智绘无极

——基于易灵思 Ti60F225 无极缩放算法实现

1 设计概述

1.1 设计目的

提供一个高性能的图像缩放解决方案，通过易灵思 Ti60F225 开发平台实现从 480P 到 1080P 的图像缩放，满足不同场景下，对图像尺寸调整的需求。同时提供用户友好的交互界面，增强用户体验。

1.2 应用领域

该作品可广泛应用于视频监控、医疗成像、工业视觉检测、多媒体展示等领域，如显示器开启、应用切换、电脑投影时，显示图像的灵动缩放。由于采用 HDMI 传输 TMDS 实时信号进行缩放，且比例可随意调节，对于实时图像处理和显示尺寸有特定要求的场景尤为适用。

1.3 主要技术特点

1. 实时信号处理：通过 HDMI 输入，实现 480P 实时信号的接收和处理。
2. 缩放比例任意：采用双线性插值等算法，支持任意输出尺寸的调节。
3. DDR 缓存：画幅缩放后的完整图片缓存至 DDR3 内存中以实现缓冲，从而实现输出的速率随画幅变化可变。
4. 用户友好交互界面：利用 STM32 单片机与 LVGL 实现图形用户界面，单片机与 FPGA 之间通过自定义通信协议传递信息

1.4 关键性能指标

采用 ADV7611 输入子卡实现 480P@60Hz 实时输入。

为后期实现对画幅进一步扩大，综合考虑开发板时钟性能，最终考虑对每帧做间隔屏蔽，从而实现 30Hz 输出。

目前已实现输出画面格式为 1080P 标准时序，尺寸缩放范围可达到最小 320*240，最大 1920*1080。

1.5 主要创新点

算法可实现对于输入图片或视频在 320*240--1920*1080 范围内任意尺寸、任意比例的放大，放大过程中图片无卡顿或抖动。

作品采用通过使用 STM32 单片机与 LVGL 搭建开发的 GUI，通过 LCD 触摸屏设置图像的尺寸大小，实现了用户友好的交互界面，并支持配置参数，长按控制，滑动条控制三种控制模式。

实现了单片机和 FPGA 采用类 SPI 协议进行通信，在 FPGA 中设计了 SPI 通信接受模块，接收从单片机传递的缩放信息，通信过程中不会出现信

息传递错误。

2 系统组成及功能说明

2.1 整体介绍

作品的系统框图如图 1 所示。

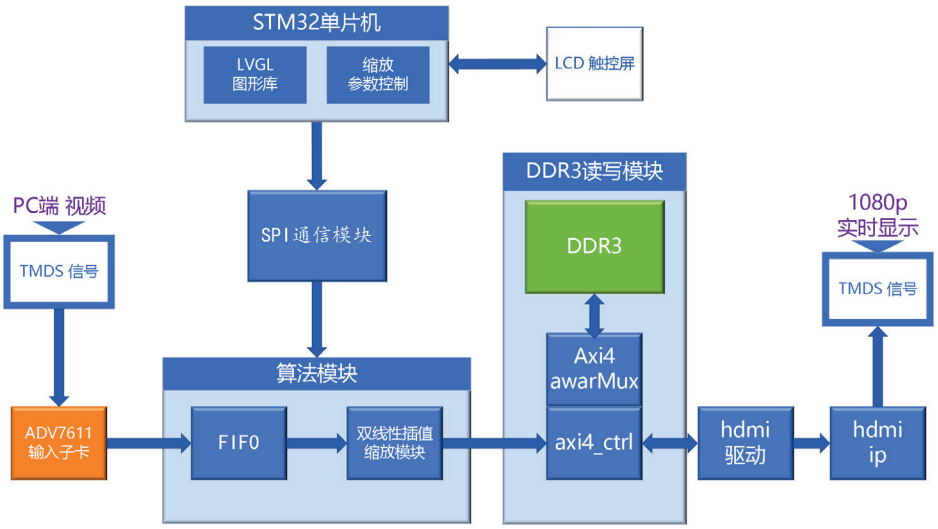

图 1 作品框图

整个系统可分为输入输出模块、算法模块、DDR3 读写模块、控制与 SPI 通信模块四个部分。

信号由 ADV7611 子卡外接 HDMI 输入，进入到开发板中的信号为 24 位并行 GRB 格式的原始数据，时序为 480P（640*480），30Hz 的时序。

数据进入算法模块时，每行数据首先进入 FIFO 缓存，然后等待缩放模块顺序读出调用并进行插值运算。

算法模块计算完后，通过 `axi4_ctrl` 读写控制模块和 `Axi4_awarMux` 地址仲裁模块，将处理后的图片存入 DDR3 中。DDR3 读写使用 axi4 协议，并规划了 4 个 8MB 大小的空间用以分块读写并做前后级缓冲。

hdmi 驱动会生成 1080P 输出的标准时序提供给 hdmi 相关 ip 做输出，同时会根据缩放后的图像尺寸间断从 DDR3 读出数据，以实现缩放图片的居中显示。

根据 DDR3 中的缓冲帧数与读写控制逻辑，外接 STM32 单片机作控制模块会先后将图像尺寸大小更新到缩放算法模块与 hdmi 驱动模块，与每一帧图片在系统中传播的流程同步。

2.2 各模块介绍

2.2.1 输入输出

输入部分采用 ADV7611 子卡，将一张 640*480 的图片写入 FPGA，输入符合 480P 的标准时序。子卡将 TMDS 串行信号转换为 24 位并行的 `hdmi_data_i`，这样，HDMI 的每个输入时钟时候进入到 FPGA 芯片内部的信号即为 RGB 三个通道 24 位的一个像素数据，该像素数据会顺序进入下一

级算法中。同时，子卡会同时给出标准时序的相关信号，如 25.2MHz 时钟信号 hdmi_pclk_i、行同步信号 hdmi_hs_i、帧同步信号 hdmi_vs_i 和数据有效信号 hdmi_de_i。

对于输出部分，hdmi 驱动利用 148.5MHz 的时钟 clk_pixel，通过 hcnt，vcnt 两个计数器计数，提供行、帧同步信号 lcd_hs、lcd_vs 和数据有效信号 lcd_de，生成 1080P（1920*1080）的标准时序，对于屏幕上需要显示图片的位置，hdmi 驱动模块会使用 request 信号向 axi4_ctrl 模块请求读取数据，其余部分将会填充统一的背景色用以区分。而 hdmi 输出 ip 会实时根据 hdmi 驱动模块给出的行、帧同步信号和数据有效信号相关生成 TMDS 信号，通过 HDMI 接口输出给显示屏。

2.2.2 算法模块

算法采用了双线性插值算法。算法模块由前置 FIFO 与算法主体组成。为了保证缩放前后仍能在一帧时间内传输整张图，算法将由两个时钟驱动。慢时钟采用 HDMI 输入的时钟，快时钟则使用 150MHz。

算法模块整体架构图大致如图 2 所示，主要功能即将串行输入的实时 RGB 信号进行短暂缓存，并对实时信号进行双线性插值运算，实现 480p 到指定分辨率的图像缩放，其中缩放算法结构如图 3 所示。为实现实时信号缩放，按照设计目的，我们的算法系统需要做到低延迟、高画质和任意比例缩放的基本需求。本节将首先简要介绍双线性插值算法原理，并主要介绍算法模块的缓存结构、控制逻辑与计算架构，其中缓存结构由 ramBlock 和 ramDualPort 组成，控制逻辑与计算架构在 Scaler 模块中实现。

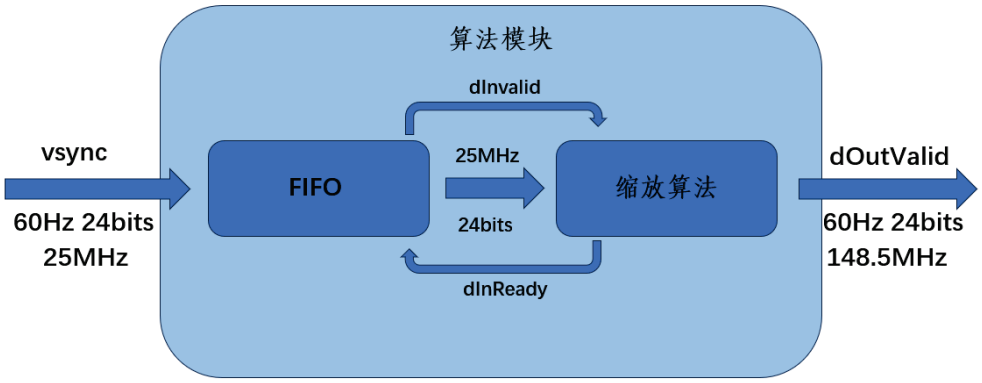

图 2 算法模块架构图

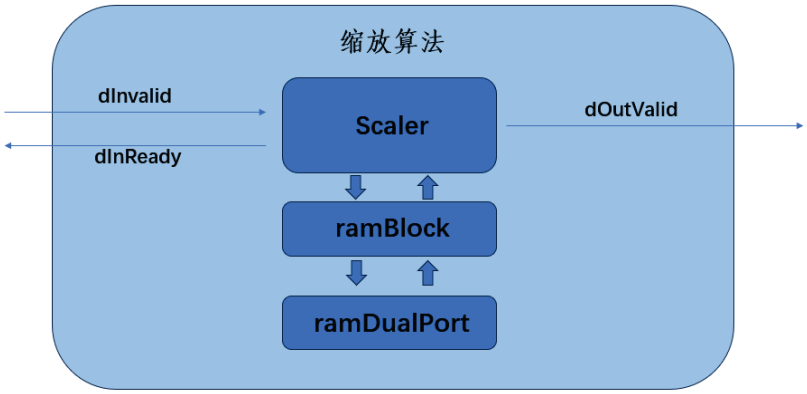

图 3 缩放算法内部结构图

2.2.2.1 双线性插值算法原理

算法模块采用双线性插值算法对实时图像进行缩放，其基本的原理如下。

算法基本思想是根据临近点的值来估算非整数坐标点的像素值，在双线性插值中，像素值的估计依赖于最近四个整数坐标点的值。假设我们有一个点 P，其在放大图中坐标为 (x,y)，根据缩放比例将其坐标映射至原图中的 Q 点，得到其映射坐标并取整数得到整数坐标为 (i,j)，那么 Q 点的像素值可以根据四个相邻整数坐标点 (i,j)，$(i+1,j)$，$(i,j+1)$和 $(i+1,j+1)$的像素值通过线性插值计算得到。

2.2.2.2 算法模块计算架构

针对高画质目标，基于双线性插值原理，算法模块运算架构均使用多位定点数运算以提高计算精度。模块内主要运算包括：

1. 根据缩放比例系数确定待插值像素点的位置：利用输入输出尺寸计算长与宽缩放比例系数，利用输出图像行列坐标与比例系数相乘得到原图中的映射坐标 $(i+u,j+v)$，其中i、j表示行列坐标整数部分，u、v表示小数部分。取整与取小数位可使用移位或切片操作，这在定点数结构中很容易做到。

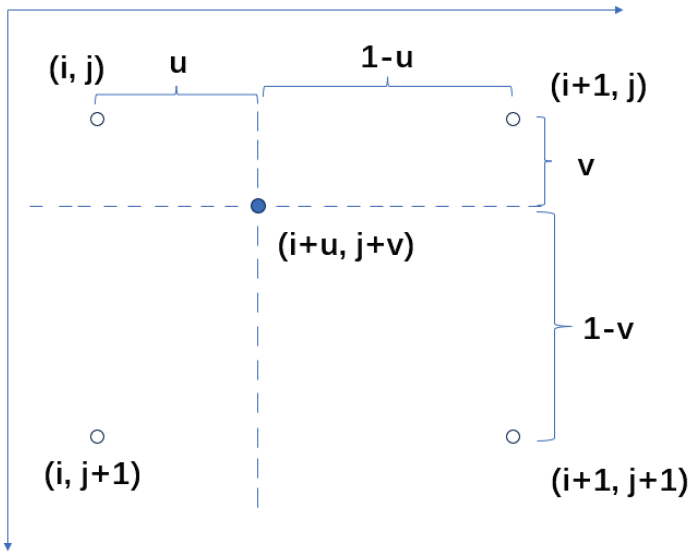

图 4 像素点映射示意图

2.计算该点与相邻四个像素点的坐标差：通过分离小数部分得到u、v即映射坐标与整数坐标$(i,\ j)$的坐标差，$1-u$、$1-v$即映射坐标与整数坐标$(i+1,j+1)$的坐标差。

3.利用坐标差值计算与相邻四个坐标点对应相乘的权重：权重计算依照双线性插值算法原理：

$f(i+u,j+v)=(1-u)(1-v)\times f(i,j)+(1-u)v\times f(i,j+1)+u(1-v)\times f(i+1,j)+uv\times f(i+1,j+1)$，

我们需要算出$(1-u)(1-v)$、$(1-u)v$、$u(1-v)$和$u\times v$。

4.将权重与四个相邻像素点数据分别进行乘加操作，便可得到插值像素点的值。

小结：计算架构主要采用并行处理、分而治之的思想，使用组合逻辑同时计算相邻四个坐标点对应相乘的权重，降低时序负担以及因复杂计算引起

的寄存器资源占用；利用定点数结构进行浮点数计算，便于分离映射坐标整数小数部分。

2.2.2.3 算法模块控制逻辑

首先考虑顶层设计思路：容易注意到缩放前后输入输出数据吞吐量差异，考虑最极端情况即输入时钟频率 25MHz、分辨率 480p 图像，输出分辨率 1080p 图像，由于要保持输入输出相同刷新率，因此算法输出时钟需要拉高至 148.5MHz。为实现实时信号低延迟缩放，我们采取读写逻辑独立处理并以关键信号指示读写模块交互。

写数据模块主要负责接收原始数据并进行缓存，其状态机控制逻辑如下：为避免原始数据拥挤和丢失情况，在输入信号与缩放算法之间布置 fifo 缓存少量数据，利用握手机制与算法模块进行交互，防止算法读满状态下原始数据丢失。在每帧开始时，状态机由起始态(W_IDLE)跳转至写数据态(WRITE)，输入数据有效信号拉高指示算法写入原始数据，此时写入行列地址分别按规律自增，原始数据按地址索引存入缓存结构；当以整行原始数据存储完成后，写数据行信号(advanceWrite)拉高，指示当前可用原始数据行数(fillCount)加一；当行地址计数指示当前已写入充足的原始数据，通过 readyForRead 信号拉高指示读数据状态机由起始态(R_IDLE)跳转至读数据态(READ)；当行地址计数指示当前正写入当前帧最后一行原始数据，状态机跳转至帧结束态(W_DONE)，写模块暂停数据写入，等待场同步信号复位。

读数据模块主要负责缓存模块原始数据读出，以及控制输出图像对应行列坐标变化，其状态机控制逻辑如下：

场同步信号置高指示状态机进入起始态(R_IDLE)，等待足够数据写入后 readyForRead 信号拉高，状态机跳转至读数据状态(READ)，输出像素行列地址分别按规律自增。读模块的核心在于，在输出行利用当前输出行与下一输出行的映射纵坐标整数部分(yPixInt、yPixIntNext)之差指示需要补充的原始数据行数，并以此指示缓存模块需要重新读入几行原始数据。如 $yPixIntNext - yPixInt = 1$，则置高 advanceRead1 信号，指示需要额外补充一行原始数据；如 $yPixIntNext - yPixInt = 2$，则置高 advanceRead2 信号，与缓存模块进行交互。

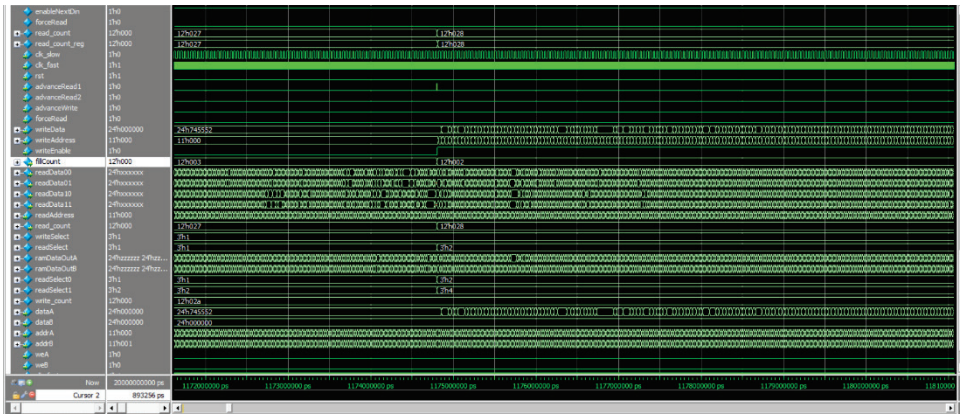

图 5 写数据行切换仿真波形图

举例说明：如果放大比例较大，计算当前行输出数据需要原始图像的第 1 行和第 2 行，计算下一行输出数据同样需要原始图像的第 1 行和第 2 行，便不需要写入新数据，此时可用原始数据行数(fillCount)不变；如果缩小

比例较小，计算当前行输出数据需要原始图像的第 1 行和第 2 行，计算下一行输出数据同样需要原始图像的第 3 行和第 4 行，便需要写入两行新数据，此时可用原始数据行数(fillCount)自减 2，进而由于可用原始数据行数过低将输出使能拉低直至写入充足数据。

小结：控制逻辑采取读写分治的思路，利用读写数据行信号(advanceWrite、advanceRead1、advanceRead2)在读模块、写模块与缓存模块之间实现交互；在读数据模块同时计算输出行与下一输出行的映射纵坐标，在输出行切换前进行原始数据补充，以减低输出行切换延迟，实现高效读写操作。

2.2.2.4 算法模块缓存结构

从算法原理可以了解到，双线性插值在某些情况需要对原始数据重复利用，在某些情况需要跳过若干行原始数据，因此如何高效利用少量缓存数据便成为了算法设计的重点。为了后续计算插值运算所需要的系数和存储插值点邻近的四个像素值，通常所采取的方法为使用两个行缓冲器来存储插值所需要的两行数据，但此方法并不适用于对于实时视频图像缩放，过少数据量缓存会影响计算速度，从而导致视频延迟、帧率下降；如果采取多行数据缓存，会导致硬件资源过度占用，降低模块性能。因此选择三行 Ram 轮转缓存结构，在尽可能减小资源占用的情况下通过读写数据行信号(advanceWrite、advanceRead1、advanceRead2)与可用原始数据行数(fillCount)灵活存写，接下来介绍缓存架构运行机制。

缓存结构主体为三个数据位宽为 24、深度为 640 的 Ram 组成，用以存储相邻的三行原始图像数据。轮转读写操作依靠独热码调用 Ram，利用三位写选择信号(writeSelect)、三位读选择信号(readSelect)形成回环移位寄存器。当写数据行信号(advanceWrite)置高，写选择信号(writeSelect)循环左移，状态跳变情况：001→010→100→001……当读数据行信号(advanceRead1、advanceRead1)置高，读选择信号(readSelect)按规律循环左移，具体跳变情况可参考图 6。

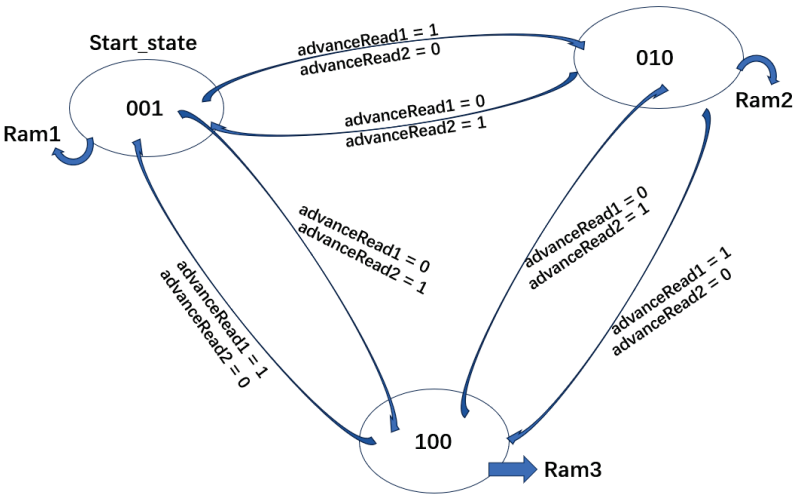

图 6 读选择信号(readSelect)状态流程图

在 Ram 写使能信号(writeEnable)置高阶段，原始图像数据按照写入模块列地址（writeColCount）写入对应 Ram；在写使能信号(writeEnable)置低阶段进行数据读取，读地址取决于映射横坐标整数部分(xPixInt)，由计算模块

传入，凭此在 ramBlock 内部得到临近四个像素点地址，读出对应四个像素数据供 Scaler 模块进行计算。此外，通过检测读写数据行信号(advanceWrite、advanceRead1、advanceRead2)，对读写操作进行计数，advanceWrite 置高指示写计数器(write_count)自增 1，advanceRead1、advanceRead2 指示读计数器(read_count)分别自增 1、2。由fillCount = write_count − read_count对可用原始数据行数进行更新。

小结：缓存结构使用回环移位寄存器实现 Ram 轮转调用，对于存储资源进行高效调用；利用可用原始数据行数(fillCount)为中转信号，实现与读写模块的交互，表明当前 Ram 占用情况，指示下一步读写操作，实现低延迟实时运算。

2.2.3 DDR3 读写模块

由于算法输出的速率（像素的速率）较快，因此为了保证视频正常输出，需要先将图片写入 DDR3 中，再读出，相当于 DDR3 在这里被用为缓存。

axi4_ctrl 分配了 4 块的空间作 DDR3 缓存设计，模块内独立设计了读、写两个状态机。由于读写共用一个地址通道，因此设置了地址仲裁模块 Axi4_awarMux。由于输入、输出信号工作的时钟域均与 axi4_ctrl 不相同，同时进出 DDR 的数据为 128 位，与填充 8 位 0 后的 RGB 格式数据（32 位）位数不同，因此在 DDR3 前、后均设置了异步读写 FIFO 即 W_FIFO、R_FIFO 作为缓冲。用户与 DDR3 交互时采用 AXI4 通信协议，因此实际上 axi4_ctrl、Axi4_awarMux 模块是在与 ddr_ctrl IP 核进行交互。

输入信号工作在 150MHz 时钟域下，当算法的输出有效信号 doutvalid 拉高时，将算法模块输出的信号（高 8 位补 0 至 32 位）data_out 写入 W_FIFO；输出信号工作在 148.5MHz 时钟域下，当 hdmi 驱动需要输出图片时，lcd_request 信号拉高，从 R_FIFO 中读出 32 位 RGB 像素数据 lcd_data。

DDR3 采用突发读写，突发长度为 8，数据长度为 128，因此每次发送 128 字节的数据，即每次突发读写结束后，地址增加 128；每一帧图像的突发读写通过若干次突发完成。

每块缓存空间为 8MBytes，以允许前期设计的放大后图片的最大 1920*1080 大小，此时读的空间地址最大为 1920*1080*4（以字节为单位，高 8 位补零后每组 RGB 数据 32 位 4 个字节）。将 4 块缓存空间以 Index0~3 标识，地址更新方式如下：

```
assign axi_awaddr = C_BASE_ADDR + {rc_wframe_index, rc_w_ptr};
assign axi_araddr = C_BASE_ADDR + {rc_rframe_index, araddr};
```

地址的高位为缓存空间下标 0~3，由于数据出入 DDR3 时，仅需顺序读写，因此读写地址的低 23 位由状态机控制累加，每次增加 128。

读写索引与地址更新方式如图 7、图 8 所示，避免了对于 DDR3 同一块分区的读写。当写区域索引需要更新时，若下一块区域没有正在被读，则写入下一块区域，否则跳过下一块区域写入；每次写区域更新时记录已经写完的区域下标，当读区域索引需要更新时，更新为上一次写区域的索引，即读取上一次写入完毕区域的数据。可见，缩放图片进入 DDR3 后，会延后一段时间读出。

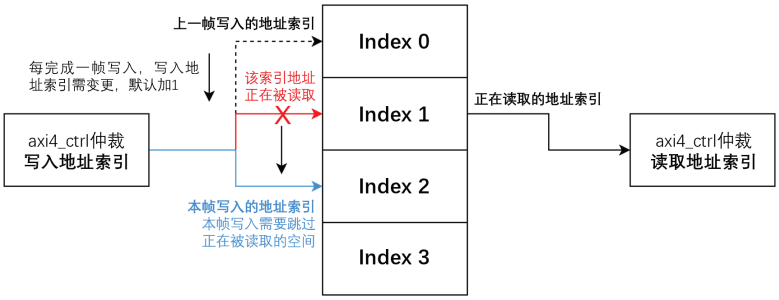

图 7 写入地址索引更新方式

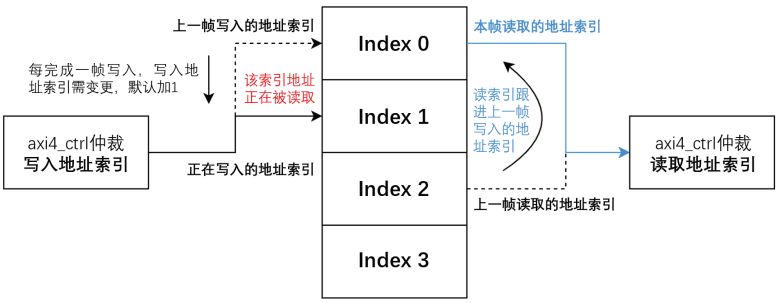

图 8 读取地址索引更新方式

写入状态机框图如图 9，分为初始状态、写入结束状态、写数据状态和写突发结束状态四种状态。

状态机位于初始状态时，当 W_FIFO 中存储的数据大于一次突发所需的数据量（128 个数据），将写有效信号 `axi_wvalid` 和写地址有效信号 `axi_awvalid` 信号拉高，并进入写数据状态；`axi4_ctrl` 模块给出的 `axi_wvalid` 和 `ddr_ctrl` 给出的写准备好 `axi_wready` 信号同时拉高，即进行一次握手时，一个 128 位数据从 W_FIFO 写入 DDR3 中，同时突发计数加 1；单次突发计数到 128 时，一次突发结束，进入到写突发结束状态，此状态中给 `rc_w_ptr` 加 128，同时状态机回到初始状态。当读取到输入子卡提供的帧同步信号 `hdmi_vs_i` 的下降沿时，代表算法输出的一帧图像结束，此时将 EOF 信号拉高。初始状态下，若 W_FIFO 中仍有数据，则继续进入写数据状态，若 EOF 拉高且 W_FIFO 中没有数据，则进入写入结束状态，写索引更新，同时写地址 `rc_w_ptr` 清零，并再次回到初始状态。

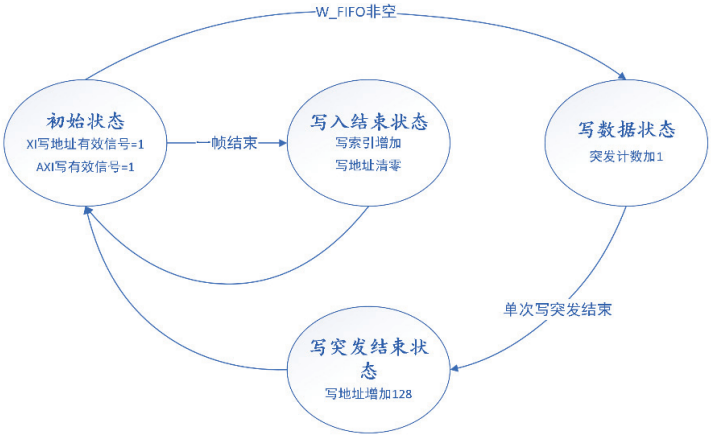

图 9 写入状态机

读出状态机框图如图 10，分为初始状态，读地址更新状态，读数据状态三种状态；当读取到 hdmi 驱动模块给出的帧同步信号 lcd_vs 的上升沿时，状态机回复到初始状态，初始状态中，当 R_FIFO 空余的容量大于一次突发读的数据量（128 个数据）时，进入读地址更新状态；读地址更新状态中，将读地址有效信号 axi_arvalid 拉高，同时状态机进入读数据状态；读数据状态中，axi4_ctrl 模块给出的 axi_wvalid 和 DDR3 给出的写准备好 axi_wready 信号同时拉高，即进行一次握手时，一个 128 位数据从 DDR3 读出到 R_FIFO 中；当 DDR 给出一次突发读信号结束的信号 axi_rlast 时，回到初始状态。

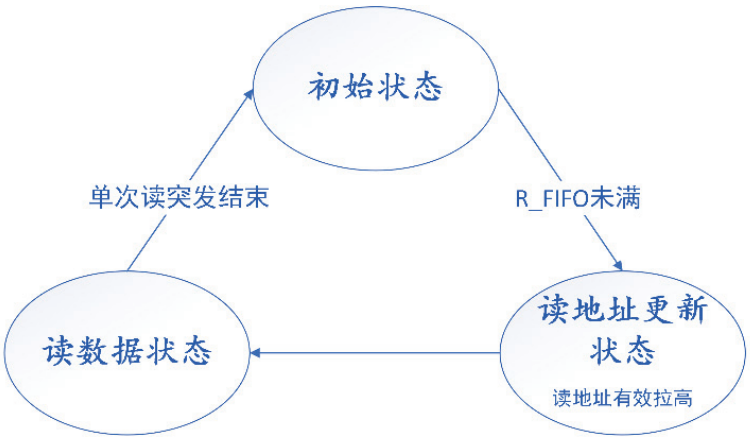

图 10 读出状态机

Axi4 读写控制模块 axi4_ctrl 中，分别给出了读、写地址和各自的有效信号，但对于 ddr_ctrl IP，仅有一个地址信号 aaddr，以及对应的有效信号 avalid、DDR3 准备好信号 aready。ddr_ctrl IP 通过输入的 atype 信号决定此时进行读操作还是写操作，从而将地址解读成读地址或写地址。因此本设计采用地址仲裁模块 Axi4_awarMux 对读写地址进行仲裁，交替将从 axi4_ctrl 接收到的读写地址传递给 ddr_ctrl，并向 ddr_ctrl 传递对应的 atype。

具体仲裁方式为交替传递读写地址。由于对 DDR3 是突发读写，因此在一次突发的过程中 axi4_ctrl 给出的读、写地址不会改变。地址仲裁状态机示意图如图 11，分为初始状态、读/写地址更新状态、读/写切换状态三种状态。寄存器 r_reqtype 的值在 0、1 间交替切换，0 对应读地址更新，1 对应写地址更新。

初始状态下，当 axi4_ctrl 模块给出的对应地址有效信号拉高时，进入读/写地址更新状态，将对应的读/写地址准备好状态拉高，与 axi4_ctrl 模块进行握手，同时将对应的读/写地址赋给 aaddr，将 atype 置 0 或 1，传递给 ddr_ctrl（例如，当 r_reqtype 为 0 且 axi_arvalid 信号拉高时，仲裁模块令 aaddr = araddr，并令 atype = 0），并进入读/写切换状态；读/写切换状态中，更新 r_reqtype 的值，并回到初始状态。状态机如此工作，交替将从 axi4_ctrl 传递过来的读、写地址信息传递给 ddr_ctrl IP 核，实现读写地址仲裁。

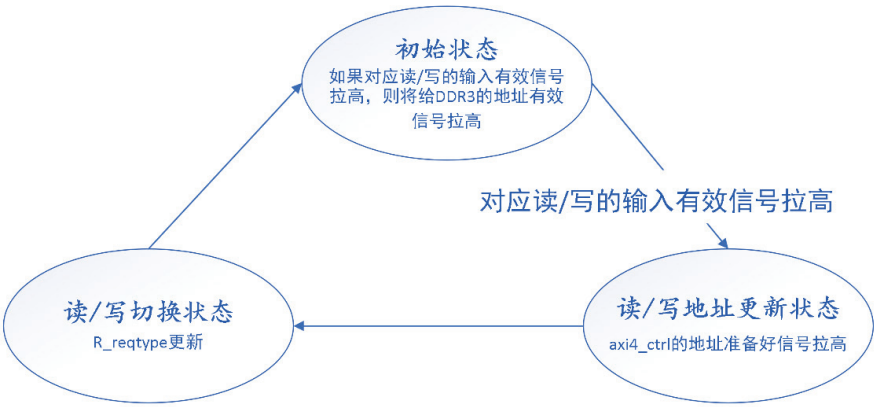

图 11 地址仲裁状态机

整体的 DDR3 读写模块的数据、地址和控制信号流通图如下图所示。

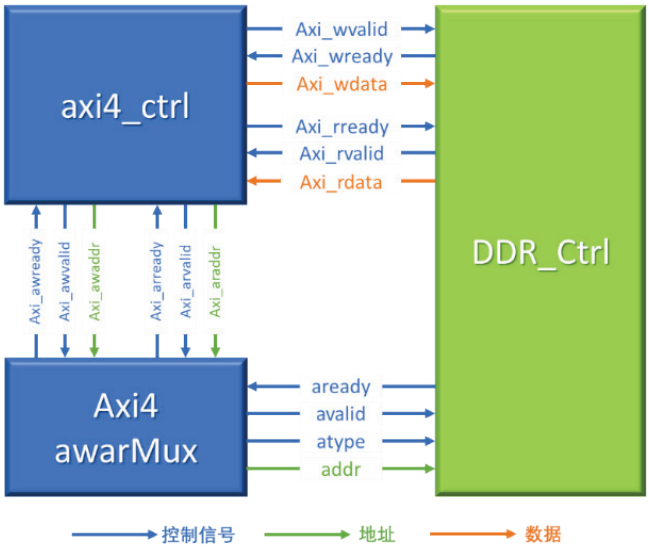

图 12 DDR3 相关数据、信号流图

2.2.4 控制与 SPI 通信模块

作品采用了使用 STM32 单片机与 LVGL 搭建开发的 GUI，通过 LCD 触摸屏设置图像的尺寸大小，并由单片机将图像尺寸增量发送给 FPGA，实现了用户友好的交互界面。

共设置了三种控制模式：

1. **配置参数模式：**使用者从下拉框中的若干标准画幅中选择要缩放到的尺寸，按下“开始缩放”键开始工作；

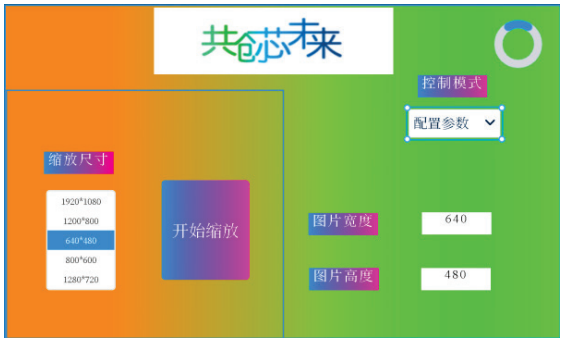

图 13 配置参数模式

2. **长按控制模式**：有放大和缩小两个按键，使用者长按按键时，对图像进行连续等比例的放大或缩小；

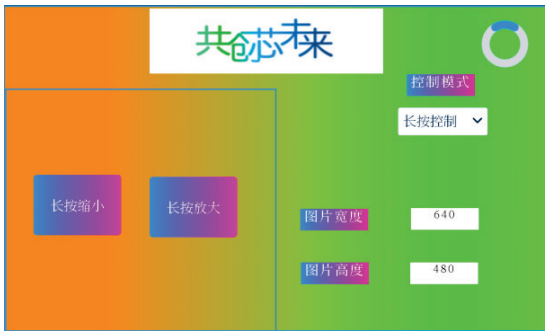

图 14 长按控制模式

3. **滑动条控制模式**：此模式支持任意 320*240 至 1920*1080 之间的任意画幅、任意比例缩放，使用者通过控制滑动条设置理想的图片大小，之后按下“开始缩放”键开始缩放。

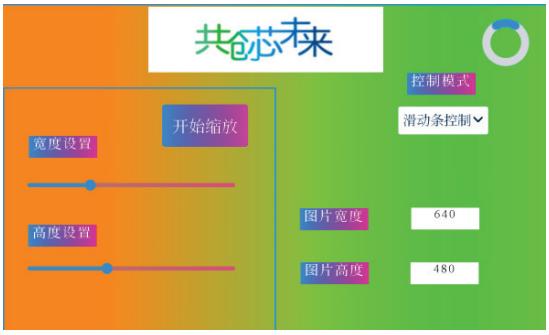

图 15 滑动条控制模式

为实现单片机与 FPGA 通信，实现了类 SPI 通信协议。此协议采用四根数据线连接单片机与 FPGA，分别为时钟线 SCLK、数据线 SDI、标志线 flag 和地线 GND；单片机每次向 FPGA 发送 16 位数据，其中第 16 位、第 8 位分别为图像长度、宽度的变化方向（1 代表放大，0 代表缩小），第 15-9 位，第 7-0 位分别为图像长度、宽度增量的绝对值。

每次发送数据时，单片机会先将标志 flag 拉高，之后单片机在 SCLK 为低时将 16 位数据从低位到高位依次放置在 SDI 线上，之后将 SCLK 拉高。串行发送完 16 位数据后，flag 会保持 4 个 SCLK 周期，以保证 FPGA 接收数据的准确性。

当 FPGA 接收到 flag 信号的上升沿时，开始一次数据接收。FPGA 在 SCLK 的上升沿对 SDI 上的每一位数据进行采样，并将数据依次存入一个 16 位缓冲器的对应位。当接收完全部 16 位信号后，延迟 3 个 SCLK 周期将 done 信号拉高，同时将缓冲器的高 8 位、低 8 位分别赋给 8 位变量 inc_length、inc_width。

当 done 信号拉高，FPGA 中的寄存器 reg_output_length、reg_output_width 会根据 inc_length、inc_width 的值自加或自减相应的数值，使图片的尺寸参数得到更新（例如当 inc_length = 8’b10000101 时，代表 reg_output_length = reg_output_length + 5）。当 ADV7611 子卡给出的 LCD_VS 信号下降沿到来时，标志着将输入一张新的图片，此时将 reg_output_length、reg_output_width 的值赋给算法的输入 OUTPUT_LENGTH、OUTPUT_WIDTH，作为图像缩

放的目标尺寸，提供给缩放算法模块。当输出 hdmi 驱动模块的 lcd_vs 信号下降沿到来时，代表将要从 DDR 中读出一帧新的图片并显示在屏幕上，此时将相应输入的 OUTPUT_LENGTH、OUTPUT_WIDTH 延迟一个 lcd_vs 下降沿，赋给 length_out_hdmi、width_out_hdmi，提供给 hdmi 驱动模块，作为刷新屏幕时所用的图像尺寸。

3 完成情况及性能参数

目前已完成 640*480 尺寸最大放大至 1920*1080，最小缩小至 320*240；支持特定比例间缩放与任意比例缩放。由于更高的尺寸需要调节前后相关参数与画幕大小，故无法体现在该工程中。

工程综合、实现后，可得硬件资源使用大致情况如下

FFs	SRLs	ADDs	LUTs	COMB4s	RAMs	DSP/MULTs
3137	576	1424	5519	0	80	18

4 总结

4.1 可扩展之处

作品目前实时显示只能实现 1080p，30 帧实时显示，后期可通过算法优化，DDR3 读写控制模块优化等方式实现 1080p，60 帧实时显示。

在 FPGA 内部资源允许的情况下，可加入双三次插值等更复杂、图像处理质量更高的算法，丰富用户的选择。

利用采集卡等设备，通过适当降低输入帧率，可实现 2K 等更高分辨率的显示。

4.2 心得体会

团队的成员都在大二下学期接触到 FPGA 的开发设计。与 stm32、Linux 开发板这种嵌入式系统开发板不同，FPGA 描述的东西更加底层，也更加贴近“芯片”这一硬件本身。

由于都对 FPGA 编程有一定经验，团队起始进展还算顺利，但是随着进一步完善功能、烧录测试，逐渐遇到了困难，甚至停滞不前。

FPGA，可编程逻辑门阵列，本质可以认为是电路的逻辑门作互相连接，实际对应的是芯片上物理电路这一实物。因而，硬件描述语言（如我们使用的 Verilog）与我们已经习惯了的、在计算机指令系统架构之上的软件编程不同，对于已经很简单的运算、存储空间的读取，往往变得非常复杂。直观地来说，相关计算规则、内存访问读写等等，完全由布尔逻辑作为搭建起点，工程量往往是很大的。

在两个多月的作品准备中，我们的开发过程也能体现出 FPGA 开发的特色。

虽然说 FPGA 开发基本逻辑是确定的，但也需要适应特定厂商的开发环境。不得不说，易灵思的 Efinity 开发软件确实提供了很新颖的交流环境与开发方式，相关 interface 的配置简明清晰，或许对于大学生教学是一个不错的选择。总体的软件风格也非常简洁大方，没有这么多繁琐的弯路。

由于需要使用厂商特定的 IP，这对于对于 Verilog 代码的阅读与理解是非常必要的，如 FIFO、DDR 控制等模块，均需要理清逻辑才能正确流畅使

用。在 10 月中旬，也因为相关数据的位宽和长度没有弄清含义而一直出现问题。

毕竟是一个团队的比赛，协作往往比单独行动更为重要。虽然没有前期制定清晰的框架，导致后期交流不够顺畅，但是整个团队都在努力、也见过很多深夜的校园，这也是参与比赛带来的磨练吧。

5 参考文献

《基于 MATLAB 与 FPGA 影图像处理教程》韩彬等著
奥唯思 VF-Ti60F225 FPGA 开发板配套资料
易灵思官方 Ti60 FPGA 数据手册与使用指

附录

Scaler 模块部分

```
always @ (posedge clk_fast or posedge start)
begin
    if(start)
    begin
        outputLine <= 0;
        outputColumn <= 0;
        xScaleAmount <= 0;
        yScaleAmount <= 0;
        readState <= R_IDLE;
        dOutValidInt <= 0;
        lineSwitchOutputDisable <= 0;
        advanceRead1 <= 0;
        advanceRead2 <= 0;
        yScaleAmountNext <= 0;
    end
    else
    begin
        case (readState)
            R_IDLE:
            begin
                xScaleAmount <= 0;
                yScaleAmount <= 0;
                if(readyForRead)
                begin
                    readState <= READ;
                    dOutValidInt <= 1;
                end
            end

            READ:
            begin
                if(dOutValidInt)
                begin
                    if(outputColumn == outputXRes)
                    begin
                        if(yPixIntNext == (yPixInt + 1))
                        begin
                            advanceRead1 <= 1;
                            if(fillCount < 3)
                                dOutValidInt <= 0;
                        end
```

```
                    else if(yPixIntNext > (yPixInt + 1))
                    begin
                        advanceRead2 <= 1;
                        if(fillCount < 4)
                            dOutValidInt <= 0;
                    end

                    if(outputLine == outputYRes)
                       readState <= R_DONE;

                    outputColumn <= 0;
                    xScaleAmount <= 0;
                    outputLine <= outputLine + 1;
                    yScaleAmount <= yScaleAmountNext;
                    lineSwitchOutputDisable <= 1;
                end
                else
                begin
                    if(lineSwitchOutputDisable == 0)
                    begin
                        outputColumn <= outputColumn + 1;
                        xScaleAmount <= (outputColumn + 1) * xScale +
0;

                    end
                    advanceRead1 <= 0;
                    advanceRead2 <= 0;
                    lineSwitchOutputDisable <= 0;
                end
            end
            else
            begin
                advanceRead1 <= 0;
                advanceRead2 <= 0;
                lineSwitchOutputDisable <= 0;
            end

            if(fillCount >= 2 && dOutValidInt == 0 || allDataWritten)
            begin
                if((!advanceRead1 && !advanceRead2))
                begin
                    dOutValidInt <= 1;
                end
            end
        end
```

```
            R_DONE:
            begin
                advanceRead1 <= 0;
                advanceRead2 <= 0;
                dOutValidInt <= 0;
            end

        endcase
        yScaleAmountNext <= (outputLine + 1) * yScale;
    end
end
```

ramBlock 模块

```
module ramBlock #(
    parameter DATA_WIDTH = 8,
    parameter ADDRESS_WIDTH = 8,
    parameter BUFFER_SIZE = 3,
    parameter BUFFER_SIZE_WIDTH =   2
)(
    input wire                      clk_slow,
    input wire                      clk_fast,
    input wire                      rst,
    input wire                      advanceRead1,
    input wire                      advanceRead2,   //将选中的读 RAM 提前两
个
    input wire                      advanceWrite,
    input wire                      forceRead,      //禁止写入

    input wire [DATA_WIDTH-1:0]     writeData,
    input wire [ADDRESS_WIDTH-1:0]  writeAddress,
    input wire                      writeEnable,
    output [11:0]                   fillCount,

    output wire [DATA_WIDTH-1:0]    readData00,
    output wire [DATA_WIDTH-1:0]    readData01,
    output wire [DATA_WIDTH-1:0]    readData10,
    output wire [DATA_WIDTH-1:0]    readData11,
    input  wire [ADDRESS_WIDTH-1:0] readAddress
);

reg [BUFFER_SIZE-1:0] writeSelect = 1;
reg [BUFFER_SIZE-1:0] readSelect = 1;
```

```
always @(posedge clk_fast or posedge rst)
begin
    if(rst)
        readSelect <= 1;
    else
    begin
        if(advanceRead1)
        begin
            readSelect <= {readSelect[BUFFER_SIZE-2 : 0],
readSelect[BUFFER_SIZE-1]};//左移
        end
        else if(advanceRead2)
        begin
            readSelect <= {readSelect[BUFFER_SIZE-3 : 0],
readSelect[BUFFER_SIZE-1:BUFFER_SIZE-2]};//左移两位
        end
        else readSelect <= readSelect;
    end
end

always @(posedge clk_slow or posedge rst)
begin
    if(rst)
        writeSelect <= 1;
    else
    begin
        if(advanceWrite)
        begin
            writeSelect <= {writeSelect[BUFFER_SIZE-2 : 0],
writeSelect[BUFFER_SIZE-1]};//左移
        end
    end
end

wire [DATA_WIDTH-1:0] ramDataOutA [2**BUFFER_SIZE-1:0];
wire [DATA_WIDTH-1:0] ramDataOutB [2**BUFFER_SIZE-1:0];

generate
genvar i;
    for(i = 0; i < BUFFER_SIZE; i = i + 1)
        begin : ram_generate

            ramDualPort #(
                .DATA_WIDTH( DATA_WIDTH ),
```

```
                .ADDRESS_WIDTH( ADDRESS_WIDTH )
            ) ram_inst_i(
                .clk_slow( clk_slow ),
                .clk_fast( clk_fast ),

                .addrA( ((writeSelect[i] == 1'b1) && !forceRead &&
writeEnable) ? writeAddress : readAddress ),
                .dataA( writeData ),

                .weA( ((writeSelect[i] == 1'b1) && !forceRead) ? writeEnable :
1'b0 ),
                .qA( ramDataOutA[2**i] ),

                .addrB( readAddress + 1 ),
                .dataB( 0 ),
                .weB( 1'b0 ),
                .qB( ramDataOutB[2**i] )
            );
        end
endgenerate

wire [BUFFER_SIZE-1:0]  readSelect0 = readSelect;
wire [BUFFER_SIZE-1:0]  readSelect1 = (readSelect << 1) |
readSelect[BUFFER_SIZE-1];

assign readData00 = ramDataOutA[readSelect0];
assign readData10 = ramDataOutA[readSelect1];
assign readData01 = ramDataOutB[readSelect0];
assign readData11 = ramDataOutB[readSelect1];

reg [11:0] write_count;
reg [11:0] read_count;

assign fillCount = write_count - read_count;

always @(posedge clk_slow or posedge rst)
begin
    if(rst)
    begin
        write_count <= 0;
    end
    else
    begin
        if(advanceWrite)
```

```
            write_count <= write_count + 1;
        else
            write_count <= write_count;
    end
end

always @(posedge clk_fast or posedge rst)
begin
    if(rst)
    begin
        read_count <= 0;
    end
    else
    begin
        if(advanceRead1)
            read_count <= read_count + 1;
        else if(advanceRead2)
            read_count <= read_count + 2;
        else
            read_count <= read_count;
    end
end

endmodule
```

spi 通信模块

```
module SpiSlave16Bits_S (
    input flag,
    input SCLK,
    input SDI,

    output reg [7:0]    inc_length,
    output reg [7:0]    inc_width,
    output reg          done,
    output [3:0]        step_de
);

assign      step_de     = step;
reg [15:0]  Read_buffer = 0;
reg [4:0]   step        = 0;
reg         VS_delay;


always@(posedge SCLK or negedge flag)
```

```
begin
    if (!flag) begin
        step<=0;
        done <= 0;
    end
    else begin
        if (step <= 15) begin
            step <= step+1'b1;
            Read_buffer[step]<=SDI;
        end
        else if (step == 19) begin
            step <= 0;
            inc_length <= Read_buffer[15:8];
            inc_width <= Read_buffer[7:0];
            done <= 1;
        end
        else
            step <= step+1'b1;
    end
end

endmodule
```