

第七届

全国大学生集成电路创新创业大赛

报告类型\*： 设计报告

参赛杯赛\*： 芯来RISC-V杯

作品名称\*： 基于E203内核的轨迹跟踪加密芯片

队伍编号\*： CICC2623

团队名称\*： 吾开天工

# 1 摘要

在本次芯来杯赛中我小组完成了一款在视频图像领域中对运动轨迹跟踪和加密传输的SOC系统。此系统可完成对图像中运动目标的实时监测，并返回带目标检测的视频图像。同时通过国密SM4算法加密其坐标，最后传输至上位机实时追踪运动目标的运动轨迹。

针对赛题要求在对E203内核做了初步性能分析后发现，取指效率和指令执行周期数构成了E203内核的性能瓶颈。为了提高取指效率，我们将原本准确率有限的静态分支预测器修改为了基于全局历史的动态分支预测器，同时为了减少多周期运算指令的执行周期，我们在不降低主频的前提下设计了一款高性能乘除法器。经过FPGA实测，改进的E203在取指效率比改进前提升了**50%**,多周期运算指令执行周期数平均提升了**13**倍。最终在嵌入式基准程序实测中跑分为：

1）Coremark得分为3.267794 CoreMark/MHz，相比baseline提升**1.5**倍。

2）Dhrystone得分为4.661278 DMIPS/MHz , 相比baseline提升**3.6**倍。

3）Whetstone得分为0.125 MWIPS/MHz，相比baseline提升**3.1**倍。

由于基础RISC-V指令集中执行SM4国密算法需要进行大量的异或、移位以及盒变换等操作，使得基础E203内核若要执行SM4算法需要耗费大量的逻辑运算和移位指令数以及相当大功耗和执行周期数。因此我们在E203内核中添加了SM4算法的加解密相关的扩展指令，并在NICE协处理器接口上添加SM4加解密硬件加速电路来完成算法的硬件加速。经过实测添加了扩展指令的E203内核在执行SM4国密算法时相较于纯软件实现SM4国密算子，扩展硬件算子的指令数减少了**530倍**，消耗的指令周期数减少了**168倍**。

为减轻E203内核在图像视频中的运算量，我团队在SOC外设系统中设计了一款ImagePocessUnit（IPU，图像处理加速器）。来加速处理系统中的图像信号。加速器中集成了接收图像，处理图像，输出图像的功能。其工作流程为：摄像头先将视频数据传输到IPU，IPU会对摄像头视野中的运动目标进行检测，并在HDMI显示器上显示摄像头的实时监控画面，并用红框标记出运动目标的位置。并把其位置坐标传入E203内核。E203会将运动目标的坐标位置实时加密，并通过串口发送给上位机。

上位机由我团队用高级语言编写，上位机会实时对加密坐标进行解密，并将解密的坐标位置动态地显示在上位机的坐标区域中。并提供API供上层应用调用完成更多复杂的应用。

我们基于Xlinx平台建立了硬件测试平台，完成了SOC系统的板级验证。也通过VCS、Modelsim等仿真平台对硬件加速电路， E203内核都进行了单元级别，系统级别的功能验证，并尝试了敏捷开发的设计模式。通过FPGA综合结果可知在此设计可工作在E203内核的官方推荐频率下（16Mhz），并在面积和功耗上均满足赛题给定的设计要求。

目录

[1 摘要 2](#_Toc136404120)

[2 需求分析 8](#_Toc136404121)

[2.1 项目需求 8](#_Toc136404122)

[2.2 硬件需求 8](#_Toc136404123)

[2.3 软件需求 9](#_Toc136404124)

[3 系统方案设计 10](#_Toc136404125)

[3.1 硬件平台 10](#_Toc136404126)

[3.2 E203微架构改进 11](#_Toc136404127)

[3.2.1 动态分支预测器设计 13](#_Toc136404128)

[3.2.2 单周期高性能乘法器设计 18](#_Toc136404129)

[3.2.3 高性能除法器设计](#_Toc136404130)[[5]](#_Toc136404130) [22](#_Toc136404130)

[3.2.4 SM4国密算法硬件加速电路设计 23](#_Toc136404131)

[3.3 IPU外设设计 25](#_Toc136404132)

[3.3.1 IPU外设架构 25](#_Toc136404133)

[3.3.2 OV5640驱动模块设计](#_Toc136404134)[[8]](#_Toc136404134) [26](#_Toc136404134)

[3.3.3 RGB2YUV模块设计 28](#_Toc136404135)

[3.3.4 DDR控制模块设计 29](#_Toc136404136)

[3.3.5 VIP模块设计 33](#_Toc136404137)

[4 硬件分系统设计方案 35](#_Toc136404138)

[4.1 E203内核的改动模块 35](#_Toc136404139)

[4.1.1 动态分支预测器 35](#_Toc136404140)

[4.1.2高性能乘法器 40](#_Toc136404141)

[4.1.3 高性能除法器 41](#_Toc136404142)

[4.1.4 SM4硬件加速电路 42](#_Toc136404143)

[4.2 SoC外设系统 47](#_Toc136404144)

[4.2.1 IPU模块 47](#_Toc136404145)

[5 系统整体测试与分析 51](#_Toc136404146)

[5.1 分支预测器的性能测试 51](#_Toc136404147)

[5.1.1 测试流程 51](#_Toc136404148)

[5.1.2 测试分析 52](#_Toc136404149)

[5.2 上板的性能跑分测试 52](#_Toc136404150)

[5.2.1 测试流程 52](#_Toc136404151)

[5.2.2 测试分析 54](#_Toc136404152)

[5.3 SM4国密扩展指令测试 54](#_Toc136404153)

[5.3.1 SM4扩展算子底层软件库设计 54](#_Toc136404154)

[5.3.2 测试流程 56](#_Toc136404155)

[5.4 整体应用测试 59](#_Toc136404156)

[5.4.1 测试流程 59](#_Toc136404157)

[5.4.2 上板测试结果 59](#_Toc136404158)

[6 总结与展望 60](#_Toc136404159)

[6.1 总结 60](#_Toc136404160)

[6.2 展望 61](#_Toc136404161)

[参考文献 61](#_Toc136404162)

[图 1系统整体框图 10](#_Toc136404163)

[图 2系统硬件平台 11](#_Toc136404164)

[图 3 E203内核改进后的微架构 13](#_Toc136404165)

[图 4分支指令对流水线影响 13](#_Toc136404166)

[图 5 BPU整体架构 14](#_Toc136404167)

[图 6 Gshare算法原理图 15](#_Toc136404168)

[图 7 BTB分支历史缓冲器工作原理图 16](#_Toc136404169)

[图 8 call指令与return指令运行示意图 16](#_Toc136404170)

[图 9 RISC-V手册中的call与return指令的判断标准 17](#_Toc136404171)

[图 10 BTB与RAS协同工作示意图 17](#_Toc136404172)

[图 11 基于Radix-4 Booth原理的A\*B表达式 18](#_Toc136404173)

[图 12 Radix-4 Booth部分积表格 18](#_Toc136404174)

[图 13 部分积计算 20](#_Toc136404175)

[图 14 SM4的加解密（左）和密钥拓展（右）流程图 23](#_Toc136404176)

[图 15 S盒查找表 24](#_Toc136404177)

[图 16 IPU顶层系统框图 26](#_Toc136404178)

[图 17 SCCB 写传输协议 26](#_Toc136404179)

[图 18 OV5640行时序图 27](#_Toc136404180)

[图 19 OV5640 QSXGA帧时序 28](#_Toc136404181)

[图 20 MIG IP核结构框图 30](#_Toc136404182)

[图 21 DDR3控制器写命令时序 30](#_Toc136404183)

[图 22 背对背写时序 31](#_Toc136404184)

[图 23 背对背读时序 31](#_Toc136404185)

[图 24 DDR控制模块顶层框图 32](#_Toc136404186)

[图 25 DDR3读写状态机 32](#_Toc136404187)

[图 26 腐蚀操作示意图 34](#_Toc136404188)

[图 27 膨胀操作示意图 34](#_Toc136404189)

[图 28 VIP模块的工作流程图 35](#_Toc136404190)

[图 29 动态分支预测器 36](#_Toc136404191)

[图 30 分支预测器命中率测试代码 39](#_Toc136404192)

[图 31 分支预测器的测试命中率 39](#_Toc136404193)

[图 32 Coremark末尾处的特殊指令 40](#_Toc136404194)

[图 33 乘法器行为级仿真结果 41](#_Toc136404195)

[图 34 除法器行为级仿真结果 42](#_Toc136404196)

[图 35 SM4硬件加速器的工作流程图 45](#_Toc136404197)

[图 36 SM4模块的仿真波形 45](#_Toc136404198)

[图 37 SM4加密测试程序 46](#_Toc136404199)

[图 38 SM4在线加密网站的结果 47](#_Toc136404200)

[图 39 C程序中对IPU外设的寄存器配置 49](#_Toc136404201)

[图 40 运动目标画框标记 50](#_Toc136404202)

[图 41 E203读到的坐标数据 50](#_Toc136404203)

[图 42 VCS上的Coremark测试 51](#_Toc136404204)

[图 43 Coremark上板测试跑分 53](#_Toc136404205)

[图 44 dhrystone上板测试跑分 53](#_Toc136404206)

[图 45 whetstone上板测试跑分 54](#_Toc136404207)

[图 46 SM4扩展算子软件驱动 55](#_Toc136404208)

[图 47 程序中调用SM4算子 57](#_Toc136404209)

[图 48 行为级仿真SM4算子的测试结果 58](#_Toc136404210)

[图 49 SM4算子的上板测试结果 58](#_Toc136404211)

[图 50 应用的上板测试结果 59](#_Toc136404212)

# 2 需求分析

## 2.1 项目需求

经过总结，该作品的设计需求为以下8点：

1. 获取E203开源IP以及配套的SoC系统；
2. 在硬件平台上能正确编译，下载软件，能够读写外设寄存器；
3. 设计动态分支预测器的微架构，并能兼容E203的微架构；
4. 用高性能乘除法器替换掉原本的长周期乘除法器；
5. 设计SM4硬件加速电路，能正确进行文本的加解密；
6. 设计IPU，能将摄像头图像显示在HDMI显示器上，并对运动物体标记；
7. 在SoC中为IPU分配地址空间，并设计相应的IPU外设寄存器；
8. 设计与SoC通信的上位机，上位机能对加密数据进行SM4解密，并能在上位机的坐标区域上动态显示运动物体的运动轨迹。

通过对设计需求进行总结，我们从硬件和软件两个方面进行需求分析：

## 2.2 硬件需求

1. 含有E203内核的SoC系统：其中SoC包括E203处理器内核，系统存储总线，私有设备总线，私有外设接口IP，常开域IP（IP复用于SIFive 公司开源的Freedom 310 ）以及系统内的存储器。
2. 动态分支预测器：动态分支预测器的微架构采用gshare，BTB以及RAS分支预测算法。相较于E203原本的静态分支预测，动态分支预测器需要系统反馈给它每次预测的结果是否正确，所以需要在E203的微架构中兼容这一点。
3. 高性能乘除法器：由于E203原本的乘除法器计算一次乘法需要17个周期，计算一次除法需要36个周期。这显然会成为含有大量乘法操作的Coremark基准测试程序以及大量乘除法操作的Dhrystone，Whetstone基准测试程序性能跑分的一大短板。所以为了提升E203的性能我们将原本的长周期乘除法器改成了高性能的乘除法器。
4. SM4硬件加速器：我们需要根据国家密码管理局发布的GM/T 0002-2012《SM4分组密码算法》标准来设计SM4硬件加速逻辑电路。
5. 摄像头驱动电路：由于IPU需要处理摄像头采集到的图像数据，所以需要一个能够驱动摄像头正常工作硬件电路，并集成到SoC中。
6. TMDS视频传输驱动电路：由于需要将摄像头采集到的数据显示在HDMI显示器上，而HDMI接口采用的是TMDS视频传输协议，因此需要一个TMDS视频传输驱动电路来将图像信息正确的发送给HDMI显示器，并将该驱动电路集成到SoC中。
7. DDR3控制器：IPU对运动目标检测的核心思想是利用帧差法对相邻两帧数据做差，所以需要先存储两帧的图像数据，而一帧1024\*768的灰度图片（每个像素占8bit内存）需要约0.78MB的内存空间，所以至少需要大约1.57MB的内存去存储图像数据。而本系统采用的FPGA芯片XC7A100T只有0.6MB的BRAM，无法满足我们需要的存储内存大小，所以需要利用片外的DDR3内存，并增加DDR3控制器。

## 2.3 软件需求

1. 以SDK为基础，建立基础工程。
2. 为IPU分配内存地址，可以通过运行程序访问IPU的外设寄存器。
3. 在蜂鸟软件开发平台HBird SDK中进行SM4加解密相关软件驱动的添加。
4. 上位机实现对串口数据的接收，并对接收到的加密数据进行解密操作，并将解密后的坐标数据动态地显示在上位机的坐标区域中。

# 3 系统方案设计

针对上面的需求分析，我们最终系统的设计架构如图1所示：

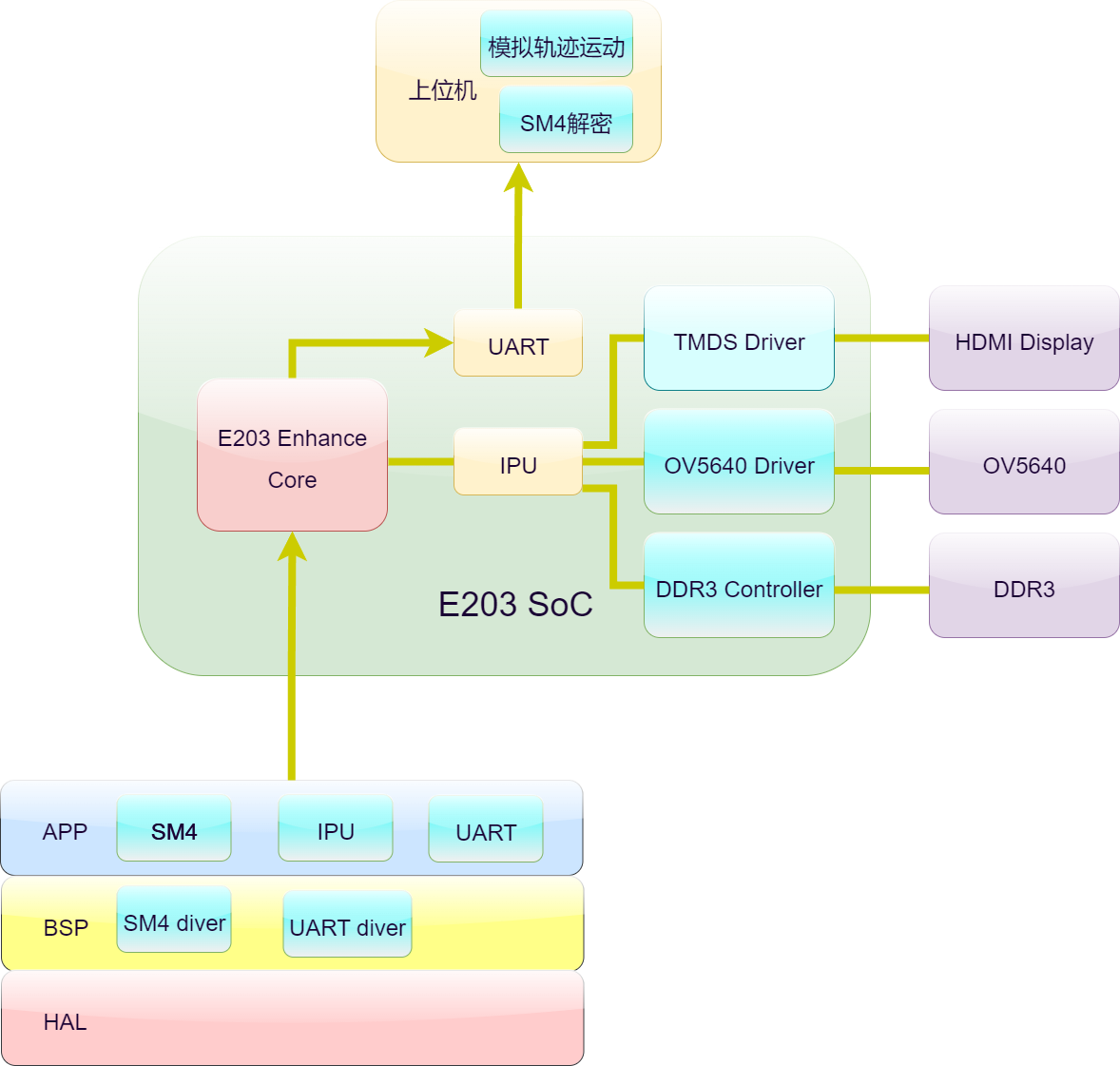


图 1系统整体框图

## 3.1 硬件平台

硬件平台采用正点原子的达芬奇PRO开发板，该开发板搭载了XC7A100T型号的FPGA芯片，含有63400个LUT，4.8Mbit的BlockRam，101K个逻辑单元，6个PLL。

开发板集成了我们系统所需的UART接口，OV5640接口以及HDMI接口。同时由于通常软件调试的时候采用的是10针的JTAG接口，所以我们用了板子上10个引出的FPGA引脚作为JTAG接口。硬件平台如图2所示：

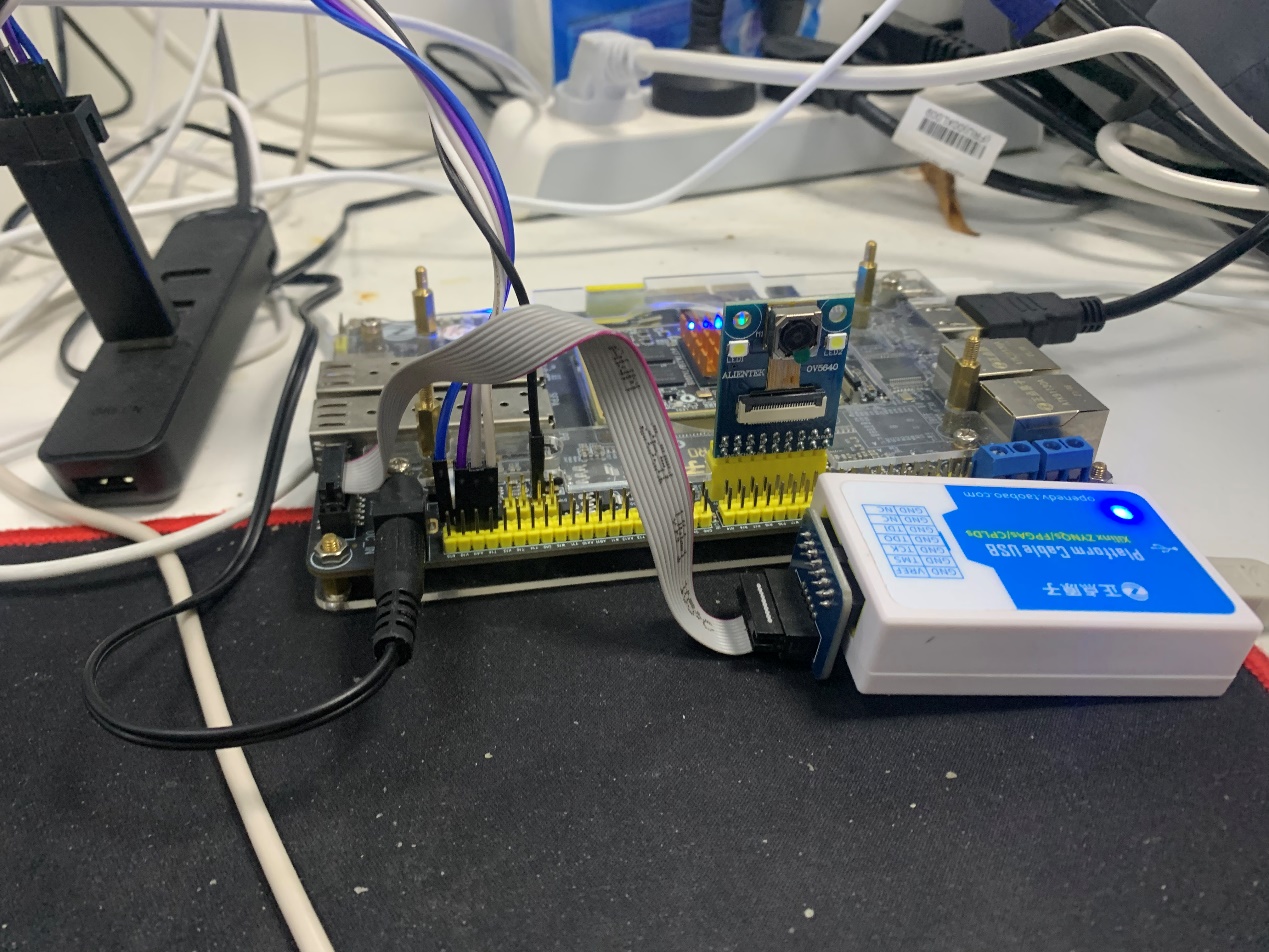


图 2系统硬件平台

整体的硬件连接是，JTAG用于下载比特流文件，UART用于与上位机进行通信，OV5640接口用于实现板子与摄像头的SCCB通信，HDMI接口用于实现板子与HDMI显示器的TMDS视频传输协议。

## 3.2 E203微架构改进

蜂鸟e203是芯来科技开源的一款2级流水线RISC-V CPU，支持RV32IMAC指令集以及带扩展的协处理器接口。它的主频为16MHz,因此主要面向超低功耗场景，提供标准的JTAG调试接口；私有的ITCM和DTCM，实现指令与数据的分离存储[1]。由于本次赛题要求对蜂鸟E203 RISC-V内核的微架构实现进行一定优化，并通过Benchmark基准测试（Dhrystone、CoreMark、Whetstone）来进行系统性能测试，要求在同等软件环境下与原本蜂鸟SoC测试分数相比有一定提升。常见的优化内核性能的方法有：减小执行单元的运算周期、多发射流水线、乱序流水、加深流水、提高分支预测的成功率、加cache来减少内核对存储器的访问时间等。

**方案论证：**

1、由于赛题的测试标准是主频在每1MHz下的跑分，所以加深流水级的方法并不适用。

2、因为E203内核内部紧耦合了ITCM和DTCM，其存储体是SRAM，所以可以做到单周期对存储器的访问，因此加cache的方式也是不适用的。

3、多发射流水线理论上可以成倍的提升处理器性能，但实际在做多发射流水线架构的时候，需要处理好指令间相关性，以及增加部件端口数，而顺序流水的多发射在发生指令相关的时候必然会引入很多流水空泡，所以要想很好地发挥出多发射流水的性能，一个比较好的解决办法是将指令乱序执行，使得可以在发生相关或者端口冲突造成的空泡中填入不相干的指令。但这种改进方法堆E203的内核改动极大，出于求稳的设计态度我们暂时放弃这种方案。

4、E203原本的分支预测器采用的是BTFN (back taken，forward not taken)预测方法，即对于向后的指令跳转预测为跳，向前的跳转则预测为不跳。向后跳转指跳转的目标地址（PC值）比当前分支指令的PC值要小。为了提高分支预测的预测成功率，可以采用一些在早期高性能CPU里用到的动态分支预测技术。在Scott McFarling的*Combining Branch Predictors*[2]一文中测试在SPEC89的负载下，256条目的gshare分支预测成功率可以达到91.5%左右。而且gshare分支预测器的架构简单，比较适合用在E203内核上。

5、由于E203内核原本的乘法器是计算一次乘法需要17个周期，而目前单周期乘法器的设计已经很成熟，并且Coremark测试基准程序中包含了大量的乘法计算，为了提高内核性能，换用单周期乘法器是一个很好的选择。E203内核原本的除法器计算一次除法或者求余需要36个周期，并且Dhrystone和Whetstone测试基准程序中包含了大量的乘除法操作，故同样可以采用高性能除法器对其进行替换优化。

综上分析，我们组最终选择的E203内核微架构的优化方案是：

1、采用gshare+BTB+RAS动态分支预测算法来对E203分支预测器进行优化设计。

2、采用单周期乘法器作为E203的乘法计算单元，采用高性能Radix2除法器作为E203的除法单元。

3、为了加速对SM4国密算法计算速度，我们利用E203的NICE协处理接口添加了SM4的加解密加速电路。

最终改进后的内核微架构如图3所示（蓝色部分是修改的模块）：

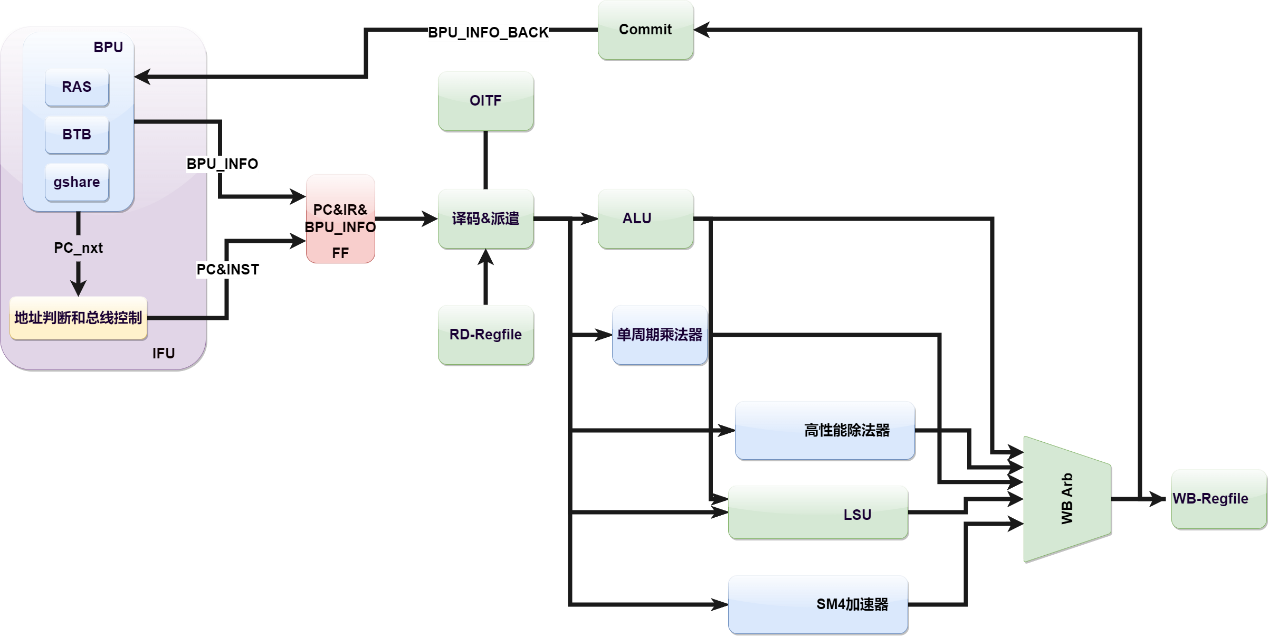


图 3 E203内核改进后的微架构

### 3.2.1 动态分支预测器设计

**分支预测概述：**

分支指令的影响是开发指令级并行性的重要障碍，一条指令流中，平均每5-7条指令中就有一条是分支指令，在完成一条分支指令之后，系统有可能会出现跳转也就是产生控制相关，进而使得处理器流水线出现空泡，如图4所示，直至分支路径的结束。如果流程越长，则处理器等待的时间也就越长，极大的降低处理器的性能。

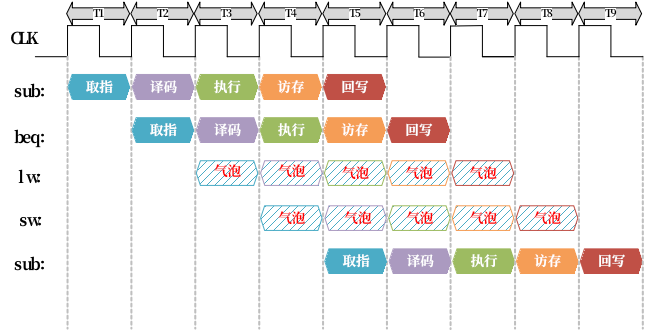


图 4分支指令对流水线影响

分支预测器正是为克服这一情况而产生的。分支预测技术是指处理器在程序分支路径执行时会预测其跳转结果的一种机制。在本设计中，为了提高蜂鸟E203的分支预测准确率，我团队将蜂鸟E203的BTFN(Back True Forward Not)的静态分支预测器修改为基于全局历史的动态分支预测器，该预测器将由转移目标缓冲(Branch Target Buffer，BTB)、全局历史寄存器(Global History Register，GHR)、返回地址栈共同构成（Return Address Stack，RAS）。[3]最终的分支预测器的总体架构如图5所示：

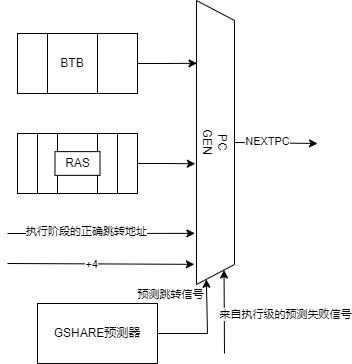


图 5 BPU整体架构

**Gshare预测器设计：**

在使用传统全局历史分支预测方法时，为了做出更好的预测，通常是通过扩展历史记录表的长度来实现。这样做会获得更高的指令地址分辨率以即更多的PHT寻址地址位数。但是PHT的寻址位数每增加1位，都会导致其大小增加一倍，因此就需要非常大的表来存储分支预测信息，这便是传统全局历史分支预测方法所存在的问题。

为了解决这个问题，可以将分支历史与条件分支命令的地址中部取哈希散列，工作方式如下图所示，在每个时钟周期程序计数器PC将与一个记录了此条指令前所有动态指令的跳转情况（这里称全局历史寄存器Gshare）做哈希散列后，得到的地址去寻址（PHT）。通过判断PHT读出的此条指令的历史跳转情况也就是PHT的值来决定是否在取指前端预测为跳转。并依据PHT的结果将GHR更新。同时并将此条分支的GHR和对应PHT值保存起来作为快照，以便在分支误预测时恢复现场。

对于某个条件分支指令而言，其全局历史是相对恒定的，让PHT同时保存全局历史的所有组合过于浪费，因此将指令地址的中部分与全局历史记录进行异或运算，而当指令地址的中部相同时，指令地址下部对应的指令共用几个PHT项，这样兼顾了全局历史表与PHT大小，改良了传统分支历史分支预测方法中PHT利用效率太差的问题。图6中10bit的全局历史记录与10bit的指令地址做哈希运算作为PHT的索引使用。

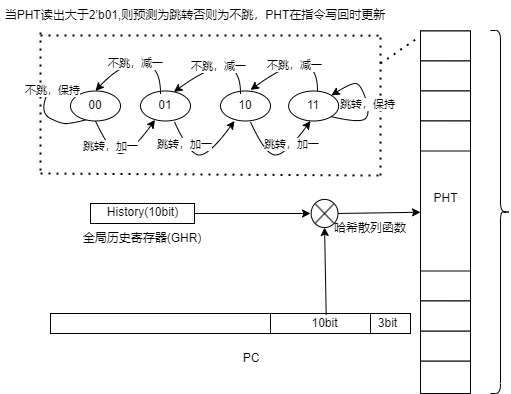


图 6 Gshare算法原理图

**BTB分支历史缓冲器设计：**

分支预测器除了需要对分支指令的方向进行预测之外，还需要对目标地址也进行预测，由于容量限制不可能对任意的PC均分配一项分支目标缓存，故BTB也采用了Cache的设计思路，利用PC的低位作为Index其PC作为Tag（EntryPC），在本设计中BTB的组织分Valid+EntryPC+TargetAdress+BrType这四部分，在每个周期PC将通过其低位Index位去选择对应的EntryPC和TargetAdress并判断其是否与有效和Match，若匹配后则TaegetPC将作为nextPC。

在指令写回阶段对于未正确预测的跳转指令会在指令更新阶段将其跳转地址和指令类型更新至BTB中，完成BTB的更新与替换。在本设计中由于增加了BrType位边可在前端直接通过指令类型判断是否跳转，从而降低了分支预测器预测错误对指令流错误取指的影响。其工作原理图如图7所示：

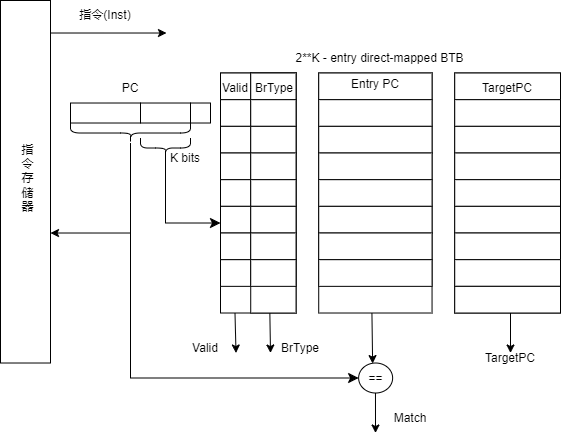


图 7 BTB分支历史缓冲器工作原理图

**RAS栈返回地址缓冲器设计：**

call指令每次调用子程序（如某个函数）是固定的，因此对应目标pc固定可用BTB进行预测。但对于子程序可能很多地方都需要调用它，故返回的地方会发生变化，但是return指令的目标地址总是最近一次执行call指令的下一条指令，如图8所示。在E203中并没有因此可设计一个后进先出存储器（LIFO）保存最近执行call指令的下一条指令地址，该存储器和堆栈的工作原理一样，称为返回地址堆栈（RAS）。现代处理器中BTB和RAS几乎是必须要使用的。在本设计中RAS的大小为16项，可以支持16层函数调用，足以支持大多数嵌入式领域的应用场景。

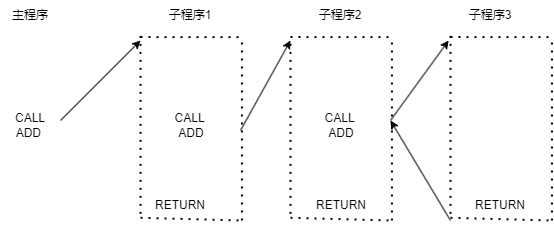


图 8 call指令与return指令运行示意图

在遇到call指令时需要能够将call指令的下一条地址放入RAS中，在RISC-V手册中定义了如下规则识别call和return指令。其判断标准手册[4]如图9所示：

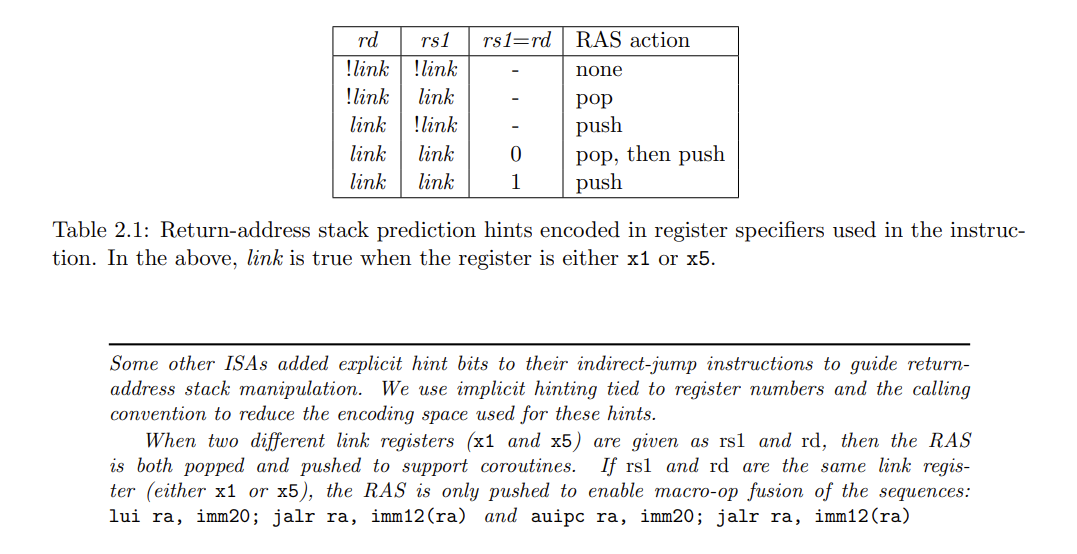


图 9 RISC-V手册中的call与return指令的判断标准

但由于分支预测器工作于流水线的前端所以并不能够在预测时得到其源操作数寄存器，所以在实际工作时要利用BTB来辅助RAS工作，利用之前提到的在BTB设计中的BrType位可以用起作为标志位将地址压入栈中，并通过BrType的Return标记位将其从RAS中读出。如图10所示：

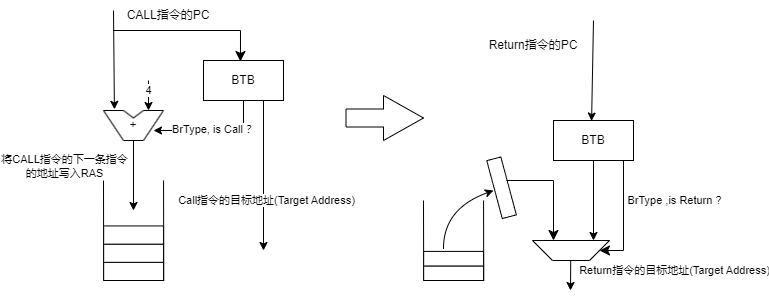


图 10 BTB与RAS协同工作示意图

最终设计的基于Gshare+BTB+RAS的分支预测器能够利用Gshare来对分支指令是否跳转做出预测，如果Gshare判定当前指令跳转则BTB与RAS根据当前PC是否有在自己的查找表中命中来确定指令跳转的目标地址，如上图5所示。

当该分支预测器在内核中工作时，会不断根据分支指令的提交结果来判断Gshare是否预测正确，以及BTB与RAS预测的跳转目标地址是否正确。如果预测错误在内核刷新流水线的同时也会将分支预测器回滚到未预测错的状态，并且根据当前的指令提交结果修正Gshare中的ghr参数以及更新BTB与RAS查找表中的信息。这使得该动态分支预测器具有学习的能力，利用历史数据为未来做预测。

### 3.2.2 单周期高性能乘法器设计

该乘法器是基于radix-4 booth编码华莱士树结构的乘法器，乘法器在一拍内完成Booth编码和华莱士树生成以及最后一级加法器加法。同时在设计该乘法器的时候，我们探索性的采用SpinalHDL语言来做乘法器设计。

**Booth编码电路：**

基于Radix-4 Booth原理A\*B可以如下图11所示：



图 11 基于Radix-4 Booth原理的A\*B表达式

因此用SpinalHDL语言实现Radix-4 Booth编码操作为：

**SpinalHDL代码：**

|  |
| --- |
| class BoothEncode(val a:SInt,val b:SInt,val cut:Int=1) extends Area{  val v:SInt = if(a.getWidth > b.getWidth) a else b  val f:SInt =(if(a.getWidth > b.getWidth) b else a)  val w:Int = v.getWidth  val n:Int = f.getWidth  val odd :Int = w & 1  val even:Int = 1 - odd  val cnt :Int = (n>>1)+odd  val pp = Vec(UInt(w+2 bits),cnt)  val pz = UInt((w + n - cut - 1) bits) |

编码电路首先定义了一些基本量，其中v代表被乘数也就是原理中表示的A，f代表其booth编码也就是B。并根据位宽选择生成部分积数量较少的哪一项作为booth编码项。同时并根据定义两个操作数的位宽以确定生成的部分积的数量，其中odd代表是否是奇数位乘法器若是则odd为1在后续多补一项部分积的暂存项。其部分积生成操作如图12所示：

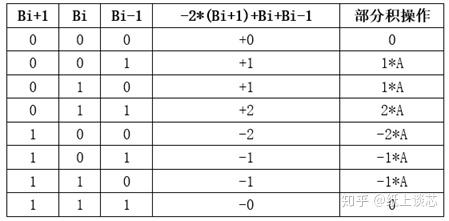


图 12 Radix-4 Booth部分积表格

根据上表对于位运算当Bi+1=0,Bi,Bi-1不同时部分积为1\*A操作，推广至整个f则1\*A的部分积定义位为pos1代码所示，同理可得2\*A如pos2所示，-1\*A如neg1所示，-2\*A如neg2所示：

**SpinalHDL代码：**

|  |
| --- |
| val pos1:Bits = (~(f |>> 1) &( f ^ (f |<< 1))).asBits  val pos2:Bits = (~(f |>> 1) & f & (f |<< 1) ).asBits  val neg1:Bits = ( (f |>> 1) &( f ^ (f |<< 1))).asBits  val neg2:Bits = ( (f |>> 1) & ~f & ~(f |<< 1) ).asBits |

但由于Radix-4是每次求部分积看隔着的两位如下图13所示，故其pos1，pos2，neg1，neg2只有偶数位有效。

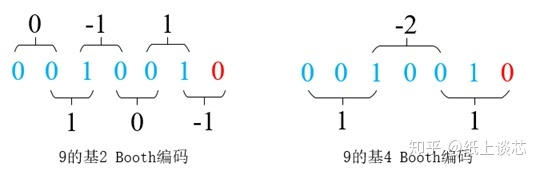


图13 Radix-4 Booth编码示例

故在循环中以2为步长只遍历部分积标志的偶数位，其中vs中存了4种部分积操作分别是1\*A，2\*A,-2\*A,-A。并通过由部分积标志位作为onehot的选择器选择对应的部分积操作作为华莱士树的输入。

**SpinalHDL代码：**

|  |
| --- |
| for(i <- Range(0,n,2)){  val oh:Array[Bool] = Array(pos1(i),pos2(i),neg2(i),neg1(i))  val vs:Array[SInt] = Array(v,v|<<1,~(v|<<1),~v)  pp(i>>1) := U(if(i==0) "11" else "10")@@ MuxOH.or(oh,vs).asUInt} |

值得注意的是有两点，其一是对于负数操作在生成部分积时并未取反后加一只是单纯取反。后续加1操作在pz的低位进行。这样的好处在于可以节省掉取反加1的所需要的加法器。由于只有偶数位低位有效故生成一组1010101…序列通过neg操作选择屏蔽所有不为负操作部分积的位数操作。

**SpinalHDL代码：**

|  |
| --- |
| pz(0, w\_lo bits) := (U("1" + "01" \* (cnt-1)) & neg.asUInt).resized |

另一个是关于符号位扩展：由于部分积的组成（其实二进制数均如此）由符号位及其扩展和其数据位组成，以4bit有符号乘法运算为例，（对应的部分积经符号位扩展后一般情况为8位），如-8在二进制表示为(1111\_1000)b其中1111\_为符号位扩展位，(\_1000)为数据位。针对其特征我们可以对其进行抽取相同项达到化简目的。

首先对于任意一个带有符号位扩展的数如+4=(0000\_0100), -4=(1111\_1100)我们可以看到其符号位及其扩展要么是0000全零或者1111全一，而其均可以写成1111+!msb的形式，例如msb为0为正数则符号位为1111+!0=0。则我们可以把部分积的符号位全部写死为全1，并将其部分积对应的msb的非放入对应的bit位，使得1111+!msb的加法运算可以放在华莱士树中去运算。这个行为定义为pz的高位。

由于只有偶数位有效故存在一个1010101…的序列其中偶数位对应着每个部分积的msb的非，当A为正数时所有的-A，-2\*A的部分积均为正数也就是!msb等于1故通过~neg来屏蔽掉1010101…序列中其偶数位对应的非neg操作的位。

**SpinalHDL代码：**

|  |
| --- |
| val ext :Bits = Mux(v.msb,pos,neg)  pz(pz.high downto w) := (U("1" + "01" \* (cnt-1)) & ~ext.asUInt).resized |

但这样做的缺点在于在部分和计算中具有大量的符号位扩展，且在求和过程中翻转频繁，故而极大耗费面积和功耗。故在这里对Booth编码做了进一步的优化。其中对于符号位来说因为无论正负其都能提取全1故可在生成部分积时就将符号位运算完毕以达到节省符号位数量的问题。

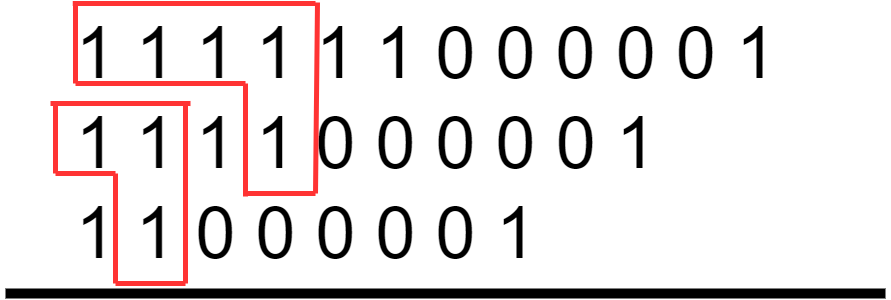


图 13 部分积计算

由上图13所示可将第二个部分积的符号位最低位与第一个部分积圈起来的部分先进行运算得0故第一个部分积的符号位只有两位11和其后面的数据位000001，同理用于第二项的部分积最低符号位已经计算故此位为零，且由第三部分积的最低符号位与第二部分积圈起来的部分做加法则第二个部分积为符号位10加其数据位000001，以此类推，发现只有第一个部分积为符号位为11其余均是10，故实际生成部分积的代码如下：

**SpinalHDL代码：**

|  |
| --- |
| pp(i>>1) := U(if(i==0) "11" else "10")@@ MuxOH.or(oh,vs).asUInt |

**华莱士树加法电路：**

将生成的部分积分别移位并与pz（高位为符号位扩展，低位为负数+1）相加得到华莱士树的和。在乘法器的设计中采用树形乘法器，可以减少关键路径和所需的加法器单元数目

**SpinalHDL代码：**

|  |
| --- |
| def getSumSim:Vec[UInt] = {  val res:Vec[UInt] = Vec(UInt(pz.getWidth bits), 2)  res(0) := (for(i <- 0 until pp.length) yield (pp(i)<<(2\*i)).resized).fold(pz)(\_+\_)  res(1) := U(0)  return res  } |

从数据最密集的地方开始，不断的反复使用全加器、半加器来覆盖“树”。全加器是一个3输入2输出的器件，因此全加器又称作3—2压缩器。通过全加器将树的深度不断缩减，最终缩减为一个深度为2的树。最后一级则采用简单的2输入加法器组成。对于综合优化的电路如下代码所示：

**SpinalHDL代码：**

|  |
| --- |
| def getSumSyn:Vec[UInt] = {  val w:Int = pz.getWidth  val n:Int = pp.length  val tmp:IndexedSeq[ArrayBuffer[Bool]] = for(i <- 0 until w) yield ArrayBuffer[Bool](pz(i))  for(i <- 0 until n;j <- 0 until pp(i).getWidth;if 2 \* i + j < w){  tmp(2\*i+j).append(pp(i)(j))  }  val buf:IndexedSeq[Bits] = for(i <- 0 until w) yield Bits(tmp(i).length bits)  for(i <- 0 until w;j <- 0 until tmp(i).length){  buf(i)(j) := tmp(i)(j)  }  val csa:IndexedSeq[Bits] = genCSA(buf)  val res:Vec[UInt] = Vec(UInt(pz.getWidth bits), 2)  for(i <- 0 until csa.length;j <- 0 to 1){  res(j)(i) := (if(j < csa(i).getWidth) csa(i)(j) else False)  }  return res  } |

用SpinalHDL完成对单周期乘法器的设计后，将其生成verilog代码，并接入E203的执行单元中。

### 3.2.3 高性能除法器设计[5]

二进制除法器的本质是多次减法，直到余数小于除数为止。对应的两个N bit二进制数的除法算法如下。

1、设置2N bit寄存器A的低N位存放被除数，设置2N bit寄存器B的高N位存放除数，设置N bit寄存器C存放商，设置计数器cnt。

2、将A左移一位，与B比较高N位大小，若A[2N-1 : N] B[2N-1:N]，则令A=A-B，令C左移一位，最低位为1；若A[2N-1:N] B[2N-1:N]，则令C左移一位，最低为0；

3、计数器cnt加1，若小于N则继续进行步骤2，否则结束计算。

4、A中剩余的高N位为余数，C中的为商。

这种计算除法的算法被称为恢复余数除法，由于恢复余数除法采用串行结构，计算速度慢，且需要的时钟周期数不确定，不利于进行控制设计。

E203自带的除法器采用基4的加减交替法计算。加减交替法是一种不恢复余数的除法。在恢复余数的除法中设第i次减法计算后的余数小于0，则恢复余数法的操作是，其中Y是除数，是余数。则第i+1次的余数如式（3-1）：

……….….（3-1）

因此不恢复余数除法采用的公式，其结果与恢复余数后左移一位再减Y是等效的。

原码交替加减法的通用公式为:

……………………………（3-2）

其中是第i次计算所得的商，则若部分余数为正，即，否则。

不恢复余数除法可以使用脉动阵列的形式，为此需引入可控加/减法单元（CAS）。

而如若要减小除法器的计算周期，可以通过并行处理的方式来提高除法器的计算性能，本设计为了尽可能减小除法器的计算Latency Cycle，故在初赛中我小组最终选择了Xilinx公司提供的除法器IP，将该IP接入到E203的执行单元中，完成内核中的除法以及求余运算。

### 3.2.4 SM4国密算法硬件加速电路设计

SM4是一种分组密码算法，由我国国家密码管理局在2012年发布，常用于无线互联网加密等领域。

SM4 算法主要包含异或、移位以及盒变换操作。它分为密钥拓展和加/解密两个模块，这两个模块的流程大同小异。其中，移位变换是指循环左移；盒变换是一个将8bit输入映射到8bit输出的变换，是一个固定的变换。

图14是 SM4 的加解密（左）和密钥拓展（右）的流程图：

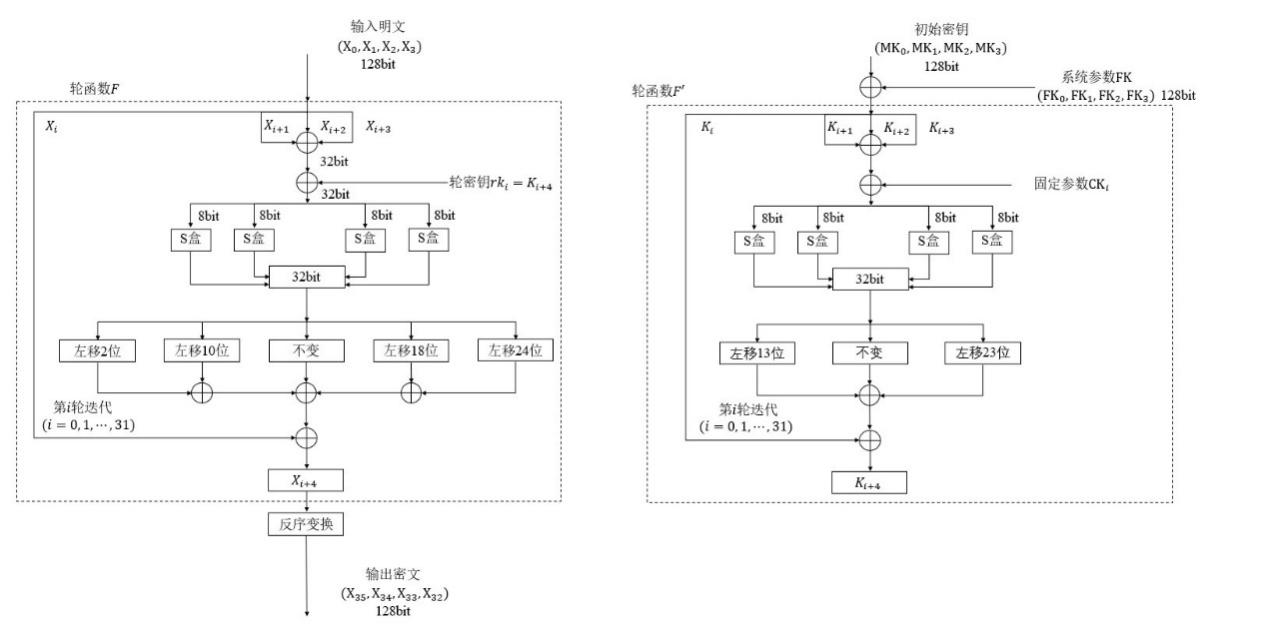


图 14 SM4的加解密（左）和密钥拓展（右）流程图

在SM4国密的GM/T 0002—2012标准中有以下规定[6]：

**轮函数F：**

本算法采用非线性迭代结构，以字为单位进行加密运算，称一次迭代运算为一轮变换。设输入为(*X* 0 , *X*1 , *X* 2 , *X* 3 ) ∈ ()4，轮密钥为rk∈，则轮函数F为：

*F* ( *X*0 , *X*1 , *X*2 , *X*3 , *rk* ) = *X*0 ⊕ *T* ( *X*1 ⊕ *X*2 ⊕ *X*3 ⊕ *rk* )……..（3-3）

**合成置换T：**

T：，是一个可逆变换，由非线性变换τ和线性变换L复合而成，即T(.)=L(τ(.))。

**非线性变换τ：**

τ 由 4 个并行的 S 盒构成。

设输入为*A* = (*a*0 , *a*1 , *a*2 , *a*3 ) ∈ ()4 ，输出为*B* = (*b*0 , *b*1 , *b*2 , *b*3 ) ∈()4 ，则(*b*0 , *b*1 , *b*2 , *b*3 ) = *τ*(*A*) = ( *Sbox* (*a*0 ), *Sbox* (*a*1 ), *Sbox* (*a*2 ), *Sbox* (*a*3 ) )

**线性变换L：**

非线性变换 τ 的输出是线性变换 L 的输入。设输入为B ∈，输出为 C ∈，则：

*C* = *L*(*B*) = *B* ⊕ (*B* <<< 2) ⊕ (*B* <<< 10) ⊕ (*B* <<< 18) ⊕ (*B* <<< 24)….（3-4）

S盒：

S盒中数据均采用16进制表示，如图15所示：

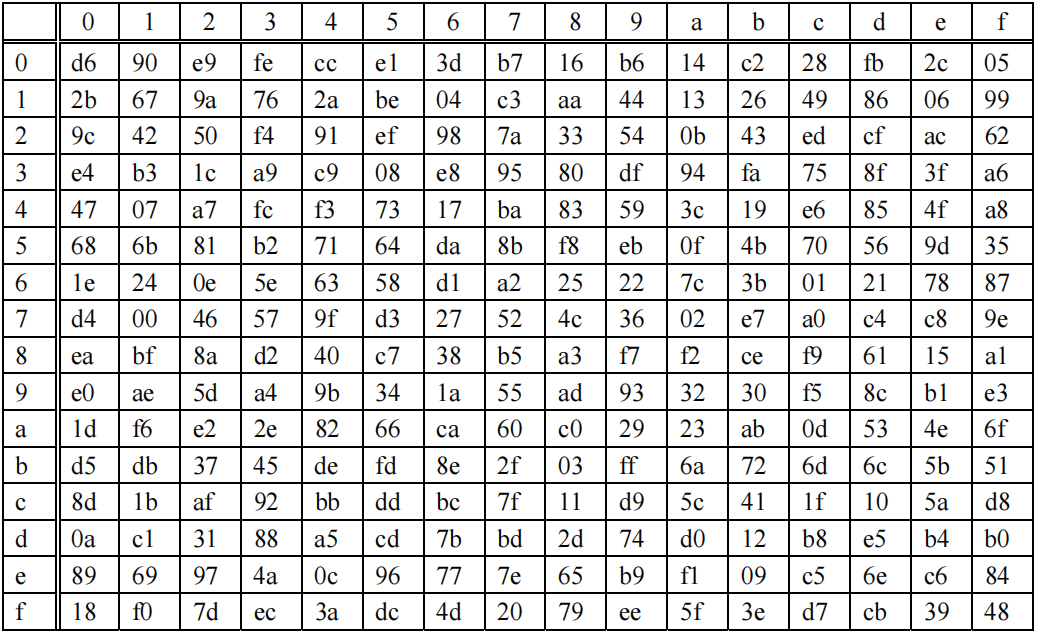


图 15 S盒查找表

**加解密算法：**

定义反序变换R为：

*R* ( *A*0 , *A*1 , *A*2 , *A*3 ) = ( *A*3 , *A*2 , *A*1 , *A*0 ) ，*Ai* ∈

设明文输入为(*X*0 , *X*1 , *X*2 , *X*3 ) ∈ ()4

密文输出为：(*Y*0 , *Y*1 , *Y*2 , *Y*3 ) ∈ ()4

轮密钥为：*rki* ∈, *i* = 0,1,2,...,31

则本算法的加密变换为：

*Xi*+4 = *F*(*Xi* , *Xi*+1 , *Xi*+2 , *Xi*+3 , *rki* ) = *Xi* ⊕ *T*(*Xi*+1 ⊕ *Xi*+2 ⊕ *Xi*+3 ⊕ *rki* ,) ……（3-5）

(*Y*0 , *Y*1 , *Y*2 , *Y*3 ) = *R*(*X*32 , *X*33 , *X*34 , *X*35 ) = (*X*35 , *X*34 , *X*33 , *X*32 )…………………（3-6）

本算法的解密变换与加密变换结构相同，不同的仅是轮密钥的使用顺序。

加密时轮密钥的使用顺序为：(*rk*0, *rk*1, …, *rk*31 )

解密时轮密钥的使用顺序为：(*rk*31, *rk*30, …, *rk*0 )

**密钥扩展算法：**

本算法中加密算法的轮密钥由加密密钥通过密钥扩展算法生成。

加密密钥MK=(MK0, MK1, MK2, MK3) ，MK*i* ∈

令*Ki* ∈ 轮密钥为*rk* *i* ∈

则轮密钥生成方法为：

*rki* = *Ki* +4 = *Ki* ⊕ *T*′(*Ki* +1 ⊕ *Ki* +2 ⊕ *Ki* +3 ⊕ *CK* *i* )……………（3-7）

在设计SM4硬件加速器的时候，我们参考了github上的一个SM4硬件RTL设计例子[7]，其设计思想是在用硬件实现SM4的加解密算法的时候，会先进行密钥扩展，得到32个子秘钥存储起来。随后当有效明文/密文数据输入的时候，每轮迭代运算开始的时候都会将对应轮数的子秘钥与当前的迭代数据一起通入到加解密模块中进行当前轮的加解密迭代操作。最终在32个迭代周期之后能够得到正确的加解密数据。

## 3.3 IPU外设设计

### 3.3.1 IPU外设架构

IPU的整体架构如图16所示：

IPU在工作的时候会首先将摄像头发送来的RGB565像素数据转为YCbCr888数据，这样做的原因是可以很容易得到图像的灰度图像，因为YCbCr888数据的Y分量就是图像的灰度值。随后将相邻两帧的灰度数据存入到DDR3中，随后将相邻两帧的灰度数据一起传输给VIP图像处理单元进行图像处理，得到图像中运动目标的大致位置；在将图像的灰度图发送给HDMI接口的同时，将代表运动目标位置的矩形框覆盖在原灰度图上，进而可以在HDMI显示器上观察到对图像中运动物体的跟踪标记。

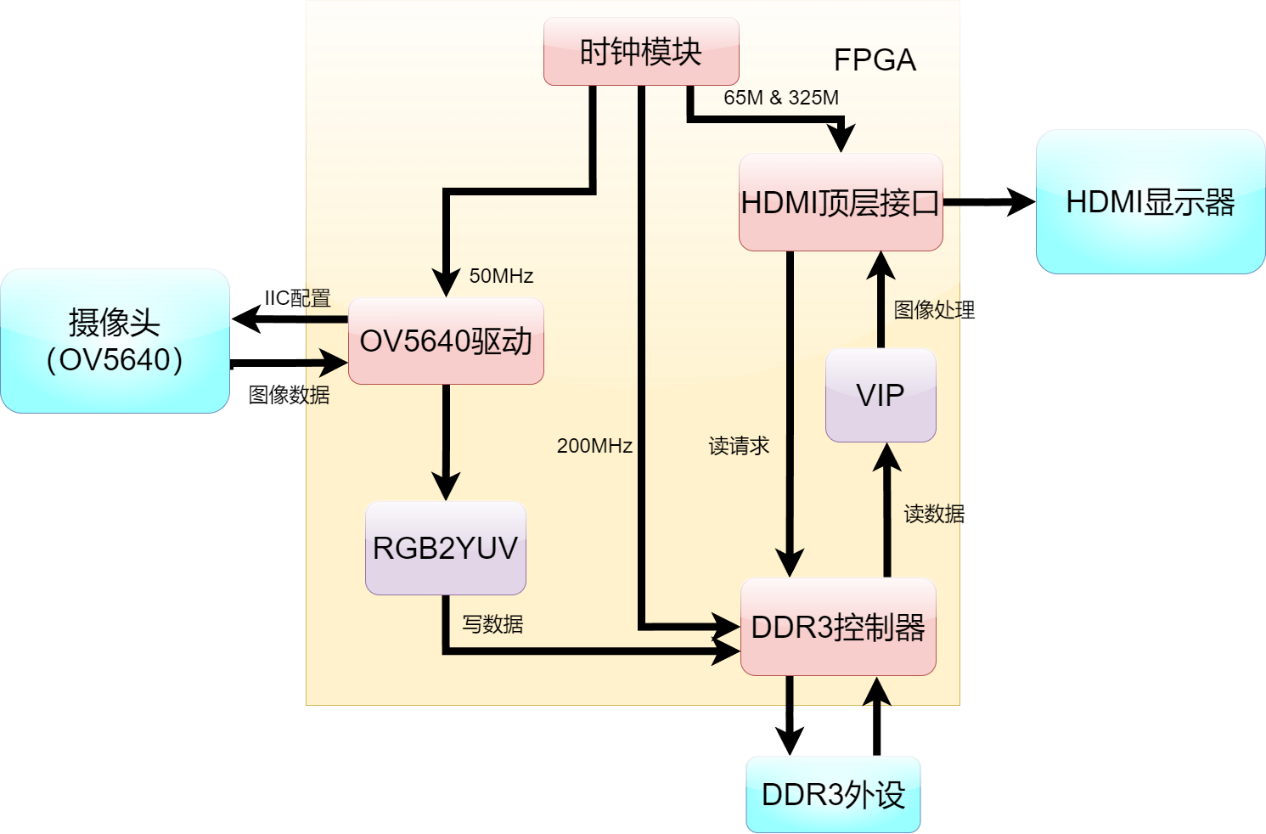


图 16 IPU顶层系统框图

### 3.3.2 OV5640驱动模块设计[8]

OV5640是一款1/4英寸单芯片图像传感器，使用的是两线式SCCB接口总线。

**OV5640的写时序：**

SCCB的写传输协议如下图17所示：

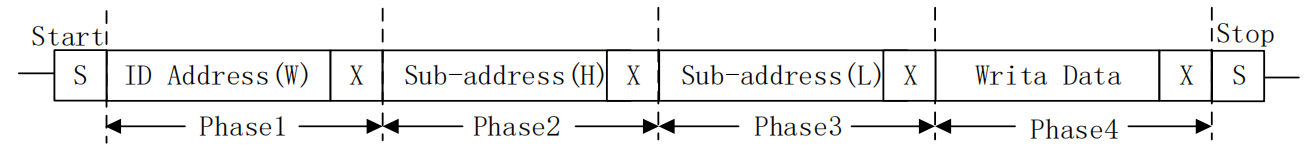


图 17 SCCB 写传输协议

上图中的ID ADDRESS是由7位器件地址和1位读写控制位构成（0：写，1：读），OV5640的器件地址为7’h3c，所以在写传输协议中，ID Address（W）= 8’h78（器件地址左移 1 位，低位补 0）；Sub-address(H)为高8位寄存器地址，Sub-address(L)为低8位寄存器地址，在OV5640众多寄存器中，有些寄存器是可改写的，有些是只读的，只有可改写的寄存器才能正确写入；Write Data为8位写数据，每一个寄存器地址对应8位的配置数据。

在OV5640正常工作之前，必须先对传感器进行初始化，即通过配置寄存器使其工作在预期的工作模式，以及得到较好画质的图像。因为 SCCB 的写传输协议和IIC几乎相同，因此我们直接使用IIC的驱动代码来配置摄像头。

**OV5640的图像输出时序：**

QSXGA，这里指：分辨率为2592\*1944的输出格式，类似的还有：QXGA(2048\*1536）、UXGA(1600\*1200）、SXGA(1280\*1024）、WXGA(1440\*900）、WXGA(1280\*800）、XGA(1024\*768）、SVGA(800\*600）、VGA(640\*480）、QVGA(320\*240）和 QQVGA(160\*120)等。

PCLK：像素时钟，一个 PCLK 时钟输出一个像素或者半个像素（像素数据的高8位或者低8位）。

VSYNC：帧同步信号。

HREF/HSYNC：行同步信号。

D[9:0]：像素数据，在RGB565格式中，只有高8位是有效的。

tPclk：一个时钟周期 。

tp：一个像素点的周期，在RGB565和YUV422输出格式下，tp=2\*tPclk；Raw 输出格式下，tp=tPclk。

下图为 OV5640 输出图像数据的行时序图。

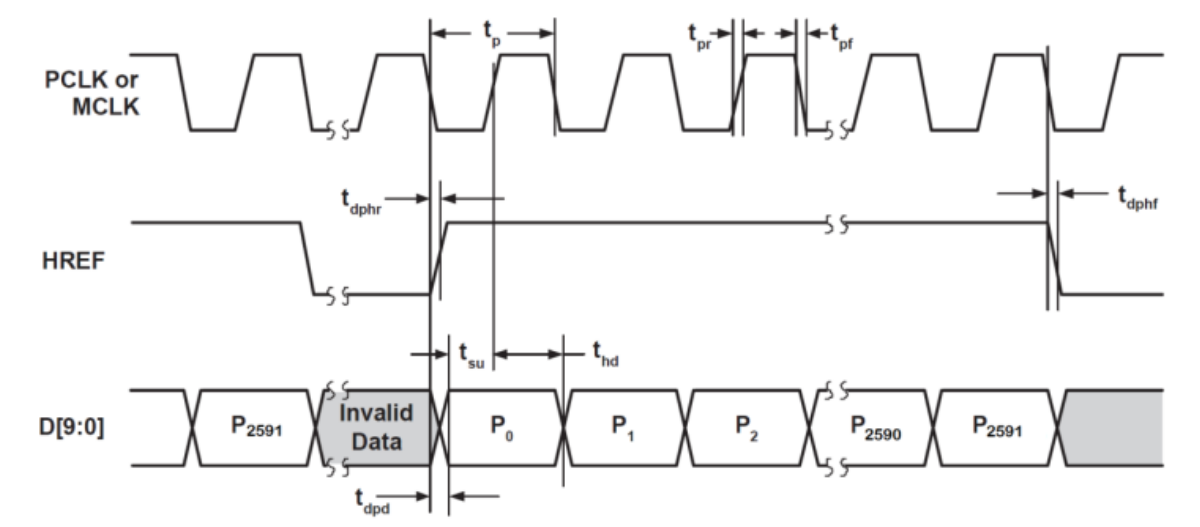


图 18 OV5640行时序图

从上图可以看出，传感器在HREF为高电平的时候输出图像数据，当HREF变高后，每一个PCLK时钟，输出一个8位或者10位像素数据。比如我们采用QSXGA时序，RGB565格式输出，tp=2\*tPclk，每2个字节组成一个像素的颜色，这样每行总共输出2592\*2个PCLK，也就是2592\*2个字节。再来看看帧时序（QSXGA模式，分辨率2592\*1944），如下图所示：

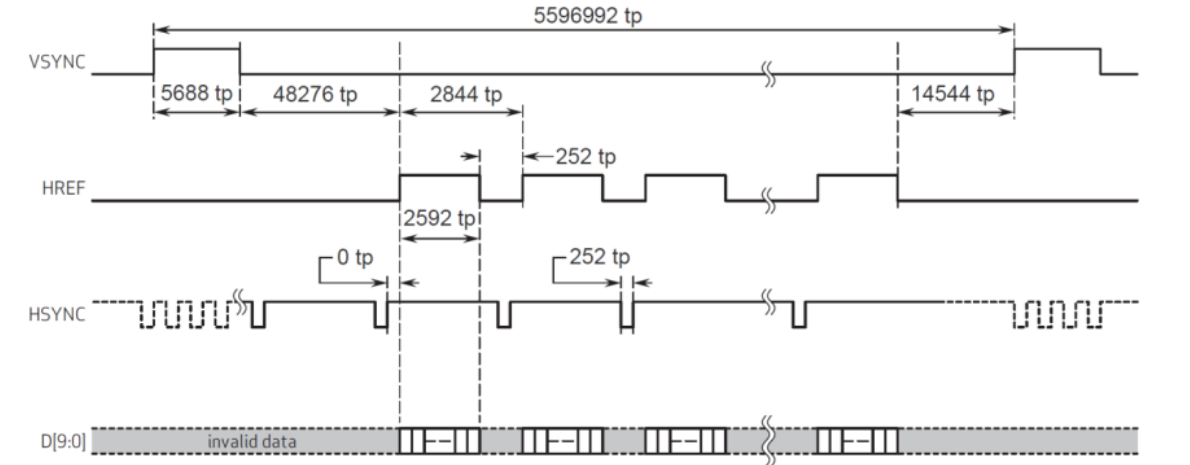


图 19 OV5640 QSXGA帧时序

由上图可知，VSYNC的上升沿作为一帧的开始，高电平同步脉冲的时间为 5688tp，紧接着等待48276tp时间后，HREF开始拉高，此时输出有效数据；HREF 由2592tp个高电平和252tp个低电平构成；最后一行图像数据输出完成之后等待14544tp时间，一帧数据传输结束。所以输出一帧图像的时间实际上是tFrame = 5596992tp。

### 3.3.3 RGB2YUV模块设计

摄像头传输的原始数据是彩色数据，彩色图像的每个像素点具有红绿蓝三种颜色分量。在图像的实际处理过程中，如果对RGB565格式的彩色像素数据直接进行处理，一方面，会直接增加整个处理器的运算量，将直接影响到系统的读写带宽，最终导致系统难以达到对图像实时、快速的处理要求。另一方面，影响图像运动目标检测的关键因素在于摄像头采集得到的图像结构而非图像颜色。因此，在整个手写数字识别系统中图像数据主要采用灰度化以及二值化的像素数据，而非原始的RGB565格式的彩色图像数据。

图像灰度化处理主要是将RGB空间的像素数据映射到YUV空间的像素数据，整个过程实际上是完成了红绿蓝颜色分量到亮度色度分量的转换。图像灰度化主要有最大值法、平均值法、加权均值法等多种方法，其中加权均值法在图像灰度化过程中获得了广泛的应用。加权均值法在图像的灰度化过程中充分考虑了人眼对不同颜色分量的敏感程度，其主要反映在转换公式中不同颜色分量的权重值，公式（3-8）是本系统采用的加权均值法灰度转换公式，公式中的R表示像素数据的8位红色分量，G表示像素数据的8位绿色分量，B表示像素数据的8位蓝色分量。

Y = 0.299R +0.587G + 0.114B ………………………………..（3-8）

对RGB565格式的像素数据进行灰度化的过程中，需要考虑两方面的内容：一是在实际的硬件电路中是无法进行小数运算，因此需要用定点化运算和位移运算来实现小数运算过程；二是公式（3-8）中所要求的所要求的红绿蓝分量的数据表示格式为8位二进制，因此需要对RGB565格式的像素数据的红绿蓝颜色分量进行位拓展，将其转换为8位二进制格式的颜色分量。

故最终采用公式（3-9）对色彩图像进行灰度化：

Y = (77R + 150G + 29B)>>8 ……………………………….（3-9）

### 3.3.4 DDR控制模块设计

DDR3读写控制器主要用于生成片外存储器DDR3 SDRAM进行读写操作所需要的时序，继而实现对片外存储器的读写访问。由摄像头采集得到的图像数据通常数据量较大，使用片内存储资源难以实现大量图像数据的缓存，因此借助开发板上存在的片外存储器DDR3 SDRAM对图像数据进行缓存。

DDR3 SDRAM(Double-Date-Rate Three Synchronous Dynamic Random Access Memory)是DDR SDRAM的第三代产品，相较于DDR2，DDR3具有更高的运行性能与更低的电压。DDR SDRAM是在SDRAM技术的基础上发展改进而来的，同SDRAM相比，DDR SDRAM的最大特点是双边沿触发，即在时钟的上升沿和下降沿都能进行数据采集的发送，同样的工作时钟，DDR SDRAM的读写速度可以比传统的SDRAM快一倍。在我们系统中使用的DDR3芯片信号是MT41K64M16TW，它的bank地址位宽为3位，行地址位宽为13位，列地址位宽为3位，所以该芯片的地址大小等于2^3\*2^13\*2^10(即2^26=64M)，数据位宽为16bit，所以芯片的容量为64M\*16bit，也就是128Mbyte。

IPU中的DDR3读写控制器可以在Xilinx公司MIG(Memory Interface Generators)IP核的基础上设计得到的。MIG IP核是Xilinx公司针对DDR存储器开发的IP，里面集成存储器控制模块，实现DDR读写操作的控制流程。图20所示是7系列的MIG IP核结构图。MIG IP核对外分出了两组接口，左侧是用户接口，右侧是DDR物理芯片接口，负责产生具体的操作时序，并直接操作芯片管脚。

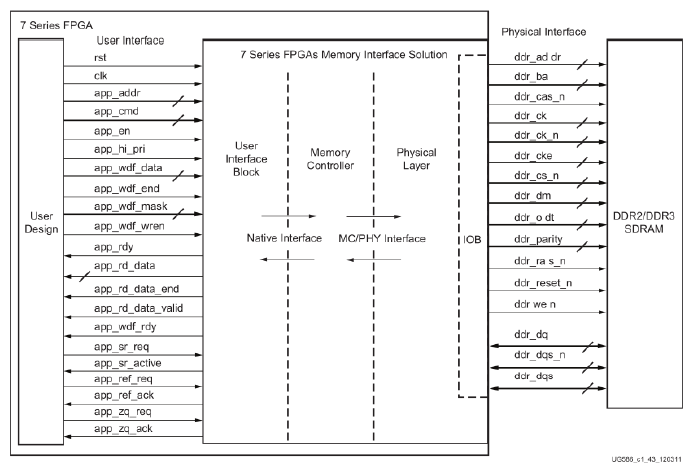


图 20 MIG IP核结构框图

DDR3的读写都包含写命令操作，其中写命令(app\_cmd)的值等于0，读操作app\_cmd的值等于1。如图21所示写命令时序，在写命令的过程中，需要首先检查app\_rdy信号的状态，当app\_rdy为高电平时则表示此时的IP核命令接收处于准备好状态，可以接收用户命令，在当前时钟拉高app\_en，同时发送命令(app\_cmd)和地址(app\_addr)，此时命令和地址被写入。

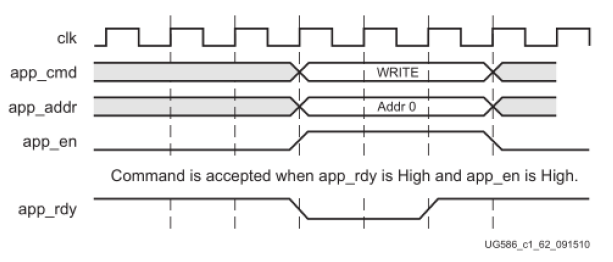


图 21 DDR3控制器写命令时序

在DDR3的写数据过程中，在完成写命令之后需要进行写数据操作，如图22所示是背靠背模式下的写时序示意图，在写数据的过程中，首先需要检测app\_wdf\_rdy信号，该信号为高表明此时IP核数据接收处于准备完成状态，可以接收用户发送的数据，在当前始终拉高写使能(app\_wdf\_wren)，给出写数据(app\_wdf\_data)。这样加上发起的写命令操作就可以成功向IP核写入数据。

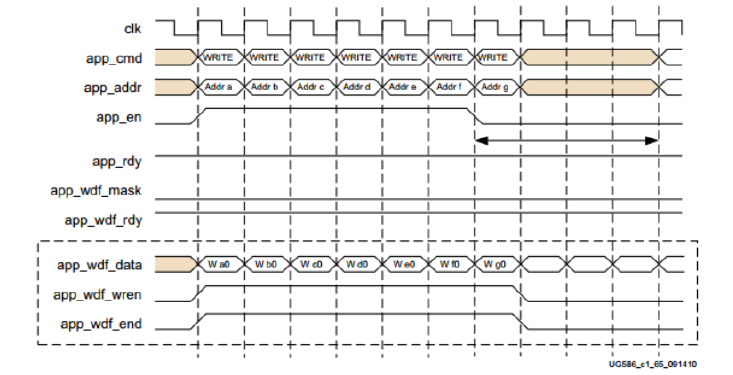


图 22 背对背写时序

如图23所示是背对背模式的读时序图，读时序比较简单，发出读命令后，用户只需要对数据有效信号(app\_rd\_data\_valid)的电平状态进行判断，当为高电平时，表明此时数据总线上的数据是有效的返回数据。需要注意的是，在发出读命令后，有效读数据会晚若干周期才能出现在数据总线上。

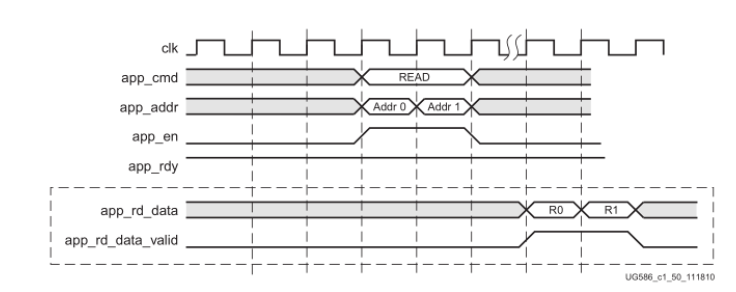


图 23 背对背读时序

图24是IPU模块中DDR控制模块的框图，DDR控制模块由DDR读写模块、MIG模块和FIFO调度模块3个子模块组成。DDR读写模块负责与MIG模块的命令和地址的交互，根据FIFO调度模块中FIFO的剩余数据量来切换DDR3的读写命令和地址。MIG模块负责连接外设和FPGA。FIFO调度模块负责对输入和输出的数据进行时钟域的切换和位宽的转换。

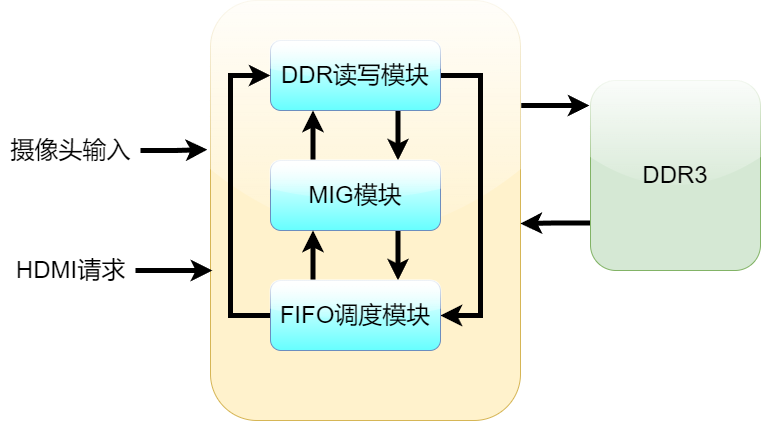


图 24 DDR控制模块顶层框图

DDR读写模块主要是产生DDR读写所需要的控制信号及地址信号等，同时在该模块内部设计了用于控制DDR3读写流程的状态机，如图25所示。整个状态机由4个基本状态组成，包括空闲状态(idle)、DDR3空闲状态(ddr3\_done)、写状态(write)、读状态(read)。在复位结束后，如果DDR3没有初始化完成，那么状态一直在空闲状态(idle)，否则就跳转到DDR3空闲状态(ddr3\_done)。

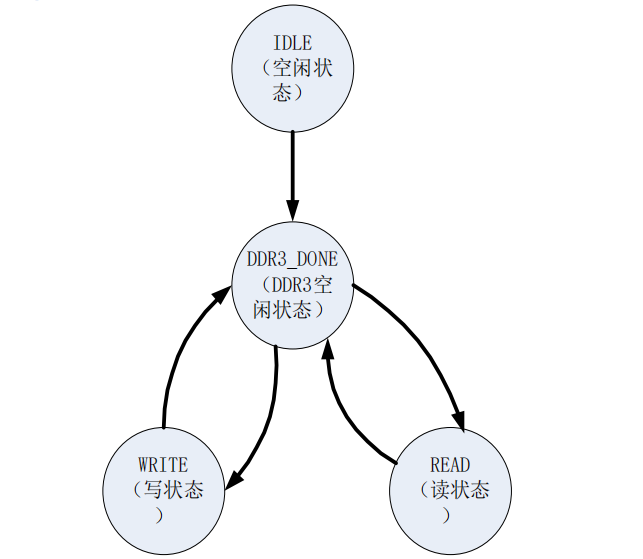


图 25 DDR3读写状态机

在DDR3空闲状态，当输入端的帧复位信号到来时，对写地址计数器和写地址进行复位操作。FIFO控制模块优先处理DDR3写请求信号，以免写FIFO溢出，用于写入DDR3的数据丢失。当写FIFO的存储数据达到一次写操作的长度时，就将状态跳转到写状态。在写状态，当写地址计数器达到一次写操作的时候并且握手信号命令使能app\_rdy和写数据有效使能app\_wdf\_rdy同时为高的时候，将状态跳转到DDR3空闲状态。在地址复位寄存器为高，且持续的时间大于1000个时钟周期后，将状态跳转到读状态(read)，否则状态继续留在DDR3空闲状态，这样做的目的是为了保证字FIFO复位的时候不会向其中写入数据。当读FIFO的数据不满一次读操作的情况下，将状态跳转到读状态(read，否则就停留在DDR3空闲状态。

FIFO调度模块主要是用于解决数据跨时钟域处理的问题，其中写FIFO主要对摄像头的像素数据进行缓存，在DDR读写模块中的状态机进入写状态时将数据从写FIFO中读出，写入到MIG IP核中，最终写入到DDR3芯片中。而读FIFO主要是对DDR3读出的数据进行缓存，在HDMI数据请求信号有效时将数据送到HMDI驱动模块进行HDMI图像显示。

### 3.3.5 VIP模块设计

VIP模块的作用是对摄像头的灰度数据进行进一步处理。由于灰度数据每个像素只占8bit，而一片DDR3芯片的数据位宽是16bit，也即可以在DDR中相同的地址区间内存入两帧灰度数据。而VIP模块每次从DDR控制器的读FIFO模块中读出的16bit数据就是两帧图像的同一像素位置的两个灰度数据。而且图像的运动目标检测常见的方法是采用帧差法。故在VIP中首先会对传入的两帧灰度数据的像素做差，并设置一个阈值。如果像素差小于这个阈值则认为该像素点不是运动目标的像素点；如果像素差大于这个阈值则认为该像素点是运动目标的像素点。不是运动目标的像素点的值全部赋为0，是运动目标像素点的值全部赋为1。这样就实现对图像中运动目标的二值化。虽然理论上这种做法已经可以得到运动目标的大致位置，但考虑到二值化之后的数据是有噪声的，所以还需要将二值化后的数据进行形态学滤波。

形态学滤波是一种对图像的滤波处理方式，它的本质和其它的滤波器相同，能够实现对图像的去噪和增强等。形态学滤波处理的基本方式是对图像进行腐蚀和膨胀操作。膨胀操作通过将像素添加到该图像中的感知边界，扩张放大图像中的明亮白色区域。腐蚀操作和膨胀操作恰好相反，处理方式主要是沿着物体边界移除像素并缩小物体的大小。

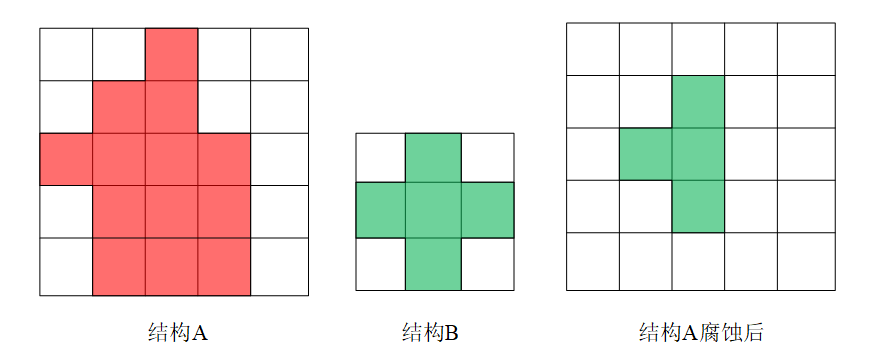


图 26 腐蚀操作示意图

图26所示是腐蚀操作的示意图，腐蚀可以理解为，移动结构B，如果结构B与结构A的交集完全数据结构A的区域内，则保存该位置点，所有满足条件的点构成结构A被结构B腐蚀的结果。而膨胀操作的示意图如图27所示，膨胀可以理解为，将结构B在结构A上进行卷积操作，如果移动结构B的过程中，与结构A存在重叠区域，则记录该位置，所有移动结构B与结构A存在交集的位置的集合为结构A在结构B作用下下的膨胀结果。



图 27 膨胀操作示意图

对帧差法后的二值化数据进行腐蚀和膨胀操作之后，可以得到比较干净的二值化图像。

在得到干净的二值化图像之后，可以根据二值化图像中为1的数据的像素位置来确定运动目标在视野中的大致位置，进而可以依据这个来得到运动目标位置的最小矩形边框。确定了运动目标的矩形框位置之后就可以在灰度图上覆盖边框了。整个VIP模块的工作流程图如图28所示：

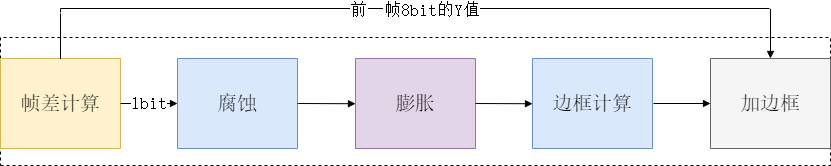


图 28 VIP模块的工作流程图

综上已经完成了对整个系统的硬件原理的介绍，下一章会针对每个模块在系统中所处的位置对每个模块的设计进行详细介绍，并单独测试每个模块。

# 4 硬件分系统设计方案

本章将介绍E203内核的改动模块以及SoC外设系统上的IPU模块与整个系统的关联设计，并分模块测试验证。

## 4.1 E203内核的改动模块

### 4.1.1 动态分支预测器

由上一章介绍可知该动态分支预测器内部包含Gshare分支预测器，BTB目标地址预测器，RAS目标地址预测器。

分支预测器的具体参数为：分支目标缓存（BTB）项数为256，栈返回地址缓存器（RAS）为16项。并通过基于10位的全局历史的Gshare预测器共同组成前端的取指预测单元。

该模块在系统中所处的位置是在原Lite BPU模块内部。其对外的输入输出接口如图29所示：

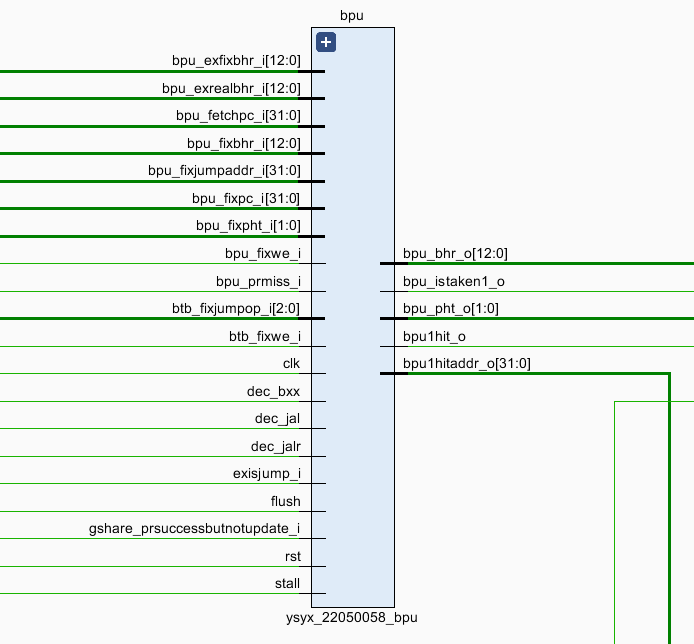


图 29 动态分支预测器

其IO的具体介绍如表1所示：

表 1 动态分支预测的IO引脚表

|  |  |  |
| --- | --- | --- |
| IO名 | 方向 | 介绍 |
| clk | Input | 时钟 |
| rst | Input | 复位，高电平有效 |
| stall | Input | IFU或者EXU产生的流水线暂停信号 |
| flush | Input | EXU的commit模块产生的流水线冲刷信号 |
| exisjump\_i | Input | EXU检测到指令跳转信号 |
| gshare\_  prsuccessbut  notupdate\_i | Input | 暂时无用，接0 |
| bpu\_prmiss\_i | Input | BPU预测错误信号 |
| bpu\_exfixbhr\_i | Input | 预测错误之前的GHR值 |
| bpu\_exrealbhr\_i | Input | 暂时无用，接0 |
| btb\_fixwe\_i | Input | 需要更新BTB信号 |
| bpu\_fixbhr\_i | Input | 在E203内核中，同bpu\_exfixbhr\_i |
| bpu\_fixpht\_i | Input | 预测错误之前的PHT值 |
| bpu\_fixpc\_i | Input | 预测错误的PC |
| bpu\_fixjumpaddr\_i | Input | 真正跳转的目标地址 |
| bpu\_fixwe\_i | Input | 需要修改PHT信号 |
| btb\_fixjumpop\_i | Input | 指示当前预测错误的跳转指令是什么类型的 |
| dec\_jal | Input | 与min decode相连，当前PC是jal类型指令 |
| dec\_jalr | Input | 与min decode相连，当前PC是jalr类型指令 |
| dec\_bxx | Input | 与min decode相连，当前PC是bxx类型指令 |
| bpu\_fetchpc\_i | Input | next pc |
| bpu1hit\_o | Output | 为1代表当前next pc的指令需要跳转，并且跳转的目标地址能在BTB或者RAS的查找表中找到 |
| bpu1hitaddr\_o | Output | 预测当前next pc的指令需要跳转的地址 |
| bpu\_bhr\_o | Output | 向流水线传入当前的GHR，用于预测错误时回滚到未预测错误时的GHR值。 |
| bpu\_pht\_o | Output | 向流水线传入当前的PHT，用于预测错误时回滚到未预测错误时的PHT值。 |
| bpu\_istaken1\_o | Output | 为1代表当前next pc的指令需要跳转 |

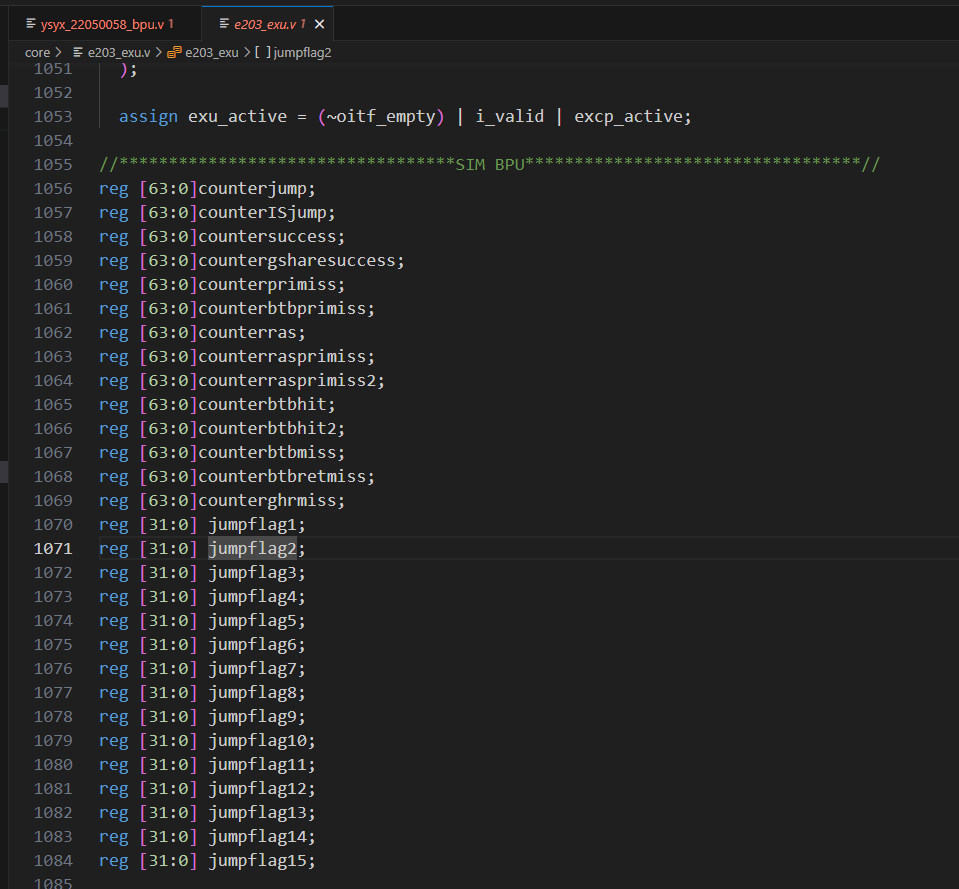
由上表可知动态分支预测器实际是与E203的整个流水线都是有关联的，需要在EXU阶段能对所有跳转指令预测错误的时候刷新整个流水线，并回滚分支预测器的状态到预测错误指令之前，能够根据实际commit的结果来对之前的分支预测器的内部参数做调整，还要充分利用了E203在IFU中自带的mini decode。

由于E203原本的分支预测器是能够在IFU阶段就能使用加法器计算next pc值的。对于jal这种类型的指令，由于知道其必跳转，且跳转的目的地址就是当前的pc值加上jal指令中的立即数。所以对于jal，原本的分支预测器是能够做到百分之百预测成功的；对于bxx类型的指令，由于不能确定当前指令是否taken，所以不能做到百分之百预测成功，只能利用BTFN来对指令是否跳转来做预测；对于jalr类型的指令，能够确定其必跳转，但不能确定跳转的地址，只能利用对x1寄存器的加速读取或者阻塞当前流水等待读取jalr指令中对应寄存器中的数据才能继续进行下一阶段的流水，所以对于jalr指令其也是百分之百预测命中的，但可能会造成流水线阻塞，产生空泡。

针对E203原始的分支预测器所结合的环境以及预测短板，我们选择对jal指令的预测仍然采用E203原本的分支预测方法；对bxx类型的指令选择用Gshare算法来预测该指令是否跳转，用原本的IFU加法器来计算跳转的目标地址；对于jalr类型的指令选择用BTB和RAS对其跳转的目标地址做预测，如果在BTB或RAS查找表中未命中当前PC，则采取E203原始分支预测器对jalr指令的预测方式。同时由于原本的E203分支预测器在处理jalr类型指令的时候，会因为OITF中有未写回的指令而阻塞对Regfile的访问，但实际OITF中的rd寄存器未必是与jalr指令中的寄存器相同的，所以这样也造成了不必要的流水线阻塞，所以新的分支预测器还直接引入了OITF中所有有效entry中的rd寄存器编号，这样能够很好的解决上述不必要的流水线阻塞问题。

综上新的分支预测器可以保证分支预测的成功率一定高于原始的E203分支预测器。

在对动态分支预测器的命中率做测试的时候，在e203\_exu模块中设置了一系列计数器来对分支预测器的命中率与miss率做统计，如图30所示。



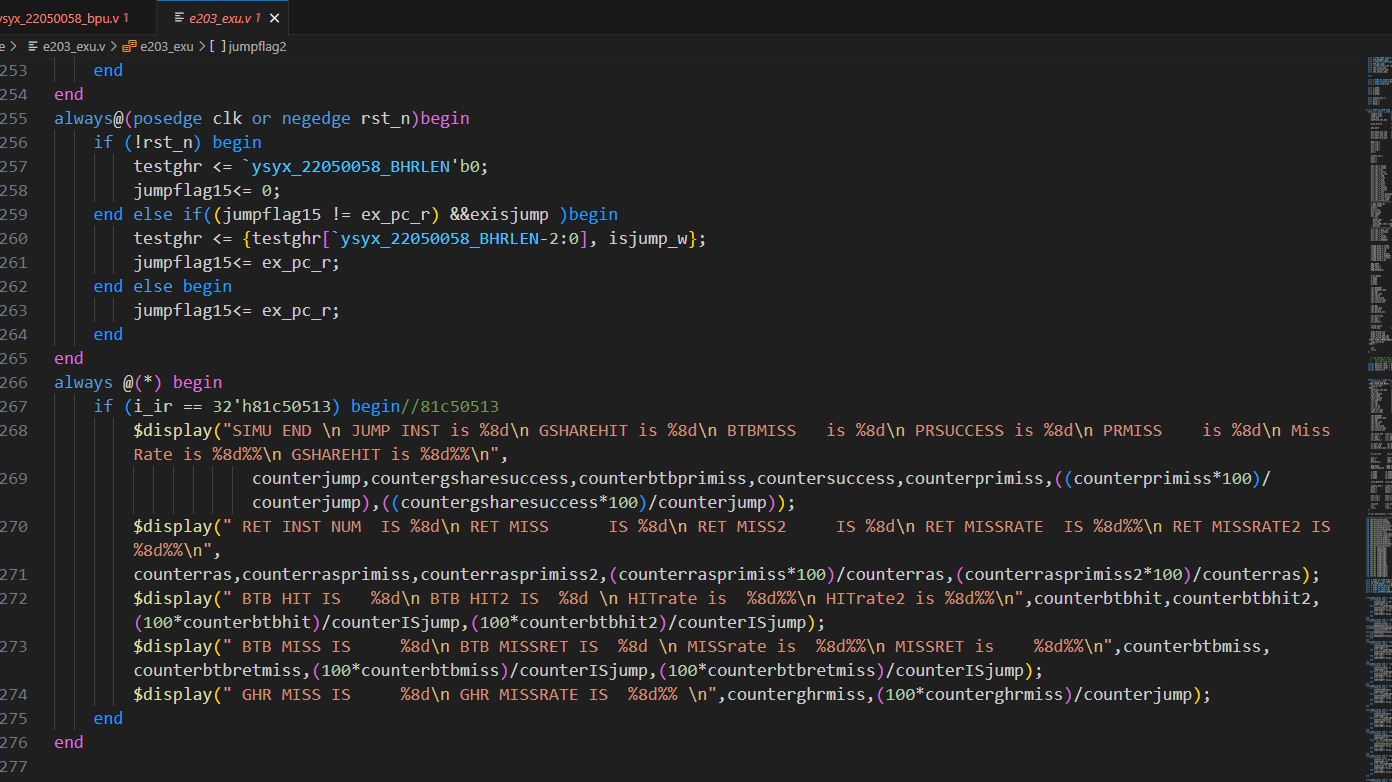


图 30 分支预测器命中率测试代码

在Coremark基准测试程序500迭代的负载下，测试其各项命中率如图31所示：

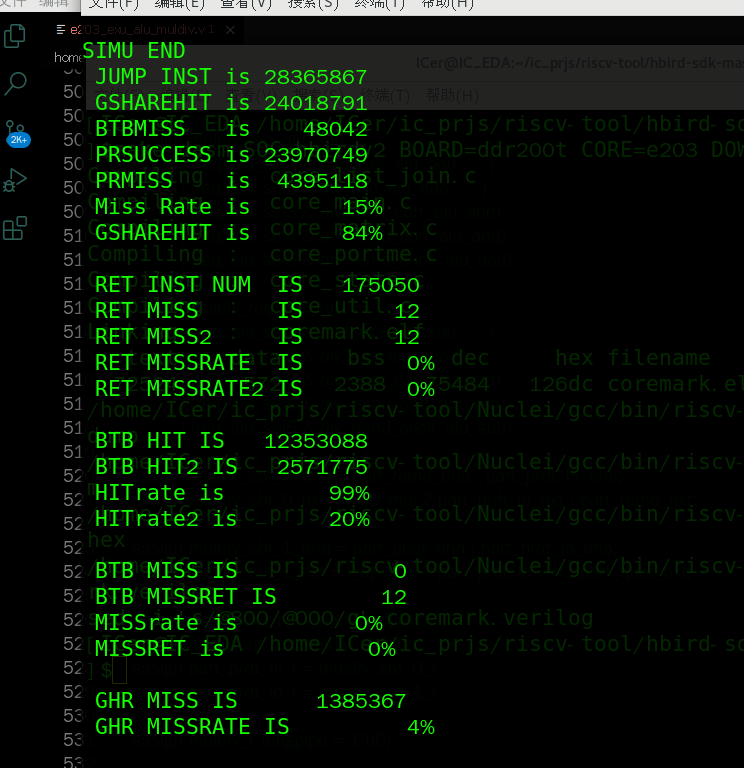


图 31 分支预测器的测试命中率

该测试结果会在Coremark程序执行末尾自动打印，因为我在Coremark的程序末尾插入了一条特殊指令如图32所示：

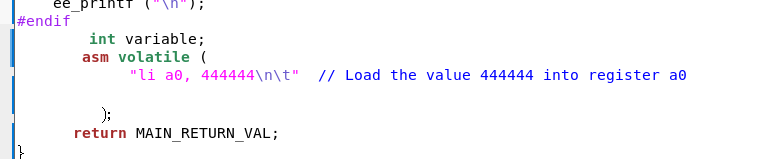


图 32 Coremark末尾处的特殊指令

该指令会最终在编译的时候生成一个值为81c50513的特殊指令，我在BPU的测试程序里对这个指令进行识别就可以做到判定程序已经执行到末尾，并触发打印命中率统计结果。

根据最终的统计结果汇总到表2为：

表 2 分支预测器在Coremark基准测试程序下的命中率统计

|  |  |
| --- | --- |
|  | Coremark负载 |
| 跳转指令总数 | 28365867 |
| Gshare命中次数 | 24018791 |
| BTB预测错误次数 | 48042 |
| Return指令次数 | 175050 |
| BTB命中次数 | 12353088 |
| Gshare命中概率 | 84% |
| BTB命中概率 | 99% |
| RAS命中概率 | 20% |
| 总预测正确率 | 84% |

由统计可知，除了RAS的命中率不尽如人意以外，其他指标的命中率都比较不错，接近它们的理论命中率。最终的整体预测正确率可以达到84%。

### 4.1.2高性能乘法器

前一章已经介绍了，该乘法器是radix-4 booth编码华莱士树结构的乘法器。并且是采用敏捷开发语言SpinalHDL做设计，最终生成verilog文件。E203原本的乘法器接口是将从寄存器读出来的操作数送入到e203\_exu\_alu\_muldiv模块中，并且利用握手协议来进行读入和输出数据。而我们的乘法器是单周期乘法器，所以在执行乘法操作的时候，一直将握手信号的vaild和ready都拉高即可。

乘法器的IO口如表3所示：

表 3 乘法器IO列表

|  |  |  |
| --- | --- | --- |
| IO名 | 方向 | 介绍 |
| mulI\_payload\_a | Input | 乘数a |
| mulI\_payload\_b | Input | 乘数b |
| mulI\_payload\_high | Input | 为1输出高32bit结果，为0输出低32bit结果 |
| mulI\_payload\_sign | Input | 为1是有符号乘法，为0是无符号乘法 |
| mulO\_payload | Output | 乘法结果（32bit） |

**对该乘法器进行测试：**

直接运行名为bao\_mul的vivado工程即可，测试结果如图33所示：

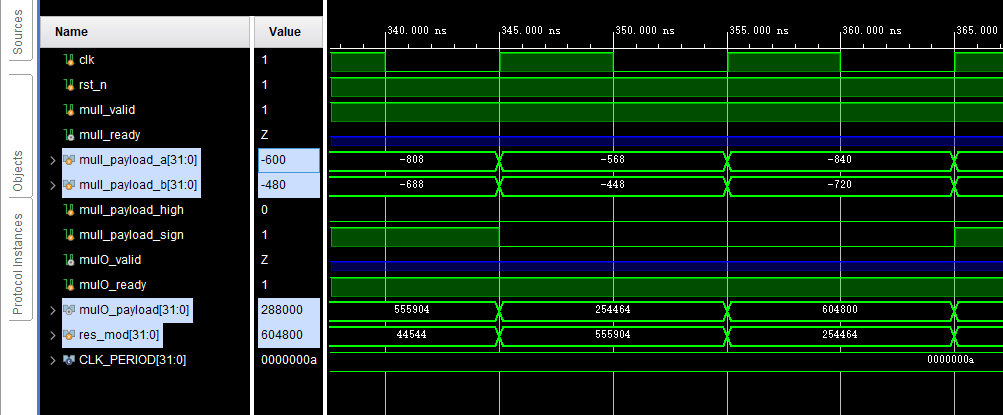


图 33 乘法器行为级仿真结果

其中res\_mod是参考结果，是直接利用乘法计算出来的结果，且由于都是非阻塞赋值，所以res\_mod的值更新的要慢一个周期。可以看出乘法器的计算结果与参考结果一致。乘法器仿真通过。

### 4.1.3 高性能除法器

上文提到，为了尽可能降低除法器的Latency Cycle，我们选择了使用xlinx公司的除法器ip。该ip配置的除法器是基2的有符号除法器，输出数据的高32bit是商，低32bit是余数，Latency Cycle为1，接口是按照AXI4-stream协议。

除法器的IO列表如表4所示：

表 4 除法器IO列表

|  |  |  |
| --- | --- | --- |
| IO名 | 方向 | 介绍 |
| aclk | Input | 时钟 |
| s\_axis\_divisor\_tvalid | Input | 除数有效信号 |
| s\_axis\_dividend\_tvalid | Input | 被除数有效信号 |
| s\_axis\_divisor\_tdata | Input | 除数 |
| s\_axis\_dividend\_tdata | Input | 被除数 |
| m\_axis\_dout\_tvalid | Output | 结果有效信号 |
| m\_axis\_dout\_tdata | Output | 计算结果（高32bit为商，低32bit为余数） |

**对除法器进行测试：**

直接运行名为div\_ip的vivado工程即可。

测试结果如图34所示：

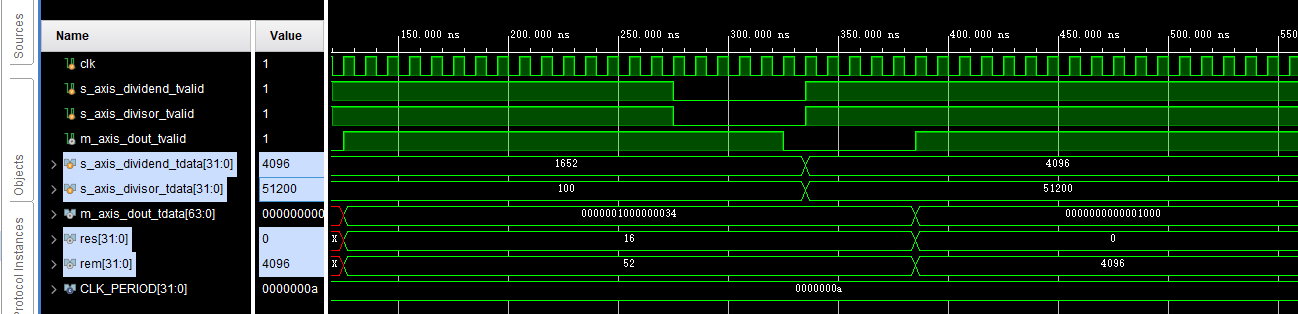


图 34 除法器行为级仿真结果

除法计算的商和余数都正确。除法器测试通过。

### 4.1.4 SM4硬件加速电路

该SM4的硬件加速电路的IO接口如表5所示：

表 5 SM4硬件加速电路IO列表

|  |  |  |
| --- | --- | --- |
| IO名 | 方向 | 介绍 |
| clk | Input | 时钟 |
| reset\_n | Input | 复位，低电平有效 |
| sm4\_enable\_in | Input | 使能sm4加速电路接口，高电平有效 |
| encdec\_enable\_in | Input | 使能SM4加解密工作模式，高电平有效 |
| encdec\_sel\_in | Input | 选择加密或者解密模式，高电平加密，低电平解密 |
| valid\_in | Input | 输入数据有效信号 |
| data\_in | Input | 输入待加/解密的明/密文 |
| enable\_key\_exp\_in | Input | 使能秘钥扩展工作模式，高电平有效 |
| user\_key\_valid\_in | Input | 有效秘钥输入信号，高电平有效 |
| user\_key\_in | Input | 输入的秘钥数据 |
| ready\_out | Output | 输出结果有效信号，高电平有效 |
| result\_out | Output | SM4硬件加速电路的输出结果 |
| key\_exp\_ready\_out | Output | 秘钥扩展完成信号，高电平有效 |

我们设计的是将SM4硬件加速电路添加到E203内核的NICE扩展协处理器接口上。在指令译码的时候发现是一条custom3类型的指令的时候，会自动将该指令派遣到NICE接口上。在nice\_core模块内部会对发送过来的指令进行进一步解码，所以我们扩展指令的解码操作实际是在nice\_core内部完成的，这样的设计极大减小了扩展算子与外部内核微架构的耦合程度，在添加扩展指令的时候只需要再nice\_core内部添加即可，不用对外部流水线架构做任何修改。

针对我们的SM4硬件加速电路的工作特性，我们为其设计了五条扩展指令，如表6所示：

表 6 SM4的扩展指令

|  |  |  |
| --- | --- | --- |
| 协处理器指令 | 介绍 | 编码 |
| load\_key | 从内存中加载秘钥 | Opcode指明使用custom3指令组  Xd位的值为0，表明此指令不需要写回任何结果  Xs1位的值为1，表示该指令需要读取操作数rs1，操作数rs1就是秘钥的数据  Xs2位的值为0，表示该指令不需要读取操作数rs2  Func7为0000001 |
| load\_data | 从内存中加载待加解密的数据 | Opcode指明使用custom3指令组  Xd位的值为0，表明此指令不需要写回任何结果  Xs1位的值为1，表示该指令需要读取操作数rs1，操作数rs1就是待加解密的数据  Xs2位的值为0，表示该指令不需要读取操作数rs2  Func7为0000010 |
| encrypt | 秘钥扩展并进行加密操作 | Opcode指明使用custom3指令组  Xd位的值为0，表明此指令不需要写回任何结果  Xs1位的值为0，表示该指令不需要读取操作数rs1  Xs2位的值为0，表示该指令不需要读取操作数rs2  Func7为0000110 |
| decrypt | 秘钥扩展并进行解密操作 | Opcode指明使用custom3指令组  Xd位的值为0，表明此指令不需要写回任何结果  Xs1位的值为0，表示该指令不需要读取操作数rs1  Xs2位的值为0，表示该指令不需要读取操作数rs2  Func7为0000111 |
| sm4\_res | 将加解密之后的数据存到内存中 | Opcode指明使用custom3指令组  Xd位的值为0，表明此指令不需要写回任何结果  Xs1位的值为1，表示该指令需要读取操作数rs1，操作数rs1就是要存储结果的地址  Xs2位的值为0，表示该指令不需要读取操作数rs2  Func7为0000100 |

在使用该加速器的时候，应该首先将秘钥读入到加速器中，再将待加/解密的数据读入到加速器中，之后发送加/解密指令进行加/解密操作，在加/解密操作完成之后将计算的结果存储到内存空间中。其流程图如图35所示：

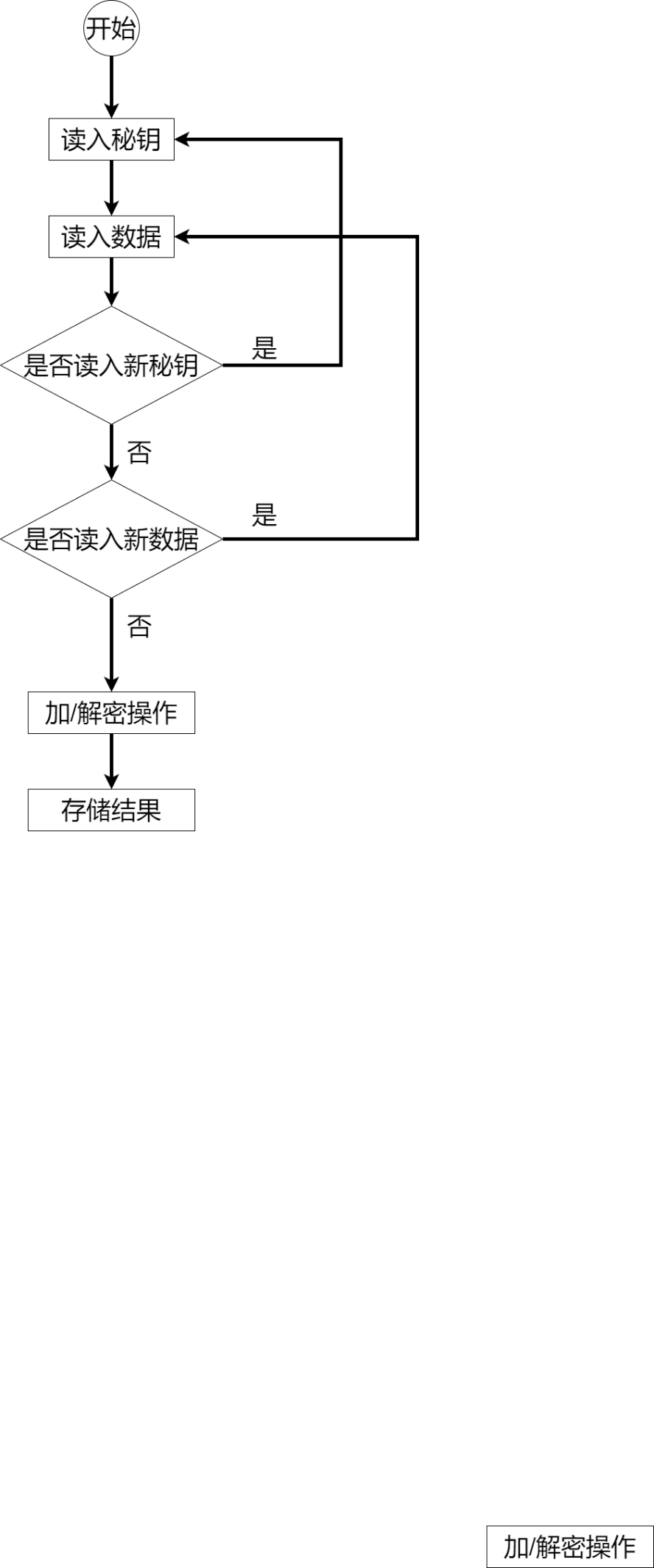


图 35 SM4硬件加速器的工作流程图

将sm4\_source文件夹中的代码添加到modelsim工程中，进行编译就可以得到采用modelsim仿真的波形图，如图36所示：

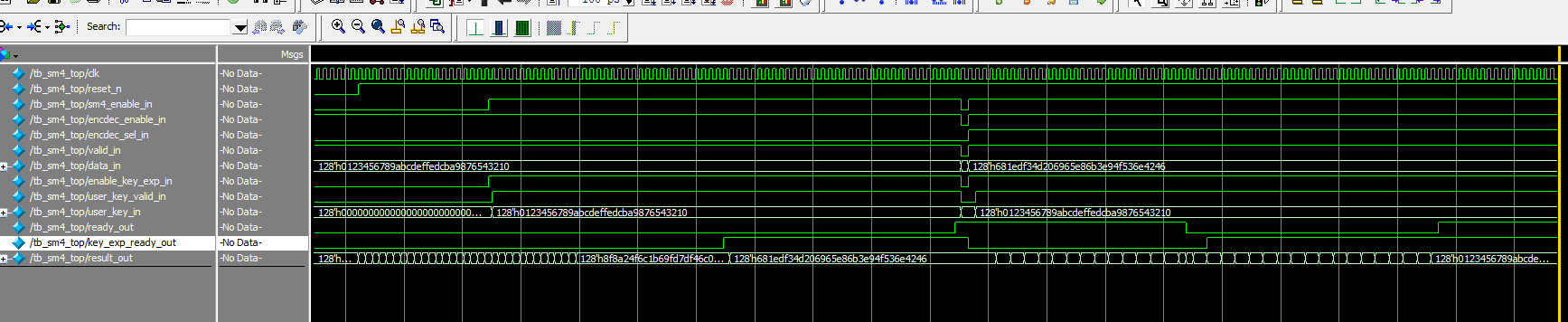


图 36 SM4模块的仿真波形

上图是先对128bit的16进制表示的密钥“01 23 45 67 89 ab cd ef fe dc ba 98 76 54 32 10”进行秘钥扩展，之后再对128bit的16进制表示的明文“01 23 45 67 89 ab cd ef fe dc ba 98 76 54 32 10”进行加密。如图37所示：

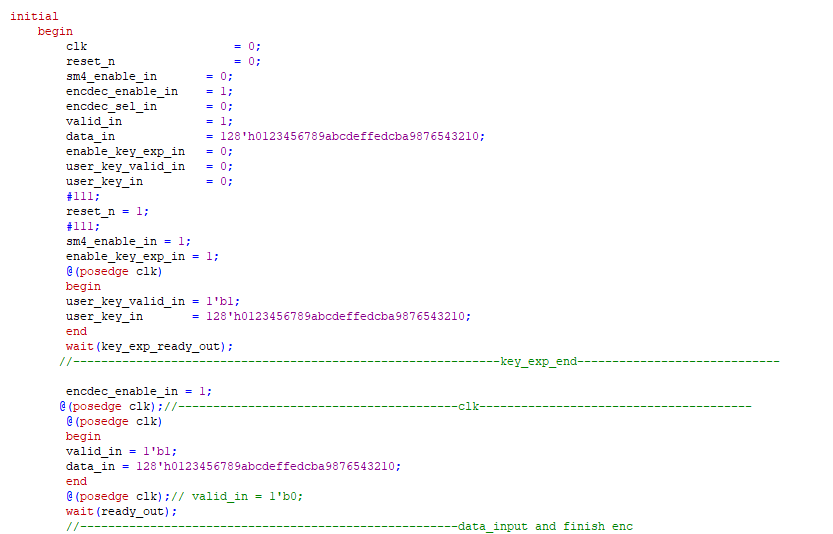


图 37 SM4加密测试程序

最终得到的加密结果为：681EDF34D206965E86B3E94F536E4246002A8A4EFA863CCAD024AC0300BB40D2

而在SM4在线加密网站上测试的结果也是681EDF34D206965E86B3E94F536E4246002A8A4EFA863CCAD024AC0300BB40D2

，如图38所示。故SM4硬件加速器能正确加密。

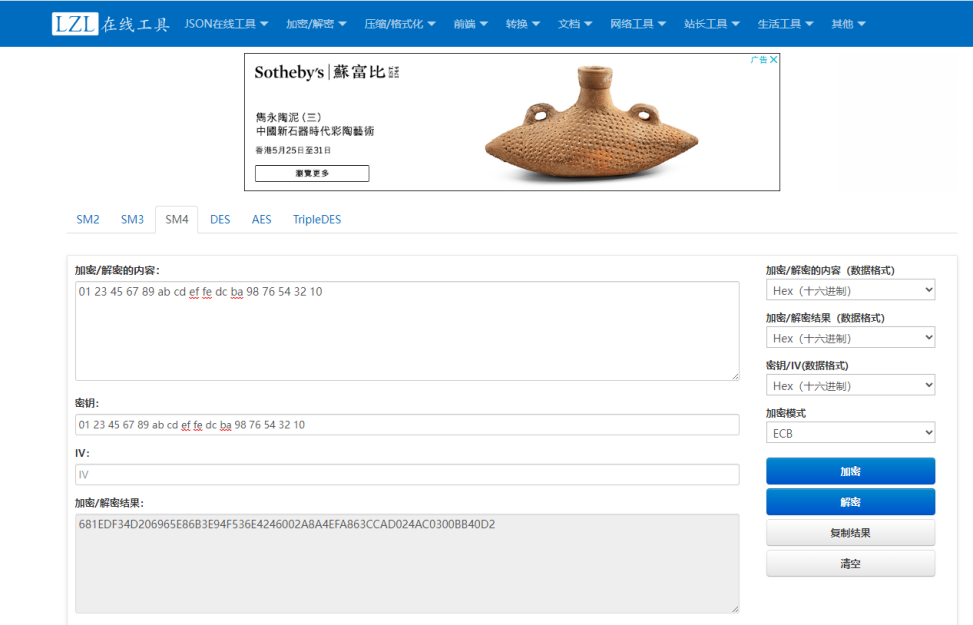


图 38 SM4在线加密网站的结果

之后再将加密的数据作为输入通入到SM4硬件加速器中解密，如上图波形图以及图39所示，得到的解密结果为01 23 45 67 89 ab cd ef fe dc ba 98 76 54 32 10，与原明文相同。SM4硬件加速器能正确解密。

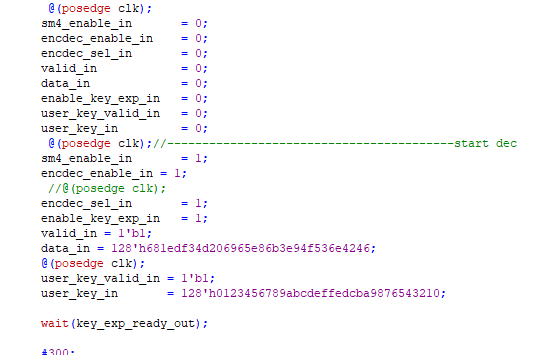


图39 SM4解密测试程序

## 4.2 SoC外设系统

### 4.2.1 IPU模块

IPU模块最终是会挂在到SoC的外设总线上，所以需要先为IPU在SoC系统上分配地址。在E203的外设总线上已经预留出一个AXI的外设接口，这个AXI外设接口的基地址是32'h1004\_1000，所以我们就利用这个接口将IPU模块挂载到SoC总线上。

IPU模块占用的物理内存空间为0x1004\_1000~0x1004\_1FFF。下表7是E203 SoC物理内存地址的空间分配：

表 7 E203 SoC物理内存地址分配

|  |  |  |
| --- | --- | --- |
| 总线分组 | 组件 | 地址区间 |
| Core直属 | CLINT | 0x02000000~0x0200ffff |
| PLIC | 0x0c000000~0x0cffffff |
| ITCM | 0x80000000~0x8001ffff |
| DTCM | 0x90000000~0x8001ffff |
| 系统存储总线接口 | Debug Module | 0x00000000~0x00000fff |
| ROM | 0x00001000~0x00001ffff |
| Off\_chip\_QSPI0 | 0x20000000~0x3fffffff |
| 私有外设总线 | 常开域 | 0x10000000~0x10007fff |
| HCLKGEN | 0x10008000~0x10008fff |
| GPIOA | 0x10012000~0x10012fff |
| UART0 | 0x10013000~0x10013fff |
| QSPI0 | 0x10014000~0x10014fff |
| PWM | 0x10015000~0x10015fff |
| UART1 | 0x10023000~0x10023fff |
| QSPI1 | 0x10024000~0x10024fff |
| IIC0 | 0x10025000~0x10025fff |
| UART2 | 0x10033000~0x10033fff |
| QSPI2 | 0x10034000~0x10034fff |
| IIC1 | 0x10035000~0x10035fff |
| GPIOB | 0x10040000~0x10040fff |
| IPU | 0x10041000~0x10041fff |
| 保留 |  |

IPU在系统中的作用是将摄像头视野中的物体坐标存储起来，并发送给CPU做进一步处理。所以对于IPU来说其主要任务是得到图像中运动物体的坐标值。而上一章中我们已经介绍了IPU检测图像中运动目标的工作流程，其中提到了在运动目标周围画框，所以可以根据画框的中心像素点位置来确定运动目标的坐标值。

由于我们对OV5640配置的分辨率是1024\*768，所以运动目标的坐标值肯定不会大于这个区间，故坐标值可以用2个11bit的寄存器去存储。

故我们对IPU配置了三个寄存器，如表8所示：

表 8 IPU寄存器

|  |  |  |  |
| --- | --- | --- | --- |
| 寄存器名 | 寄存器位宽 | 寄存器描述 | 寄存器地址 |
| IPU\_EN | 1 | IPU使能寄存器，给寄存器的第0位配置1代表使能IPU，给第0位配置0代表禁止IPU | 0x10041000 |
| AIM\_X | 11 | 运动目标的X坐标值 | 0x10041004 |
| AIM\_Y | 11 | 运动目标的Y坐标值 | 0x10041008 |

E203内核可以直接通过仿存程序访问IPU的寄存器，通过控制IPU使能，再读取AIM\_X和AIM\_Y寄存器上的数据来获得运动目标的坐标数据。

E203内核对外的总线接口是ICB总线，不过芯来公司已经在SoC外设总线上做好了ICB转AXI协议的模块，所以我们的IPU外设最终是挂载到AXI接口上，再将AXI接口转成ICB接口连接到系统的外设总线上，最终能被E203内核访问到。

现将IPU挂载到SoC外设总线上，并在C程序中对其寄存器进行读写操作如图所示：

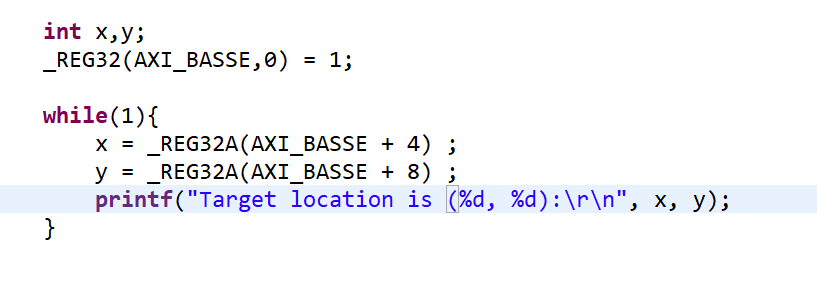


图 39 C程序中对IPU外设的寄存器配置

最终测试结果如图40，图41所示:

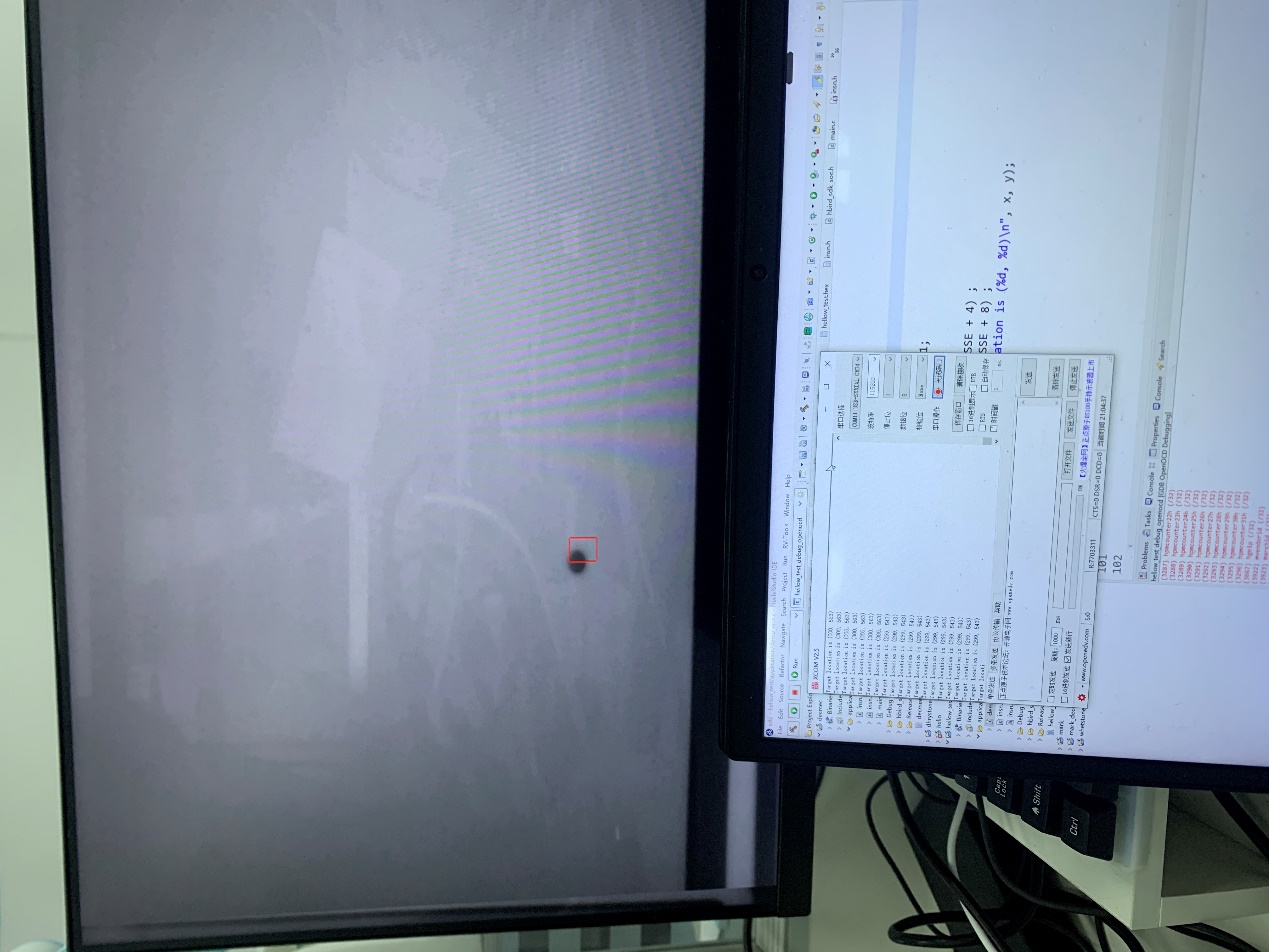


图 40 运动目标画框标记

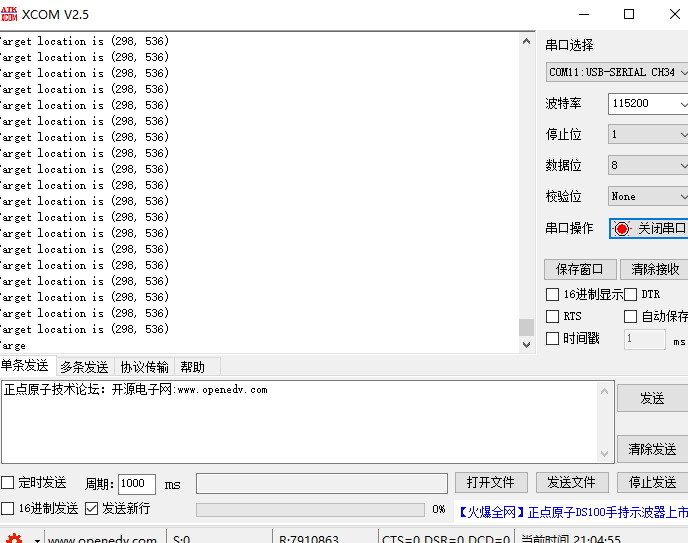


图 41 E203读到的坐标数据

由测试结果可知，IPU能对运动目标进行画框操作，对于显示器来说左上角是坐标零点，显示器的x坐标区间为（0~1024），y坐标区间为（0~768）。而图中目标的位置在屏幕的左偏下的位置，内核读取到检测坐标为（293，536），该坐标值与目标在屏幕中的位置坐标直观上感觉比较吻合。故IPU能正确检测运动物体的坐标。

综上已经对设计的每一个模块进行了测试验证，下一章将对整体系统进行分功能测试。

# 5 系统整体测试与分析

本章将对最终系统进行以下测试：

1、仅修改分支预测器之后的Coremark仿真跑分（基于VCS）

2、上板之后的Dhrystone、CoreMark、Whetstone跑分测试

3、整体系统仿真测试SM4国密算子以及上板测试

4、整体系统上板SM4算子+上位机（基于python自研上位机）的应用测试

## 5.1 分支预测器的性能测试

### 5.1.1 测试流程

在已经安装芯来编译工具链的linux系统中（已安装vcs），进入jichaung\_bpu文件夹，再进入到e203/vsim文件夹下，在终端一次输入make clean、make install、make compile SIM=vcs、make run\_test SIM=vcs即可启动对Coremark基准测试。该基准测试迭代次数为500。最终的测试结果如图42所示：

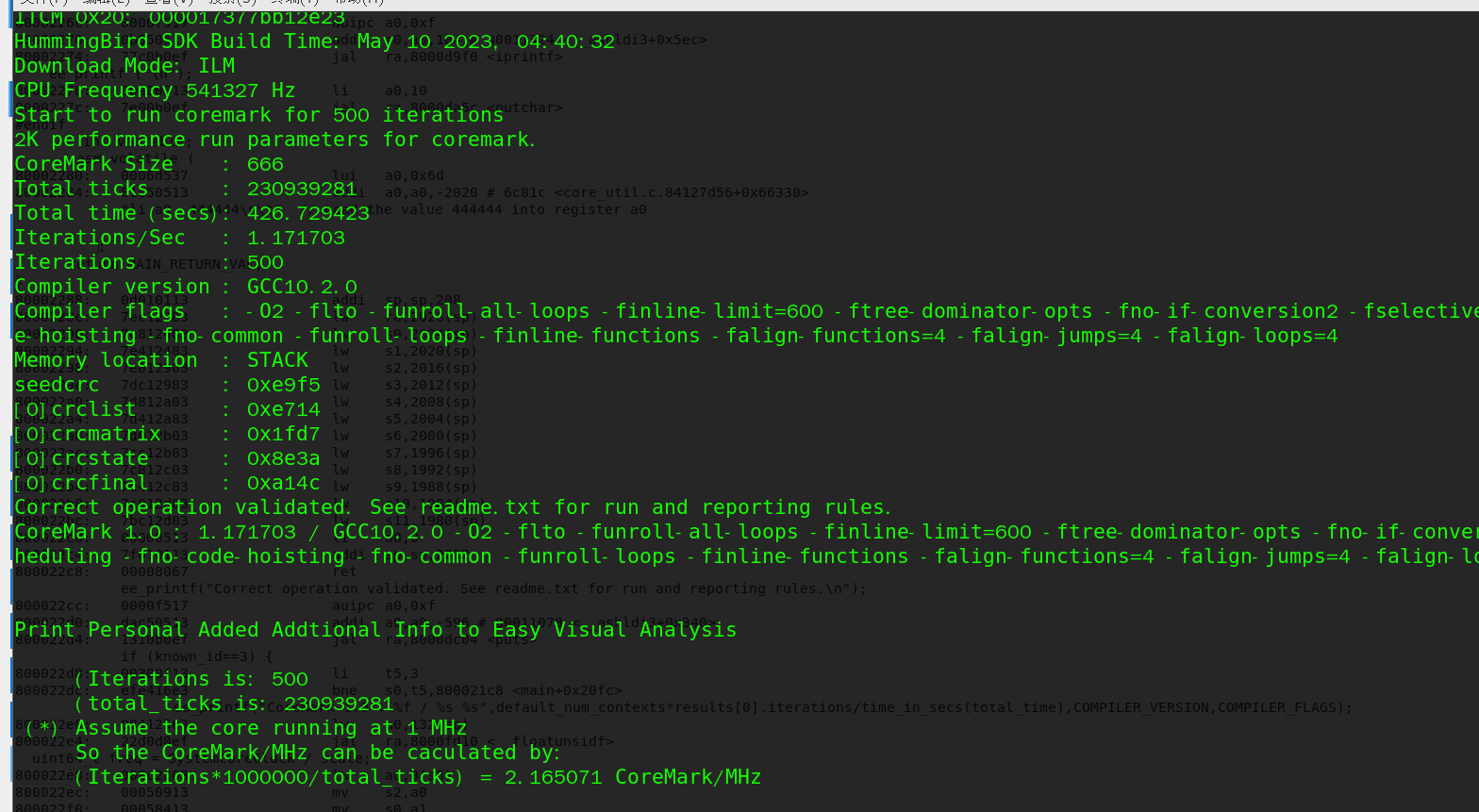


图 42 VCS上的Coremark测试

### 5.1.2 测试分析

该测试结果是仅修改分支预测器之后的结果，原本的E203的Coremark 500迭代次数的跑分为2.13 Coremark/MHz左右，可以看出仅将原本的静态分支预测器改成动态分支预测器并不能明显提升系统的性能跑分。

经过分析我们认为导致动态分支预测不能很好的提升性能的原因主要是：

1. E203的流水线深度太短，当发生预测错误的时候，带来的惩罚较小，所以相较于原始E203内核的性能提升比例不高。
2. 原本的E203分支预测器在mini decode和IFU加法器的加持下，已经能对jal指令百发百预测正确，对bxx指令的跳转地址百分百预测正确，对jalr也是百分百预测正确，只是会不定引入一些流水线空泡。所以虽然E203原本的BTFN预测算法只是静态预测，但已经能达到很高的预测正确率了，所以在原本E203环境的基础上分支预测能够提高的空间不大。

动态分支预测还是比较适合用在高性能处理器上，尤其是对有cache的处理器来说。对于短流水的处理器的性能提升并不大。

## 5.2 上板的性能跑分测试

### 5.2.1 测试流程

将最终修改后的E203软核下载进XC7A100T的FPGA中，进入NucleiStudio文件夹下，启动NucleiStudio.exe。默认的工程目录下已经包含了后面所有测试的工程项目。依次在IDE的当前工程选择框中选择mark\_debug\_openocd，dhrystone\_debug\_openocd，whetstone\_debug\_openocd，（所有基准测试程序的配置都与原本IDE自带例程中的配置相同，但要注意生成的bit流文件不可以含有压缩指令C，因为我们的动态分支预测器不支持对压缩指令的兼容）并依次将其下载进芯片中。最终运行结果如图43、图44、图45所示：

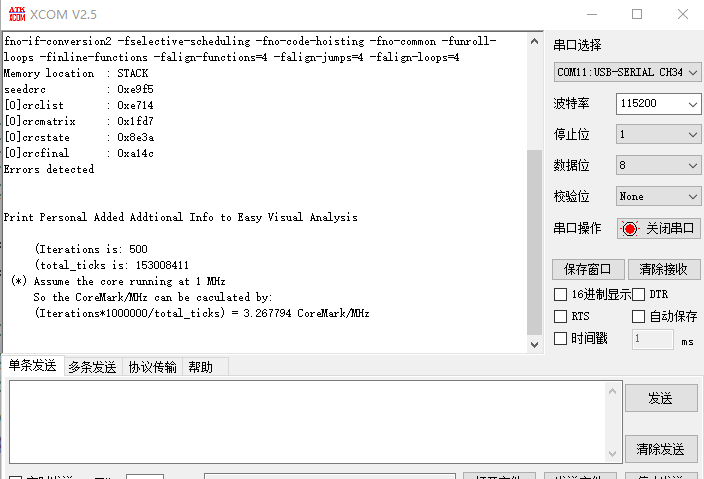


图 43 Coremark上板测试跑分

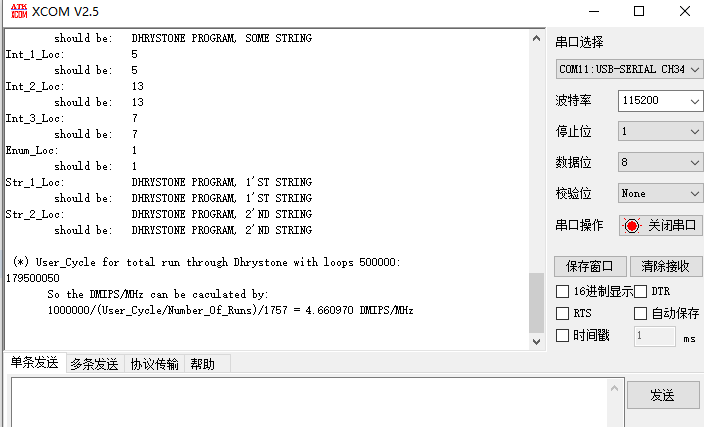


图 44 dhrystone上板测试跑分

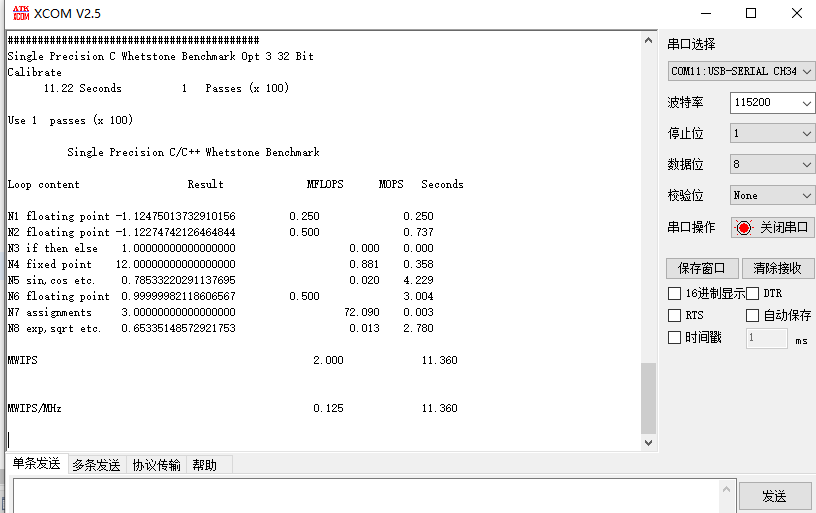


图 45 whetstone上板测试跑分

### 5.2.2 测试分析

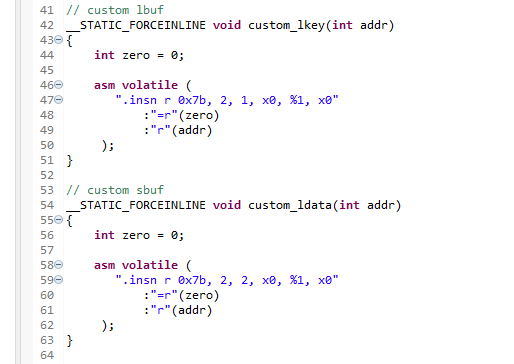
由上板测试可知，最终的Coremark跑分相较于原本的E203的内核Coremark跑分提升了65%，dhrystone跑分提升了3.64倍，whetstone跑分提升了3.125倍。可见最终对性能的提升还是很明显的。

虽然whetstone的跑分相较于原始的E203内核有较大的提升，但跑分其实依然较低。由于whetstone中用到了大量的浮点运算，所以在内核中加入FPU才能最大化的提升最终跑分结果。这是我们未来对作品要继续努力的一个地方。

## 5.3 SM4国密扩展指令测试

### 5.3.1 SM4扩展算子底层软件库设计

上一章中已经介绍了SM4的扩展指令设计，现在我们还需要在软件中能够使用这些扩展指令。我们在C代码工程中，建立一个inst.h用来存放我们扩展算子的底层软件驱动。根据对SM4五条扩展指令的设计，我们在inst.h中添加5条指令驱动，如图46所示：



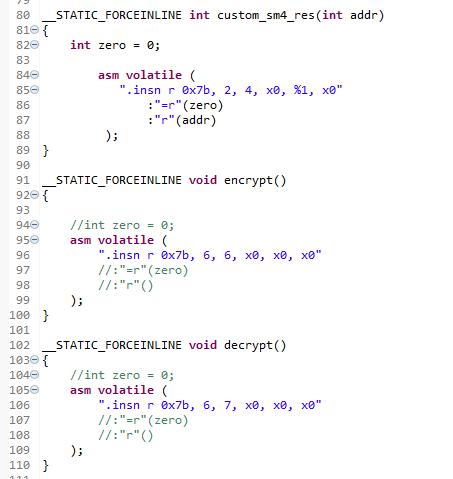


图 46 SM4扩展算子软件驱动

该软件驱动的原理是利用在C程序中内嵌汇编的方法。其语法格式为：

asm volatile( /\* volatile : 可选，禁止编译器对汇编代码进行优化 \*/

"汇编指令" /\* 汇编指令间使用'\n'分隔 \*/

:"=限制符"(输出参数)

:"限制符"(输入参数)

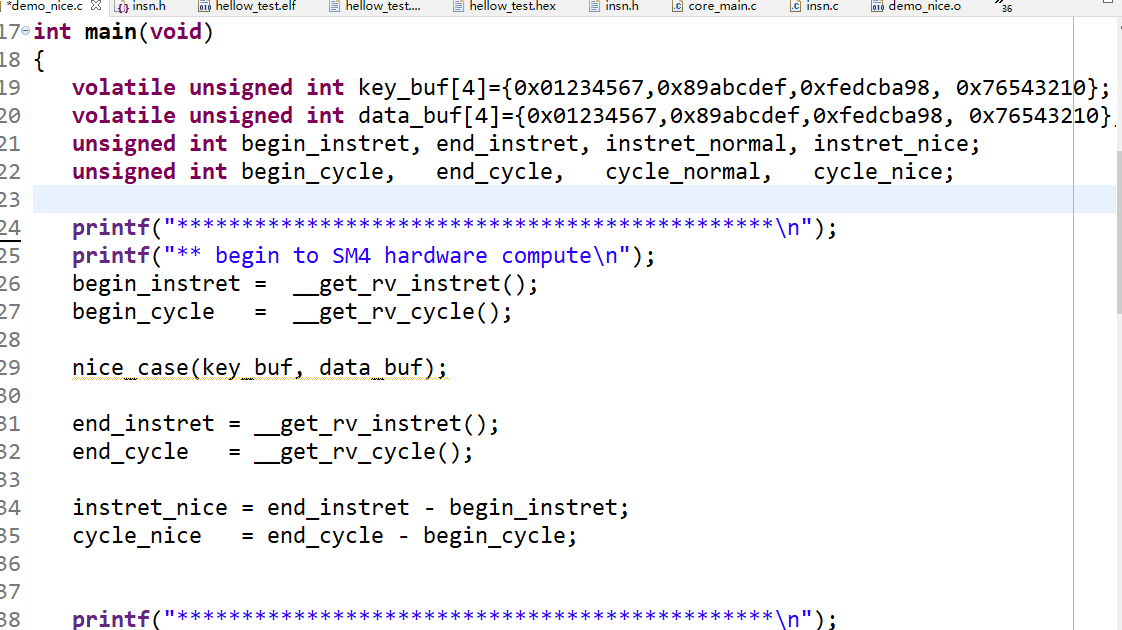
:保留列表

)

其作用是在C语言代码中嵌入汇编指令，由于我们自定义的指令是不能被普通的编译器所编译出来的，所以通过内嵌汇编指令的方式将自定义指令嵌入到编译器生成的汇编文件中是一种为扩展指令提供软件驱动的比较简单的方式。

### 5.3.2 测试流程

在提交材料的bin文件夹中存放了一些用于仿真的.verilog文件，其中的hellow\_test.verilog文件包含了用于行为级仿真测试SM4算子的比特流文件。在e203\_tb\_top.v中将$readmemh函数的这个文件的路径修改一下，就可以在vivado上进行SM4算子的行为级描述仿真了。我们的程序中调用SM4算子的方式如图47所示：



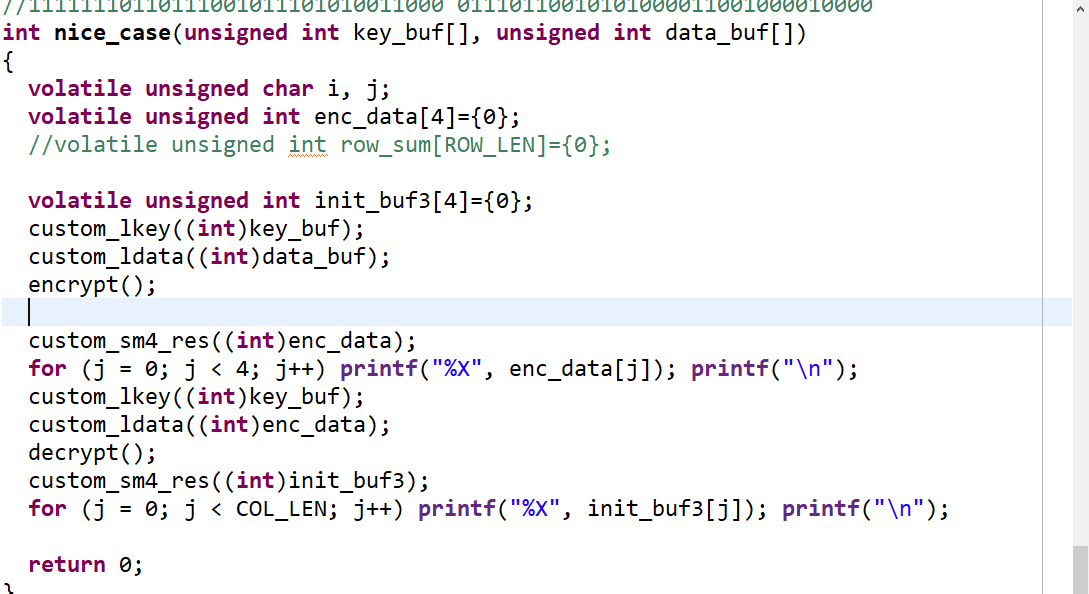
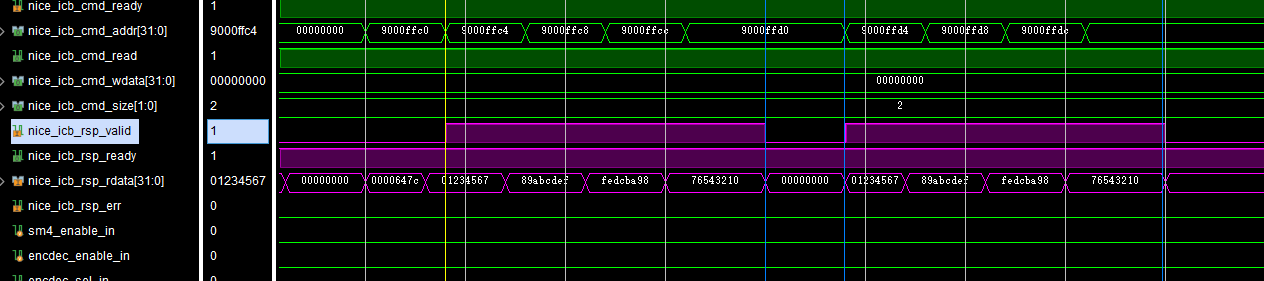
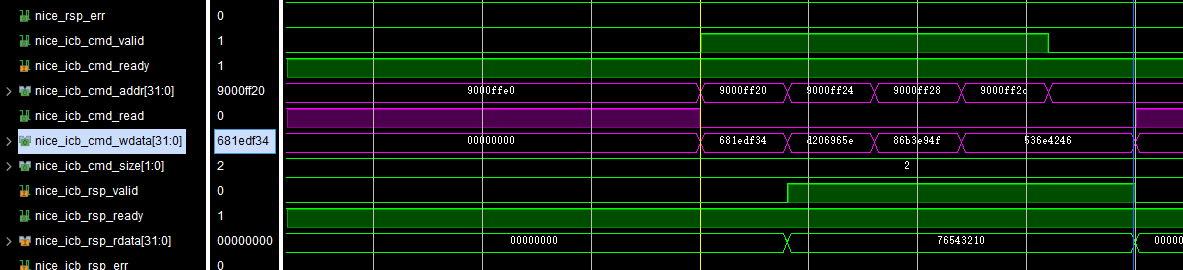
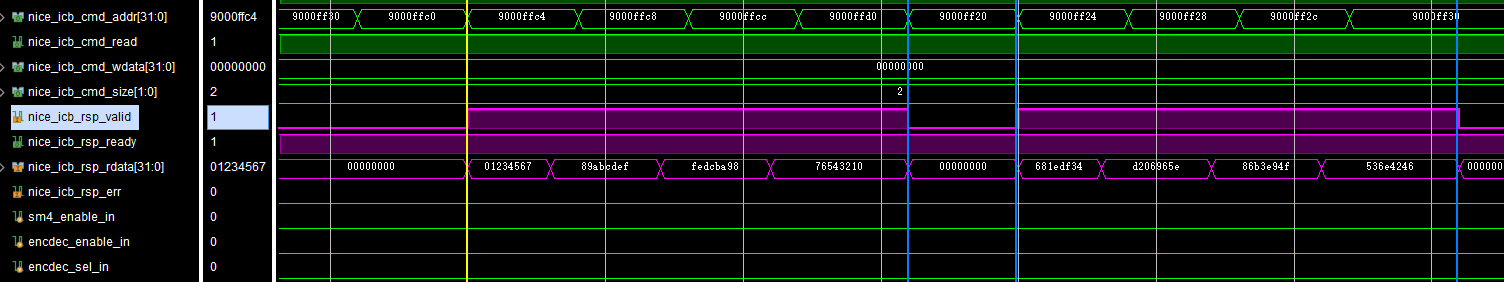


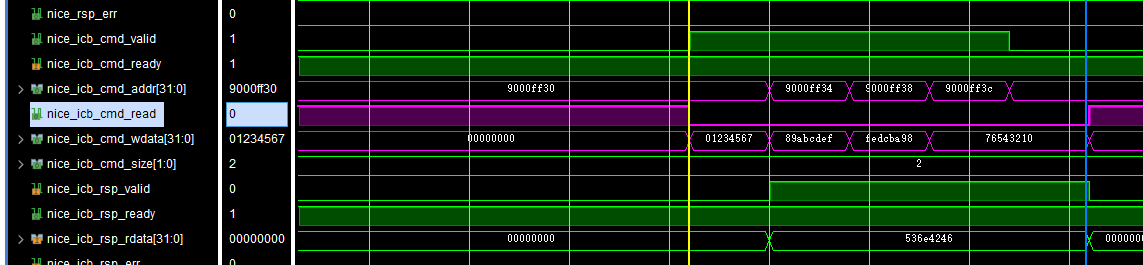
图 47 程序中调用SM4算子

最终的行为级仿真的测试结果如图48所示：









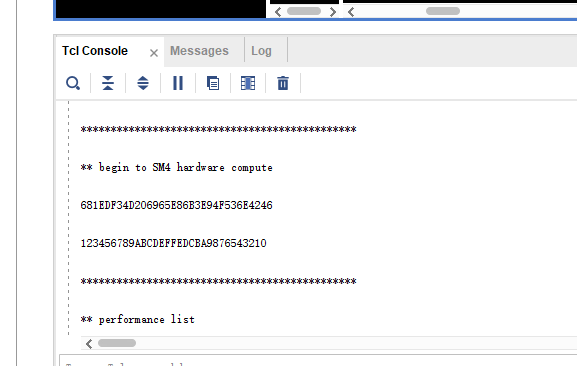


图 48 行为级仿真SM4算子的测试结果

如波形图以及vivado打印的结果所示，E203的SM4扩展算子能够正确的进行加解密操作。

随后再将软核代码下载进FPGA板中，并将C程序编译生成的比特流文件以ilm的形式烧录进芯片中，改c代码是用软件进行SM4解密以及用硬件加速器进行SM4解密分别做对比，用的也是hellow\_test工程。最终的上板测试结果如图49所示：

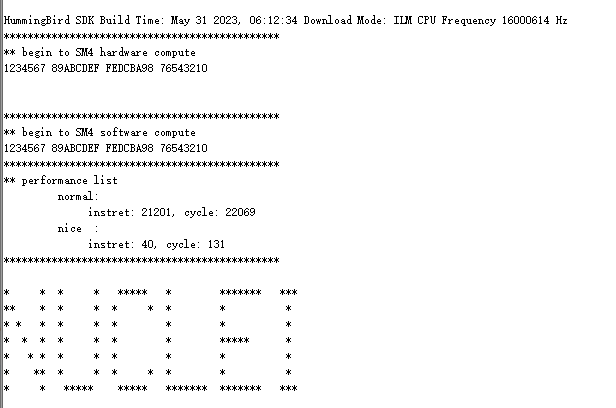


图 49 SM4算子的上板测试结果

如图所示，最终用硬件实现解密，指令数减少了530倍，消耗的指令周期数减少了168倍。使用我们设计的SM4计算算子能够很明显的对SM4算法的运算做出性能提升。

## 5.4 整体应用测试

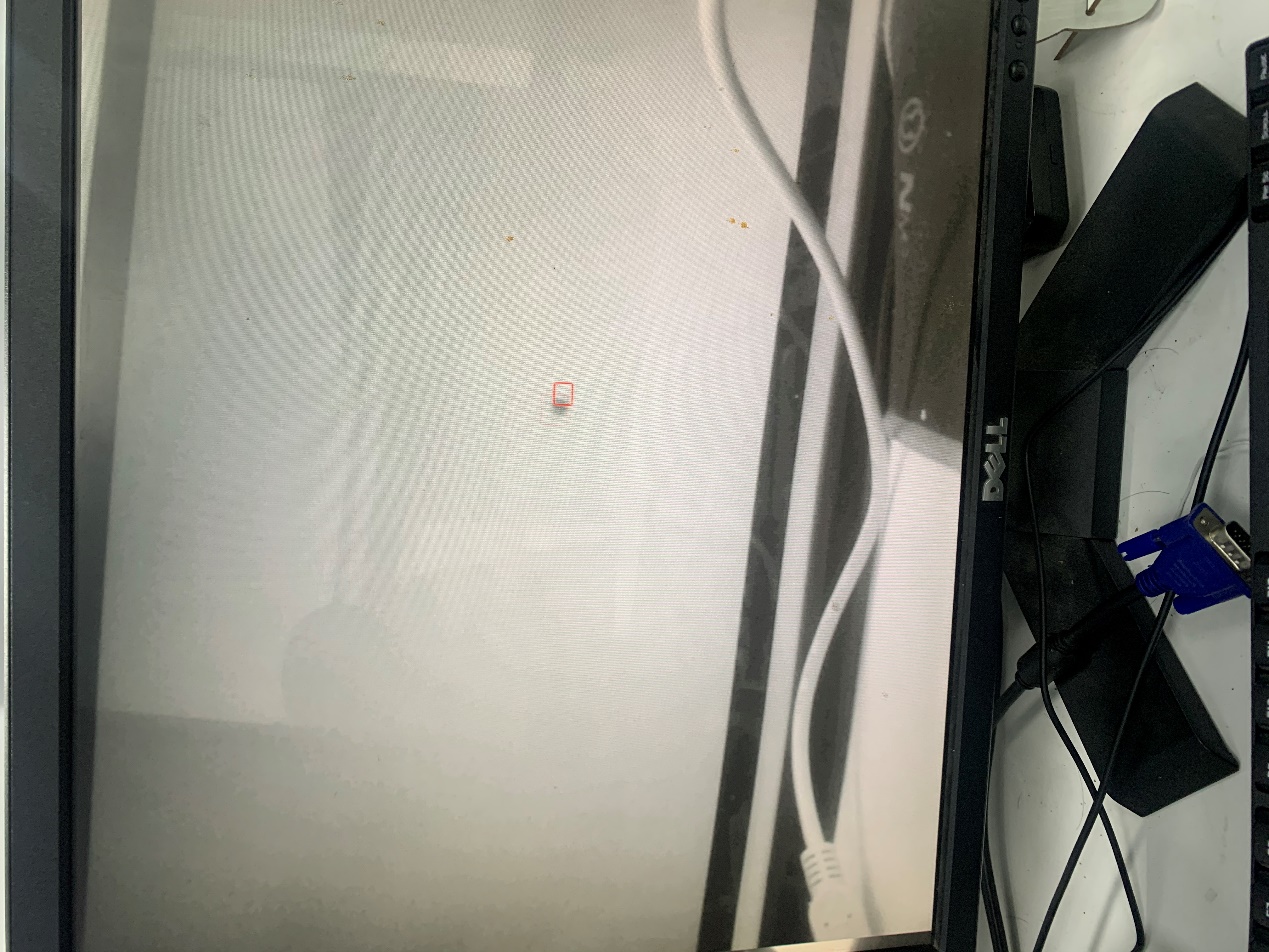
该应用是针对当前安全芯片的一种使用场景来进行设计的。现如今很多手机应用APP都具有以人脸作密码的功能，所以手机中的安全芯片要求在能够采集到人脸的情况下，不能泄露用户的人脸隐私信息，也就是要对采集到的信息进行加密。故本应用的设计就是IPU对摄像头视野中的运动物体进行坐标检测的时候，E203内核对坐标信息要先加密再对外传输。将加密后的信息传输到我们自研的上位机上之后，上位机能根据秘钥来对加密信息进行解密，并且将解密的坐标以小红点的形式动态的显示在上位机的坐标区域上。

### 5.4.1 测试流程

将jichuang\_final文件夹下的工程下载进FPGA板子中，并将NucleiStudio中的decmec工程烧录进芯片中。随后再启动上位机，在上位机开始界面输入当前开发板与电脑串口连接的COM口即可进入应用工作模式。

### 5.4.2 上板测试结果

最终上板测试的结果如图50所示：



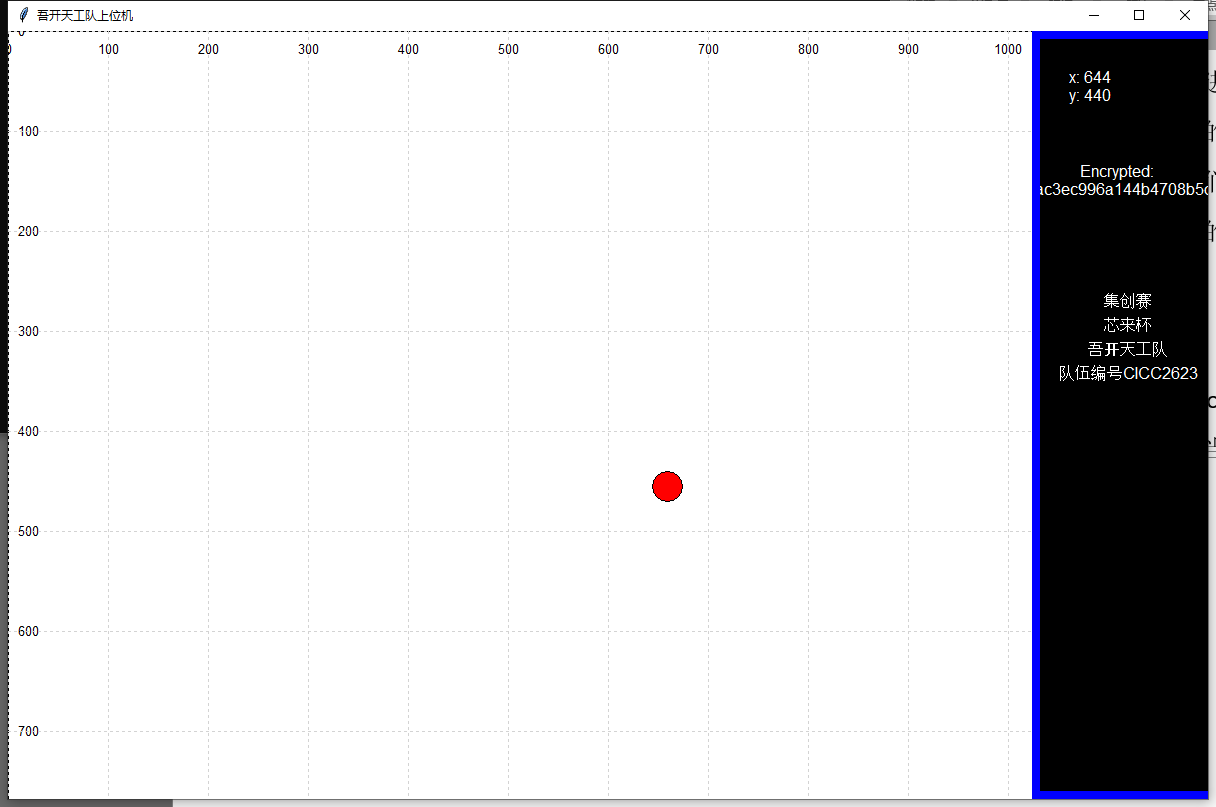


图 50 应用的上板测试结果

如图所示，上位机能对芯片发送过来的加密坐标进行解密。并使用一个小红点来模拟运动目标的运动轨迹。与我们预期相同。

# 6 总结与展望

## 6.1 总结

本设计工作主要在两个方面，一个是对内核进行优化，另一个是结合扩展算子进行系统级应用。

在内核优化方面，本设计对内核的分支预测，乘除法计算做了一定优化，最终的性能测试Coremark跑分相较于原本的E203的内核Coremark跑分提升了65%，dhrystone跑分提升了3.64倍，whetstone跑分提升了3.125倍，比较符合我们对内核优化的预期。

在扩展算子方面，本设计扩展了SM4国密算法算子，使得能够对SM4的加解密算法进行硬件加速。

在系统应用方面，本设计在外设系统上挂载了一个IPU，该IPU能够对摄像头中运动的物体进行坐标检测。在结合SM4扩展算子做系统级应用的时候，SM4扩展指令能够对IPU测得的物体坐标进行加密，并将加密信息传输到我们自研的上位机上，上位机能够利用秘钥对加密信息进行解密，并且能在上位机上重新模拟出运动物体的运动轨迹，符合我们一开始的设计预期。

## 6.2 展望

虽然本设计看起来像是一个比较完善的作品了，但我们的工作到目前为止仍有三个遗憾。

首先第一个遗憾就是没能来得及设计出FPU，这使得whetstone基准测试无法达到我们的理想结果。

第二个就是想在E203的外设总线或者NICE接口上挂载一个yolo神经网络，改神经网络能够对图像中的物体做出识别。最终可以把E203做成一种NPU芯片。

第三个是对上位机进行进一步优化，美化。使得上位机界面更美观，功能更加多元化。

# 参考文献

[1] 胡振波. 教你设计CPU——RISC-V 处理器. 人民邮电出版社. 2018.

[2] McFarling, Scott. “Combining Branch Predictors.” (1993).

[3] 姚永斌. 超标量处理器设计[M]. 清华大学出版社, 2014.

[4] Waterman A , Lee Y , Patterson D A , et al. The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA[J]. eecs department, 2011.

[5] <https://zhuanlan.zhihu.com/p/168096788>

[6] 吕述望, 李大为, 张超, 等. GM/T 0002—2012 SM4分组密码算法

[7] https://github.com/raymondrc/FPGA\_SM4

[8] 正点原子. 达芬奇Pro之FPGA开发指南 V1.3