汇编代码

```
.model tiny
        .code
        .8086

        ; --- 内存/端口定义 ---
        ROM_SEGMENT     EQU     0FC00h

        ; --- RAM 变量定义 (DS=0000h) ---
        VAR_TICKS       EQU     0500h   ; [Word] 1ms 计数器
        VAR_SCAN_IDX    EQU     0502h   ; [Byte] 扫描索引
        VAR_DIGITS      EQU     0510h   ; [Bytes] 显存数组 (6字节,
0510=个位 ... 0515=十万位)

        VAR_LED_TIMER   EQU     0520h   ; [Word] LED 计时
        VAR_LED_STATE   EQU     0522h   ; [Byte] LED 状态

        VAR_SEND_REQ    EQU     0530h   ; [Byte] 串口发送请求标志 (1=
需要发送, 0=空闲)

        ; --- 硬件端口 ---
        ; 8259 PIC
        PIC_ICW1        EQU     20H
        PIC_ICW2        EQU     21H
        PIC_ICW4        EQU     21H

        ; 8254 PIT
        PIT_CNT0        EQU     40H
        PIT_CTRL        EQU     43H

        ; 8255 PPI
        PPI_PORTA       EQU     60H
        PPI_PORTB       EQU     61H
        PPI_PORTC       EQU     62H
        PPI_CTRL        EQU     63H

        ; 16550 UART (COM1)
        COM1_BASE       EQU     3F8H
        COM1_RBR        EQU     3F8H    ; Read Buffer
```

```
        COM1_THR        EQU     3F8H    ; Transmit Holding
        COM1_DLL        EQU     3F8H    ; Divisor Low (DLAB=1)
        COM1_DLM        EQU     3F9H    ; Divisor High (DLAB=1)
        COM1_IER        EQU     3F9H    ; Interrupt Enable
        COM1_LCR        EQU     3FBH    ; Line Control
        COM1_LSR        EQU     3FDH    ; Line Status

        org     0h

; --- ROM 表格 ---
SegTab  db 0C0h,0F9h,0A4h,0B0h,099h,092h,082h,0F8h,080h,090h
BitTab  db 01h, 02h, 04h, 08h, 10h, 20h

; 中断服务程序 (每 1ms 触发)
IR0_ISR PROC FAR
        PUSH AX
        PUSH BX
        PUSH CX
        PUSH SI
        PUSH DS

        MOV  AX, 0000h
        MOV  DS, AX

        ; 秒计数逻辑
        MOV  BX, VAR_TICKS
        MOV  AX, [BX]
        INC  AX
        MOV  [BX], AX

        CMP  AX, 1000
        JB   TASK_LED

        ; 秒脉冲处理
        MOV  WORD PTR [BX], 0   ; 清零毫秒VAR_TICKS

        ; 设置串口发送请求标志 (通知主循环)
        MOV  BX, VAR_SEND_REQ
        MOV  BYTE PTR [BX], 1
```

```
        ; 6位 BCD 进位加法
        MOV  SI, VAR_DIGITS
        MOV  CX, 6
INC_LOOP:
        MOV  AL, [SI]
        INC  AL
        CMP  AL, 10
        JB   NO_CARRY
        MOV  BYTE PTR [SI], 0
        INC  SI
        LOOP INC_LOOP
        JMP  TASK_LED
NO_CARRY:
        MOV  [SI], AL

        ; LED 流水灯逻辑
TASK_LED:
        MOV  BX, VAR_LED_TIMER
        MOV  AX, [BX]
        INC  AX
        MOV  [BX], AX

        CMP  AX, 200            ; 200ms 速度
        JB   TASK_SCAN

        MOV  WORD PTR [BX], 0
        MOV  BX, VAR_LED_STATE
        MOV  AL, [BX]
        SHL  AL, 1
        TEST AL, 10h            ; 检查是否溢出低4位
        JZ   UPDATE_LED
        MOV  AL, 01h
UPDATE_LED:
        MOV  [BX], AL
        NOT  AL
        MOV  DX, PPI_PORTC
        OUT  DX, AL

        ; 数码管扫描
TASK_SCAN:
```

```
        MOV   DX, PPI_PORTB
        MOV   AL, 00h
        OUT   DX, AL              ; 消隐

        MOV   BX, VAR_SCAN_IDX
        MOV   AL, [BX]
        XOR   AH, AH
        MOV   SI, AX

        INC   AL
        CMP   AL, 6
        JB    SAVE_IDX
        MOV   AL, 0
SAVE_IDX:
        MOV   [BX], AL

        MOV   BX, OFFSET BitTab
        MOV   CL, CS:[BX + SI]    ; 查位选

        MOV   BX, VAR_DIGITS
        MOV   AL, [BX + SI]
        XOR   AH, AH
        MOV   SI, AX
        MOV   BX, OFFSET SegTab
        MOV   AL, CS:[BX + SI]    ; 查段码

        MOV   DX, PPI_PORTA
        OUT   DX, AL
        MOV   DX, PPI_PORTB
        MOV   AL, CL
        OUT   DX, AL

        MOV   AL, 20H             ; EOI
        OUT   20H, AL

        POP   DS
        POP   SI
        POP   CX
        POP   BX
        POP   AX
```

```
        IRET
IR0_ISR ENDP

; 串口发送子程序 (辅助函数)
; 输入: AL = 要发送的字节
UART_SendByte PROC NEAR
        PUSH DX
        PUSH AX

        MOV  DX, COM1_THR
        OUT  DX, AL             ; 写入发送寄存器

WAIT_TX:
        ; 等待发送完毕 (查询 LSR Bit 5: THRE)
        MOV  DX, COM1_LSR
        IN   AL, DX
        AND  AL, 20h            ; 0010 0000
        JZ   WAIT_TX            ; 如果为0，继续等待

        POP  AX
        POP  DX
        RET
UART_SendByte ENDP

; 主程序
start:
        cli
        mov     ax, 0000h
        mov     ds, ax
        mov     es, ax
        mov     ss, ax
        mov     sp, 04000h

        ; 变量初始化
        mov     bx, VAR_TICKS
        mov     word ptr [bx], 0
        mov     bx, VAR_LED_TIMER
        mov     word ptr [bx], 0
        mov     bx, VAR_LED_STATE
        mov     byte ptr [bx], 01h
```

```
        mov     bx, VAR_SEND_REQ
        mov     byte ptr [bx], 0   ; 初始不发送

        ; 清显存
        mov     cx, 6
        mov     di, VAR_DIGITS
        xor     ax, ax
CLR_RAM:
        mov     [di], al
        inc     di
        loop    CLR_RAM

        ; 硬件初始化
        ; 8259A
        MOV AL, 13H
        OUT PIC_ICW1, AL
        MOV AL, 20H
        OUT PIC_ICW2, AL
        MOV AL, 09H
        OUT PIC_ICW4, AL

        ; 中断向量
        MOV BX, 80H
        MOV AX, OFFSET IR0_ISR
        MOV WORD PTR [BX], AX
        MOV BX, 82H
        MOV AX, ROM_SEGMENT
        MOV WORD PTR [BX], AX

        ; 8254
        mov dx, PIT_CTRL
        mov al, 36h
        out dx, al
        mov dx, PIT_CNT0
        mov al, 0E8h
        out dx, al
        mov al, 03h
        out dx, al

        ; 4. 8255
```

```
        mov dx, PPI_CTRL
        mov al, 80h
        out dx, al

        ; 16550 UART 初始化 (115200, 8N1)
        ; 设置 DLAB=1 访问除数锁存器
        mov dx, COM1_LCR
        mov al, 80h
        out dx, al

        ; 设置波特率 115200 (50MHz / 16 / 27)
        mov dx, COM1_DLL
        mov al, 1Bh
        out dx, al
        mov dx, COM1_DLM
        mov al, 00h
        out dx, al

        ; 设置 DLAB=0, 8数据位, 1停止位, 无校验
        mov dx, COM1_LCR
        mov al, 03h          ; 0000 0011
        out dx, al

        sti

        ; 主循环
forever:
        ; 检查是否有发送请求
        mov bx, VAR_SEND_REQ
        mov al, [bx]
        test al, al
        jz  forever              ; 如果是0，继续空转

        ; --- 开始发送数据 ---
        mov byte ptr [bx], 0     ; 清除标志位，避免重复发送

        ; 发送换行符 (CR LF) 让显示整齐
        mov al, 0Dh
        call UART_SendByte
        mov al, 0Ah
```

```
        call UART_SendByte

        ; 倒序发送 6 位数字 (从高位到低位: 0515h -> 0510h)
        mov si, 5
SEND_LOOP:
        mov bx, VAR_DIGITS
        mov al, [bx + si]       ; 读取数字 (0-9)
        add al, 30h             ; 转换为 ASCII ('0'-'9')
        call UART_SendByte

        dec si
        jns SEND_LOOP           ; 如果 si >= 0 继续

        jmp forever

; Reset Vector
        org     03ff0h
        db      0EAh
        dw      offset start
        dw      ROM_SEGMENT
        org     03fffh
        db      0

        end
```

系统总线

```
module system_bus(
    // CPU 侧控制信号
    input  wire          cpu_rd_n,      // CPU 读选通 (低电平有效)
    input  wire          cpu_wr_n,      // CPU 写选通 (低电平有效)
    input  wire          cpu_iom,       // IO/Memory 选择 (1=IO,
0=Memory)
    input  wire [19:0]   cpu_addr,      // 20位地址总线
    input  wire [7:0]    cpu_dout,      // CPU 输出的数据 (用于写入
RAM/IO)
    output wire [7:0]    cpu_din,       // CPU 输入的数据 (从
RAM/ROM/IO 读取)
```

```
    // 系统总线控制信号 (仲裁后的输出)
    output wire iorc_n,
    output wire iowc_n,
    output wire mrdc_n,
    output wire mwtc_n,

    // CPU 中断请求信号 (低电平有效)
    input  wire          cpu_inta_n,

    // 外设数据输入 (从外设读出的数据)
    input  wire [7:0]    ram_q,          // RAM 数据输出
    input  wire [7:0]    rom_q,          // ROM 数据输出
    input  wire [7:0]    pic_dout,       // PIC (8259) 数据输出

    // RAM 控制信号
    output wire          ram_wren,       // RAM 写使能 (高电平有效)
    output wire [13:0]   ram_addr,       // RAM 地址 (16KB -> 14位)
    output wire [7:0]    ram_data,       // 写给 RAM 的数据

    // ROM 控制信号
    output wire [13:0]   rom_addr,       // ROM 地址 (16KB -> 14位)

    // PIC (8259) 控制信号
    output wire          pic_cs_n,       // PIC 片选 (低电平有效)
    output wire          pic_a0,         // PIC 寄存器选择 (地址位0)
    output wire [7:0]    pic_din,        // 写给 PIC 的数据
    output wire          pic_inta_n,     // 中断响应信号 (给 PIC)

    // PIT (8254) 控制信号
    output wire          pit_cs_n,
    output wire          pit_a0,
    output wire          pit_a1,
    output wire [7:0]    pit_din,
    input  wire [7:0]    pit_dout,

    // PPI (8255) 控制信号
    output wire          ppi_cs_n,
    output wire [1:0]    ppi_addr,
    output wire [7:0]    ppi_din,
    input  wire [7:0]    ppi_dout,
```

```
    // UART (16550) 控制信号
    output wire [7:0]   uart_din,
    input  wire [7:0]   uart_dout,
    output wire [2:0]   uart_addr,
    output wire         uart_cs_n,

    // DMA (8237) 控制信号
    output wire         dma_cs_n,       // DMA 片选 (CPU 访问 DMA 寄存
器时使用)
    output wire [3:0]   dma_ain,        // CPU 访问 DMA 时的地址输入
    output wire [7:0]   dma_din,        // 送给 DMA 的数据 (RAM读取数据
或 CPU配置数据)
    input  wire [7:0]   dma_dout,       // DMA 输出的数据 (DMA配置读取
或 内存写数据)

    // DMA 主控信号 (当 DMA 获得总线权时由 DMA 输入)
    input  wire         dma_mrdc_n,
    input  wire         dma_mwtc_n,
    input  wire         dma_iorc_n,
    input  wire         dma_iowc_n,
    input  wire         dma_aen,        // DMA 地址允许 (1=DMA主控,
0=CPU主控) -> 关键仲裁信号
    input  wire         dma_dben,
    input  wire         dma_adstb,
    input  wire [3:0]   dma_dack,       // DMA 响应信号 (用于选择外设)
    input  wire [7:0]   dma_aout        // DMA 输出的高位地址 (配合外部
锁存器)
);

    // 总线仲裁逻辑 (Bus Arbitration)

    // 内部地址总线：根据 AEN 选择 CPU 地址还是 DMA 地址
    wire [19:0] sys_addr;
    wire [19:0] dma_addr;
    assign sys_addr = (dma_aen) ? dma_addr : cpu_addr;

    // CPU 侧产生的控制信号 (预处理)
    wire cpu_mrdc_n_int, cpu_mwtc_n_int, cpu_iorc_n_int,
cpu_iowc_n_int;
```

```
    assign cpu_mrdc_n_int = (cpu_iom == 1'b0 && cpu_rd_n == 1'b0) ?
1'b0 : 1'b1;
    assign cpu_mwtc_n_int = (cpu_iom == 1'b0 && cpu_wr_n == 1'b0) ?
1'b0 : 1'b1;
    assign cpu_iorc_n_int = (cpu_iom == 1'b1 && cpu_rd_n == 1'b0) ?
1'b0 : 1'b1;
    assign cpu_iowc_n_int = (cpu_iom == 1'b1 && cpu_wr_n == 1'b0) ?
1'b0 : 1'b1;

    // 最终系统控制信号：根据 AEN 选择 CPU 信号还是 DMA 信号
    assign mrdc_n = (dma_aen) ? dma_mrdc_n : cpu_mrdc_n_int;
    assign mwtc_n = (dma_aen) ? dma_mwtc_n : cpu_mwtc_n_int;
    assign iorc_n = (dma_aen) ? dma_iorc_n : cpu_iorc_n_int;
    assign iowc_n = (dma_aen) ? dma_iowc_n : cpu_iowc_n_int;

    // 地址译码逻辑 (Address Decoding)

    // 内部片选信号 (高电平表示选中)
    wire ram_cs, rom_cs, pic_cs, pit_cs, ppi_cs, uart_cs, dma_cs;

    // RAM: 0x00000 - 0x03FFF
    // 选中条件：A19..A14 = 000000，且必须是存储器操作 (dma_aen=1 时默认为
存储器传输或由 dma_dack 区分)
    // 这里简化处理：只要地址匹配即选中
    assign ram_cs = (sys_addr[19:14] == 6'b000000) ? 1'b1 : 1'b0;

    // ROM: 0xFC000 - 0xFFFFF
    assign rom_cs = (sys_addr[19:14] == 6'b111111) ? 1'b1 : 1'b0;

    // IO 空间译码 (注意：DMA 模式下地址可能无效，主要靠 DACK，但这里保留地址
映射供 CPU 访问) ---

    // DMA (8237): IO Space, Address 0x00 - 0x0F (标准 PC 映射)
    // 只有在 CPU 控制总线时 (dma_aen=0) 且地址匹配时选中
    assign dma_cs = (!dma_aen && sys_addr[7:4] == 4'b0000) ? 1'b1 :
1'b0;

    // PIC (8259): IO Space, Address 0x20 - 0x21
```

```
    assign pic_cs = (!dma_aen && sys_addr[7:4] == 4'b0010) ? 1'b1 :
1'b0;

    // PIT (8254): IO Space, Address 0x40 - 0x43
    assign pit_cs = (!dma_aen && sys_addr[7:4] == 4'b0100) ? 1'b1 :
1'b0;

    // PPI (8255): IO Space, Address 0x60 - 0x63
    assign ppi_cs = (!dma_aen && sys_addr[7:4] == 4'b0110) ? 1'b1 :
1'b0;

    // UART (16550): IO Space, Address 0x3F8 - 0x3FF
    assign uart_cs = (!dma_aen && sys_addr[9:3] == 7'b1111111) ?
1'b1 : 1'b0;

    // 输出信号分配 (Output Assignments)

    // --- RAM 接口 ---
    assign ram_addr = sys_addr[13:0]; // 使用仲裁后的地址
    assign ram_data = (dma_aen) ? dma_dout : cpu_dout;
    assign ram_wren = (ram_cs == 1'b1 && mwtc_n == 1'b0) ? 1'b1 :
1'b0;

    // --- ROM 接口 ---
    assign rom_addr = sys_addr[13:0];

    // --- PIC (8259) 接口 ---
    assign pic_cs_n   = !pic_cs;
    assign pic_a0     = sys_addr[0];
    assign pic_din    = cpu_dout;    // PIC 只能被 CPU 配置
    assign pic_inta_n = cpu_inta_n;

    // --- PIT (8254) 接口 ---
    assign pit_cs_n = ~pit_cs;
    assign pit_a0   = sys_addr[0];
    assign pit_a1   = sys_addr[1];
    assign pit_din  = cpu_dout;

    // --- PPI (8255) 接口 ---
    assign ppi_cs_n = ~ppi_cs;
```

```
    assign ppi_addr = sys_addr[1:0];
    assign ppi_din  = cpu_dout;

    // --- UART (16550) 接口 ---
    assign uart_cs_n = ~uart_cs;
    assign uart_addr = sys_addr[2:0];
    assign uart_din  = cpu_dout;

    // --- DMA (8237) Slave 接口 (CPU 读写 DMA 寄存器) ---
    assign dma_cs_n = ~dma_cs;
    assign dma_ain  = sys_addr[3:0]; // DMA 内部通常有16个寄存器地址空间
    // dma_din 的数据来源:
    // 1. 当 CPU 写 DMA 寄存器时 (dma_aen=0, dma_cs=1), 数据来自
cpu_dout。
    // 2. 当 DMA 读 存储器时 (dma_aen=1, Mem -> IO), 数据来自
ram_q/rom_q。
    assign dma_din  = (dma_aen) ? ram_q : cpu_dout;

    // 数据总线多路复用 (Data Bus Multiplexer to CPU)

    // cpu_din: CPU 读取到的数据
    // 注意: 增加了读取 DMA 寄存器的情况 (dma_cs)
    assign cpu_din = (!cpu_inta_n) ? pic_dout :
                     (ram_cs && !mrdc_n) ? ram_q :       // 读 RAM
                     (rom_cs && !mrdc_n) ? rom_q :       // 读 ROM
                     (pic_cs && !iorc_n) ? pic_dout :    // 读 PIC
                     (pit_cs && !iorc_n) ? pit_dout :    // 读 PIT
                     (ppi_cs && !iorc_n) ? ppi_dout :    // 读 PPI
                     (uart_cs && !iorc_n) ? uart_dout :  // 读 UART
                     (dma_cs && !iorc_n)  ? dma_dout :   // 读 DMA
寄存器
                     8'h00;

endmodule
```

exe2hex.py

```
import os
import struct
```

```
import sys

# =================配置区域=================
# 这里写死 ROM 大小：16384 * 8bits = 16384 字节
ROM_SIZE_BYTES = 16384
# ==========================================


def calculate_checksum(record_bytes):
    """计算 Intel HEX 校验和: 0x100 - (sum(bytes) & 0xFF)"""
    total = sum(record_bytes)
    return (-total) & 0xFF


def exe_to_bin_content(exe_data):
    """
    尝试解析 DOS MZ 头并提取代码段。
    """
    # 检查是否有 MZ 签名 (0x4D 0x5A)
    if len(exe_data) > 2 and exe_data[0:2] == b"MZ":
        print("检测到 DOS EXE 文件头，正在提取代码段...")
        # 解析 MZ 头, 0x08: Header size in paragraphs
        header_paragraphs = struct.unpack_from("<H", exe_data,
0x08)[0]
        header_size = header_paragraphs * 16

        # 提取实际代码
        code_data = exe_data[header_size:]
        print(f"头大小: {header_size} 字节, 代码净荷: {len(code_data)}
字节")
        return code_data
    else:
        print("未检测到 EXE 头，按纯二进制文件处理。")
        return exe_data


def bin_to_hex(bin_data, output_file):
    """将二进制数据转换为定长 Intel HEX 格式"""

    current_len = len(bin_data)
```

```
    # === 强制大小检查与填充逻辑 ===
    print(f"目标 ROM 大小: {ROM_SIZE_BYTES} 字节")

    if current_len > ROM_SIZE_BYTES:
        print(
            f"【错误】输入文件过大! ({current_len} 字节) 超过了 ROM 上限
({ROM_SIZE_BYTES} 字节)。"
        )
        sys.exit(1)
    elif current_len < ROM_SIZE_BYTES:
        pad_len = ROM_SIZE_BYTES - current_len
        print(f"输入数据 {current_len} 字节，正在填充 {pad_len} 字节的
0xFF...")
        # 填充 0xFF (通常代表空指令或无数据)
        bin_data = bytearray(bin_data) + (b"\xff" * pad_len)
    else:
        print("输入数据大小正好等于 ROM 大小，无需填充。")

    start_address = 0

    with open(output_file, "w") as f:
        print(f"正在生成 {output_file} ...")

        # 每次读取 16 字节
        for i in range(0, len(bin_data), 16):
            chunk = bin_data[i : i + 16]
            byte_count = len(chunk)
            address = start_address + i
            record_type = 0  # 00 = 数据记录

            # [Byte Count, Addr Hi, Addr Lo, Record Type, Data...]
            record_bytes = [
                byte_count,
                (address >> 8) & 0xFF,
                address & 0xFF,
                record_type,
            ]
            record_bytes.extend(chunk)
```

```
            # 计算校验和
            checksum = calculate_checksum(record_bytes)

            # 生成 HEX 行 :LLAAAATTDD...DDCC
            hex_line = ":{:02X}{:04X}{:02X}".format(byte_count,
address, record_type)
            hex_line += "".join(["{:02X}".format(b) for b in
chunk])
            hex_line += "{:02X}\n".format(checksum)

            f.write(hex_line)

        # 写入文件结束记录 :00000001FF
        f.write(":00000001FF\n")
        print("转换完成! ")


if __name__ == "__main__":
    if len(sys.argv) < 3:
        print("用法: python exe2hex.py <输入文件> <输出文件.hex>")
    else:
        input_path = sys.argv[1]
        output_path = sys.argv[2]

        try:
            with open(input_path, "rb") as f_in:
                raw_data = f_in.read()

            # 1. 提取有效数据
            clean_data = exe_to_bin_content(raw_data)

            # 2. 转换并填充到固定大小
            bin_to_hex(clean_data, output_path)

        except FileNotFoundError:
            print(f"错误: 找不到文件 {input_path}")
        except Exception as e:
            print(f"发生未知错误: {e}")
```