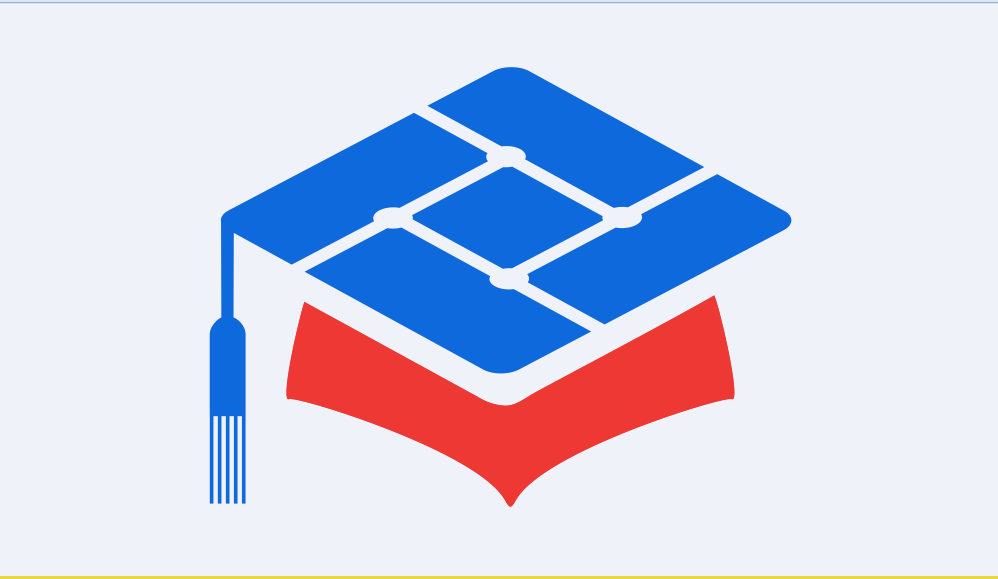
**第七届全国大学生集成电路创新创业大赛**

**CICIEC**

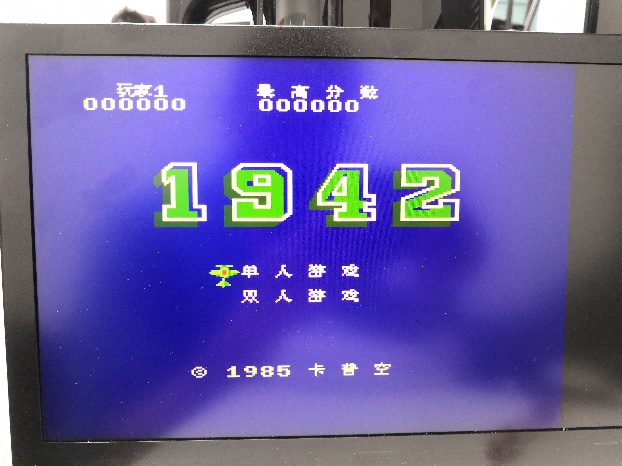
****

**Arm杯**

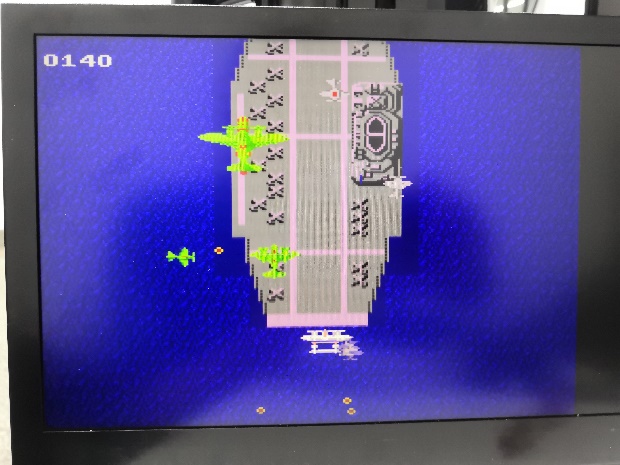
|  |  |
| --- | --- |
| **参赛题目：** | **基于arm处理器的智能游戏机设计** |
| **队伍编号：** | **CICC3370** |
| **队伍名称：** | **3bits** |

**1. 展示**

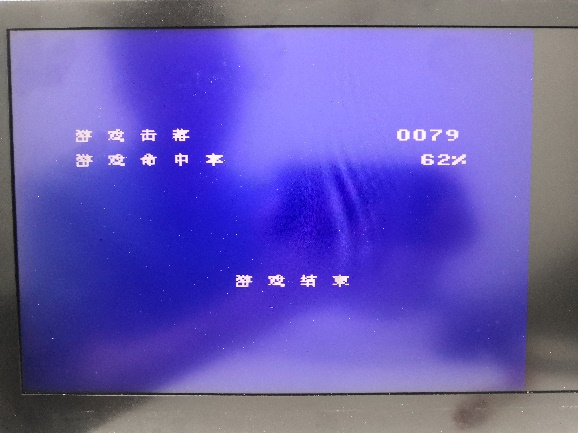
* 开始界面



* 游戏进行



* 结算界面

****

**2. 系统方案的设计与介绍**

**2.1 系统方案的选择**

对于题目“基于arm处理器的智能游戏机”进行分析：主要内容有两个：一是实现什么样的游戏，二是实现什么样的智能化。

对于游戏的实现，首先需要具备基础的硬件条件，包括输入控制设备：摇杆，手柄，鼠标以及键盘等等。综合考虑便携性和操作的丰富程度，我们最终选择PS2手柄作为我们的输入设备。根据实验室条件，游戏的画面通过VGA或者HDMI即可，不考虑LCD。游戏音乐采用普通的蜂鸣器实现即可。

对于游戏的设计上，我们认为有简单的贪吃蛇，打地鼠等等，但是玩法单一，带给玩家的体验不够“刺激”。我们最终选择了射击类游戏飞机大战作为我们的游戏进行实现，该类游戏节奏紧张，敌机类型丰富多样，能够扩展出多样的爆炸效果，buff加成效果。同时加上丰富的，紧张刺激的音效，更够带给玩家一个紧张有趣的体验。难点在于：采用VGA进行显示且保证游戏一个流畅的帧率，必须要实现游戏单位和游戏背景的实时显示，这需要采用速度尽量快的且寻址方便的BRAM资源进行游戏单位图案和背景图案数据的存储。游戏在刷新每一帧之前都必须保证该帧游戏画面单位的完整性。倘如不对每一帧的图像进行压缩，FPGA的小容量的BRAM将会被很快的消耗，而留给其他模块的BRAM更是很少。因此我们采用FC游戏的显示原理，实现了我们自己的PPU图像处理模块，实现了图像数据的有效压缩，实现了我们自己的图像显示引擎。图案素材借鉴经典FC游戏《1942》。

对于智能游戏的环节上，我们计划采用在FPGA上部署手势识别的神经网络模型实现游戏的操作。难点在于：由于神经网络参数量大，不能在有限的资源上实现全并行计算，需要考虑资源与速度的平衡。

**2.2 软硬件介绍**

说明：由于厂商提供的FPGA板卡和配套的底板可自由连接的引脚较少，在目前开发阶段我们使用了两套板卡。一套是安路PH1A60板卡，另一套是Xilinx Artix7 75T板卡。

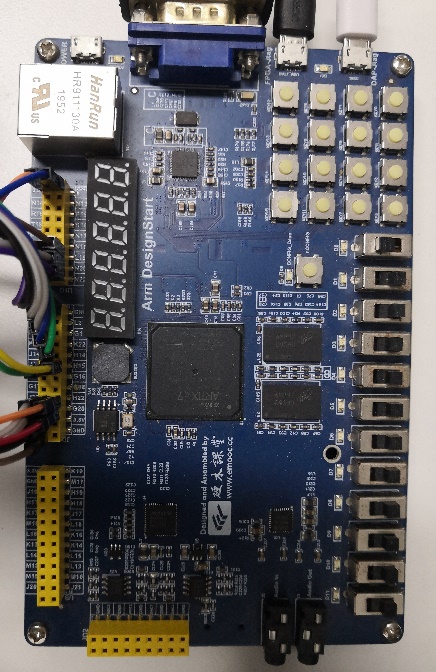
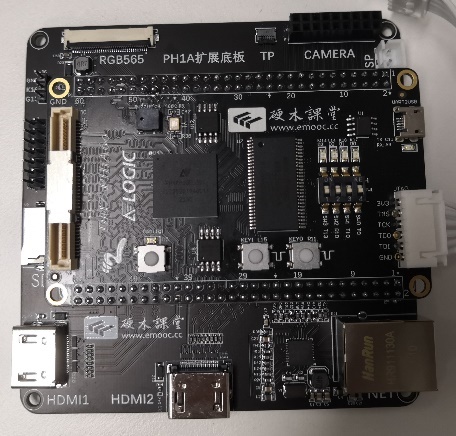
****

图1-1 前期开发阶段使用的两套开发板

硬件：

* PS2手柄
* 13寸显示器
* DAPLINK Debugger
* Xilinx Artix7 75T板卡（游戏开发阶段暂用后续进行移植）
* PH1A60板卡（神经网络手势识别加速器开发阶段使用）

软件：

* TD 5.6.1
* KEIL V5.06
* Modelsim 10.5
* Vivado 2019.1（开发阶段暂用）
* PyCharm

**3.SoC方案**

**3.1 SoC总体**

我们目前的SoC如下图所示。其中包含：

* Cortex M3 Core
* AHB-Interconnect
* CODE\_ROM
* DATA\_RAM
* UART
* LED&KEY
* PS2
* SPI
* GAME\_ENGINE(PPU&APU)
* CNN Accelerator

所有模块通过AHB-Lite总线连接到AHB-Interconnect总线矩阵实现与Cortex-M3软核的交互。

其中与游戏功能实现相关的是：

PS2接口，完成游戏手柄数据的接收（完成）；

SPI\_FLASH：完成游戏相关数据的不易失存储，通过软件层面进行操作（完成）；

PPU\_FLASH：在游戏运行过程中加载一些图像显示数据，通过硬件层面进行读操作；与SPI\_FLASH通过一个2选1的MUX实现对同一个FLASH的操作（完成）；

GAME\_ENGINE中的PPU：完成游戏图像数据的显示处理及相关控制（完成）；

GAME\_ENGINE中的APU：完成游戏音效的控制（未完成）；

CNN\_ACC：完成对ov5640摄像头的采集的手势的识别（未完成）；

VGA\_INTR：VGA显示器游戏画面完成一帧渲染的中断信号，用于配合软件实现下一步操作。



图2-1 SoC架构

**3.2 Memory Map示意**

暂时没有最终确定，后续可能会修改。

**3.3 游戏图像处理单元PPU**

我们设计的PPU是用于图形压缩数据的解码和一些控制功能实现。由于我们的游戏是基于名叫tile的单元实现，有必要对tile的显示原理进行说明。Tile的本质其实就是一个8bits\*8bits的图案点阵。以显示彩色数字“1”为例，显示效果如下图所示。实际上在每一个这样的图案表（tile）中，每一个像素点只能显示四种颜色。至于每一个像素点显示哪种颜色，可以采用2bit位宽的数据进行表示：2’b00，2’b01，2’b10及2’b11。由此，一个图案表需要2个8bits\*8bits的点阵数据，也就是一个图案表需要16Byte表示。示例中的“1”的图案表数据如图2-3所示。至于每一个像素点的2bit颜色索引对应的具体的RGB色彩则需要由调色板（palette）确定。将图2-3中两张表进行高低位拼接并对从调色板索引RGB值便能得到这一张彩色数字“1”的正确显示，如图2-4所示。



图2-2 彩色数字“1”的显示



图2-3 彩色数字“1”的图案数据



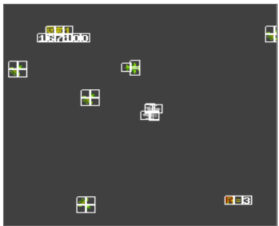
图2-4 对调色板的索引

PPU整体如下图所示，分为两大部分：sprite精灵绘制和background背景地图绘制。这两个模块需要结合VGA\_Driver的扫描坐标X，Y进行一定的计算。



**3.3.1 游戏活动单位sprite绘制的硬件实现**

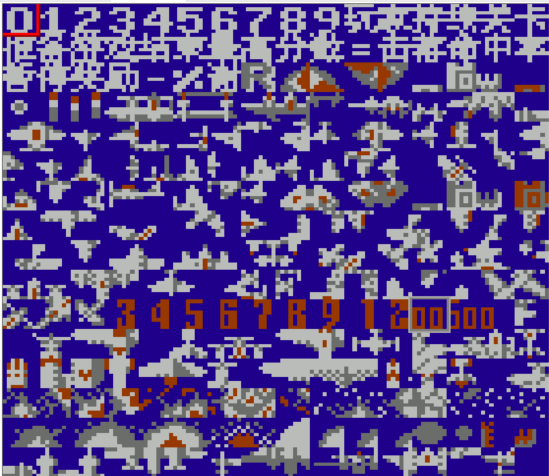
对于FC游戏而言，活动的单位称为sprite，也就是“精灵”，一个精灵就是一个tile。若干个sprite进行有序排列便能够形成有效的显示单位。以一个灰色向下的小飞机为例，如下图所示。这个灰色小飞机实际上是由3个tile组成，这三个tile都能在上文的tile库中找到。一次场景中显示的去背景的效果实际上如下图所示。



这一部分的硬件实现如下图所示。



spriteTileRom：作为精灵图案的素材库。在我们借鉴的游戏《1942》中，存在各式各样的游戏单位，每一个单位都有数量或多或少的tile组成，同时相同单位存在不同的运动姿态，如果采用直接存图片的方式，将会消耗掉较多的BRAM空间。由此我们采用游戏《1942》的tile库，如下图所示。这张表一共由256个tile组成，只有4KB。



palletteSprite：则存储了精灵显示的调色板数据（4个调色板）。

spriteRam：为了确定每一个tile的横坐标（Byte3）和纵坐标（Byte2），图案表索引（Byte1），调色板的选择（Byte0[5:4]），是否左右翻转（Byte0[7]），是否上下翻转（Byte0[6]）。由此每一个tile的控制字需要4Byte。游戏精灵数量不需要太多，我们在FPGA中使用256KB用于精灵控制字的存储，一共64个精灵，该模块的数据由M3软核进行数据写入。



tileDraw0~ tileDraw7：为了完成显示画面每一行的绘制，我们需要使用绘图单元进行绘制。由于可能会出现64个精灵同时出现在同一行的情况，理论上应该例化64个绘制模块进行绘制，但是这会消耗较多的资源，因此我们只采用8个绘制模块tileDraw，实现资源与显示效果的平衡。每一个tileDraw结合当前显示器对应的像素点完成对每一个tile精灵数据的解析译码，从而从spriteTileRom和palletteSprite中找出正确的RGB像素值输出。

eightRam8：由于显示器在对每一行进行扫描时，不可能在当前行扫描时间内去遍历64个精灵数据，因此需要在前一行的非显示时段去对遍历64个精灵数据进行扫描，暂存下一行需要扫描显示的精灵。直至扫描完成。



pixelMux：8个绘制模块还会出现其中精灵重合的情况。我们将精灵编号更小的进行显示。

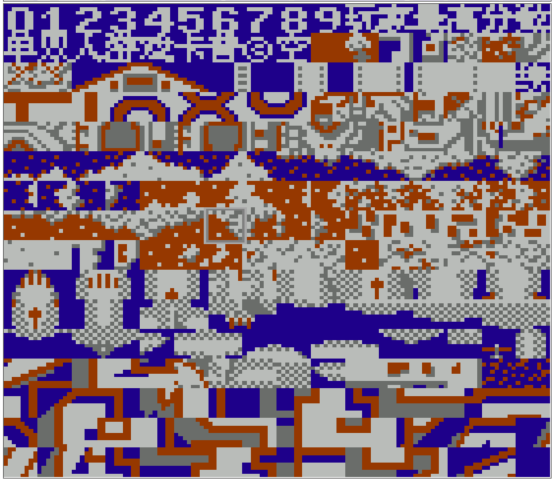


**3.3.2 游戏背景background显示与滚动控制**

标准的FC游戏画面是256\*240像素。同样的，游戏的背景也由一个个tile游戏排列组成。游戏背景贯穿整个游戏过程，包括开始，进行及结算。背景的绘制模块如下图所示。



backTileRom：作为背景图案的素材库。由此我们借用游戏《1942》的背景tile数据库，如下图所示。这张表同样只有256个tile组成，也就是4KB。

****

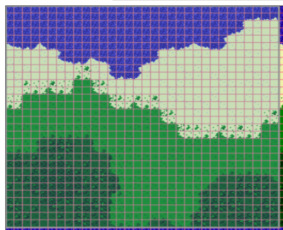
paletteBack：则存储了背景显示的调色板数据（4个调色板）。

nameTableRam：前文说到游戏运行是以256\*240像素运行，而游戏又是以8\*8大小的tile排列组合而成，因此背景需要由32\*30=960个tile组成。由于有256个背景图案tile，所以需要8bits的数据对背景图案库进行索引。因此一幅背景图案需要960Byte的数据量，被称为名称表nameTable。

attriTableRam：由于nameTableRam只是存储了tile的索引值，并没有存储调色板的索引值，所以属性表attriTableRam便存储了960个tile的调色板索引值。attriTableRam只有64Byte，每一个Byte决定了相邻的4\*4个tile的调色板，每一个Byte中的2bits决定了相邻4个tile的调色板的选择（相邻4个tile的调色板一致）。



由此，一幅完整的背景只需要1KB的存储空间。相比于直接存储12bit的彩色图案，压缩了近90倍。下面是一幅背图案的示意。



scrollCtrl：为了增加游戏背景的丰富程度，需要实现游戏背景的移动。我们存储两个名称表和属性表并且采用类似于编程中的快慢指针的方式实现背景的显示和移动。我们采用一个慢指针scrollPtr指向每一帧背景图的起始行的名称表位置并加载下一帧图像需要的名称表和属性表。

第一帧，第二帧及第六帧如下所示（简化示意图）。







为了控制背景移动速度，我们采用一个计数器实现对游戏显示帧的计数，每当一帧扫描完成时，该计数器加一。当计数器达到上限时，名称表指针scrollPtr移动一次。

flashToNametable：在整个游戏中，存在许多幅背景，不可能将所有的名称表和属性表全部存储到FPGA稀少的BRAM资源中，同时考虑到速度和M3搬运数据能力的问题，我们最终决定将每一幅背景对应的名称表和属性表预先存储到外部flash中，并在游戏进行中有scrollCtrl控制模块控制flashToNametable模块从外部flash中读取正确对应的名称表和属性表数据并写入到PPU的名称表和属性表内存中。

**3.4 游戏音频处理单元APU**

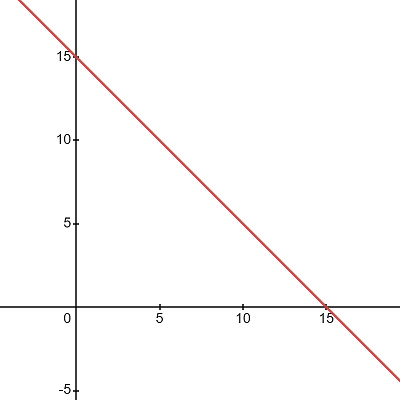
为了更好地还原NES游戏的音效，我们准备使用硬件实现游戏音频处理单元（APU），并结合软件部分，通过代码实现在特定的游戏事件发生时，播放相应的音效。合理利用板上资源，在不增加多余外设的前提下，利用板子自带的蜂鸣器实现音频播放，这不仅契合大赛要求，同时也更好地还原了NES游戏的实现。

**3.4.1 音乐与音效的原理介绍**

当我们按一定的节奏更改声音的频率，就形成了旋律。例如，按以下顺序更改频率就会听到“小星星”的旋律：

|  |  |
| --- | --- |
| 时间（毫秒） | 目标频率 |
| 0 | 261.63 |
| 500 | 261.63 |
| 1000 | 392.00 |
| 1500 | 392.00 |
| 2000 | 440.00 |
| 2500 | 440.00 |
| 3000 | 392.00 |

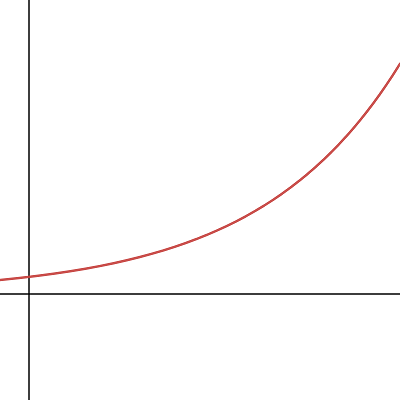
游戏中除了背景音乐外，还有各种音效，它们往往在跳跃、攻击、爆炸时触发。音效的原理与旋律一样，也是随着时间修改声音的一个或者多个属性，只不过音效一般时长很短、旋律简单。例如，按以下时间函数修改音量，就得到了一个淡出音效：



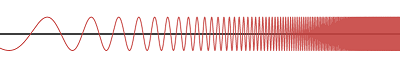
示例波形：

音量淡出示例波形

而按以下时间函数修改频率，就得到一个类似滑音的音效：



示例波形：



**3.4.2 APU的组成**

有了上面的基本知识，就很容易理解NES APU的组成了。NES APU包含4个波形生成器，称为声音通道。它们分别是，2个方波生成器、1个三角波生成器和1个噪声产生器



每个生成器至少包含以下组件，用于生成指定频率的波形：

1.定时器（Timer）。如同前文sine\_wave函数里面有循环，硬件上的循环用定时器实现。定时器用于驱动序列产生器（Sequencer），从而控制波形频率。

2.序列产生器（Sequencer）。用于产生波形，例如方波波形或者三角波波形。

其他各种组件：

3.包络生成器（Envelope Generator）。用于控制音量，支持固定音量和淡出音量（线性递减），音量最大值 15，最小值 0。它的时钟源为帧计数器（Frame Counter），但它内部包含分频器（Divider），可以进一步减小频率。

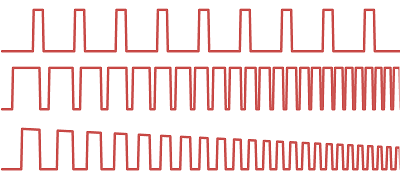
4.滑音单元（Sweep Unit）。用于实现滑音音效。它实现滑音的原理是，按一定算法定时修改通道的定时器。定时器时间间隔改变将导致波形频率发生变化，从而产生滑音效果。该单元也由帧计数器驱动，但输入频率被降低了一半。它内部也包含分频器，可以进一步减小频率。

5.长度计数器（Length Counter）。用于控制时长，它也由帧计数器驱动，但输入频率被降低了一半。

**3.4.3 方波通道**

方波包含全部组件，除了频率、振幅等常规属性，还有一个独特属性叫占空比（Duty Cycle）。NES APU 支持 4 种占空比，分别是：12.5%、25%、50%和75%。

以上各组件的搭配，可以生成各种方波：



**3.4.4 三角波通道**

三角波通道与其它通道不同，它不支持音量控制，只支持频率与时长控制。因此，除了定时器和序列产生器，还包含以下组件：

1.线性计数器（Linear Counter）。用于精细的控制时长，它由帧计数器驱动。

2．长度计数器（Length Counter）。它的作用、功能、特性等与方波通道的完全一样。虽然，它也能控制三角波的时长，但由于它的时钟信号频率太小，所以控制精度不如线性计数器高。

上面的各种组件的搭配，只能让三角波通道生成频率和时长不同的三角波。

**3.4.5 噪声**

噪声通道通常用来模拟枪炮、爆炸等音效，比如“魂斗罗”（Contra）游戏开场音乐最后的一声爆炸。这个通道除了定时器和序列产生器，还包含以下组件：

1.包络生成器（Envelope Generator）。它的作用、功能、特性等与方波通道的包络生成器完全一样。

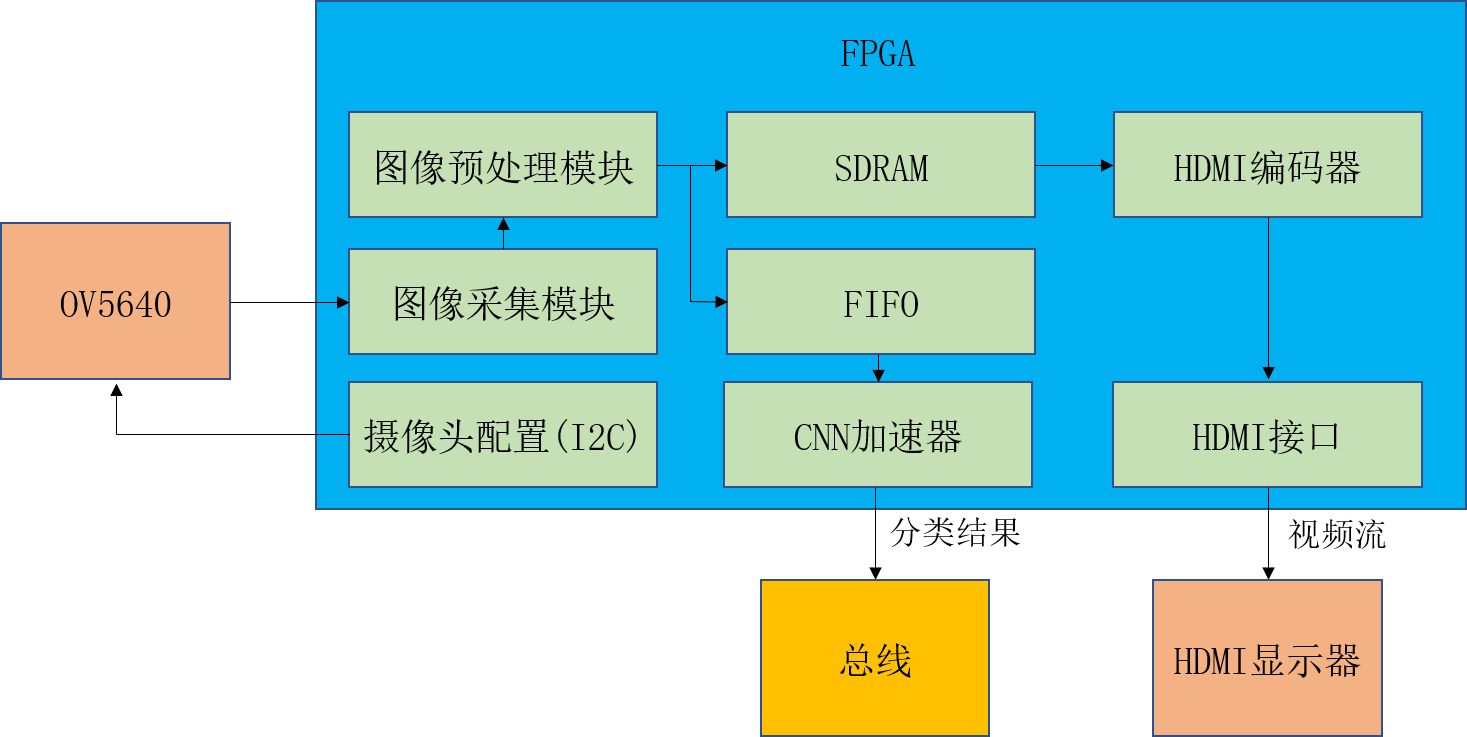
2.长度计数器（Length Counter）。它的作用、功能、特性等与方波通道的完全一样。

噪声通道的序列产生器比较有意思，它是由线性反馈移位寄存器（LFSR）实现的，线性反馈移位寄存器在此处的作用是产生随机的0和1，所以噪声通道的序列产生器能够产生0、1随机序列。

**3.5 CNN神经网络加速器与图像采集、显示设备**

**3.5.1 手势交互系统结构**

手势交互系统硬件包括OV5640摄像头、HDMI显示屏、FPGA板卡。程序框图如下图所示：

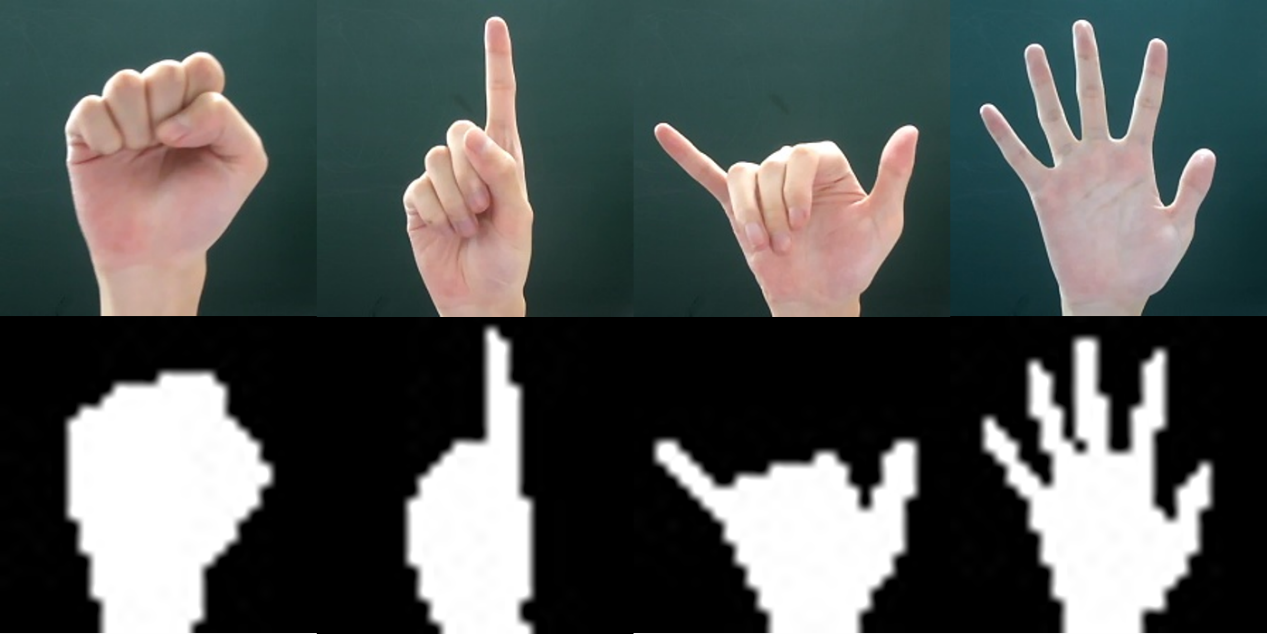


视频流从OV5640输入图像采集模块，单个像素为RGB565格式。图像预处理模块的功能有：截取2个224\*224像素区域作为双手的图像采样区域，并在区域周边添加可视边框；将图像采样区域的像素灰度化；将图像采样区域的8bit灰度流输入FIFO模块，整个视频流输入SDRAM暂存，视频流从SDRAM读出，通过HDMI编码器输出到显示器。两个区域的灰度流暂存到FIFO中，依次输入CNN加速器进行运算，在总线上输出双手的手势分类结果，实现手势与SoC系统的交互。

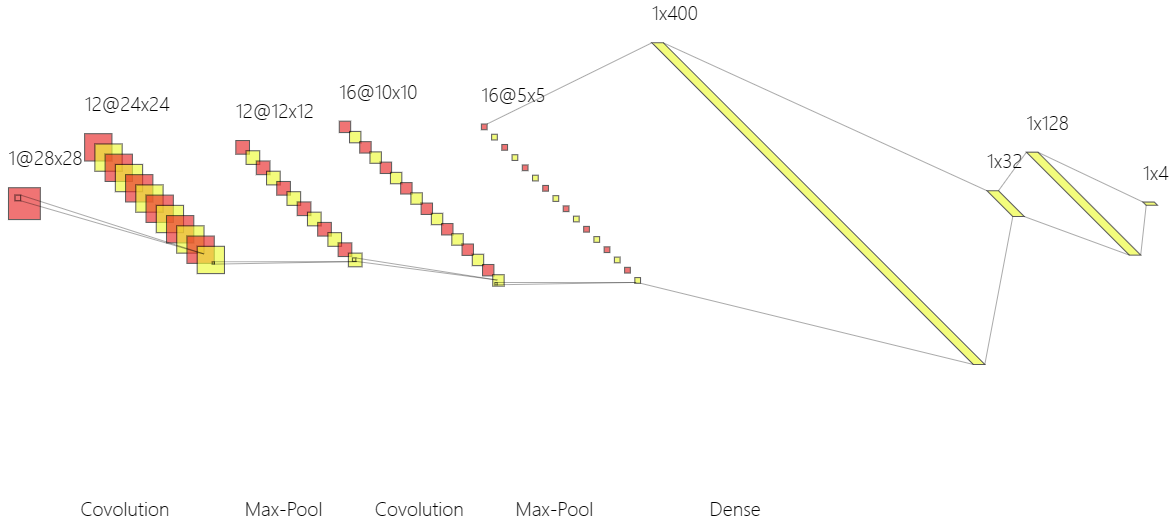
**3.5.2 CNN加速器设计**

相比传统算法，卷积神经网络在图像识别分类任务上有着稳定性好，泛化能力强等优点，已逐渐成为手势识别、人机交互领域的主流算法。卷积神经网络涉及多个卷积核的同时运算，十分适合发挥FPGA的并行计算优势，进行计算的加速。

在游戏的交互中，我们设置了4种手势，分别是fist、one、six、five。配合双手可以完成8种游戏操作。在输入神经网络前对224\*224的灰度图片降采样为28\*28并进行二值分割，能够保留手势的识别特征并降低神经网络计算量。

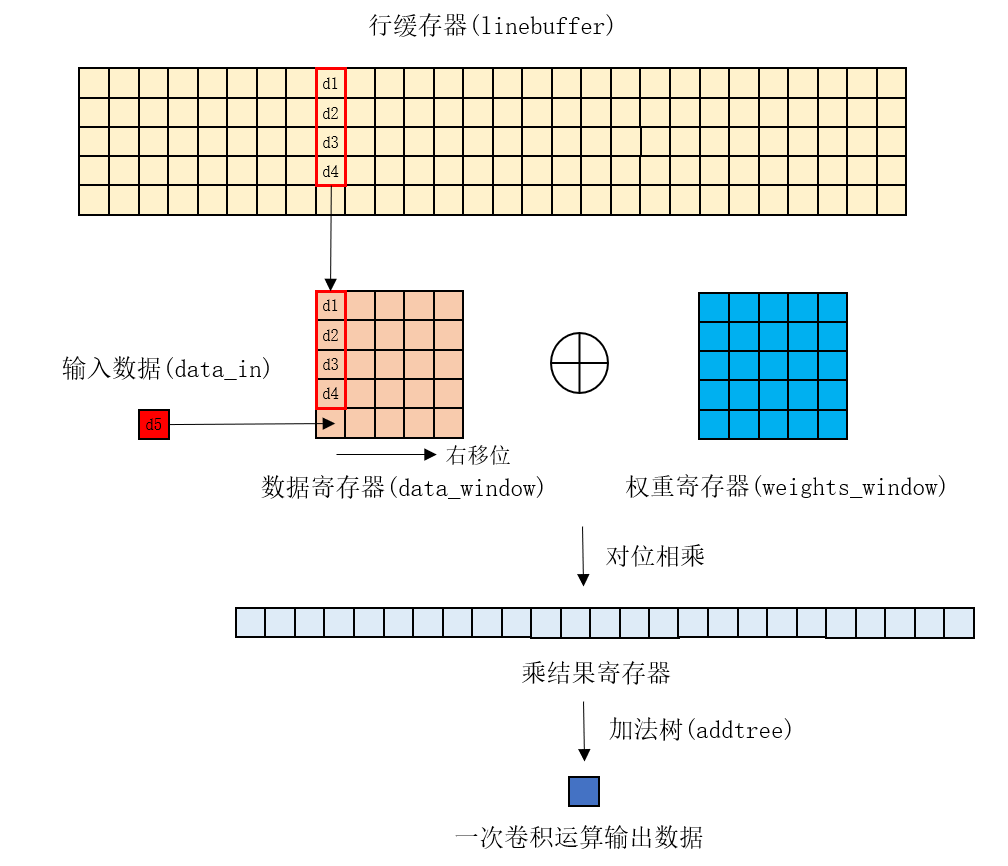


CNN网络结构采用2层卷积和3层全连接分类器，第一个卷积层的卷积核尺寸为5\*5，输出12个24\*24的特征图；第一个池化层将12个特征图降采样为12\*12；第二个卷积层的卷积核尺寸为3\*3，输出16个10\*10的特征图；第二个池化层将16个特征图降采样为5\*5。全连接层的节点数分别是32、128、4，输出4个数值，其中最大的一个即代表神经网络判断的手势类别。



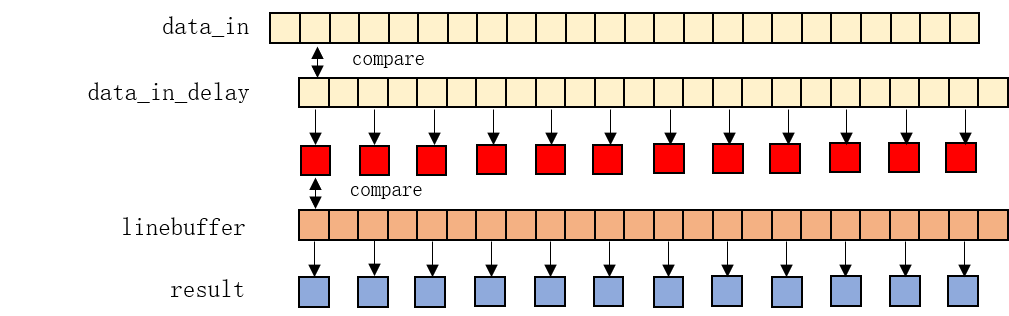
（1）第一个卷积层设计

二值化的数据流是串行输入，一次推理连续输入28\*28个数据，CNN加速器的第一层卷积采用全并行设计：每输入一个数据，在一个时钟周期内并行地进行12个通道的卷积操作，输入速度与输出速度相同。如下图所示，对于5\*5的卷积核，图像预先缓存4行才能进行第一次运算，使用双口RAM来实现linebuffer，预存5行像素，并读出和当前输像素相匹配的缓存像素来组成一个完整的计算窗，计算窗每个时钟周期更新一次。在每个时钟周期内，窗内的25个数据和卷积核对位相乘得到25个结果，再使用加法树流水线将25个数在3个时钟周期内相加，输出一次卷积操作的结果。



（2）池化层设计

池化操作是一种图像降采样方法，能够压缩特征图，防止过拟合。这里使用的CNN网络中池化核尺寸都是2\*2，计算方式为保留池化核内的最大值（MaxPool）。由于池化核尺寸为2\*2，只需要缓存一行图像数据。首先将输入数据寄存一个周期，得到data\_in和data\_in\_delay，二者进行比较即可实现输入的一行的数据两两比较，结果存放在linebuffer中。下一行输入时data\_in和data\_in\_delay比较一次，再和linebuffer对应内存位置的数据比较即可实现4个数取最大值的操作。



（3）第二个卷积、池化层设计

考虑到FPGA的DSP资源只有120个，第一层全并行卷积已经消耗50个DSP，第二层卷积只能采用串行设计。第二层输入12通道，输出16通道，这些通道之间串行运算，消耗的DSP资源很少。数据缓存、计算与第一层相同，但由于第二层的计算速度比第一层慢，第一层池化输出的数据不能直接输入到第二层卷积层，需要一个FIFO来暂存数据。当第二层卷积完成一组串行计算得出正确输出时再向FIFO请求下一个数据。

（4）全连接层设计

由于第二层的串行设计，只需将输出数据乘以对应的权重并进行累加，只需要消耗一个乘法器来进行运算。

（5）权重的内存分配

卷积神经网络的权重在网络训练完成时就已经确定，在FPGA前向传播计算时不会发生改变。在训练时采用的是float32类型，在FPGA中计算和储存浮点数都要消耗大量资源，需要将权重量化为定点数。量化操作实际上是一种映射方法，将浮点数映射为int8(16位定点数)或int4(8位定点数)类型，又称作int8/int4量化。经大量学者的研究，神经网络会将量化误差视为一种噪声，并不会显著降低神经网络的特征提取能力。使用的CNN网络约2万参数，量化为16位定点数消耗40KB存储资源，这对于指定的FPGA型号来说是可以接受的。

**4. 软件方案**

**4.1 游戏机制与玩法**

整个游戏过程如下图的状态机所示，包括游戏开始界面，游戏进行界面和游戏结算界面。游戏开始界面包括一些相关的选项内容（后续有待完善），游戏进行界面主要包括敌机的创建，各单位的各种数据更新及碰撞检测。如果检测到我方飞机被碰撞则结束游戏并转到游戏结算界面，再次按下确认键返回游戏开始界面。



**4.1.1 游戏各类单位设计**

我方飞机：发射子弹及一次发射的子弹数量，闪避技能（闪避期间不被撞击或者击中）且有动画；

我方子弹：直线路径；

灰色小型敌机：折返随机且折返具有动画；是否携带子弹随机；方向随机；

中型敌机：从底部出现；直线向上；不携带子弹；

绿色大型敌机：生命值高，发射子弹数量多，停留时间长；

敌机子弹：具有方向计算，始终射向我方飞机；

爆炸：具有动画（一共四帧）。



**4.1.2 碰撞检测**

游戏中涉及到许多单位的碰撞检测，且每一帧都需要做出碰撞检测，碰撞的类型也很多样。对于本次模仿实现的射击游戏《1942》，单位类型主要是，我方飞机，我方子弹，敌方飞机，敌方子弹及游戏BUFF。

* 我方飞机与敌方飞机，敌方子弹的碰撞。
* 我方子弹与敌方飞机的碰撞。
* 我方飞机与BUFF的碰撞。

同一帧画面中，如果我方飞机能够最多发射20颗子弹，敌方子弹最多发射20颗，敌方飞机单位20架次，我方飞机1架次，BUFF一个。如果采用距离的方式进行运算（涉及到加减，乘法甚至开方运算），需要计算20\*（20）+1\*（20+20）+1=441次距离计算，将会占用CPU较多的资源。

BitMask位掩码碰撞检测：由于本次游戏的画面的基础单位是8\*8的tile，因此我们不必从像素点级别去计算，我们可以考虑从tile角度去实现碰撞检测。全屏的tile数量为32\*30，也就是将256\*240的画面分成32列\*30行的网格，每一个网格用1bit表示是否存在碰撞体，有碰撞体为1’b1，无碰撞体为1’b0。碰撞位图可以用数组表示，如下。

bit hitMap[32][30]

实际碰撞检测，如下图所示。我方飞机位于图中下方四格（绿色数字标注），我方子弹又黄色数字标注，敌机由红色数字标注。如果要检测我方飞机是否发生了碰撞，我们便从敌机的hitMap中去索引相同位置的bit数值，如果存在1，则我方飞机发生了碰撞。以下图的例子来看我方飞机并没发生碰撞，而从上往下数的第二个敌方飞机被我方右下角的那颗子弹击中。



**4.1.3 敌机生成与阵列设计**

后续计划，还没完成。

**4.1.4 关卡地图设计**

目前完成了第一关地图的设计。

并将名称表和属性表存储在外部flash中（画出flash地图的数据的存储位置分布）。后续有待完善。

**4.1.5 游戏音效的软件控制**

后续需要完成。

**4.2 手势识别的软件部分（与游戏如何结合）**

采用手势识别结果代替手柄进行操作。目前具体操作还未落实。

**5. 总结和后续计划**

**5.1 后续计划**

代码移植：由于游戏实现是在Artix7板卡上完成，后续确定完响应的外设之后，需要移植到官方指定的安路PH1A60板卡上。

游戏功能：目前基本实现了游戏功能，具有一定的完整性和趣味性。后续计划增加游戏buff的设置；完善游戏关卡的设置；完成敌机生成的机制，增加敌机阵列的编排；增加游戏boss的设定。

游戏智能部分：继续完善CNN加速器部分。

**5.2 总结**