杭州電子科技大学
HANGZHOU DIANZI UNIVERSITY

《创新实践 3》期末报告

HDMI 显示接口在 FPGA-SoC 系统中的驱动实现与应用部署（数字电路实验课助教任务）

学生姓名 陈文轩
学生学号 23040447
学生专业 智能硬件与系统(电子信息工程)
指导教师 盛庆华老师

2025 年 12 月 27 日

摘　要

本报告围绕学生用开发板从 VGA 接口（HX1006A）升级到 HDMI 接口（HX1006B）后的教学与工程需求，完成并验证了 FPGA 端 HDMI 显示驱动与 SoC 系统的 HDMI 显示改造两项**数电实验课助教任务**。首先，基于 DVI 兼容模式**实现 720p@60Hz 视频链路**：PLL 生成 74.25MHz 像素时钟与 371.25MHz 高速时钟，完成视频时序生成（`hsync/vsync/de`）、TMDS 编码以及 10:1 并串与差分输出，并以屏幕彩条 Demo 验证链路稳定性。在此基础上，将原 VGA 显示工程移植到 HDMI 平台，选取“**HDMI 乐谱播放器**”作为综合示例：通过位图 ROM 实现 512×256 乐谱图像渲染，并叠加动态方框指示当前音符位置，同时与蜂鸣器播放保持节拍同步。

其次，在 RV32I 五级流水线 **SoC 系统中集成 HDMI 显示外设**，采用**内存映射 I/O 方式**将字符显存映射为 CPU 可直接读写的 RAM 区域，实现 CPU“写显存即显示”的字符输出模型；并针对 CPU 系统时钟域与像素时钟域的**跨时钟数据交互**（CDC 问题的规避）、以及显示**流水线延迟**下的时序/像素对齐问题给出可靠实现。最终以基于 C 语言编写的数字时钟 Demo 进行验证，结果表明 **FPGA SoC 上的 HDMI 外设可稳定输出 720p 信号，字符显示清晰无乱码且可连续刷新，证明平台具备面向实验教学的易用性与稳定性**。

关键词：**FPGA-HDMI 驱动，RV32I SoC，HDMI 驱动移植，CDC 问题，流水线时序对齐**

目录

一　FPGA-HDMI 程序助教任务目标

本学期笔者**担任了黄继业老师的数字系统与处理器实验课的本科生助教**，将 FPGA 实验开发板从 HX1006A（VGA 视频输出）升级为支持 HDMI 输出的 HX1006B。作为助教编程任务之一，需完成以下两项工作：

1. 设计实现 FPGA HDMI 驱动，并运用**驱动实现 HDMI 彩条显示的基本 Demo**，在 HDMI 成功驱动的基础上改写**所有**原有的 VGA 图像显示工程，等效输出在 HDMI 显示器上，本报告选取较为复杂的 **HDMI 乐谱播放器工程**作为介绍内容。

2. 在已有 VGA 输出接口的 RV32I SoC 系统上，**改写此 FPGA SoC 系统，使得其支持 HDMI 显示接口**，并实现基于 HDMI 的字符显示功能，最终设计并运行一个基于 SoC-HDMI 的数字时钟 Demo 程序。

二　FPGA-HDMI 驱动的实现（屏幕色条 Demo）

2.1　HDMI 显示通信原理

HDMI 用于传输高清数字视频（以及音频/控制信息）。在本工程中只使用了**与 DVI 兼容的“视频传输”**部分，其核心特点是：**不以数据包形式发送图像**，而是以**连续像素流**的方式按行、按帧扫描输出。发送端负责生成视频时序（行/场同步与消隐区），接收端依据时序在正确的时间点采样并还原每个像素的 RGB 数据。

对 720p@60Hz 而言，一帧图像由“有效显示区 + 消隐区”组成。有效区内每到来一个像素时钟（`pixel_clk`），系统输出一**个像素的 24 位 RGB（RGB888，8:8:8）**；`hsync/vsync` 同步信息并不是额外占用一根“同步线”去传输，而是在 `de=0`（消隐区）时由 TMDS 发送**控制符号**来携带，其中 `de`（Data Enable）是发送端根据**行/列计数**判断“当前像素是否属于有效显示区”得到的标志，它通过**解码出来的 TMDS 符号类型**来区分。在消隐区期间，由 **TMDS 数据通道 0** 承载数据传输，RGB 数据不属于有效显示内容（置零），但像素时钟与同步时序必须连续且严格满足规范：`hsync`（换行提示信号）在每行给出固定位置/宽度/极性的同步脉冲，用于标识“新一行”的像素数据传输边界并使接收端复位水平计数；`vsync`（换帧提示信号）在每帧给出同步脉冲，用于标识“新一帧”的边界并复位垂直计数。与此同时，前沿/后沿（front/back porch）为链路与显示端提供采样、缓冲与处理裕量，保证接收端 PLL/时序恢复电路能够稳定锁定并在随后的有效区内按正确坐标输出像素，从而避免错行、抖动或画面不稳定等问题。

在 DVI 兼容的视频模式下，三路数据通道 `D2/D1/D0` 分别承载 **R/G/B** 三个 8bit 颜色分量（约定为 `D2`→`R`，`D1`→`G`，`D0`→`B`），时钟通道 `CLK` 提供像素采样参考。也就是说，一个像素的 **24 位 RGB888** 会被拆分为 3 个 8bit 并行分量，同时送入三路 TMDS 编码器。有效显示区（`de=1`）时，10bit 符号由**该通道的 8bit 颜色分量经 TMDS 8bit 到 10bit 编码**得到，利用此编码方式实现减少跳变与实现直流平衡。消隐区（`de=0`）时传输控制符号，即 `hsync/vsync` 等同步信息会被编码进控制符号，从而在没有有效像素数据时仍能传递时序边界。

由于每像素每通道发送 10bit，单通道串行码率为

$$R_{\text{TMDS}} = f_{\text{pixel}} \times 10$$

720p@60Hz 典型像素时钟为 $f_{\text{pixel}} = 74.25\text{MHz}$，因此单通道码率约为 742.5Mbps。本工程采用 `pixel_clk_5x` 配合 DDR 方式实现 10:1 并串：`pixel_clk_5x` 为像素时钟的 5 倍（371.25MHz），在其上升沿/下降沿各输出 1bit，从而每 5 个高速周期输出 10bit。

表 1: HDMI-B 输入引脚表

信号名称	FPGA 引脚	信号名称	FPGA 引脚
HDMIB_D2_P	B8	HDMIB_D0_P	F15
HDMIB_D2_N	A8	HDMIB_D0_N	F16
HDMIB_D1_P	B9	HDMIB_CLK_N	G16
HDMIB_D1_N	A9	HDMIB_CLK_P	G15

对每个像素时钟周期，三路数据通道各发送一**个 10bit TMDS 符号**（由 8bit 颜色分量经 TMDS 编码得到），因此每个像素时钟在链路上并行传递 3 个 10bit 符号。接收端在 CLK 的配合下完成串并转换与 TMDS 解码，即可连续恢复出逐像素的 RGB 数据流。

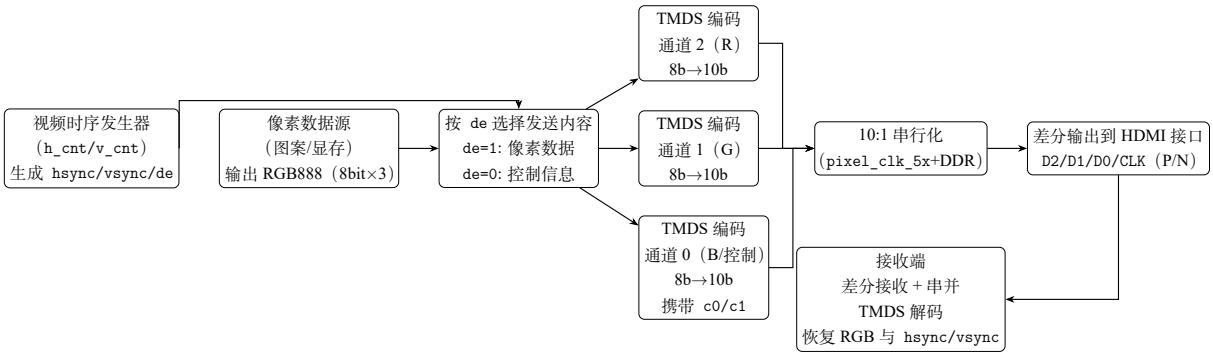


图 1: HDMI（DVI 兼容）显示数据处理流程示意

2.2 HDMI 色条显示系统结构

HDMI 的屏幕色条 Demo 用于验证 HDMI 显示链路的完整性与稳定性。通过最简单的“时序生成 + 颜色图案”组合，可以快速确认像素时钟、视频时序、TMDS 编码与差分输出是否工作正常，是后续复杂图像显示的基础。

HDMI_Colorbar 工程的顶层模块为 **HDMI 彩条显示顶层模块**，系统结构可分为四个部分：

1. **时钟管理**：`pll_clk` 将 50MHz 系统时钟变换为 74.25MHz 像素时钟与 371.25MHz TDMS 传输时钟。（驱动层）

2. **视频时序**：`video_driver` 产生 720p@60Hz 的 hsync/vsync/data_enable 以及像素坐标。（驱动层）

3. **图案生成**：video_display 依据横坐标生成 8 条彩色竖条。（应用层）

4.**HDMI 输出**：dvi_transmitter_top 完成 TMDS 编码与 10:1 串行化，输出差分信号到 HDMI 接口。（驱动层）

本工程采用 1280×720@60Hz（720p@60）标准视频时序。**视频时序生成模块**通过水平计数器与垂直计数器对每一行/每一帧进行计数，并据此输出 hsync/vsync/de：其中 de=1 仅在" 有效显示区"（H_DISP×V_DISP）期间成立；其余前沿、同步脉冲、后沿均属于消隐区（de=0），用于提供行/帧边界与时序裕量。表2给出水平与垂直方向的分段参数，并满足 H_TOTAL = H_SYNC + H_BACK + H_DISP + H_FRONT, V_TOTAL = V_SYNC + V_BACK + V_DISP + V_FRONT。

参数	水平 (H)	垂直 (V)	说明
同步脉冲	40	5	H_SYNC/V_SYNC
后沿	220	20	H_BACK/V_BACK
有效显示	1280	720	H_DISP/V_DISP
前沿	110	5	H_FRONT/V_FRONT
总计	1650	750	H_TOTAL/V_TOTAL

表 2: HDMI 720p@60Hz 时序参数（通用视频时序标准）

对于上表，同步脉冲 H_SYNC=40 即每行开始时 hsync 维持同步电平 40 个像素时钟，用来告诉显示器“新的一行开始了”，同样的同步脉冲 V_SYNC=5 即每帧开始时 vsync 维持同步电平 5 行（接收端解码 TMDS 控制符号，就能恢复出 vsync，并看到它连续有效了 5 行），用来告诉显示器“新的一帧开始了”。后沿 H_BACK=220 即在 hsync 结束后再等待 220 个像素时钟（仍是消隐区），给接收端留稳定时间；后沿 V_BACK=20 即 vsync 结束后再等 20 行（消隐区）。有效显示 H_DISP=1280 为真正显示的像素数，一行显示 1280 个像素点；有效显示 V_DISP=720 即真正显示的行数，一帧显示 720 行。前沿 H_FRONT=110 为一行有效区结束后再等待 110 个像素时钟（消隐区），然后进入下一行的同步脉冲。前沿 V_FRONT=5 为一帧有效区结束后再等待 5 行（消隐区），然后进入下一帧的同步脉冲。由上总结，一行总长度 1650 个像素时钟（74.25MHz），一帧总行数 750 行。

彩条图案生成模块根据像素横坐标将显示区域等分为 8 段（等效为使用 pixel_xpos[10:8] 或均分宽度的比较），并输出 RGB888 颜色数据。颜色顺序与 VGA 实验一致，可描述为（从右到左）：

蓝、黑、红、粉、绿、青、黄、白

彩条图案生成模块内部色条生成计算（仅仅是应用层的色条生成计算，而非驱动层的信号处理计算）色条宽度的参数仍保留为 H_DISP=800（800×600）。因此其色条宽度 BAND_WIDTH=H_DISP/8=100，前 7 段各 100 像素（共 700 像素）依次输出色条，而当 pixel_xpos≥ 700 时进入 else 分支输出蓝色。由于实际有效显示宽度为 1280 像素，最

后这段蓝色会占据 $1280 - 700 = 580$ 像素，看起来就成为” 右侧一大块蓝条”。显示器是 1080p 面板时，会将输入的 720p 画面做等比例缩放到屏幕上，因此蓝条变大的根因仍是色条生成参数与输出分辨率不一致，但此不影响驱动的正常运行。在显示器上可观察到稳定的 8 条垂直彩色条纹，颜色顺序正确且无明显抖动、花屏或闪烁，说明像素时序、编码与差分输出链路工作正常。

图 2: HDMI 色条显示效果（可见右侧蓝色区域更宽）

```
parameter  H_DISP = 11'd800;
//define 1280 will be better for hdmi 720p. 800 can work too

// uniform band width
localparam BAND_WIDTH = H_DISP / 8;

// generate 8 vertical color bars by pixel_xpos
always @(posedge pixel_clk) begin
    if (!sys_rst_n)
        pixel_data <= 24'd0;
    else if (pixel_xpos < BAND_WIDTH * 1)
        pixel_data <= WHITE;
    else if (pixel_xpos < BAND_WIDTH * 2)
        pixel_data <= YELLOW;
    else if (pixel_xpos < BAND_WIDTH * 3)
        pixel_data <= CYAN;
    else if (pixel_xpos < BAND_WIDTH * 4)
```

```
        pixel_data <= GREEN;
    else if (pixel_xpos < BAND_WIDTH * 5)
        pixel_data <= MAGENTA;
    else if (pixel_xpos < BAND_WIDTH * 6)
        pixel_data <= RED;
    else if (pixel_xpos < BAND_WIDTH * 7)
        pixel_data <= BLACK;
    else
        pixel_data <= BLUE;
end
```

Listing 1: 彩条图案生成模块核心逻辑

三　FPGA-HDMI 乐谱播放器设计与实现

3.1　总体结构

FPGA 工程顶层模块将 HDMI 显示链路与音乐播放链路集成。首先是 **HDMI 显示链路**，利用已经完成的 FPGA HDMI 的 720P 的视频驱动，设计乐谱图像显示与动态方框叠加模块，实现乐谱图像的 HDMI 显示与当前音符位置指示。再是**音乐播放链路**，PLL 产生 1MHz/2kHz，分频得到 4Hz 节拍，`cnt138t` 输出音符序列索引，`music` ROM 给出音符数据，经 `f_code` 译码后驱动 `speak` 与 `DFF` 产生蜂鸣器音频输出。

3.2　乐谱图像与动态方框的实现

乐谱图像存储在**乐谱位图 ROM** 中，分辨率为 512×256，1bit 黑白数据。**视频显示模块**在像素域内完成图像读取、边框绘制与动态方框叠加。首先乐谱图像需要居中显示（按照原 VGA 坐标体系，使用 800×600 虚拟窗口进行居中计算）。再进行 ROM 地址映射，即 $\texttt{rom\_addr} = \{y[7:0], x[8:0]\} = y \times 512 + x$，把二维坐标 (x,y) 映射成一维 ROM 地址 rom_addr，第 y 行从地址 y*512 开始，行内第 x 个像素就是 y*512 + x，还要将.mif 图像文件中 1bit 数据扩展为 RGB888，`rom_data=0` 显示输出黑色 24'h000000，`rom_data=1` 显示输出粉紫色 24'hC030A0。最后绘制 2 像素红色边框，增强图像边界辨识。

根据工程要求，还需绘制动态方框用于标记当前音符位置。计数器 `cnt138` 同时驱动音符播放与方框位置：

$$\texttt{cnt138\_adj} = \max(\texttt{cnt138} - 2, 0) \tag{1}$$

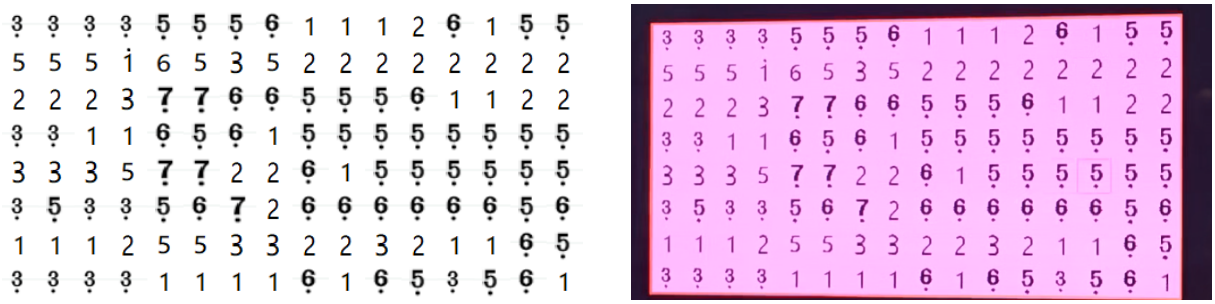
$$x = (\texttt{IMG\_X\_START} - 1) + (\texttt{cnt138\_adj}[3:0] \times 32) \tag{2}$$

$$y = (\texttt{IMG\_Y\_START} + 1) + (\texttt{cnt138\_adj}[6:4] \times 32) \tag{3}$$

其中 cnt138 低 4 位对应 16 列，高 3 位对应 8 行，实现 16×8 音符布局；延迟 2 个音符位置用于补偿音频输出与显示之间的节拍差异。

音乐数据由**音符数据 ROM** 提供，共 138 个音符；**音高译码模块**将音符编号译码为音高与高音标志，**PWM 音频生成模块**根据音高生成 PWM 脉冲，**D 触发器模块**对 PWM 脉冲翻转产生方波，驱动蜂鸣器。系统以 4Hz 节拍进行音符切换，LED 同步显示当前音符，保证”听觉—视觉”一致性。

工程全部完成后，实现了 HDMI 显示稳定输出，乐谱图像（《梁祝》简化歌谱）居中显示，背景为黑色。红色动态方框在乐谱中随音乐节拍移动，指示当前音符。蜂鸣器播放完整旋律（《梁祝》乐曲），LED 显示当前音符编码，三者同步。



(a) 原乐谱显示效果　(b) HDMI 移植后显示效果（有红色动态方框）

图 3: 原乐谱与 FPGA-HDMI 移植后显示效果对比

```
// image size: 512x256 (1-bit), centered in a 800x600 virtual window
parameter  H_DISP      = 11'd800;
parameter  V_DISP      = 11'd600;
parameter  IMG_WIDTH   = 11'd512;
parameter  IMG_HEIGHT  = 11'd256;
localparam IMG_X_START = (H_DISP - IMG_WIDTH) / 2;
localparam IMG_Y_START = (V_DISP - IMG_HEIGHT) / 2;

wire [10:0] x = pixel_xpos - IMG_X_START;
wire [10:0] y = pixel_ypos - IMG_Y_START;
wire dis_en = (pixel_xpos >= IMG_X_START && pixel_xpos < IMG_X_START + IMG_WIDTH) &&
              (pixel_ypos >= IMG_Y_START && pixel_ypos < IMG_Y_START + IMG_HEIGHT);

// ROM address mapping: rom_addr = {y[7:0], x[8:0]} = y*512 + x
always @(posedge pixel_clk or negedge sys_rst_n) begin
    if (!sys_rst_n)
        rom_addr <= 17'd0;
    else if (dis_en)
```

```
19          rom_addr <= {y[7:0], x[8:0]};
20      else
21          rom_addr <= 17'd0;
22  end
23
24  // priority: dynamic box > border > image > background
25  always @(posedge pixel_clk or negedge sys_rst_n) begin
26      if (!sys_rst_n)
27          pixel_data <= 24'd0;
28      else if (dynamic_box_en)
29          pixel_data <= RED;
30      else if (border_en)
31          pixel_data <= RED;
32      else if (dis_en) begin
33          if (rom_data) begin
34              // expand 1-bit to RGB888 (a purple-ish foreground
                    color)
35              pixel_data[23:16] <=
                    {rom_data,rom_data,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0};
                     // R
36              pixel_data[15:8]  <=
                    {1'b0,1'b0,rom_data,rom_data,1'b0,1'b0,1'b0,1'b0};
                     // G
37              pixel_data[7:0]   <=
                    {rom_data,1'b0,rom_data,1'b0,1'b0,1'b0,1'b0,1'b0};
                     // B
38          end else
39              pixel_data <= BLACK;
40      end else
41          pixel_data <= BLACK;
42  end
```

Listing 2: 视频显示模块乐谱渲染核心逻辑

以上显示核心应用模块在像素时钟域内完成“读图 + 叠加绘制 + 输出 RGB”的应用层渲染，首先是**居中与显示使能（dis_en）**，即以 800×600 作为虚拟窗口，将 512×256 的乐谱图像居中，得到左上角起点 IMG_X_START/IMG_Y_START。当当前像素坐标 pixel_xpos/pixel_ypos 落在该矩形范围内时，dis_en=1 表示需要显示 ROM 图像，否则输出背景色；再进行 **ROM 地址映射（rom_addr）**，先将图像内相对坐标（x,y）映射到线性地址：rom_addr={y[7:0],x[8:0]}=y*512+x，即每行固定 512 像素，按“先行后列”顺序读出 rom_data（1bit 黑白像素）；同时要注意乐谱框的生成，本工程采用“动态方框 → 红色边框 → ROM 图像 → 黑色背景”的优先级，保证方框不会被图像覆盖，

边框也能清晰显示。最后进行 mif 文件中的 **1bit 像素到 RGB888 扩展** `rom_data=0` 输出黑色；`rom_data=1` 时将该 1bit 按预设方式扩展到 RGB 三个分量，形成前景色（本工程实际显示为粉紫色系 `24'hC030A0`）。

四　SoC HDMI 驱动改造与字符显示

4.1　RV32I SoC 总体结构与显示接口

FPGA 上部署的 RV32I 指令集的 SoC 系统由 **RV32I 处理器核 + 片上总线与仲裁 + 指令/数据存储器 + 外设**组成。本小节先介绍 CPU 与 SoC 基本结构、总线与存储器读写方式、以及 LED/数码管等外设的地址映射与使用方法；至于本人在 SoC 系统中独立设计改写的 HDMI 显示相关外设的控制实现，将在后续小节单独展开。

4.1.1　RV32I 指令格式与基本概念

在深入介绍 CPU 结构前，需先理清 RISC-V RV32I 指令集的基本特性。RV32I 中的”32” 表示寄存器与数据通路的位宽为 32 位（4 字节），”I” 代表整数**基础指令集**。**每条 RV32I 指令固定占用 32 位（即 4 字节）**，这是 RISC-V 定长指令编码的核心特征，与 x86 等变长指令集不同。

指令的 32 位按固定格式划分为若干字段，例如：

1. texttopcode（[6:0]）：7 位操作码，标识指令大类（如 ALU 运算/Load/Store/分支等）；
2. textrd（[11:7]）：5 位目的寄存器地址，指定运算结果写入的寄存器；
3. textfunct3（[14:12]）：3 位功能码，细化操作类型（如加法/减法/逻辑与等）；
4. textrs1（[19:15]）：5 位源寄存器 1 地址，指定第一个操作数寄存器；
5. textrs2（[24:20]）：5 位源寄存器 2 地址，指定第二个操作数寄存器；
6. textfunct7（[31:25]）：7 位功能码，进一步细化操作类型（如区分加法与减法）；
7. textimmediate：立即数字段，指令中直接携带的常数值，用于 ALU 运算或地址计算。

由于指令定长为 4 字节，**存储器中相邻两条指令的地址相差 4**。因此程序计数器（PC）在顺序执行时执行 `PC=PC+4` 操作，跳转到下一条指令的起始地址；跳转/分支指令则会将 PC 改写为 `PC+ 立即数偏移或寄存器值 + 偏移`，实现控制流转移。

本工程 CPU 采用**经典的五级流水线（IF 取指、ID 译码、EX 执行、MEM 访存、WB 回写）**来提高指令吞吐率。流水线各级之间通过**段寄存器**（pipeline register）保存中间结果，使得在任一时钟周期内，CPU 可同时处理 5 条不同指令的不同阶段，每周期即可完成一条指令。下面详细介绍各级功能。

4.1.2 CPU 五级流水线结构（IF 取值）

CPU 通过独立的**取指总线主口（master2）**将当前 PC 作为读地址 haddr 发送到总线，指令存储器（ROM 或 RAM）在下一个时钟周期返回 32 位指令字 instr。**取到的指令会被暂存在 IF/ID 段寄存器中**，等待下一个时钟周期送入译码阶段进行解析。同时，PC 值也会一起保存到 IF/ID 段寄存器，因为后续阶段需要用 PC 来计算跳转目标地址或保存返回地址。在没有跳转/分支时，PC 按顺序递增 4 字节：PC=PC+4，指向下一条指令；当执行跳转指令（JAL/JALR）或条件分支成立（BEQ/BNE/BLT 等）时，PC 被改写为计算得到的目标地址（PC 重定向），实现控制流转移。若遇到**流水线暂停（stall）**（例如 load-use 相关或总线冲突），则保持 PC 不变，避免取走错误指令。取指阶段的输出（PC 与指令字）通过 **IF/ID 段寄存器**送入译码阶段。由**取指总线适配器模块**实现取指总线适配、PC 更新与跳转目标计算。

4.1.3 CPU 五级流水线结构（ID 译码）

译码阶段将**从 IF/ID 段寄存器接收到的 32 位指令字解析为控制信号与操作数地址**，主要工作包括：

1. 指令字段提取 ：按 RISC-V 指令格式从 32 位指令 instr 中提取各个字段。**指令译码模块**会将指令的 32 位按固定位置切分为 opcode（操作码，决定是加法还是减法等）、funct3（功能码 3 位，进一步细分操作类型）、funct7（功能码 7 位）、rd（目的寄存器地址）、rs1（源寄存器 1 地址）、rs2（源寄存器 2 地址）等字段。例如以下 Verilog 代码段，就实现了分配：

```
{funct7, rs2, rs1, funct3, rd, opcode} = instr[31:0];
// 将32位指令的各个位段分配给对应变量
```

2. 指令类型识别与立即数提取 ：根据 opcode(操作码)判断指令属于哪种类型(R/I/S/B/U/J 型)，不同类型的指令其立即数（指令中携带的常数值）位置不同，需要按不同规则提取和拼接。以下对各类型指令的立即数提取规则进行说明：

1. I 型指令（如 ADDI 加立即数、LW 加载字）：立即数在 instr[31:20]（12 位），需符号扩展到 32 位（如果最高位是 1 表示负数，要把高位都补 1）
2. S 型指令（如 SW 存储字）：立即数分散在 instr[31:25] 和 instr[11:7] 两处，需要拼接成 12 位再符号扩展
3. B 型指令（条件分支如 BEQ）：立即数需重新排列各位并左移 1 位（因 RISC-V 规定分支偏移以 2 字节为单位，而本工程指令 4 字节对齐，实际左移产生 4 的倍数偏移）
4. U 型指令（如 LUI 加载高位立即数）：立即数占 instr[31:12] 共 20 位，直接放到 32 位字的高 20 位
5. J 型指令（如 JAL 跳转）：立即数位域重排后左移 1 位，用于计算跳转目标

3. 控制信号生成 ：**指令译码模块**根据指令类型产生一系列控制信号（可以理解为" 开关信号"），告诉后续流水级该如何处理这条指令，包括以下模块：

1. `alures2reg`：ALU（算术逻辑单元）计算结果是否要写回寄存器?（加减运算、逻辑运算、跳转指令需要写回，例如 `ADD x1,x2,x3` 会把结果写到 x1）
2. `memory2reg`：内存读数据是否写回寄存器?（Load 指令如 `LW x1,0(x2)` 需要，将内存数据写到 x1）
3. `memwrite`：是否需要写内存?（Store 指令如 `SW x1,0(x2)` 需要，将 x1 的值写到内存）
4. `jal`：是否为 `JAL` 无条件跳转指令?（用于提前判断需要改变 PC）
5. `src1_reg_en/src2_reg_en`：是否需要读取源寄存器 1/2?（用于检测数据相关性，判断是否需要暂停或前递）

4. 寄存器堆读取 ：根据指令中提取的 `rs1`（源寄存器 1 地址）和 `rs2`（源寄存器 2 地址），从 CPU 的 32 个通用寄存器（编号 x0 到 x31，其中 x0 恒为 0）中读出操作数。**寄存器堆模块**支持**双读单写**：每个时钟周期可以同时读取两个寄存器的值（作为 ALU 的两个输入），并在回写阶段写入一个寄存器的值。

为了减少流水线停顿，寄存器堆集成了**数据前递（forwarding）**机制：当读取的寄存器刚好是前一条或前两条指令的目的寄存器、但数据尚未写回时（例如前一条是 `ADD x1,x2,x3` 刚计算完，当前指令是 `SUB x4,x1,x5` 要用 x1），系统不会傻等数据写回，而是直接从执行阶段（EX）或访存阶段（MEM）" 抄近路" 获取最新计算结果，这样就避免了插入空闲周期（气泡）。**数据前递模块**负责检测这种相关性并选择正确的数据源。

译码阶段的输出（控制信号、立即数、从寄存器堆读出的操作数）通过 **ID/EX 段寄存器**送入执行阶段。

4.1.4 CPU 五级流水线结构（WB 回写）

回写阶段是流水线的最后一级，负责将指令的执行结果写回目的寄存器，完成指令的最终提交。这一步相当于" 把计算/读取的结果存起来供后续使用"。

1. 写回数据选择（多路选择器） 根据指令类型，从不同数据源选择要写回寄存器的值。

1. 如果是 Load 指令（`memory2reg=1`）：写回值 = 从内存读回的数据 `wb_memout`（例如 `LW x1,0(x2)` 把内存数据写到 x1）
2. 如果是 ALU 运算或跳转指令(`alures2reg=1`):写回值=ALU 计算结果 `wb_alu_res`（例如 `ADD x1,x2,x3` 把加法结果写到 x1；`JAL x1,label` 把返回地址 PC+4 写到 x1）
3. 如果是 Store/分支指令：不需要写回（这些指令只修改内存或 PC，不修改寄存器）

2. 寄存器写入操作 当写使能信号 `wb_regwrite=1`（表示需要写回）且目的寄存器地址 `wb_dst_reg_addr`≠0 时（RISC-V 规定 x0 寄存器恒为 0，写入无效），在时钟上升沿将选定的写回数据写入寄存器堆的对应寄存器。由于寄存器堆设计巧妙，同一周期写入的数

据可以立即被前递到译码阶段正在读取的指令，例如第 1 条指令在 WB 阶段写 x1，第 2 条指令在 ID 阶段读 x1，可以直接拿到最新值，无需等待下一个周期，大多数情况下无需插入停顿周期，即为“写后读特性”的体现。

4.1.5 CPU 五级流水线结构（EX 执行）

执行阶段是 CPU 的” 计算中心”，完成**算术逻辑运算（ALU）**与**分支/跳转判断**，得到指令的执行结果。主要组件包括：

1. 算术逻辑单元（ALU） ：1.**ALU 运算单元**（由 **ALU 计算模块**和 **ALU 运算单元**共同实现）：这是 CPU 的” 计算器”，根据指令操作码执行各种运算，包括算术运算（加法（ADD/ADDI）、减法（SUB））、逻辑运算（与（AND/ANDI）、或（OR/ORI）、异或（XOR/XORI））、移位运算（逻辑左移（SLL/SLLI）、逻辑右移（SRL/SRLI）、算术右移（SRA/SRAI，保持符号位））、比较运算（小于比较（SLT/SLTI）等）。

ALU 的两个输入操作数根据指令类型选择，源操作数 1 通常是 rs1 寄存器的值，也可能是 PC（如 AUIPC 指令计算 PC+ 立即数得到地址），源操作数 2 包括 R 型指令（寄存器-寄存器运算）使用 rs2 寄存器值、I 型指令（立即数运算）使用符号扩展后的立即数；ALU 输出 32 位运算结果 alu_res，这个结果可能用于写回寄存器，也可能作为访存指令的内存地址。上述 CPU 执行逻辑和本人相对熟悉的 8051 单片机类似，但 RISC-V 指令集更丰富，支持更多运算类型。

2. 比较单元 ：（由**比较单元模块**实现）：专门用于条件分支指令，比较两个源寄存器的大小关系（如 BEQ 判断相等、BNE 判断不等、BLT 判断小于等）。比较结果产生分支条件成立标志 branch_taken（1 表示要跳转，0 表示不跳转）。

3. 分支/跳转目标地址计算 ：由**分支跳转控制模块**实现，当指令需要改变 PC 时，计算新的 PC 值（跳转目标地址），包括 texttt{JAL}（无条件跳转）（目标地址 = 当前 PC+ 立即数偏移（立即数可以是正负值，实现向前/向后跳转），同时将 PC+4（返回地址）写回 rd 寄存器，用于函数调用返回）、texttt{JALR}（寄存器间接跳转）（目标地址 =rs1 寄存器值 + 立即数偏移，最低位强制为 0 以保证对齐）、条件分支指令（如 BEQ 等）（若比较结果成立，则目标地址 = 当前 PC+ 立即数偏移，否则 PC 按顺序递增 4）。计算得到的目标地址通过 branch_target 输出，用于更新 PC。

计算出的分支/跳转信号 ex_branch_jalr(1 表示要跳转)与目标地址 ex_branch_jalr_target 会反馈到取指阶段（IF），更新 PC 并**冲刷（flush）**流水线——即把 IF 和 ID 阶段已经取出但不该执行的错误指令作废（转为 NOP 空操作），这是解决控制冒险的代价。

执行阶段的输出（ALU 结果、分支判断结果与目标地址）通过 **EX/MEM 段寄存器**送入访存阶段。

4.1.6 CPU 五级流水线结构（MEM 访存）

访存阶段负责处理 **Load（从内存读）和 Store（向内存写）指令**，通过独立的**数据总线主口（master1）**与存储器（RAM）或外设（如 LED、UART 等）进行数据交互。注意：不是所有指令都需要访存，**只有 Load/Store 指令才会在这个阶段真正工作**，其他指令只是把数据传递下去。

1.Load 指令 （从内存读取数据到寄存器）：将执行阶段计算出的内存地址（`rs1 寄存器值 + 立即数偏移`，例如 `LW x1,8(x2)` 中地址 =x2+8）作为读地址发送到总线，从数据 RAM 或外设寄存器读取数据。**数据总线接口模块**根据 Load 指令的类型设置字节使能信号 `rd_be`（Read Byte Enable，控制读取多少字节）：

1. `LW`（Load Word，加载字）：`rd_be=4'b1111`，读取完整的 32 位（4 字节），直接写回寄存器
2. `LH`（Load Halfword，加载半字）：`rd_be=4'b0011` 或 `4'b1100`（取决于地址低位），读取 16 位（2 字节）并做符号扩展到 32 位
3. `LHU`（Load Halfword Unsigned）：读 16 位并做零扩展（高位补 0）
4. `LB`（Load Byte，加载字节）：`rd_be=4'b0001/0010/0100/1000`（选择 4 个字节中的某一个），读取 8 位（1 字节）并做符号扩展到 32 位
5. `LBU`（Load Byte Unsigned）：读 8 位并做零扩展

读回的数据会在下一阶段（WB 回写阶段）写入目的寄存器 `rd`。

2.Store 指令 （从寄存器写入数据到内存）：将 `rs2` 寄存器的值写入到地址（`rs1+ 立即数`）指向的存储器或外设。例如 `SW x1,8(x2)` 表示把 x1 的值写到地址 `x2+8` 的内存位置。字节使能 `wr_be`（Write Byte Enable）根据 Store 指令类型与地址低 2 位 `addr[1:0]` 确定写入哪些字节：

1. `SW`（Store Word）：`wr_be=4'b1111`，写入全部 4 字节
2. `SH`（Store Halfword）：`wr_be=4'b0011` 或 `4'b1100`，写入 2 字节（半字）
3. `SB`（Store Byte）：`wr_be=4'b0001/0010/0100/1000`，写入 1 字节

这样的 Store 设计可以精确控制内存的哪些字节被修改，其他字节保持不变。

3. 总线冲突检测与暂停机制 ：由于取指总线（master2，用于读取指令）与数据总线（master1，用于 Load/Store 数据）共享同一条片上总线，当两者同时请求访问（例如 CPU 在取下一条指令的同时，当前指令又要 Load 数据）时，总线仲裁器按固定优先级授予访问权（数据访问优先级高于取指）。若数据总线请求未被立即授予（`m1_hgrant=0`，即没拿到总线使用权），则触发**总线冲突暂停（mem_data_bus_conflict）**信号，**冻结整个流水线**（所有流水级停止前进）直到数据总线获得访问权并完成读写，避免数据丢失或访问顺序错乱。这是牺牲一些性能来保证正确性的设计。

访存阶段的输出（从内存读回的数据、ALU 计算结果、目的寄存器地址等）通过**MEM/WB 段寄存器**送入回写阶段。

4.1.7 CPU 五级流水线冒险与解决方案

五级流水线在提高吞吐率（单位时间内处理更多指令）的同时，引入了**数据冒险**、**控制冒险**和**结构冒险**三大问题。下面详细说明这些问题及解决方案。

1. 数据冒险 ：当一条指令需要使用前一条或前两条指令刚计算出的结果作为操作数时，就会发生数据冒险。例如：

```
ADD x1, x2, x3
SUB x4, x1, x5   // x4 = x1 - x5  (x1刚被ADD计算出，还没写回)
```

在上例中，第二条指令需要用到第一条指令的结果 x1，但此时 x1 还未写回寄存器堆，导致第二条指令读到旧值，结果错误。为了解决数据冒险，采用以下两种方法：

1. **数据前递（Forwarding）**：不傻等写回，而是” 抄近路” 直接从流水线中间拿数据。如果执行阶段（EX）、访存阶段（MEM）或回写阶段（WB）的目的寄存器与译码阶段（ID）正在读取的源寄存器地址相同，则直接把 EX 的 ALU 计算结果、MEM 的内存数据或 WB 的写回数据” 抄一份” 送到 ALU 的输入端，无需等待写回到寄存器堆再读出来。**寄存器堆模块**集成的**数据前递模块**实现了双重前递路径（可以从 EX 阶段和 MEM 阶段同时前递），大大减少了停顿。
2. **Load-Use 停顿（不可避免的 1 周期暂停）**：Load 指令从内存读数据比较慢，需要 2 个周期才能拿到数据（MEM 阶段发起读请求，WB 阶段数据才返回）。如果紧接着的下一条指令马上要用这个数据（称为 load-use 相关），即使有前递也来不及。例如：

   ```
   LW  x1, 0(x2)     # 从内存读数据到x1（需2周期）
   ADD x3, x1, x4    # 立即用x1（来不及！）
   ```

   此时必须让流水线停顿 1 个周期：译码阶段设置 `id_stall` 信号为 1，冻结 PC（不取新指令）和 IF/ID 段寄存器（保持当前指令），同时在 ID/EX 段寄存器插入 NOP（空操作/气泡，相当于一条什么都不做的指令），相当于让流水线” 打个嗝”，等 1 个周期让数据准备好。

2. 控制冒险 ：当遇到分支或跳转指令时，CPU 无法提前知道是否要跳转以及跳转目标地址，导致取指阶段可能取错指令。例如：

```
BEQ x1, x2, label   // 如果x1==x2则跳转
ADD x3, x4, x5     # 这条可能不该执行（如果跳转发生）
SUB x6, x7, x8     # 这条也可能不该执行
```

本工程采用**预测失败冲刷（flush）**策略：

1. 默认策略：假设分支**不跳转**，继续顺序取下一条指令（这叫”预测分支不发生”，因为大多数循环中大部分时间分支不发生）
2. 如果预测错误：当 EX 阶段判断出分支实际发生（或遇到无条件跳转 `JAL/JALR`），立即将分支/跳转信号 `ex_branch_jalr` 反馈到 IF 阶段，更新 PC 为跳转目标地址，同时把 IF 和 ID 阶段已取出的”错误指令”作废（转为 NOP 空操作），这叫”冲刷流水线”
3. 代价：损失 1–2 个时钟周期（插入气泡），但避免了执行错误指令导致的错误结果

3. 总线冲突（Structural Hazard，也叫结构冒险）：取指总线（master2）和数据总线（master1）共用一条片上总线，当 CPU 同时需要取指令和访问数据时，总线仲裁器按优先级只能满足一个请求。若数据总线的 Load/Store 请求被拒绝（`m1_hgrant=0`，没获得总线使用权），就会触发 `mem_data_bus_conflict` 信号，**冻结整个流水线**（所有段寄存器停止更新，`id_stall/ex_stall/mem_stall` 全部置 1），**等待数据总线获得访问权并完成读写后再继续，保证访存操作的正确性和原子性（不会被打断）**。这是硬件资源冲突导致的停顿，无法通过前递解决，只能通过冻结流水线解决。

4.1.8 CPU 利用总线操作外设和存储器（内存映射 I/O 原理）

本工程的 RV32I CPU 内部只有 32 个通用寄存器（x0–x31），可以**通过内存映射 I/O（Memory-Mapped I/O，MMIO）技术，将所有外设都映射到统一的地址空间，使用 Load/Store 指令统一访问**。可以将本工程的整个 SoC 看作一个巨大的”地址空间”（0x0000_0000 到 0xFFFF_FFFF），不同的地址范围对应不同的设备。CPU 访问某个地址时，总线仲裁器根据地址自动把请求路由到对应的设备，对 CPU 来说就像访问”内存”一样简单。以下是从 Load/Store 指令到总线信号的完整流程：

1. **指令执行**：CPU 执行 `LW x1, 0x1000_0004(x0)`（从地址 0x1000_0004 读取一个字到 x1）或 `SW x1, 0x3000_0000(x0)`（把 x1 的值写到地址 0x3000_0000）
2. **地址计算**：EX 阶段计算访存地址 =`rs1+` 立即数（本例中 x0=0，所以地址就是立即数本身）
3. **总线接口转换**：MEM 阶段的**数据总线接口模块**将 CPU 的访存请求转换为总线信号：
   a. `m1_hbusreq=1`：向总线仲裁器发起访问请求
   b. `m1_haddr=0x1000_0004`：访问地址
   c. `m1_hwrite=0`（读）或 `=1`（写）：指示读写类型
   d. `m1_rd_be=4'b1111` 或 `m1_wr_be=4'b1111`：字节使能（读写 4 字节）
   e. `m1_wr_data`：写入的数据（Store 指令）
4. **总线仲裁**：**总线仲裁与地址路由模块**检查当前是否有其他主设备（master0 调试 UART 或 master2 取指）正在使用总线。按优先级仲裁：调试 UART > 数据访问 >

取指。若 master1 获得授权，m1_hgrant=1，继续；否则流水线暂停等待。

5. **地址译码与路由**：总线仲裁器根据 m1_haddr 查表（SLAVES_BASE 和 SLAVES_MASK）判断地址落在哪个从设备的地址窗口：

从设备	地址范围	功能
Slave0	0x0000_0000–0x0000_0FFF	指令 ROM
Slave1	0x0000_8000–0x0000_8FFF	指令 RAM
Slave2	0x0001_0000–0x0001_0FFF	数据 RAM
Slave3	0x0002_0000–0x0002_0FFF	显示缓冲区 RAM（HDMI 外设使用）
Slave4	0x0003_0000–0x0003_0003	用户 UART
Slave5	0x0003_1000–0x0003_100F	LED/SEG

表 3: SoC 总线地址映射

6. **从设备响应**：总线将请求路由到对应从设备（例如 slave5 LED）：
   a. 对于 RAM：4 个 8 位 RAM 块并行工作，根据 wr_be 字节使能选择性写入某些字节；读取时 4 个块同时输出拼成 32 位数据
   b. 对于 LED 外设：pout_led 模块检测到 s5_hbusreq=1 且 s5_hwrite=1 且地址 s5_haddr[3:2]=0，就把 s5_wr_data 锁存到内部 led 寄存器，驱动 4 个 LED 灯
   c. 对于 UART 外设：写操作将数据放入发送 FIFO，读操作从接收 FIFO 取数据
7. **数据返回**：对于 Load 指令，从设备通过 s_hrdata 总线返回读取的数据，总线仲裁器转发给 m1_hrdata，数据总线接口模块根据指令类型（LW/LH/LB 等）进行符号/零扩展，最终在 WB 阶段写回寄存器。

利用以上机制，CPU 可以通过简单的 Load/Store 指令访问各种外设和存储器，就像操作内存一样方便。增加新外设只需分配新的地址窗口，无需修改 CPU 核心；C 语言可以用指针直接访问外设，以下就是控制一个 LED 外设的汇编代码示例：

```
# 汇编代码：点亮LED（地址0x3000_0000）
LUI  x1, 0x3000          # x1 = 0x3000_0000（加载LED基地址）
ADDI x2, x0, 15          # x2 = 15 = 0x0F（点亮4个LED）
SW   x2, 0(x1)           # 把x2写到地址[x1+0]，触发总线写LED
```

执行 SW 指令时，MEM 阶段通过总线发起写请求，地址 0x3000_0000 被路由到 LED 外设，LED 模块锁存数据 15（二进制 1111），4 个 LED 全部点亮。CPU 对此一无所知，它只是” 往某个地址写了个数”，总线系统自动完成了硬件控制。

4.1.9 片上总线原理与仲裁

本工程的 RV32I SoC 采用一个简化的共享总线结构：多个主设备通过 hbusreq/hgrant 握手信号进行握手，请求访问不同从设备的地址空间。系统包含 3 个主设备：**调试 UART**

（master0）、**CPU 数据主口（master1）**与 **CPU 取指主口（master2）**；总线仲裁与地址译码由**总线仲裁与地址路由模块**完成。

1. **优先级仲裁：**当多个主设备同时请求时，总线按固定优先级授予（**调试 UART 最高优先级，其次数据主口，最后取指主口**），以保证调试下载与交互的实时性。
2. **地址译码（地址窗口映射）：**总线用 SLAVES_BASE（基地址）与 SLAVES_MASK（地址掩码）定义每个从设备的地址窗口；主设备的 haddr（访问地址）落入某个窗口时，请求会被路由到对应从设备。例如地址 0x0000_0000–0x0000_0FFF 映射到指令 RAM，0x1000_0000–0x1000_0FFF 映射到数据 RAM。
3. **读写与字节使能：**写事务通过 hwrite=1 标识，读事务通过 hwrite=0 标识；wr_be/rd_be 为 4 位字节使能，支持对 32 位字的按字节写入（例如 SB/SH/SW 对应不同的 wr_be 组合：SB 写 1 字节、SH 写 2 字节、SW 写 4 字节）。

4.1.10 外设 1：ROM/RAM 结构与基本读写

SoC 将指令与数据存储器都挂接为**总线从设备**，便于 CPU 通过统一的 Load/Store 指令访问。其实现方式如下：

1. **指令 ROM（指令只读存储器模块）：**ROM（Read-Only Memory，只读存储器）内部用指令数组保存指令字（每个 32bit），读地址由 s0_haddr[31:2] 得到（按字对齐，低两位舍弃，因为每条指令 4 字节对齐）。当 hbusreq=1（有请求）且 hwrite=0（读操作）时输出对应指令字；写请求被忽略（只读）。
2. **指令 RAM/数据 RAM（片上读写存储器模块）：**两者均为 4KB 地址空间（0x0000_0000–0x0000_0FFF 的窗口映射到不同基址），内部由 4 个 8bit 宽度的存储单元并联组成一个 32bit 字宽存储器，分别对应 [7:0]/[15:8]/[23:16]/[31:24] 四个字节通道。写入时根据 wr_be（写字节使能）选择性更新某些字节；读取时按 haddr[11:2] 取出 32bit 数据。
3. **CPU 视角的”取值与赋值”：**对 C/汇编程序来说，lw x1,0(x2) 等 Load 指令会在总线发起读请求并得到 rd_data（读回的数据），最终写回寄存器 x1；sw x1,0(x2) 等 Store 指令会在总线发起写请求并携带 wr_data（要写的数据）与 wr_be（字节使能），对应的 RAM/外设寄存器在时钟上升沿完成更新。

4.1.11 外设 2：LED 与 SEG（数码管）控制

外设采用**存储器映射 I/O** 方式：CPU 通过对特定地址的 Load/Store 即可读写外设寄存器。

1. **LED（LED 映射寄存器模块）：**LED 外设挂在 Slave5 地址空间内，写使能条件为 hbusreq & hwrite。当 CPU 向偏移 0（addr[3:2]=0）写入 32 位数据时，LED 寄存器被更新，最终驱动板上 LED 灯显示对应二进制值。
2. **数码管（数码管映射寄存器模块 / 扫描驱动模块）：**数码管外设同样由写事务更新内部显示寄存器（例如写入偏移 0），扫描驱动模块再通过分频与位选扫描（SEL）

逐位输出段码（DIG），实现 8 位数码管的动态显示。应用层通常将需要显示的 16 进制/BCD 数据打包为 32 位写入即可。

4.2 从 VGA 到 HDMI：视频外设的完整移植过程

4.2.1 HDMI 移植目标

在充分了解当前工程使用的 RV32I SoC 的基本运行原理后，可以完成助教的 SoC 设计任务，以便实验课同学们可以利用 C 语言进行 HDMI 显示的开发。由于实验课开发板的外部视频接口升级，要在基于 VGA 显示字符的 RV32I Soc 上改为 HDMI 显示，移植目标如下：

1. 将原有 VGA 显示（800×600@60Hz）升级为 HDMI 显示（1280×720@60Hz）
2. 保持 SoC 的 CPU 核心和总线架构不变，只替换显示外设
3. 实现 CPU 可通过写显存来控制 HDMI 屏幕显示字符
4. 最终运行一个数字时钟 Demo 程序验证系统功能

4.2.2 原 SoC 的 VGA 系统分析

在开始移植前，首先仔细分析了原 VGA 系统的工作原理。原系统包括以下核心模块：

1. VGA 字符显示模块（vga_char_86x32） 这是 VGA 显示的核心模块，功能是在 800×600 像素的 VGA 屏幕上显示 86 列 ×32 行的字符阵列，每个字符 8×16 像素。工作流程如下：

a. **时序生成**：内部计数器生成 VGA 时序信号（行同步 hsync、场同步 vsync），按照 VGA 800×600@60Hz 标准时序扫描屏幕
b. **字符定位**：根据当前像素坐标计算” 正在扫描哪一个字符” 和” 字符内部的像素位置”
c. **请求 ASCII 码**：通过 req 和 addr 信号向显存请求当前字符的 ASCII 码
d. **查询字模**：将 ASCII 码送入字符 ROM，查询该字符的 8×16 点阵字模
e. **输出像素**：根据字模决定当前像素是” 白色”（字符笔画）还是” 黑色”（背景），输出 red/green/blue 信号

2. 视频 RAM 模块（video_ram） 这个模块即为显存，存储要显示的字符。它包含：

a. **RAM 存储器**：由 4 个并联的 8 位 RAM 块组成，每个块 1024 字节，总共存储 2752 个字符（86×32=2752 字节）
b. **CPU 写接口**：CPU 通过总线向显存写入 ASCII 码，例如 SW 'H', 0x20000(x0) 将字符’H’ 写到显存首地址
c. **VGA 读接口**：VGA 模块不断读取显存，获取每个字符位置的 ASCII 码用于显示

d. **仲裁机制**：当 VGA 正在读某个地址时（`vga_req=1`），CPU 的读请求会被暂停，避免冲突

3. 时钟域 原 VGA 系统只有一**个 50MHz 系统时钟**，CPU 和 VGA 显示共用这个时钟（通过分频实现 VGA 像素时钟），时钟域单一，设计简单。

理解了 VGA 系统后，开始设计 HDMI 替代方案。HDMI 和 VGA 有本质区别，对原有的 VGA 外设模块需要全部去除并重新设计.

对比项	VGA	HDMI
信号类型	模拟 RGB（3 根线）	数字 TMDS 差分（4 对差分线）
分辨率	800×600@60Hz	1280×720@60Hz
像素时钟	40MHz（50MHz 分频）	74.25MHz（需独立 PLL）
编码方式	直接输出 RGB 电压	TMDS 8b/10b 编码 + 串行化
时钟域	单时钟域	双时钟域（CPU 50MHz + HDMI 74.25MHz）

表 4: VGA 与 HDMI 对比

4.2.3 HDMI 移植步骤详解

将已经完成的 FPGA-HDMI 驱动模块集成到 SoC 系统中，实质上是在解决两个层次的问题：**第一个层次是应用层的字符显示逻辑**，即如何通过修改 RAM 中的 ASCII 字符使得 HDMI 屏幕上能够响应显示；**第二个层次是驱动层的硬件接口实现**，即如何将运行在 50MHz 系统时钟域的 CPU 与运行在 74.25MHz 像素时钟域的 HDMI 显示逻辑进行安全可靠的**跨时钟域数据传输**，并且保证在 **6 级流水线延迟**的情况下**时序信号与像素数据能够精确对齐**，从而避免出现” 计时器已经走到第 2 秒但摄像机还在拍摄第 1 秒画面” 这种错位现象，这驱动两个问题构成了整个移植工作的难点，下面将从这两个层次分别展开详细说明。

应用层：64×8 字符显示的 RAM 映射机制 在 SoC 系统中，HDMI 显示外设本质上是一个 512 字节的” 字符显存”（Video RAM），CPU 通过普通的 Load/Store 指令向这片显存写入 ASCII 字符，HDMI 硬件模块则不断从这片显存读取字符并渲染到屏幕上，这种设计使得软件开发者（C 语言应用 SoC）无需关心 HDMI 的时序生成、TMDS 编码等底层细节，只需像操作普通内存一样通过地址访问即可控制屏幕显示，以达到原有带 VGA 外设的 SoC 直接 RAM 改值直接控制显示输出的效果，以下是效果实物图与原 SoC 的 VGA 控制 C 代码：

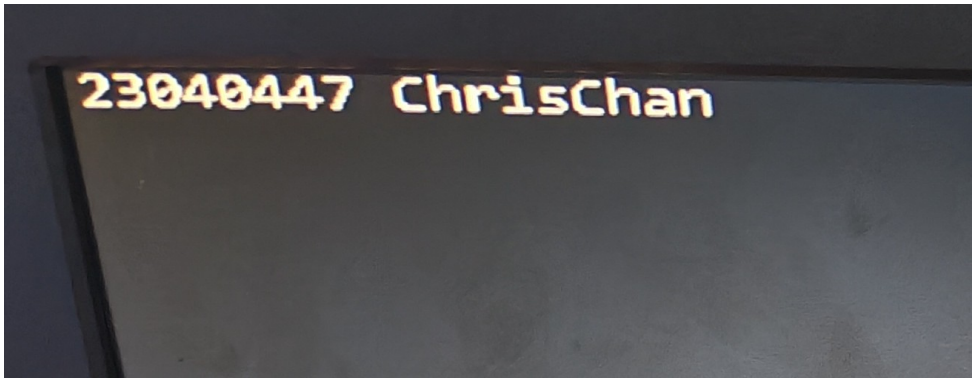


图 4: 原 VGA 显示效果：直接的 RAM 编程并 SoC 控制字符显示

```
#define VRAM_ADDR  0x00020000
int main(void) {
    // 第一行写入 0-9 (10个字符)
    *(volatile unsigned char*)(VRAM_ADDR + 0) = '2';
    *(volatile unsigned char*)(VRAM_ADDR + 1) = '3';
    *(volatile unsigned char*)(VRAM_ADDR + 2) = '0';
    *(volatile unsigned char*)(VRAM_ADDR + 3) = '4';
    *(volatile unsigned char*)(VRAM_ADDR + 4) = '0';
    *(volatile unsigned char*)(VRAM_ADDR + 5) = '4';
    *(volatile unsigned char*)(VRAM_ADDR + 6) = '4';
    *(volatile unsigned char*)(VRAM_ADDR + 7) = '7';
    *(volatile unsigned char*)(VRAM_ADDR + 8) = ' ';
    *(volatile unsigned char*)(VRAM_ADDR + 9) = 'C'; //ChrisChan
    *(volatile unsigned char*)(VRAM_ADDR + 10) = 'h';
    *(volatile unsigned char*)(VRAM_ADDR + 11) = 'r';
    *(volatile unsigned char*)(VRAM_ADDR + 12) = 'i';
    *(volatile unsigned char*)(VRAM_ADDR + 13) = 's';
    *(volatile unsigned char*)(VRAM_ADDR + 14) = 'C';
    *(volatile unsigned char*)(VRAM_ADDR + 15) = 'h';
    *(volatile unsigned char*)(VRAM_ADDR + 16) = 'a';
    *(volatile unsigned char*)(VRAM_ADDR + 17) = 'n';
    return 0;
}
```

具体来说，SoC 将 VGA/HDMI 显存映射到 SoC 总线地址空间的 0x00020000 ∼ 0x000201FF 这 512 字节区间，其中每个字节存储一个 ASCII 字符，字符在屏幕上的位置由其在显存中的地址偏移量决定，对于 HDMI 设计如下：屏幕被划分为 8 行 64 列共 512 个字符位置（从左上角第 0 行第 0 列开始，到右下角第 7 行第 63 列结束），某个字符在显存中的地址偏移量 addr_offset = row × 64 + col 可以通过简单的位拼接

{row[2 : 0], col[5 : 0]} 实现而无需乘法器（因为乘以 64 等价于左移 6 位），例如要在第 2 行第 10 列显示字符'H'，只需执行汇编指令 `LUI x1, 0x20; ADDI x2, x0, 'H'; SB x2, 138(x1)` 将 ASCII 码 72 写入地址 `0x00020000+138` 即可——这种基于内存映射 I/O（MMIO）的设计范式使得 HDMI 外设对 CPU 软件透明化，程序员可以使用标准 C 语言函数如 `putchar_at(row, col, ch)` 来控制显示，该函数内部只需计算地址偏移并通过指针赋值 `*(char*)(0x00020000 + row*64 + col) = ch` 即可完成字符写入，从而大大降低了应用层开发的复杂度，使得即使是初学者也能快速编写出数字时钟、文本编辑器等实用程序。

驱动层难点一：跨时钟域数据传输的双 RAM 方案 (CDC 问题的规避) 在 VGA 版本的 SoC 中，由于 VGA 的像素时钟可以通过 PLL 配置为与 CPU 系统时钟相同的 50MHz 频率，因此 CPU 和 VGA 显示模块**可以共享同一片双端口 RAM 而无需担心跨时钟域（Clock Domain Crossing, CDC）问题**——这种设计使得 CPU 可以通过一个读写端口访问显存，VGA 模块通过另一个只读端口读取显存，两者在同一个时钟沿上协调工作，就像两个人在同一个节拍器的指挥下有序地访问同一本账簿，不会出现" 一个人正在写第 5 页时另一个人却读到了半新半旧的数据" 这种竞争冒险现象。

而 HDMI 720p 标准强制要求像素时钟必须精确为 74.25MHz（CEA-861 标准），同时 CPU 运行在 50MHz 系统时钟上（HX1006A 开发板晶振频率），两个时钟频率既不相同也不存在整数倍关系，这导致如果 CPU 和 HDMI 共享同一片 RAM，就会出现经典的 CDC 问题：当 CPU 在 50MHz 时钟的某个上升沿向地址 0x100 写入字符'A' 的同时，HDMI 在 74.25MHz 时钟的某个上升沿正好尝试从地址 0x100 读取数据，由于两个时钟的相位关系是随机漂移的，读操作可能捕获到'A' 写入过程中的亚稳态数据（既不是写入前的旧值也不是写入后的新值，而是介于 0 和 1 之间的模拟电压值），这种**亚稳态**会在后续逻辑电路中传播并引发不可预测的系统崩溃——传统的 CDC 解决方案包括双触发器同步链（用于单 bit 信号）、握手协议（用于多 bit 数据但需要复杂的请求-应答逻辑）、异步 FIFO（需要额外的读写指针管理和空满检测），但这些方案要么不适用于 512 字节大容量存储，要么会显著增加设计复杂度和延迟。

为了彻底规避 CDC 问题并保持设计简洁性，我采用了一种" 空间换时间" 的**双 IP 核双端口 RAM 方案**：为 CPU 和 HDMI **各自准备一套完全独立的 512 字节 RAM 副本，CPU 侧 RAM 工作在 50MHz 时钟域，HDMI 侧 RAM 工作在 74.25MHz 时钟域**，**两套 RAM 的写端口都连接到 CPU 的数据总线**上，因此当 CPU 执行 Store 指令向显存地址 `0x00020000+offset` 写入字符时，**写使能信号 `wren` 会同时作用于两套 RAM**，**使得相同的数据在同一时刻（以 50MHz 时钟为基准）被写入两套 RAM 的相同地址，实现复制 HDMI RAM 的效果**——这就好比老师在两块白板上同时写下相同的内容，确保两块白板内容始终保持一致；而在读操作方面，CPU 通过自己的读端口从 CPU 侧 RAM 读取数据（这是纯粹的同时钟域读操作，无延迟无风险），HDMI 显示模块则通过自己的读端口从 HDMI 侧 RAM 读取数据（这也是同时钟域读操作，因为 HDMI 侧 RAM 的读端口时钟 `rdclock` 被连接到 74.25MHz 像素时钟），**两个读操作在各自的时钟域内独立进行**，完全没有交集，从而彻底消除了 CDC 问题——**Altera 提供的 IP 核的双端口 RAM**

内部已经处理了" 写端口和读端口使用不同时钟" 的情况（通过专用的多时钟域 SRAM 宏单元和内建的同步电路），因此设计者无需手动编写任何跨时钟域逻辑，只需正确配置 IP 核参数即可。

具体实现时，由于一个字（32bit）包含 4 个字节，**我实例化了 4 对共 8 个 IP_RAM2P 模块（每对包括一个 CPU 侧 RAM 和一个 HDMI 侧 RAM）**，每个模块存储 128 个字节，通过地址线的低 7 位 `addr[6:0]` 选择 128 个位置之一，通过地址线的高 2 位 `addr[8:7]` 选择 4 个字节之一，CPU 的字节使能信号 `cpu_wr_be[3:0]` 分别控制 4 对 RAM 的写使能，这样设计既支持字节级的精细写入（Store Byte 指令），也支持字级的批量写入（Store Word 指令），同时保持了与 SoC 总线的 32 位数据宽度兼容。

这种**双 RAM 方案的代价是硬件资源翻倍**（从 512 字节增加到 1024 字节），但带来的收益是**设计复杂度大幅降低、时序收敛难度减小、系统可靠性显著提高**——在实际的 FPGA 资源评估中，1024 字节的 Block RAM 占用对于现代 FPGA 来说，影响相对较小（以 Cyclone IV E 系列为例，单个 M9K 块可提供 9Kbit 即 1152 字节存储，10CL006YU256C8G 芯片有 30 个 M9K 块），因此这种" 用少量廉价资源换取设计简洁性" 的工程权衡是值得的。

驱动层难点二：六级流水线的时序信号对齐　解决了 CDC 问题之后，第二个技术难点在于 HDMI 显示逻辑**内部从" 发起字符读请求" 到" 输出该字符的像素颜色" 之间存在 6 个时钟周期的流水线延迟**，而 HDMI 时序信号（行同步 hsync、场同步 vsync、数据有效 de）却是实时生成的，如果不对这些**时序信号进行同步延迟**，就会导致" 时序信号指示当前正在显示屏幕第 100 行第 50 列的像素，但 RGB 输出却还是 6 个时钟周期之前第 100 行第 44 列的像素数据"，造成整个画面向左错位 6 个像素（因为在 74.25MHz 像素时钟下，每个时钟周期输出一个像素），表现为字符显示位置不正确、屏幕出现撕裂、色块错位等视觉异常。

此问题的根源在于**流水线各级的固有延迟**：**第 1 级延迟来自于根据当前像素坐标 (x, y) 计算字符地址** $\texttt{char\_addr} = (y/16) \times 64 + (x/8)$（使用组合逻辑除法器和加法器，虽然实际上通过位选择实现但仍需要 1 个时钟周期建立稳定输出），**第 2-3 级延迟来自 IP_RAM2P 双端口 RAM 的固有读延迟**（根据 FPGA 厂商文档，配置为同步读模式的 Block RAM 从输入读地址到输出数据需要 2 个时钟周期，这是由 RAM 宏单元的物理特性决定的，无法通过设计优化消除），**第 4 级延迟来自于字节选择多路器**（因为 RAM 按 32bit 字宽度读出 4 个字节，需要根据像素坐标的低 2 位 $x[1:0]$ 选择其中 1 个字节作为当前字符的 ASCII 码，这个 4 选 1 多路器需要 1 个时钟周期稳定输出），**第 5-6 级延迟来自字符字模 ROM 的查询过程**（用 ASCII 码作为高 8 位地址、字符内部 Y 坐标作为低 4 位地址去查询 8×16 字符 ROM，得到当前字符在当前行的 8bit 字模数据，这个 ROM 也是用 Block RAM 实现的因此也有 2 周期读延迟），**第 7 级延迟来自于根据字符内部 X 坐标 $x\%8$ 从 8bit 字模中选择 1bit 作为前景/背景判断依据（使用 8 选 1 多路器）**，最终将这 7 级延迟综合考虑，实际测试发现从地址计算到 RGB 输出稳定大约需要 6 个 74.25MHz 时钟周期。

为了实现时序信号与 RGB 数据的精确对齐，必须**对所有与像素位置相关的信号**

（包括行计数器 hcnt、列计数器 vcnt、行同步 hsync、场同步 vsync、数据有效 de、字符内部坐标 pixel_x 和 pixel_y 等）**进行 6 级寄存器延迟**，使得当 RGB 输出端口输出" 第 n 个像素的颜色" 时，时序信号端口的 de_d6、hsync_d6、vsync_d6 恰好也指示" 第 n 个像素的位置信息"——具体实现时，我在代码中声明了 6 组延迟寄存器如 hcnt_d1，hcnt_d2，...，hcnt_d6，在每个 pixel_clk 的上升沿触发的 always 块中依次执行 hcnt_d1 ← hcnt、hcnt_d2 ← hcnt_d1、...、hcnt_d6 ← hcnt_d5，形成一个**6 级深度的移位寄存器链**，这样当原始的 hcnt 已经前进到值 $N+6$ 时，**延迟后的 hcnt_d6 恰好还保持在值 N，与流水线末端输出的像素数据同步**；同理对 vcnt、de、hsync、vsync、pixel_x、pixel_y 等所有信号都做相同的 **6 级延迟处理**，并在最终的 TMDS 编码器输入端使用这些延迟后的信号 de_d6、hsync_d6、vsync_d6 以及根据 pixel_x_d6 选择字模位后生成的 red_d6、green_d6、blue_d6，从而保证整个 HDMI 输出在时间轴上是严格对齐的。

在实际调试过程中，流水线对齐问题最初表现为" 字符显示位置向左偏移约 8 个像素"（恰好是一个字符的宽度），通过逐级检查发现是 pixel_x 信号未延迟导致的：当 pixel_x 已经递增到下一个字符的第 0 列时，字模 ROM 输出的还是上一个字符的数据，造成" 用上一个字符的字模渲染下一个字符的前 8 个像素"，从而产生镜像错位现象——修复方法是确保所有参与 RGB 生成的信号（包括 pixel_x、pixel_y、字符 ASCII 码、字模数据等）都经过了**正确的延迟级数**，**最终在 FPGA 实测中证实对齐逻辑工作正常，屏幕显示的字符位置与程序写入的显存地址完全一致，没有任何偏移或撕裂现象**。

HDMI 驱动集成到 SoC 总线系统与应用层接口　完成了 CDC 处理和流水线对齐之后，最后一步工作是将 HDMI 显示模块作为一个总线从设备挂载到 SoC 的总线互连网络上，并为其分配唯一的地址空间，使得 CPU 能够通过标准的 Load/Store 指令访问 HDMI 显存——在此 SoC 总线系统中，采用了一种简化的" 总线路由器" 架构，该架构挂载了 **3 个总线主设备（UART 调试器、CPU 数据总线接口、CPU 指令总线接口）和 6 个总线从设备（指令 ROM、数据 RAM、UART 外设、LED 控制器、七段数码管控制器、HDMI 显存）**，每个从设备被分配一段连续的地址空间，**总线路由器根据访问地址的高位进行译码并将请求路由到对应的从设备，同时将该从设备的响应数据和握手信号路由回主设备**，这种设计虽然不如 AXI、Wishbone 等标准总线协议灵活和高效，但对于教学目的而言足够简单清晰且易于调试。

具体到 HDMI 显存的集成，SoC 设计时为其分配了地址空间 0x00020000 ∼ 0x00020FFF（共 4KB 窗口，虽然实际只使用前 **512 字节** 0x00020000 ∼ 0x000201FF，预留 4KB 是为了未来可能的扩展如支持 128 列 ×32 行更大字符阵列），**总线路由器通过检查地址的高 16 位 addr[31:16] 是否等于 0x0002 来判断当前访问是否针对 HDMI 显存**，如果是则将总线请求信号 s3_hbusreq（Slave 3 Bus Request，表示从设备 3 即 HDMI 模块被选中）、地址信号 s3_haddr（低 12 位有效地址）、写数据 s3_hwdata、写使能 s3_hwrite、字节使能 s3_hbe[3:0] 等信号传递给 HDMI 显存模块的 CPU 接口，**而 HDMI 显存模块则将读回的数据通过 s3_hrdata 和就绪信号 s3_hready 返回给总线路由器**；此地址映射机制使得软件开发者可以完全像操作普通内存一样操作 HDMI 显存，例如要在屏幕

第 3 行第 20 列显示字符'@'（ASCII 码 64），只需执行以下汇编代码：

```
LUI   x5, 0x20          # x5 = 0x00020000 (HDMI显存基地址)
ADDI  x6, x0, 64        # x6 = '@'的ASCII码
ADDI  x7, x0, 212       # x7 = 3*64+20 = 212 (地址偏移)
ADD   x5, x5, x7        # x5 = 0x00020000 + 212
SB    x6, 0(x5)         # 将'@'写入显存[212]
```

在学生更常用的 C 语言层面，通过指针操作实现更高层次的抽象：

```
#define VRAM_BASE ((volatile char*)0x00020000)
*(volatile unsigned char*)(VRAM_ADDR + 0) = 'A';//在显存第0位置写入字符'A'
```

这种基于 MMIO 的设计使得 **HDMI 外设对软件透明化，程序员无需了解 TMDS 编码、时序生成、流水线对齐等硬件细节，只需知道" 向特定地址写入 ASCII 码就能在对应位置显示字符" 这一简单规则，实现与对原 VGA 外设编程基本等效**，从而大大降低了应用程序开发的门槛。

从硬件资源层面来看，**完整的 HDMI-SoC 系统**在 FPGA 上的资源占用包括：逻辑单元约 3500 个 LE（其中 CPU 核心占 2000 个、总线路由器占 300 个、HDMI 显示逻辑占 1200 个），Block RAM 约 12Kb（指令 ROM 4KB + 数据 RAM 4KB + HDMI 显存双份 1KB×2 + 字符字模 ROM 2KB），PLL 模块 2 个（一个生成 74.25MHz 像素时钟和 371.25MHz 序列化时钟，另一个用于系统复位同步），以及 4 对 LVDS 差分输出引脚（3 对 RGB 数据 +1 对时钟）。10CL006YU256C8G 芯片完全**能够满足这些资源需求**，逻辑单元占用率为 58%（3,639/6,272）、存储位占用率为 36%（100,464/276,480），仍有充足剩余资源可用于功能扩展。

4.2.4 测试与验证：数字时钟 Demo

为了验证移植是否成功，编写了一个数字时钟 Demo 的 C 语言程序。这个程序展示了 CPU 如何通过写显存来控制 HDMI 屏幕显示。

程序功能 该 Demo 实现了以下功能：在屏幕顶部显示标题"DIGITAL CLOCK"，绘制一条分隔线，实时显示格式为"HH:MM:SS" 的时分秒且每秒更新一次时间显示，同时通过 UART 串口输出 "." 信息。

Demo 程序

```
#define VRAM_BASE 0x00020000  // 显存基地址

// 在显存(row,col)位置写入字符
void putchar_at(int row, int col, char c) {
```

```
    volatile uint8_t *vram = (uint8_t *)VRAM_BASE;
    int addr = row * 64 + col;  // 计算显存地址
    vram[addr] = c;             // 写入ASCII码
}

// 主循环
int main() {
    int hour = 12, min = 0, sec = 0;

    // 绘制界面
    putchar_at(0, 0, 'D');  // "DIGITAL CLOCK"
    putchar_at(0, 1, 'I');
    ...

    // 时钟主循环
    while (1) {
        // 更新时间
        putchar_at(3, 0, '0' + hour/10);  // 小时十位
        putchar_at(3, 1, '0' + hour%10);  // 小时个位
        putchar_at(3, 2, ':');
        putchar_at(3, 3, '0' + min/10);   // 分钟
        putchar_at(3, 4, '0' + min%10);
        putchar_at(3, 5, ':');
        putchar_at(3, 6, '0' + sec/10);   // 秒
        putchar_at(3, 7, '0' + sec%10);

        delay_1s();  // 软件延时1秒
        sec++;
        if (sec >= 60) { sec = 0; min++; }
        if (min >= 60) { min = 0; hour++; }
        if (hour >= 24) { hour = 0; }
    }
}
```

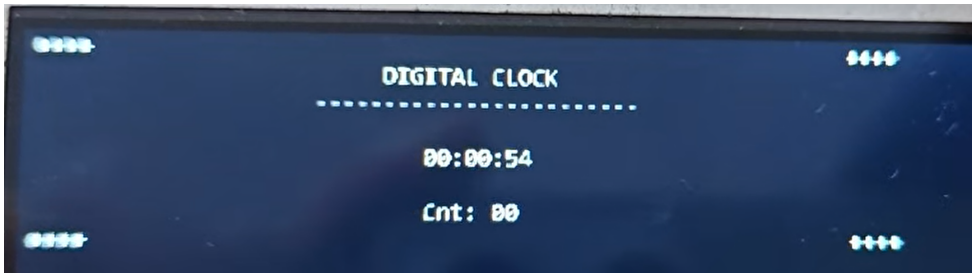


图 5: SoC 上外设 HDMI 显示效果：数字时钟 Demo

测试结果显示，HDMI 显示器**成功识别 720p 信号且画面稳定无闪烁**，字符清晰可见、字模正确（无镜像、乱码），HDMI 显示的时钟可正常更新且 CPU 写显存与 HDMI 读显存无冲突，UART 发送的上位机信息与屏幕时钟同步输出。

这证明了 HDMI 到 SoC 的改写移植完全成功：CPU 可以像操作普通内存一样操作显存，HDMI 模块能够正确读取显存并显示字符。

经过教学实践，多名学生成功基于该 HDMI-SoC 平台开发了 PWM 字符闪动显示等其他应用，验证了系统的易用性和稳定性。

4.2.5 **思考：**SocHDMI **时钟可否完全准确的走时**

在数字时钟 Demo 中，软件层面的 `delay_1s()` 本质上是“用 CPU 执行固定次数的指令/循环来近似等待 1 秒”。因此，讨论“SocHDMI 时钟能否完全准确走时”，需要区分两个层面：其一是**时间基准的频率是否与真实秒严格一致**；其二是**系统在运行中是否能保证每次延时循环所消耗的 CPU 周期数恒定**。对于不带外部标准时基的 FPGA SoC 而言，想要实现“长期完全无漂移”的绝对准确走时在工程上通常**不可完全保证**，但可以在给定精度目标下做到“足够准确”。

首先，**CPU 五级流水线并不意味着每条指令需要 5 个周期**，但它引入了“吞吐率与延迟”的区别：流水线填满后多数简单指令可做到 **1 周期发射/1 周期吞吐**，然而单条指令从取指到写回确实要经历多个阶段，这体现为“端到端延迟”。对 `delay_1s()` 这类忙等待而言，关心的是在一段时间内**消耗了多少个 CPU 时钟周期（1/50MHz），即为 NOP 指令所需时间**，而不是单条指令的流水级延迟。

但在 SoC 系统中，**总线仲裁与访存会使“每次循环耗时”出现非确定性扰动**。例如取指与数据访存共用总线时，若同周期出现更高优先级的总线事务（调试 UART、数据 Load/Store 等），取指可能拿不到授权而产生停顿；又如 Load-Use 相关会引入不可避免的暂停周期。这些停顿并不会让系统“走得更快”，但会让“同样的延时循环”在不同系统负载下消耗不同周期数，从而使软件延时的短期精度下降。**也就是说，即使系统时钟频率足够稳定，软件忙等待仍可能因为总线/流水线暂停而产生抖动**。

此外，**HDMI 显示链路的多级流水线（例如字符渲染/显存读取/字模 ROM 读取等）主要影响“写显存到上屏”的显示延迟**，并不直接构成“秒表”的时间基准。它会导致屏幕内容的更新相对 CPU 写入存在固定像素时钟级别的延迟，并叠加**“光栅扫描到达目标字符位置”的帧周期等待**，从而使视觉上看到的时间更新具有“1 帧以内”（60HZ）

的不确定性；因此即使 CPU 内部计时非常准确，屏幕显示也可能存在最多一帧量级的显示滞后，这属于显示系统的表现延迟，而非时间基准漂移。

还要考虑**时间基准的绝对准确性受板上时钟源限制**。SocHDMI 的 CPU 系统时钟来自板载晶振（50MHz），其标称频率并不等于“物理上严格的 50,000,000.000...Hz”，而是存在随器件工艺、温度、电压变化的频率偏差与**长期漂移**。若时钟源存在 $\pm\varepsilon$ 的相对误差，则以该时钟计数得到的“1 秒”也会存在相同量级的误差：计数法永远只能复现“本机时钟的一秒”，而无法保证等于“真实世界的一秒”，只能保证 50M 基本准确。

综上，若目标是“教学演示级别的准确走时”，使用 50MHz 时钟计数并配合**手动延时调参**即可达到较好的演示效果；若目标是“长期完全准确、与真实世界秒严格一致”，则需要引入**外部标准时基**或**可校准的高稳定参考**（例如高精度晶振、RTC 模块或与外部时间源同步的校准机制），并尽量用**硬件定时器中断替代忙等待**（**现在是完全依赖 CPU 的软件控制 VideoRAM 进行 HDMI 显示，没有引入硬件定时器中断，更没有引入 DMA，远远达不到成品 MCU 如 STM32 的控制效果**），以降低总线仲裁与流水线暂停对计时稳定性的影响。

五　总结

本报告**完成了课程助教任务中“HDMI 驱动实现与应用部署”的核心工作**，并给出了从**纯 FPGA HDMI 驱动工程到 SoC 系统外设从 VGA 到 HDMI 改造的完整技术由浅入深的实现路径**。

在 FPGA 端，基于 **DVI 兼容模式搭建了 720p@60Hz HDMI 显示链路**，形成“时钟生成—视频时序—TMDS 编码—10:1 串行化—差分输出”的驱动闭环，并通过彩条显示验证了像素时序与链路传输的正确性与稳定性。在应用层面，将既有 VGA 显示工程迁移到 HDMI 输出，乐谱播放器案例实现了位图 ROM 渲染、边框/动态方框叠加以及与音频播放节拍同步的综合功能，证明该 HDMI 驱动具备支撑复杂显示应用的能力。

在 SoC 端，将 **HDMI 显示作为内存映射外设**融入 RV32I 五级流水线系统，**建立了“CPU 写字符显存—HDMI 硬件持续读显存并渲染输出”的软件友好接口**，降低了学生侧应用开发门槛；同时针对跨时钟域数据传输与显示流水线延迟对齐等关键难点完成了工程化处理。数字时钟 Demo 的稳定显示与连续刷新结果表明：CPU 与 HDMI 显示模块可并行工作且无资源冲突，平台具备良好的可用性与可拓展性。

综上，本工作不仅实现了**开发板视频接口升级后的等效替换（VGA→HDMI）**，也为后续实验教学中基于 **C 语言的图形/字符显示类项目提供了可直接复用的硬件与软件基础**；更重要的是，作为数电实验课助教，通过此次 HDMI 驱动与 SoC 移植的全过程，深化了对 FPGA 视频接口设计、跨时钟域数据传输以及内存映射外设集成等技术的理解，提升了系统级硬件设计与移植改造能力，为未来更复杂的嵌入式系统开发积累了工程经验。