《视频图像处理》

[《视频图像处理》 1](#_Toc184119092)

[第一节:视频图像基础 3](#_Toc184119093)

[1.1图像 3](#_Toc184119094)

[1.1.1像素点 3](#_Toc184119095)

[1.1.2像素时钟 4](#_Toc184119096)

[1.2视频 5](#_Toc184119097)

[1.2.1帧率 5](#_Toc184119098)

[1.3视频图像处理 5](#_Toc184119099)

[1.3.1邻域 6](#_Toc184119100)

[1.3.2邻域选取 6](#_Toc184119101)

[第二节：图像滤波 7](#_Toc184119102)

[2.1均值滤波 8](#_Toc184119103)

[2.1.1均值滤波算法验证 8](#_Toc184119104)

[2.1.2验证结果 9](#_Toc184119105)

[2.1.3均值算法优化 10](#_Toc184119106)

[2.2中值滤波 11](#_Toc184119107)

[2.2.1冒泡排序算法 12](#_Toc184119108)

[2.2.2冒泡排序算法HDL实现 12](#_Toc184119109)

[2.2.3中值滤波验证 13](#_Toc184119110)

[第三节：视频图像处理通用性程序设计与验证 14](#_Toc184119111)

[3.1视频图像处理电路设计原理 17](#_Toc184119112)

[3.2视频图像处理数字电路通用性模版设计 17](#_Toc184119113)

[3.2.1运算电路程序实现 19](#_Toc184119114)

[3.2.2地址计算电路设计与实现 19](#_Toc184119115)

[3.2.2.1字地址 22](#_Toc184119116)

[3.2.2.2地址计算程序实现 23](#_Toc184119117)

[3.2.3人机交互设计 24](#_Toc184119118)

[3.2.4视频图像处理控制程序设计与实现 25](#_Toc184119119)

[3.2.4.1流水线运行特征 26](#_Toc184119120)

[3.2.4.2视频图像处理控制程序设计 27](#_Toc184119121)

[3.3视频图像处理验证 29](#_Toc184119122)

[3.3.1仿真 29](#_Toc184119123)

[3.3.2视频图像处理验证仿真模型构建 30](#_Toc184119124)

[3.3.2.1仲裁器 31](#_Toc184119125)

# 第一节:视频图像基础

生活中音像数据是一个庞大的数据，自然离不开对影像数据的 各种处理，比如GPU是视频图像处理专用的ASIC数字电路，它和DSP，cpu等数字电路在实际应用中扮演者不同的角色，视频图像处理的应用是非常广泛。

## 1.1图像

图像就是**影像数据**在计**算机或者电路中**的一种表示，在一个平面上能够展现出连续画面，在电路或者计算机中表示连续的画面是不现实的，通过离散的表示方法来展现连续的画面。

通过以上可以看出，任何影像数据在平面上展现，都是离散表示，被整个平面分割成较小的区域，只要区域足够的小，肉眼看到的是连续的，我们把这个较小的区域叫做像素点。

### 1.1.1像素点

任何图像在计算机和电路中都是以像素点为最小的表示单位，换句话说像素点是图像的最小单位，任何像素点作为最基本的单位，像素点展现出来的是色彩。

像素点在实际中，它的值不同表示出不同图像，下来我们探讨像素值由哪些分类

1. 二值图像

二值图像值的是每个像素点的值只有2个值，一般情况下二至图像的像素点值用1比特表示，比如点阵图。

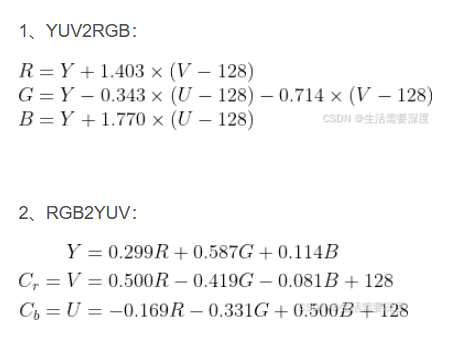
除了以上标准的二值图像以外，黑白图像也可以认为是二值图像

1. 灰度图

灰度图每个像素点用8比特表示，取值范围[0:255]，叫做灰度值，黑白图也可以认为是灰度图的一种特殊情况。

1. 彩色图像

彩色图像通常一个像素点需要3个值表示，这三个值表示颜色，构成了色彩空间，比较常见的色彩空间由RGB,YUV,HSV,CMY，其中RGB和YUV是2中最基本的色彩空间，这个空间转换如下：



**YUV色彩空间中Y分量构成的图像就是灰度图像，当我们需要把彩色图像转换为灰度图，我们只需要转换到YUV色彩空间获取Y分量即可得到灰度图**。

### 1.1.2像素时钟

在实际中图像在平面上连续的展示是通过离散的方式表示，在平面上所有像素点不是同时出现，而是分时出现，整个平面所有像素点显示需要的时间，一般用它的倒数，即频率表示，叫做刷新频率，时间叫做刷新时间，每个像素点显示需要的时间的倒数叫做像素时钟。

## 1.2视频

视频本质上是多福图像在时间轴上动态显示的过程，每幅图像之间的时间间隔足够小，肉眼是分别不出来，展现出来的是连续的动态过程。

视频本质上还有由图像构成，图像的基本单位是像素，视频也是以像素作为基本单位。

### 1.2.1帧率

视频中一副图像我们通常叫做一帧图像，每一帧图像动态显示的时间间隔我们用帧率表示，应用有的场景用fps表示，有的用hz表示。

帧率越高对硬件需求越高，图像的刷新率也越高。

## 1.3视频图像处理

视频图像处理是一个典型的应用，直观的说就是对视频或者图像数据进行各种操作，都叫做视频图像处理，归根结底是对像素点的操作，可以对一副图像的像素点操作，也可以对多福图像的像素操作。

对图像的操作，即视频图像处理我们归纳起来有以下2种：

变换：针对一副或者多福图像的像素点经过科学运算，实现相应变换。

控制：针对一副或者多福图像的的相关控制，比如DDRx读写控制

### 1.3.1邻域

在视频图像处理中，有个关键的概念，我们称之为领域，在诸多应用中都离不开邻域，所谓邻域就是视频图像处理中参与运算的所有像素点的集合。

当我们在对视频图像数据进行运算时候，我们需要考虑哪些像素点参与运算，有的运算需要单个像素点参与运算，有的时候需要多个像素点参与运算，这里我们多个像素点就构成了邻域。

邻域

分辨率=K1\*K2

### 1.3.2邻域选取

在应用中关于邻域的选取也有要求，由于运算基本都是基于中心像素点运算，因此一般选取完全对称邻域。

|  |  |
| --- | --- |
| P00 | P01 |
| P10 | P11 |

2\*2邻域（非对称邻域，共4个像素点）

|  |  |  |
| --- | --- | --- |
| P00 | P01 | P02 |
| P10 | P11 | P12 |

2\*3邻域（左右对称邻域，共6个像素点）

|  |  |
| --- | --- |
| P01 | P01 |
| P10 | P11 |
| P20 | P21 |

3\*2邻域（上下对称邻域，共6个像素点）

|  |  |  |
| --- | --- | --- |
| P00 | P01 | P02 |
| P10 | P11 | P12 |
| P20 | P21 | P22 |

3\*3邻域（完全对称邻域，共9个像素点）

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| P00 | P01 | P02 | P03 | P04 |
| P10 | P11 | P12 | P13 | P14 |
| P20 | P21 | P22 | P23 | P24 |
| P30 | P31 | P32 | P33 | P34 |
| P40 | P41 | P42 | P43 | P44 |

5\*5邻域（完全对称邻域，共9个像素点）

在实际中邻域选取以完全对称邻域作为邻域，对称中心对应的像素点叫做中心像素点，也叫做目标像素点，我们往往以最小完成对称邻域3\*3作为我们的邻域。

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |

注意：边缘像素点没有邻域

# 第二节：图像滤波

图像滤波是视频图像处理中常见的一种处理变化算法，主要是对图像左平滑处理。

## 2.1均值滤波

用邻域的平均值代替源目标像素点的值，构成新的图像，这个图像就是滤波后的图像。

Oval = (p00+p01+p02+p10+p11+p12+p20+p21+p22)/9

### 2.1.1均值滤波算法验证

img\_filter

P11[7:0]

P10[7:0]

p02[7:0]

p01[7:0]

p00[7:0]

oval[7:0]

P20[7:0]

P12[7:0]

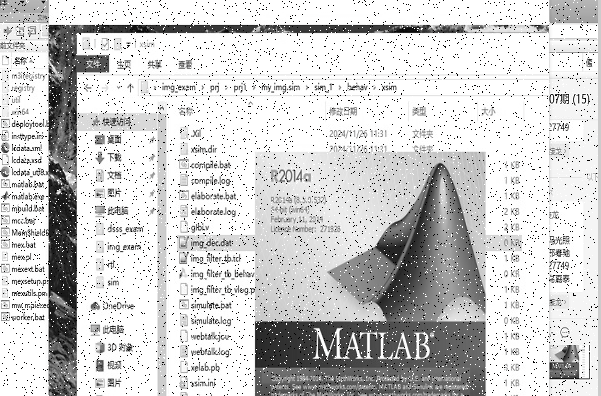
P22[7:0]

P21[7:0]

端口说明

|  |  |
| --- | --- |
| 信号名 | 描述 |
| 邻域像素点 | |
| p00[7:0] | 邻域1\*1像素点 |
| p01[7:0] | 邻域1\*2像素点 |
| p02[7:0] | 邻域1\*3像素点 |
| p10[7:0] | 领域2\*1像素点 |
| p11[7:0] | 领域2\*2像素点，中心像素点 |
| p12[7:0] | 领域2\*3像素点 |
| P20[7:0] | 领域3\*1像素点 |
| P21[7:0] | 领域3\*2像素点 |
| P22[7:0] | 领域3\*3像素点 |
| 输出平均值 | |
| oval[7:0] | 邻域平均值 |

### 2.1.2验证结果



滤波后

滤波前



### 2.1.3均值算法优化

在硬件设计中设计按照抽象层次分为系统级，算法级，RTL级，门极开关级，当我们在进行算法实现的时候，能够把算法级的算法转化为RTL级则转换，在设计中尽量避免使用乘法器，触发器。

在硬件设计中求均值可通过加权平均法实现求平均值，加权平均法是可以通过加减移位等操作实现。

假如一学年的考试分为3次，第一季度考试，其中考试，期末考试，小明第一季度的数学成绩是85分，其中开始数学成绩是60分，期末考试数学成绩是70分，求小明1学年的数学平均成绩。

数学平均成绩数学平均 = （85分+60分+70分）/3 = 71.67分

数学平均可以反应小明这一学年的数学学习的基本状况，但是不是绝对很全面，为了全面反应小明这学年的数学学习基本情况，可采用如下的方式计算：

数学平均成绩 = （1/4 \* 85分） + （1/4\*60分） + (2/4)70分

= 21.25+15+35 = 71.25

**数学平均成绩加权平均 = （（1 \* 85分） + （1\*60分） + 2\*70分 ）/4**

权值：根据实际应用情况合理分配

权值：根据实际应用情况合理分配

权值：根据实际应用情况合理分配

**数学平均成绩加权平均 = （（1 \* 85分） + （1\*60分） + 1\*70分 ）/3**

**数学平均成绩加权平均 = （（1 \* 85分） + （2\*60分） + 3\*70分 ）/6**

通过以上的加权平均法分析，我们可以将均值滤波算法按照加权平均的方式实现，合理的分配权值便可以更好的通过硬件实现。

|  |  |  |
| --- | --- | --- |
| 1 | 2 | 1 |
| 2 | 4 | 2 |
| 1 | 2 | 1 |

|  |  |  |
| --- | --- | --- |
| P00 | P01 | P02 |
| P10 | P11 | P12 |
| P20 | P21 | P22 |

Oval = (p00\*1 + p01\*2 +p02\*1 +

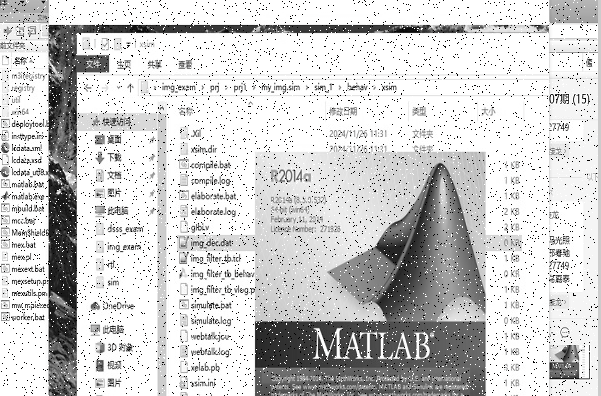
P10\*2 + p11\*4 +p12\*2 +

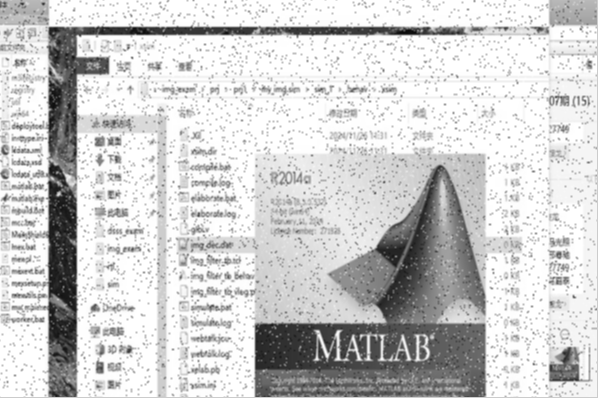
P20\*1 + p21\*2 +p22\*1 )/16;

= **(p00\*<<0 + p01<<1 +p02<<0 +**

**P10<<1 + p11<<2 +p12<<1 +**

**P20<<0 + p21<<1 +p22<<0 )>>4;**





滤波后

滤波前

## 2.2中值滤波

所谓中值滤波就是用邻域的中间值代替原来中心目标像素点的值，构成的图像就是滤波后的图像。

在硬件实现中值滤波的时候，核心就是邻域的数据的排序问题，排序的算法有很多，选择一种适合硬件实现且占用硬件资源不多的算法即可，常见的排序算法有;



以上排序算法是计算机数据结构原理中的基本排序算法，各种排序算法所需要的时间（计算器软件运行），稳定性各不相同，在硬件设计中常使用冒泡排序算法，也经典排序算法，适合硬件实现。

### 2.2.1冒泡排序算法

MLB

MSB

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| N-1 | N-2 | N-3 | N-4 | … | 1 | 0 |

**第N比较交换**

D0

将1个数经过1次比较交换

**第N-1比较交换**

D1

将3个数经过2次比较交换

**第4比较交换**

Dn-4

将N-3个数经过N-4次比较交换

**第3比较交换**

Dn-3

将N-2个数经过N-3次比较交换

**第2比较交换**

Dn-2

将N-1个数经过N-2次比较交换

**第1比较交换**

Dn-1

将N个数经过N-1次比较交换

遍历 ：如上图所示，我们把每一次的比较交换过程叫做遍历

交换比较：2个数比较大小，同时交换

### 2.2.2冒泡排序算法HDL实现

比较交换电路模块

p01[7:0]

p00[7:0]

min[7:0]

max[7:0]

D1[7:0]

D0[7:0]

switch

遍历

第1次遍历

max43[7:0]

max42[7:0]

max42[7:0]

max41[7:0]

max41[7:0]

max40[7:0]

max40[7:0]

max33[7:0]

max332[7:0]

max33[7:0]

max32[7:0]

max32[7:0]

max31[7:0]

max31[7:0]

max30[7:0]

max30[7:0]

max25[7:0]

max24[7:0]

max24[7:0]

max23[7:0]

max23[7:0]

max22[7:0]

max22[7:0]

max21[7:0]

max21[7:0]

max20[7:0]

max20[7:0]

max16[7:0]

max15[7:0]

max15[7:0]

max14[7:0]

max14[7:0]

max13[7:0]

max13[7:0]

max12[7:0]

max12[7:0]

max11[7:0]

max11[7:0]

max10[7:0]

max06[7:0]

max07[7:0]

max05[7:0]

min06[7:0]

max06[7:0]

P20 [7:0]

max04[7:0]

min05[7:0]

max05[7:0]

P12 [7:0]

max03[7:0]

min04[7:0]

max04[7:0]

P11 [7:0]

max02[7:0]

min03[7:0]

max03[7:0]

min02[7:0]

max02[7:0]

P10 [7:0]

max01[7:0]

min01[7:0]

max01[7:0]

P02[7:0]

max00[7:0]

max00[7:0]

min00[7:0]

Switch07

Switch06

Switch05

Switch04

Switch03

Switch02

min00[7:0]

Switch01

Switch00

第5次遍历

第4次遍历

第3次遍历

第2次遍历

Switch12

Switch11

Switch10

min43[7:0]

Switch43

min42[7:0]

Switch42

min41[7:0]

Switch41

min40[7:0]

min30[7:0]

Switch40

min34[7:0]

Switch34

min33[7:0]

Switch33

min32[7:0]

Switch32

min31[7:0]

Switch31

min30[7:0]

min20[7:0]

Switch30

min25[7:0]

Switch25

min24[7:0]

min15[7:0]

Switch24

Switch23

min23[7:0]

Switch23

min22[7:0]

Switch22

min21[7:0]

Switch21

min20[7:0]

min10[7:0]

Switch20

min16[7:0]

min07[7:0]

Switch16

Switch15

min14[7:0]

Switch14

min13[7:0]

Switch13

min12[7:0]

min11[7:0]

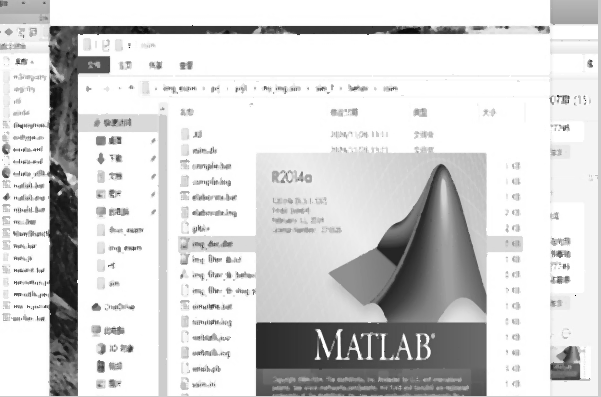
max10[7:0]

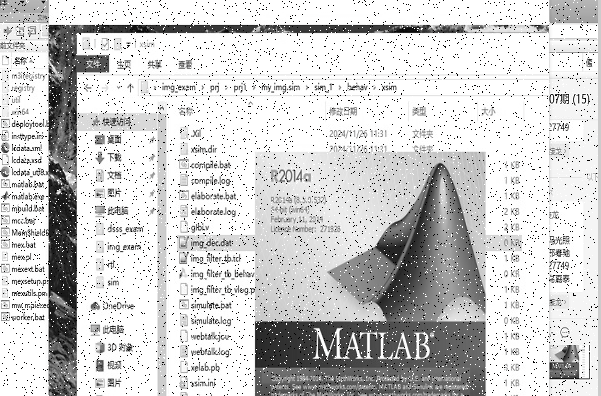
min10[7:0]

P22 [7:0]

P21 [7:0]

### 2.2.3中值滤波验证





滤波后

滤波前

# 第三节：视频图像处理通用性程序设计与验证

视频图像处理的本质上来说围绕像素点进行的数字电路设计，电路处理的元素就是像素点，在电路设计中根据其自己特点，程序的设计有一套通用的设计模版。

视频图像处理的电路设计特点

1. 数据量大

图像的数据量较大，尤其是视频，比如600\*400的恢复图像，数据一共240000字节的数据，0.24M字节的数据量，如果10帧图像就是2.4M字节的数据量，对于ON-chip-memery而言，这样数据量是相当庞大的，将视频图像数据存放在ON-chip-memery是不显示的，因此需要存放在片外存储器，如，sdram,ddrx,falsh…。

对于电路设计而言，我们需要将视频图像处理的数据先进行缓存，并且要缓存到片外存储器上，换句话说关于视频图像处理的电路设计需要片外存储器作为数据存储。

1. 运算

在视频图像处理中，大分部情况下都需要对图像做变化，即算法，在电路设计中，增加并行，适当采用一些技巧，比较常见的，比较重要的就是流水线设计。

在这里我们将流水线设计做详细的介绍，流水线设计数字电路设计的一种比较重要的设计技巧，它能够提高设计数据流的处理效率。

假如工厂组装生产车，我们通过传统的生产方式对比流水线生产方式

生产第3辆车

生产第2辆车

生产第1辆车

生产第…辆车

生产第N辆车

TN

T…

T3

T2

T1

传统生产方式

传统的生产方式是每一辆车组装生产完成才能组装生产下一辆车。

安装电器

组装

天窗等组件你

组装发动机

组装其它动力组件

底盘和相关组件

完整车辆

半成品

半成品

半成品

流水线生产方式

流水线生产方式，车辆的生产分成很多个不走完成的，后一个步骤加工的是前一个步骤已经加工完的产品，这种产生产方式叫做流水线生产线。

在很多工厂已经证明流水线生产线的方式效率要比传统的生产方式效率高很多，因此现在工厂基本都采用流水线产生方式，实践证明流水线产生适合批量产生。

在数字电路设计中，如果我们在进行数据处理的时候，数据量比较大，类似工厂的批量生产，如果我们把数据处理过程划分为多个步骤，把数据比作工厂配件或者半成品，后一个步骤处理的数据时前一个步骤处理的结果，采用这种方式就行数据处理同样会提高数据处理效率。

流水线设计适合数据量比较大的应用场景，视频图像处理数据量比较大，所以一般情况下采用流水线设计能够提高处理处理效率。

步骤….处理结果

步骤2处理结果

步骤1处理结果

步骤0处理结果

目标数据

步骤2

源数据

步骤0

步骤N-1

步骤…

步骤1

电路流水线设计结构

流水线电路设计特点：

* 步骤

流水线电路设计需要划分多个步骤，**每个步骤是一个独立的数据运算与控制的过程（在电路设计中每个步骤相对独立，不需要要等待前一个步骤）**。

流水线有几个步骤我们称之为流水线的级，比如4个步骤就叫做4级流水线。

* 数据依赖

所谓数据依赖指的是后一个步骤处理的数据来自于前一个步骤处理的结果。

* 并行性

由于每个步骤是相对独立的，所以同一时刻，所有步骤都在对数据进行运算处理。

步骤0

步骤1

步骤2

步骤2

步骤1

步骤2

步骤1

步骤2

步骤1

步骤2

步骤0

步骤0

步骤0

步骤1

步骤0

1. 数据操作

视频图像处理的为像素，像素数据被存放在片外存储器上，在设计中需要再存储器中划分存储器区域，分别为源图像存储器和目标图像存储器。

视频图像处理电路

其它电路读

视频图像处理电路写

视频图像处理电路读

视频图像源设备写

|  |
| --- |
|  |
|  |
| 源图像区域 |
|  |
| 目标图像区域 |
|  |
|  |

视频图像处理电路本质上就是从片外存储器读取原图像数据进行处理，在将处理结果写入到片外存储器的目标存储器区域。

## 3.1视频图像处理电路设计原理

目标存储器区域写

源存储器区域读

视频图像处理

目标存储器区域读

源存储器区域写

片外存储器

视频图像目标设备

如：显示器，软件

视频图像源设备

如：摄像头，软件

人机交互（可选，取决于场景需求）

软件

注意：在实际中，关于片外存储器的读写可能有2种方式

1：直接读写

所谓直接读写就是视频图像处理电路通过sdr sdram或者ddrx接口（比如AXI）直接读写数据.

这种应用主要用于数字电路独立进行视频图像处理的过程

2:间接读写

有时一个视频图像处理的较为复杂，需要软件，数字电路协议同工作完成，此时图像数据被存放在运行存储内部，我们可能需要间接通过系统总线读写数据，别入通过PCIE读写数据。

## 3.2视频图像处理数字电路通用性模版设计

通过以上视频图像处理电路设计原理，可以看出，视频图像电路的交互对象有片外存储器以及根据需求选择选择软件。

系统总线或者存储器接口

Axi,pcie，存储器接口

视频头图像处理电路

片外存储器

软件

适用于软件和电路协同工作

系统总线

比如AXI,PCIE，…

注意：软件交互一般适用于软硬协同处理的，并且大部分分情况下，存储器采用的是间接读写，和软件交互**采用同一个系统总线**。

本设计模型默认为软硬协同处理的电路设计模型，由于实际的总线或者接口较为复杂，这里我们在设计电路的时候自定义系统总线代替实际总线或者接口，系统总线定义如下：

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| bus\_cs\_n | In(slave) out(master) | 当bus\_cs\_n==0表示总线有效 |
| bus\_we | In(slave) out(master) | 当 we==1表示写  we==0表示读 |
| bus\_addr[31:0] | In(slave) out(master) | 总线地址 |
| Bus\_data[31:0] | inout | 总线数据 |
| bus\_ack | out(slave) in(master) | 从机应答，当bus\_ack==1表示读写成功 |

自定义总线功能定义

rst\_n

rst\_n

rst\_n

clk

bus\_ack\_o

bus\_we\_i

bus\_ack\_o

load0

load2

load1

bus\_we\_o

bus\_cs\_n\_o

Img\_ctl

（视频图像处理控制）

Img\_process

bus\_data\_i[31:0]

img\_kenerl

（像素数据运算）

en\_pipe

clk

done

start

result[31:0]

ui

（可选）

bus\_data\_o[31:0]

bus\_data\_i[31:0]

bus\_addr\_i[3:2]

rst\_a

inc\_dec

inc\_src

Ibase\_dec[31:2]

Ibase\_src[31:2]

clk

Img\_addr\_gen

bus\_addr\_o[31:0]

result[31:0]

bus\_cs\_n\_i

Irq

### 3.2.1运算电路程序实现

本模块运算默认求邻域均值，采用5级流水线的设计

clk

en\_pipe

load2

load1

load0

bus\_data\_i[31:0]

result[31:0]

img\_kenerl

（像素数据运算）

rst\_n

端口说明

|  |  |
| --- | --- |
| 信号名 | 描述 |
| 系统相关 | |
| clk | 系统时钟 |
| rst\_n | 系统复位 |
| 系统总线相关 | |
| bus\_data\_i[31:0] | 系统总线数据 |
| 控制相关 | |
| load0 | 当load0==1的时候默认从bus\_data\_i[31:0]  加载第1个32位的数据 |
| load1 | 当load1==1的时候默认从bus\_data\_i[31:0]  加载第2个32位的数据 |
| load2 | 当load12==1的时候默认从bus\_data\_i[31:0]  加载第3个32位的数据 |
| en\_pipe | 流水线控制信号，当en\_pipe==1流水线的每一个处理步骤输出处理结果 |
| 输出运算结果 | |
| result[31:0] | 视频图像运算结果 |

### 3.2.2地址计算电路设计与实现

对于视频图像处理而言，被处理的像素点的数据被存放外部存储器中，在处理的时候边处理变边读取边加载数据，同时处理结果也要写入外部存储器中，因此需要正确计算存储器的地址。

对于视频图像处理在外部存储器中会划分2个区域，分别源图像存储区域，和目标图像存储区域。

* 源图像存储区域 ：用来存放被处理前的图像像素点的数据

基地址

|  |
| --- |
|  |
|  |
| 源图像存储区 |
|  |
|  |

30

32=30+2

偏移地址地址

在地址计算的时候我们采用基地址和偏移地址的计算方式，这种计算方式适用于任何存储区的地址计算，包括计算器和其它电路设计。

源存储器地址 = 基地址 + 偏移地址

基地址：存储区域在整个存储器中的起始位置

偏移地址：相对于起始位置的偏移量

* 目标图像存储区域：用来存放被处理后的图像像素点的数据

该区域提前规划，大小根据实际需要确定其的存储区域大小。

|  |
| --- |
|  |
|  |
|  |
|  |
| 目标图像存储区 |
|  |
|  |

基地址

240032

240030

偏移地址地址

在地址计算的时候我们采用基地址和偏移地址的计算方式，这种计算方式适用于任何存储区的地址计算，包括计算器和其它电路设计。

源存储器地址 = 基地址 + 偏移地址

基地址：存储区域在整个存储器中的起始位置

偏移地址：相对于起始位置的偏移量

以上2个存储如果采用同一个存储芯片，这个区域不能重叠，如果是2个不同存储芯片，可分别存储。

目标数据写入

目标数据读出

源数据读出

源数据写入

存储器1

存储器0

目标数据读出

目标数据写入

源数据读出

源数据写入

存储器

单个存储芯片操作示意图

2个存储芯片示意图

无论我们使用几个存储芯片，地址计算按照通一地址空间进行计算，一律使用基地址，偏移地址的计算方式进行计算。

#### 3.2.2.1字地址

在数字电路设计中数据有大小，在数据处理的时候，当数据被存放在存储器的时候，有的时候一次读写单个数据，也可以读写多个数据，现在大部分片外存储器都支持这种操作（突发操作），在设计中我们把每次读写的多个数据叫做一个字。

字表示数据存储单位，在数字电路设计中字的大小取决于实际电路，如单个数据为16bit，每次读写数据128Bit,则一个字在这里表示128bit,即8个数据（128bit/16bit）。

在计算存储器地址的时候我们一般使用字地址，比如单个数据是8位，每次读写32位，这里一个字表示32位，即4个数据（32bit/8bit）字地址表示如下：

|  |  |  |
| --- | --- | --- |
| 字地址 | 存储器物理地址 | 映射关系 |
| 0 | 1 | 0\*4+1 |
| 1 | 5 | 1\*4+1 |
| 2 | 9 | 2\*4+1 |
| 3 | 13 | 3\*4+1 |
| 4 | 17 | 4\*4+1 |
| …. | | |
| N-1 | (N-1)\*4+1 | (N-1)\*4+1 |

我们把字地址和存储器物理地址之间的映射关系用一个公式表示：

存储器物理地址 = （字地址）\*4+k （K为非对齐偏移量）

是以上公式中如果k==0，则公式里的字地址是对齐的，我们叫做**字对齐地址**，几乎所有存储器在使用字地址的时候都需要使用字对齐地址。

|  |  |  |
| --- | --- | --- |
| 字地址 | 存储器物理地址 | 映射关系 |
| 0 | 0 | 0\*4 |
| 1 | 4 | 1\*4 |
| 2 | 8 | 2\*4 |
| 3 | 12 | 3\*4 |
| 4 | 6 | 4\*4 |
| …. | | |
| N-1 | (n-1)\*4 | (N-1)\*4 |

存储器物理地址 = 字对齐地址 \*4

字对齐地址 = 存储器物理地址/4

如果：存储器物理地址[31:0]表示，字地址用addr[31:2]表示

addr =  bus\_addr[31:2];

bus\_addr[31:0] ={ addr[31:2],2’b00 };

**通过以上分析，在地址计算的时候使用字地址，并且使用对齐后的字地址**。

#### 3.2.2.2地址计算程序实现

load2

load1

load0

rst\_a

inc\_dec

inc\_src

Ibase\_dec[31:2]

Ibase\_src[31:2]

rst\_n

clk

bus\_addr\_o[31:0]

img\_addr\_gen

端口说明

|  |  |
| --- | --- |
| 信号名 | 描述 |
| 系统相关 | |
| clk | 系统时钟 |
| rst\_n | 系统复位 |
| 控制相关 | |
| inc\_src | 当inc\_src==1源存储区地址自加1 |
| inc\_dec | 当inc\_dec==1目标存储区地址自加1 |
| rst\_a | 当rst\_a==1一次处理结束后，源存储区和目标存储区的地址回到起始地址 |
| load0 | 当load0==1表示当前正在读取源存储区第1行数据 |
| load1 | 当load1==1表示当前正在读取源存储区第2行数据 |
| load2 | 当load2==1表示当前正在读取源存储区第3行数据 |
| 人机交互相关 | |
| ibase\_src[31:2] | 源存储区的基地址，该地址为字对齐地址 |
| ibase\_dec[31:2] | 目标存区的基地址，该地址为字对齐地址 |
| 输出源或者目标存储区的物理地址 | |
| bus\_addr\_o[31:0] | 系统总线地址 |

### 3.2.3人机交互设计

有时候我们需要和软件进行交互，一般通过系统总线直接和软件进行交互的，采用命令参数解析法，本设计定义寄存器如下：

|  |  |  |
| --- | --- | --- |
| 寄存器名 | 属性 | 描述 |
| Ibase\_src[31:2] | 0（W） | 源存储区基地址 |
| Ibase\_dec[31:2] | 1（W） | 目标存储区基地址 |
| start[0:0] | 2（W） | 启动视频图像处理 |
| done\_r[0:0] | 3（R） | 表示当前一次处理结束 |

Ibase\_dec[31:2]

Ibase\_src[31:2]

rst\_n

clk

bus\_ack\_o

done

start

result[31:0]

bus\_data\_o[31:0]

bus\_data\_i[31:0]

bus\_addr\_i[3:2]

bus\_we\_i

bus\_cs\_n\_i

ui

（可选）

Irq

端口说明

|  |  |
| --- | --- |
| 信号名 | 描述 |
| 系统相关 | |
| clk | 系统时钟 |
| rst\_n | 系统复位 |
| 交互控制相关 | |
| srtat[0:0] | 启动视频图像处理 |
| Ibase\_dec[31:2] | 目标存储区基地址 |
| Ibase\_src[31:2] | 源存储区基地址 |
| done | 表示当前一次处理结束 |
| 与软件相关 | |
| irq | CPU中断信号 |
| bus\_cs\_n\_i | 系统总线片选信号 |
| bus\_we\_i | 系统总线写使能信号 |
| bus\_addr\_i[3:2] | 系统总线地址，字对齐地址 |
| bus\_data\_i[31:0] | 系统总线数据 |
| bus\_data\_o[31:0] |
| bus\_ack\_o | 系统总线从机应答 |
| 运算相关 | |
| result[31:0] | 视频图像处理运算结果 |

### 3.2.4视频图像处理控制程序设计与实现

在视频处理电路设计中，视频图像从一开始进行处理（必须保证运算所有需的像素点已经被存储在存储器中），到处理结束（源存储器数据从开始位置到存储区域最大位置的数据被运算算一次处理）是一次完整的处理过程，

T：视频图像处理时间

如24帧/秒的帧率，意味着在1秒时间内，必须处理24帧图像

T：视频图像处理时间

如24帧/秒的帧率，意味着在1秒时间内，必须处理24帧图像

处理结，done==1指示

处理开始

start==1指示

处理开始

start==1指示

处理结，done==1指示

空闲

视频图像处理中，一次处理从开始到结束，代表源存储区域数据一次处理过程，整个处理时间需要满足源数据写入源存储区域的时间。

以上从开始到结束整个过程需要用到本设计进行控制，现在我们讨论从开始到结束整个处理过程如何进行控制。

在控制中如果使用了流水线设计，在控制的时候也需要满足流水线运行的特征。

#### 3.2.4.1流水线运行特征

任何流水线在运行的时候都会经过3个阶段，分别是流水线启动，流水线稳定，流水线导出。

步骤N-1

步骤…

步骤01

步骤0

中间数据

中间数据

中间数据

目标数据

源数据

流水线启动：当在进行流水线处理的 时候，从有效数据进入到流水线开始，到出现第一个有效的目标数据之间的所有时间段，我们叫做流水线启动。

流水线稳定：当流水钱启动完成之后，有效的源数据不断进入流水线，当出现第一个目标有效数据之后，也同时会源源不断出现后续的目标数据，我们把这个时间段叫做流水线稳定，该时间段是流水线真正的工作时间。

流水线导出：当源数据没有有效的数据进入到流水线的时候，说明流水线即将停止，这里不是立马就停止，而是逐渐停止，在没有真正的停止之前目标数据任然有效。

从上面可以看出，流水线的处理也是一个过程，从启动到导出是流水线运行的一个过程，在这个过程源数据有效。

源数据有效

源数据有效

空闲

稳定

导出

启动

稳定

导出

启动

目标数据

目标数据

注意：流水线的每一次运行源数据必须一致有效

流水线的运行在实际中可能需要多次启动，每次流水线运行的时候只要数据不是一致有效，中间某一时刻数据无效，当重新有效的数据，流水线也要重新启动，我们就视图像处理就行分析，我们以\*3邻域运算为例

流水线导出

流水线启动

3\*3

3\*3

3\*3

3\*3

3\*3

3\*3

3\*3

3\*3

3\*3

3\*3

3\*3

3\*4

3\*4

#### 3.2.4.2视频图像处理控制程序设计

视频图像处理的基本过程

* 从源3\*4行数据并加载，保存在寄存器中(buf\_data\_i0, buf\_data\_i1, buf\_data\_i2)
* 读取的所有数据进行运算
* 判断运算结果是否有效，如果有效写入目标存储区域

流水线运行特征：当3\*4行数据每次下移一行的时候需要重新启动流水线

**If(bus\_ack\_i)**

**Bus\_cs\_n<=1;**

**Else bus\_cs\_n<=0;**

**Bus\_we<=0;**

**If(bus\_ack\_i)load2<=0;**

**Else load2<=1;**

**If(bus\_ack\_i)**

**Bus\_cs\_n<=1;**

**Else bus\_cs\_n<=0;**

**Bus\_we<=0;**

**If(bus\_ack\_i)load1<=0;**

**Else load1<=1;**

**If(bus\_ack\_i)**

**Bus\_cs\_n<=1;**

**Else bus\_cs\_n<=0;**

**Bus\_we<=0;**

**If(bus\_ack\_i)load0<=0;**

**else load0<=1;**

rst\_n

**读参与运算的像素数据**

**读数据并加载**

bus\_ack\_i

bus\_ack\_i

bus\_ack\_i

start==1

READ4\_2

READ4\_1

READ4\_0

**IDLE**

**en\_pile==1**

if（cn==3） cnt<=0;

else cnt<=cnt+1;

**if(cnt ==3)**

**if(col==`COL\_MAX/4)col<=0;**

**else col<=col+1;**

cnt==3 &col ==0

**运算**

**读出的数据都参与运算**

bus\_ack\_i&（（col==0） &（row！=`ROW\_MAX-3））:重新启动

（col>0 & (col<`COL\_MAX/4-1)）：稳定

en\_run

cnt==3 &col >0

（col==`COL\_MAX/4-1）&bus\_ack\_i

**写**

**运算结果写**

WRITE

col ==0 &row==`ROW\_MAX-3 & bus\_ack\_i

**If(bus\_ack\_i)**

**Bus\_cs\_n<=1；**

**Else bus\_cs\_n<=0;  
bus\_we<=1;**

**If(bus\_ack\_i & (col ==0))**

**If( row == `ROW\_MAX-3)**

**Row<=0;**

**Else row<=row+1;**

rst\_n

clk

start

bus\_ack\_o

rst\_a

inc\_dec

inc\_src

en\_pipe

load2

load1

load0

bus\_we\_o

bus\_cs\_n\_o

Img\_ctl

（视频图像处理控制）

done

端口说明

|  |  |
| --- | --- |
| 信号名 | 描述 |
| 系统相关 | |
| clk | 视频图像处理时钟 |
| rst\_n | 系统复位 |
| **系统总线相关** | |
| bus\_cs\_n | 总线片选 |
| bus\_we\_o | 总线写使能 |
| bus\_ack\_i | 总线从机应答 |
| **人机交互** | |
| start | 当start==1表示启动视频图像处理 |
| done | 当done==1表示当前一次处理处理结束，默认一帧图像处理完成表示结束 |
| **控制相关** | |
| load0 | load0==1表示加载第1个数据 |
| Load1 | load1==1表示加载第2个数据 |
| Load2 | load2==1表示加载第3个数据 |
| en\_pipe | 流水线控制，当en\_pipe==1每个运行步骤输出结果 |
| inc\_src | inc\_src==1源存储器偏移地址加1 |
| inc\_dec | inc\_dec==1目标存储器偏移地址加1 |
| rst\_a | 当rst\_a==1表示存储器区域地址回到初始位置 |

### 3.3视频图像处理验证

在数字电路设计中，所有的电路设计都需要验证，简单的说验证就是检测电路设计的正确性，包括逻辑验证和时序验证，功能型验证。

逻辑验证:通过某种方式对电路设计进行检验，根据验证方式，有在线验证（比嵌入式如逻辑分析仪）和虚拟验证（比如仿真），

时序验证：一般指的是静态时序分析

功能型验证：主要用于IC设计，在ASCI设计之前需要将功能在FPGA中验证设计方案的可行性，以及稳定性。

### 3.3.1仿真

**仿真是数字电路设计中常用的一种验证手段**，通过工具对设计进行分析，按照设计模拟信号值之间逻辑关系达到验证电路的木目的，标准的仿真是需要架构**建搭建测试验证模型**，其中用于搭建测试验证模型的顶层电路叫做tesebcnh.

testbench

设计电路

仿真电路模型

设计电路

设计电路

设计电路

设计电路

仿真电路模型…

仿真电路模型…

仿真电路模型…

仿真电路模型..

仿真验证顶层为testbench,包含仿真电路模型和设计电路2部分，其中设计电路就是我们需要验证的设计电路，为可综合的电路，另外一分部是仿真电路模型，在有的场合下为了对设计电路进行仿真验证，我们还需要编写其它一些虚拟的电路模块，这部分电路模块为非综合电路，在有的情况下是不需要电路仿真模型。

‘’

### 3.3.2视频图像处理验证仿真模型构建

在构建testbench的时候，有的场合构建起来是比较简单，但是有的时候需要构造一个较为复杂testbench，在这里我们要对视频图像处理进行仿真，除了设计文件还需要一些其他的仿真电路模型，才能进行仿真。

rst\_n

clk

**仿真电路模型**

clk

req[1:0]

bus\_ctl

系统总线控制器

bus\_ack\_i

bus\_we\_o

bus\_cs\_n\_o

rst\_n

clk

memery

存储器

bus\_ack\_i

bus\_we\_i

bus\_cs\_n\_i

rst\_n

clk

bus\_ack\_o

bus\_we\_i

bus\_cs\_n\_i

bus\_ack\_i

bus\_we\_o

bus\_cs\_n\_o

bus\_addr\_i[31:0]

bus\_addr\_o[31:0]

bus\_data\_o[31:0]

bus\_data\_i[31:0]

bus\_addr\_i[3:2]

sof\_cfg

CPU软件

bus\_data\_o[31:0]

bus\_data\_i[31:0]

gnt[1:0]

bus\_data\_i[31:0]

bus\_data\_o[31:0]

bus\_addr\_o[31:0]

arbiteri

临界冲突控制机制

### 3.3.2.1仲裁器

仲裁器是数字电路设计中一种设计，用于，由于在仿真过程中，sof\_cfg电路模块和img\_process电路模块之间存在临界冲突，因此需要引入仲裁器电路到仿真模型中。

临界冲突：有的电路中可能存在存在多个电路**共享式访问**总线或者存储器等电路，我们把被访问的电路叫做临界资源，共享式访问临界资源会存在冲突，我们叫做临界冲突。

仲裁器在数字电路设计中的作用就是临界冲突控制，设计仲裁器就是如何有效的控制临界冲突，仲裁器的控制方式可能会有很多，但是需保证基本的公平。

我们一存储器为临界资源，对仲裁器的工作原理进行分析

存储器读

存储器写

存储读写控制

存储器读

存储器写

存储器写

存储器读

存储器写

存储器

Dev2

Dev1

Dev0

当前时刻，如果dev0，dev1,dev2三个电路设备只有一个操作存储器的时候，允许它操作，如果三个设备有2个或者2个以上的电路操作存储器的时候，我们认为的规定优先级，优先级越高的设备优先操作存储器。

如果我们只规定当前时刻的电路设备的优先级，很难保证公平，为了保证公平，我们需要规定不同的时间段（称之为时隙slot），在不同的时间段内合理划分优先级。

以上例子中有三个电路设备，假如我们划分3个时间段，每个时间段的优先级规定如下：

Slot0: dev0 > dev1>dev2

Slot1: dev1>dev2> dev0

Slot2: dev2> dev0> dev1

通过分析可以看出以上三个电路设备划分了3个时隙，并且每个时隙都有一个设备的优先级最高，这样理论上是保证了公平性，并且绝对公平。

在实际中**时间段的划分为设备个数的正数倍**，优先级按照时间段的先后顺序，循环左移。

Slot0: dev0 > dev1>dev2

Slot1: dev1>dev2> dev0

Slot2: dev2> dev0> dev1

Slot3 dev0 > dev1>dev2

Slot4: dev1>dev2> dev0

Slot5: dev2> dev0> dev1

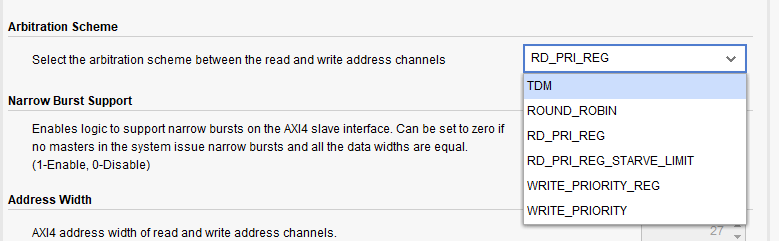
标准的仲裁器

时隙个数 = 设备数\*N

优先级 ：按照时隙的先后顺序循环左移

当前时刻：按照时隙的先后顺序依次为当前时刻，绝对公平的情况下每个时隙所持续的时间段时固定的。

以上是标准的仲裁器的工作原理，在实际中我们根据情况做适当的调整，也可以采用其它我们自定义的方式，比如TDM，round robin 等，绝对公平经常会造成时间的浪费，如AXI读写仲裁器



gnt[1:0]

req[1:0]

arbiteri

临界冲突控制机制

rst\_n

clk

端口说明

|  |  |  |
| --- | --- | --- |
| 信号名 | | 描述 |
| 系统相关 | | |
| clk | 系统时钟 | |
| rst\_n | 系统复位 | |
| 总线请求 | | |
| req[1:0] | req[1:0]  2’b01:表示sof\_cfg电路模块请求总线  2’b10:表示img\_process电路模块请求总线  2’b11:表示sof\_cfg和img\_process同时请求总线  2’b00:没有任何总线请求 | |
| gnt[1:0] | gnt[1:0]  2’b01：当前时刻允许sof\_cfg电路模块请求总线  2’b10：当前时刻允许img\_process电路模块请求总线  2’b00: 当前时刻不允许请求总线  2’b11:这种不允许存在 | |

Case(req[1:0])

2’b00:gnt<=2’b00;

2’b01:gnt<=2’b01

2’b10:gnt<=2’b10;

2’b11:gnt<=2’b10;

endcase

Case(req[1:0])

2’b00:gnt<=2’b00;

2’b01:gnt<=2’b01

2’b10:gnt<=2’b10;

2’b11:gnt<=2’b01;

endcase

req==2’b01

req==2’b10

SLOT0

rst\_n

SLOT1

img\_process> sof\_cfg

sof\_cfg>img\_process