória.

1.4 Memória Estendida

O barramento de 20 bits do modo real limitava a RAM endereçável a 1 MiB. As máquinas de 1992 eram equipadas com mais memória, tipicamente 2 MiB e em alguns casos até 4 MiB. Essa memória além do espaço endereçável era chamada de *Extended Memory* (Memória Estendida). A solução para acessar esses recursos era instalar drivers especializados.

O acesso à memória estendida não era padronizado. Os usuários podiam carregar um dos drivers fornecidos com o DOS:

- **EMS** (*Expanded Memory Specification*): `EMM386.EXE`
- **XMS** (*eXtended Memory Specification*): `HIMEM.SYS`

Ou podiam simplesmente não saber que precisavam instalar um driver, não carregar nada na inicialização e não utilizar nenhuma RAM além de 1 MiB. Esse último caso era um grande problema. Muitos clientes não conseguiam entender por que, apesar de terem instalado (e pago caro por) megabytes de RAM em suas máquinas, o jogo recém-adquirido se recusava a iniciar com a mensagem "Not enough memory". A id Software chegou a publicar uma nota explicativa junto com seus jogos.

1.4.1 API XMS

O driver XMS funciona de maneira semelhante ao `malloc`/`free` da libc e é o mais intuitivo para programadores. Ele permite manipular dados na memória estendida não endereçável por meio de operações como alocar, liberar, realocar e mover. O aspecto central do XMS é que a memória precisa ser copiada entre a RAM estendida e a memória convencional.

1.4.2 API EMS

O driver EMS abre uma janela além da RAM endereçável. A ideia é baseada em mapeamento de memória. O driver permite manipular quatro unidades de 16 KiB chamadas "páginas", por meio de uma área de 64 KiB denominada *Page Frame*. Sob demanda ao driver, uma página pode ser trocada para o *page frame* sem necessidade de cópia.

1.4.3 EMS vs. XMS

A abordagem de mapeamento do EMS era diversas vezes mais rápida do que a abordagem de cópia do XMS. Essa diferença de desempenho influenciou significativamente o gerenciamento de memória do *Wolfenstein 3D*.

1.4.4 Um sistema "impossível de amar"

Ao longo dos anos, diversas descrições marcantes foram cunhadas sobre essa arquitetura:

> "*The x86 is an architecture that is difficult to explain and impossible to love.*"
> — David Patterson & John Hennessy, *Computer Organization and Design*

> "*Software poison.*"
> — Steve Morris, Co-Arquiteto do Intel 8086

2 Vídeo

Os PCs eram conectados a monitores CRT: grandes, pesados, com diagonal pequena, baseados em tubo de raios catódicos e com superfície curva. A maioria tinha diagonal de 14 polegadas e proporção 4:3.

2.1 Histórico dos Adaptadores de Vídeo

O *Monochrome Display Adapter* (MDA) foi lançado em 1981 com o IBM PC 5150. Oferecia apenas duas cores, permitindo 80 colunas por 25 linhas de texto. Embora modesto, era padrão em todos os PCs. Muitos outros sistemas se seguiram ao longo dos anos, cada um preservando a retrocompatibilidade.

Tabela 1: Histórico das interfaces de vídeo

Nome	**Ano de Lançamento**
MDA (Monochrome Display Adapter)	1981
CGA (Color Graphics Adapter)	1981
EGA (Enhanced Graphics Adapter)	1985
VGA (Video Graphics Array)	1987

Cada iteração adicionou novos recursos, e em 1991 o sistema gráfico predominante era o VGA. Todas as placas de vídeo instaladas em PCs precisavam seguir o padrão definido pela IBM. A universalidade desse sistema era uma faca de dois gumes: enquanto os desenvolvedores tinham apenas um sistema gráfico para programar, não havia como escapar de suas limitações.

2.2 Arquitetura VGA

O VGA pode ser resumido em três sistemas principais: entrada, armazenamento e saída.

- **Controlador Gráfico e Controlador de Sequência**: controlam o acesso à RAM do VGA (interface CPU–VRAM)
- **Framebuffer (VRAM)**: composto por quatro bancos de memória de 64 KiB (em vez de um banco linear de 256 KiB)
- **Controlador CRT e DAC**[3]: responsável por converter o framebuffer indexado por paleta em RGB e depois em sinal analógico (interface VRAM–CRT)

A parte mais surpreendente da arquitetura é o framebuffer. Por que ter quatro bancos pequenos e fragmentados em vez de um único grande e linear? Parte da explicação está na retrocompatibilidade. O EGA, predecessor do VGA, possuía apenas 64 KiB de RAM, tornando fácil projetar um sistema retrocompatível que utilizasse apenas um banco de 64 KiB.

A razão mais relevante era a latência da RAM e a necessidade de largura de banda mínima. Um CRT rodando a 60 Hz e exibindo 640×480 em 16 cores precisava de um pixel a cada $\frac{1}{640\times480\times60}$ segundos. Nessa resolução, um pixel era codificado em 4 bits. Cada nibble era traduzido para uma cor RGB 6-6-6 pelo DAC, o que dividia a largura de banda por 4,5 — mas ainda exigia um byte a cada 108 ns. A latência de acesso à RAM era de 200 ns, insuficiente para atualizar a tela a 60 Hz. A solução foi ler dos quatro bancos em paralelo, resultando em uma latência amortizada de $200/4 = 50$ ns.

2.3 A Loucura Planar do VGA

> "*Right off the bat, I'd like to make one thing perfectly clear: The VGA is hard — sometimes very hard — to program for good performance.*"
> — Michael Abrash, *Graphics Programming Black Book*

O primeiro problema desse design é sua falta de intuitividade. Não há um framebuffer linear e determinar qual byte corresponde a qual pixel na tela é difícil. Essa arquitetura é chamada de *planar*. No modo 13h, em que um pixel é codificado em um byte, escrever quatro pixels consecutivos em uma linha exige gravar um byte em cada banco — todos mapeados para o mesmo endereço de memória da UMA.

Para configurar esse emaranhado de planos e controladores, são necessários 50 registradores internos mal documentados.

[3] Conversor Digital-Analógico.

2.4 Modos VGA

O BIOS podia ser chamado para configurar o VGA conforme a tabela a seguir:

Tabela 2: Modos VGA disponíveis

Modo	Tipo	Resolução	Cores	RAM (endereço)	Hz
0	texto	40×25	16 (mono)	B8000h	70
1	texto	40×25	16	B8000h	70
2	texto	80×25	16 (mono)	B8000h	70
3	texto	80×25	16	B8000h	70
4	CGA	320×200	4	B8000h	70
5	CGA	320×200	4 (mono)	B8000h	70
6	CGA	640×200	2	B8000h	70
7	MDA	9×14	3 (mono)	B0000h	70
0Dh	EGA	320×200	16	A0000h	70
0Eh	EGA	640×200	16	A0000h	70
0Fh	EGA	640×350	3	A0000h	70
10h	EGA	640×350	16	A0000h	70
11h	VGA	640×480	2	A0000h	70
12h	VGA	640×480	16	A0000h	60
13h	VGA	320×200	256	A0000h	70

2.5 Programação VGA: Modo 12h

O modo 12h oferece resolução de 640×480 a 60 Hz com 16 cores. Cada pixel é codificado em 4 bits (um nibble) distribuídos pelos quatro bancos. Para escrever a cor do primeiro pixel, o desenvolvedor precisa escrever o primeiro bit do nibble no plano 0, o segundo no plano 1, o terceiro no plano 2 e o quarto no plano 3. O Controlador CRT então lê 4 bytes de uma vez (um de cada plano), resultando em 8 pixels na tela.

As principais desvantagens desse modo são:

- Sem duplo buffer: $640 \times 480/2 =$ `0x25800` bytes, mais da metade dos 256 KiB disponíveis de VRAM
- Alta resolução implica mais pixels, mais cálculos e mais operações de desenho
- Dezesseis cores limitavam severamente os artistas

2.6 Programação VGA: Modo 13h

O modo 13h é mais atraente, pois oferece resolução de 320×200 com 256 cores a 70 Hz. Ele também simula um buffer linear por meio de um chip especial chamado *Chain-4*, que utiliza os 2 bits menos significativos do endereço de RAM para programar automaticamente a máscara e rotear a operação para o banco de VRAM apropriado.

Para configurar o VGA no modo 13h usando o BIOS, são necessárias apenas duas instruções:

Listing 2: Configuração do modo 13h via BIOS

```
mov ax, 0x13    ; AH=0 (mudar modo de v deo), AL=13h (modo)
int 0x10        ; Interrup  o de v deo do BIOS
```

Exemplo de código para limpar a tela em preto:

Listing 3: Limpar a tela em modo 13h

```
char far *VGA = (byte far *) 0xA0000000L;

void ClearScreen(void) {
    asm mov ax, 0x13
    asm int 0x10
    for (int i = 0; i < 320 * 200; i++)
        VGA[i] = 0x00;
}
```

As principais desvantagens do modo 13h são:

- Com o Chain-4 todo o espaço de endereços de RAM é utilizado e não há como fazer duplo buffer
- A proporção 320×200 (1,6) não corresponde à proporção do monitor (1,333), distorcendo o framebuffer ao ser transferido para o CRT

2.7 A Importância do Double Buffering

O duplo buffer (*double buffering*) é fundamental para animações fluidas. Com apenas um buffer, o software precisa operar exatamente na frequência do CRT (70 Hz). Caso contrário, ocorre o fenômeno conhecido como *tearing* (rasgamento de imagem).

Considere uma animação de um círculo se movendo da esquerda para a direita: a CPU escreve o framebuffer enquanto o feixe de elétrons do CRT já o está varrendo para a tela. Se a CPU for mais rápida que a frequência do CRT, ela pode reescrever o framebuffer antes que a varredura seja concluída, exibindo na tela uma composição de dois quadros diferentes — o resultado visual é como se dois quadros fossem rasgados e remendados, daí o nome *tearing*.

Com dois buffers, a CPU pode escrever no segundo framebuffer sem interferir no que está sendo varrido para a tela. Resultado: sem *tearing*.