SHANGHAI JIAO TONG UNIVERSITY

研究生课程项目报告

GRADUATE COURSE PROJECT REPORT

课程： SoC 设计

基于蜂鸟 SoC 的神经网络加速器设计

学生：胡起（121039910045）

刘增达（121039910051）

袁理轶（121039910070）

学院(系)： 电子信息与电气工程学院

开课学期： 2022 年 （春季）

目录

1 课程项目背景及设计目标

神经网络是今天重要的计算机应用方向，是人工智能技术的基础。包括人工神经网络、卷积神经网络、循环神经网络、生成对抗网络等均是基础神经网络结构的发展或者是神经网络拓扑结构的延展。在目标检测识别、分类、数据分析等众多领域有非常广泛的应用和发展前景。神经网络的运行需要大量的数据处理单元支持高效处理，CPU 的处理能力十分受限，如果仅依赖 CPU 进行神经网络的计算，将极大地影响整体系统的性能及能效。为此，目前提出神经网络加速器的设计，通过硬件加速的方式，提高神经网络的处理能力，实现高性能 SoC 系统。

本课程项目将实现 RepVGG 网络的单个 Block 块，RepVGG 基本架构如图 1，需要实现的单个 block 块配置如红框所示。本课程项目将在规定运算、存储资源的约束下对其进行特定的加速器硬件设计。

具体的运算、存储资源约束如下：

(1) 内部 memory 不能超过 256KB，sram 有 4k * 64 和 8k * 32 两种规格。

(2) 每个乘法器的输入为两个 8 比特有符号数，输出为 16 比特有符号数；不能用超过 320 个乘法器。

(3) 每个加法器的输入为两个 32 比特有符号数，输出为 32 比特有符号数。

在此基础上，将进一步把加速器挂载在蜂鸟 SoC 上，通过总线实现对加速器的控制，以及数据在内存与加速器之间的传输，完成对计算速度、数据复用的优化。最后，将完成加速器的 DC 综合以及 ICC 后端设计。综上所述，本课程项目目标可概括如下：

(1) 根据两个优化目标在加速器设计限定内进行了合理的加速器设计。

(2) 编写加速器源码、Golden Model 与 Testbench 验证加速器功能正确。

(3) 将加速器与蜂鸟 SoC 系统连接，编写 C 程序让 CPU 控制加速器运行。

(4) 后端设计及优化。

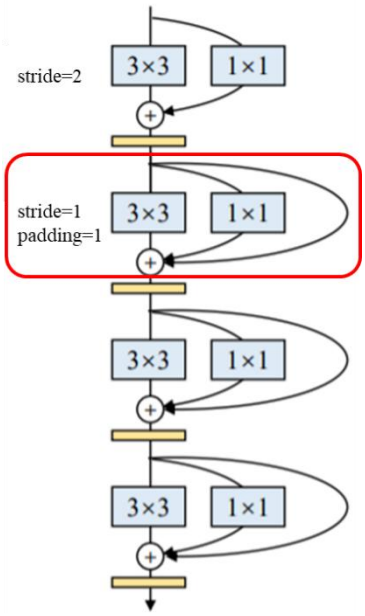


图 1 RepVGG[1]模型基本架构

2 加速器设计

2.1 加速器设计目标

本课程项目将针对 RepVGG 的第二个网络基本模块进行硬件加速设计。该模块的计算流程如图 2 所示。输入 56*56*64 的特征图，分别经过 3*3*64 的深度卷积得到 56*56*64 的特征图、经过 1*1*64 的深度卷积得到 56*56*64 的特征图，再将；之后将经过 3*3*64 卷积核卷积的特征图、经过 1*1*64 卷积核卷积的特征图、原特征图数据逐项加和，得到 56*56*64 的输出特征图。在本课程项目中，将针对上述计算过程进行针对性电路实现。与上述计算流程相对应，加速器设计时，将主要包含三个部分：（1）卷积核尺寸为 3*3 的卷积模块、（2）卷积核尺寸为 1*1 的卷积模块（3）ReLU 模块。

下面 2.2、2.3、2.4 小节将对每个部分进行详细介绍。

2.2 3*3 卷积模块设计

2.2.1 处理单元设计

处理单元（Process Element, PE）结构设计如图 2.2 所示。其主体由一个时钟上升沿出发的寄存器和多路选择器组成。PE 的输入信号包括控制信号(CLK、RST_N、ENA)以及输入数据 PE_i。输出信号分别是 PE_o 与 PE_2_AdderTree。其中 PE_o 是输入 PE_i 延时一个时钟周期的结果。PE_2_AdderTree 是 PE 输出参加乘累加运算的数据。值得注意的是，在本单元设计中，PE 输出参加乘累加的数据并不一定是 PE 中的真实数据，也可能是一个 8bit 的 0 数据，输出数据的具体内容将取决于 ENA 信号。该设计将被用于后续 padding 操作。利用将 ENA 信号拉成低电平的方法，可以在不影响数据流的前提下“忽略”该 PE，使不耽误周期的 padding 操作成为可能。

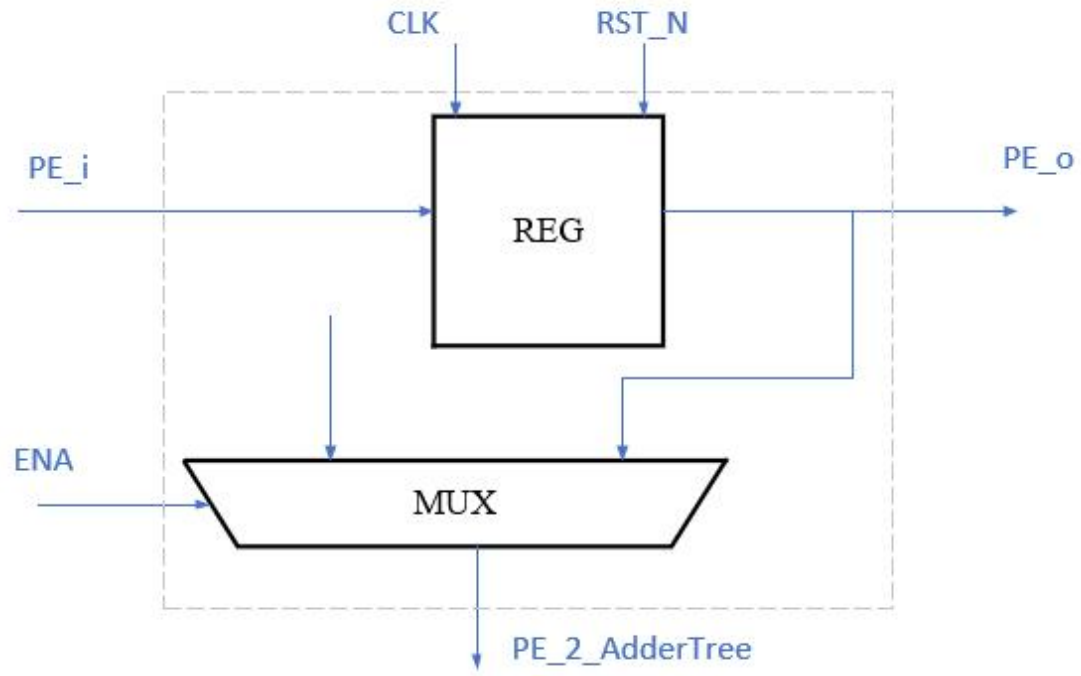


图 2.2 处理单元 PE 设计

2.2.2 滑动窗模块设计

滑动窗模块负责卷积的计算，由 3 部分组成：PE 阵列（包括 2 个 FIFO）、乘法器组（内含 1 个加法树）以及计数器组。其中 PE 阵列负责数据流的推动，乘法器组负责将 PE 阵列送来的数据计算出卷积结果，计数器组负责一些控制信号，如权重的刷新、PE 的开关、FIFO 的推动等。这部分的硬件结构设计如图 2.3

所示。接下来将分别介绍上述的 3 部分。

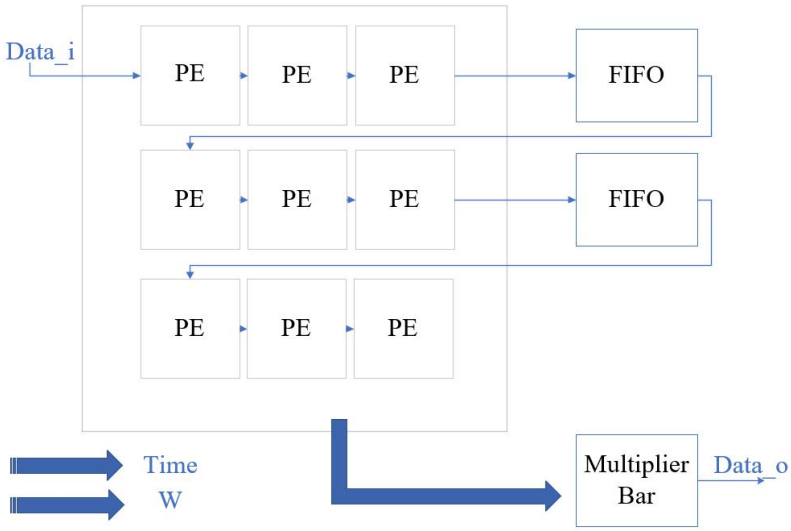


图 2.3 滑动窗（3×3 卷积）模块的硬件结构设计

在图 2.3 中，功能比较简单的是乘法器组部分，由 9 个乘法器和 1 个加法树组成。如图 2.4 所示，为了缩短关键路径，综合时成网表时提高工作效率。加法运算按照树形结构实现。

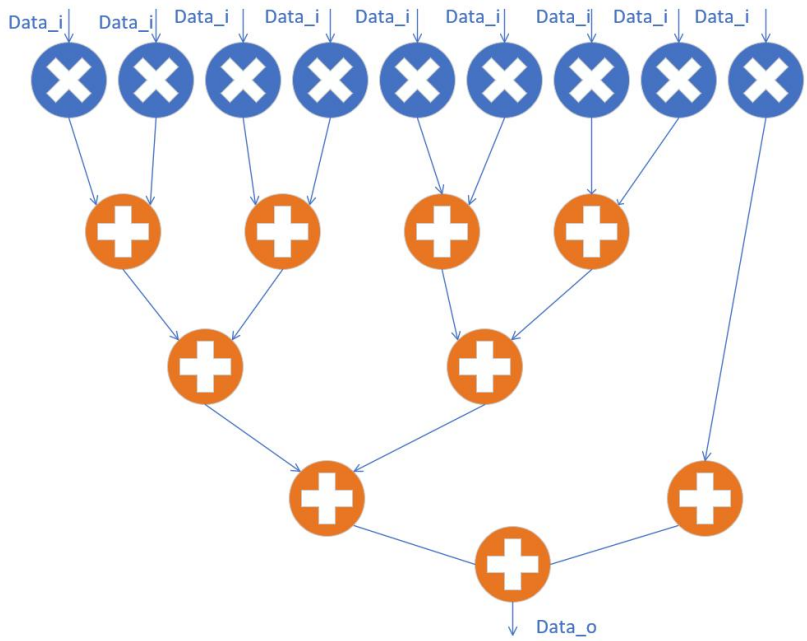


图 2.4 乘法器组的结构

在此，将以一个的特征图为例说明卷积运算以及计数器的功能，这里对利用 PE 的 ENA 信号完成 PADDING 操作。算法意义上的卷积过程如图 2.5 所示，一个 3×3 的窗口在特征图上滑动。

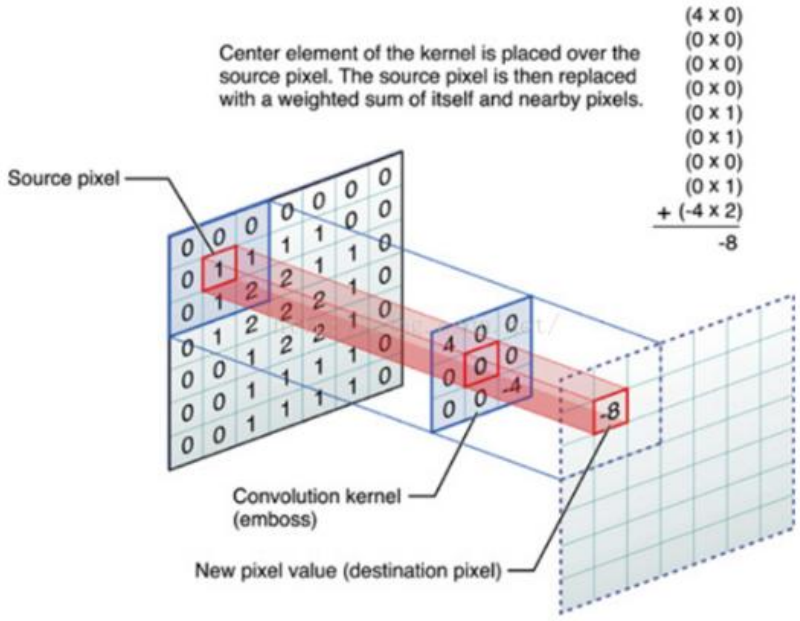


图 2.5 算法意义上的卷积过程

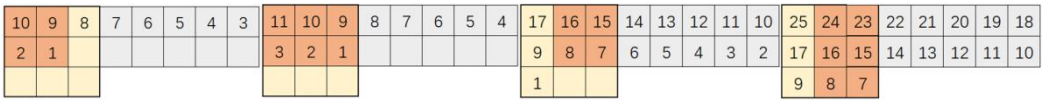


图 2.6 PE 阵列的卷积过程

如图 2.5 和图 2.6 所示，实际运算并不需要真的在图像周围插入一圈 0 而是只需要定期开关 PE 的 ENA 信号即可，而 PE 的 ENA 是由计数器组模块发出的。计数器组内有 3 个计数器，分别是初始化计数器、行计数器和列计数器。

2.3 ReLU 模块设计

利用一个 MUX 即可完成 ReLU 模块（Multiplexer，多路选择器）即可完成功能。根据输入数据的最高位可直接判断输入数据正负性，输入数据的最高位即是 MUX 的 select 信号。ReLU 模块的结构设计如图 2.7 所示。

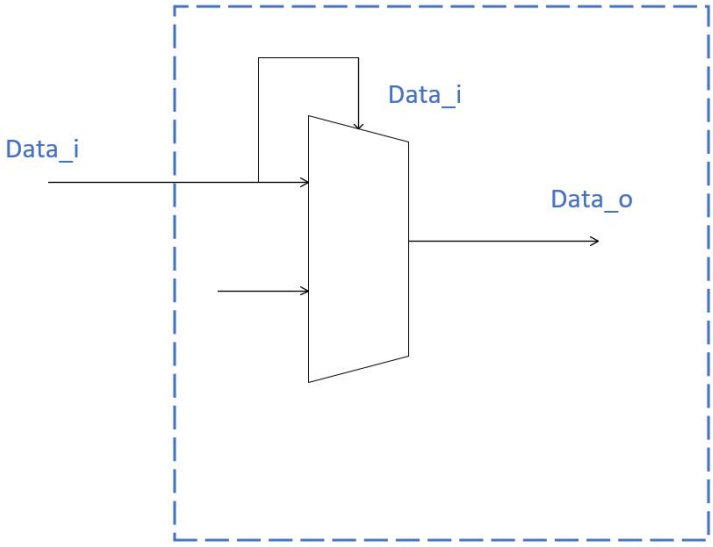


图 2.7 ReLU 模块结构设计

2.4 卷积模块设计(1 × 1)

1×1 卷积模块与 3×3 相比比较复杂，因为要做不同通道之间的累加操作。1×1 卷积模块的整体设计如图 2.8 所示。其中 Multiplier Group 是 1×1 卷积的乘法计算部分，Crossbar 则是同一通道的部分累加。为了减小数据传输量，通常采用输出通道并行的设计方法。因此 Crossbar 的输出等于输出通道的个数。

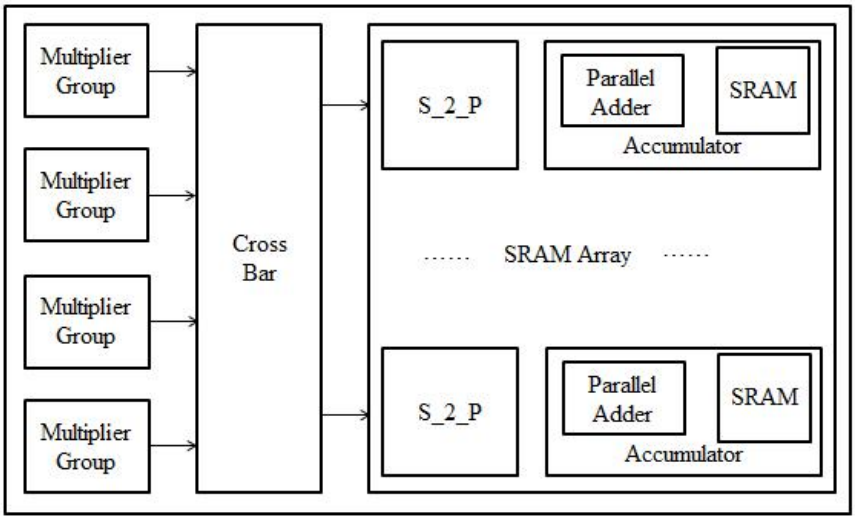


图 2.8 1×1 卷积模块的整体设计

1×1 卷积模块的难点在于如何实现累加。通常累加操作都是使用双端口 SRAM 来实现。双端口 SRAM 的优势在于可以同时进行读写，符合累加操作的特性。相比于双端口 SRAM，单端口 SRAM 来实现累加就复杂一些。我使用了单端口 32bit 端口宽度的 SRAM 来实现累加操作。

由于 SRAM 端口的宽度是 32bit，所以对于 8bit 数来说可以 4 并行。图 3.7 中的 S_2_P 模块就是串并转换（Sequential to Parallel）模块，其输入输出端口如图 2.9 所示。

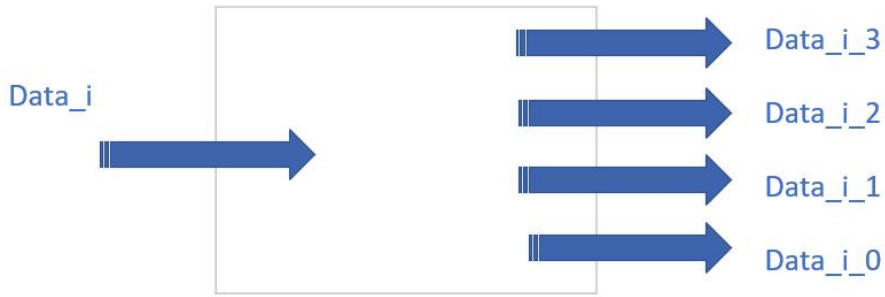


图 2. 9 串并转换 S_2_P 模块的端口

鉴于单端口 SRAM 不能同时读写的特性，SRAM 的时序需要仔细设计。Read 信号为地址的变化信号，在每组的第二个数据拉高。SRAM 的部分时序如图 2.10 所示。

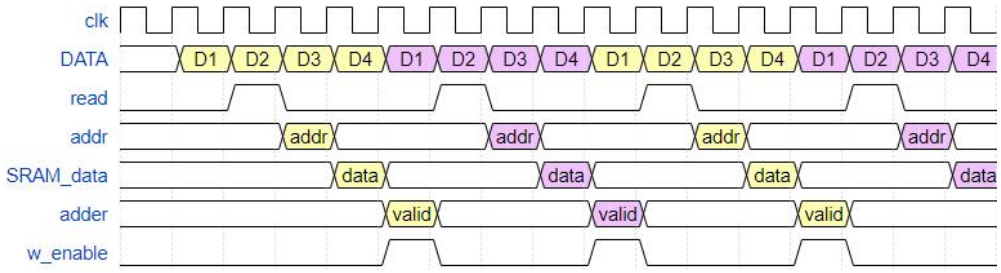


图 2.10 SRAM 的部分时序

如图 2.10 所示，令 read 信号在每组的第二个数据拉高，可以充分利用每一个周期：SRAM 的地址信号在 read 的下一个周期变化，SRAM 读取的数据在地

址的下一个周期得到。SRAM 的数据和来自 FIFO 的数据正好同时进入加法器，加法器的结果在下一个周期算出，同时将 SRAM 的写使能信号拉高。下一个周期 read 信号拉高的同时正好完成了 SRAM 的写操作，开始下一组数据的处理。通过 4 级 FIFO 的串并转换，完成了单端口 SRAM 的调度，充分利用了每一个周期，尽可能地缩短了关键路径。

SRAM 的设计如上段所述，利用图 2.10 所示的时序，设计了如图 2.11 所示的累加器硬件结构。Read 和 Valid 信号分别控制了 SRAM 的地址以满足读写要求。经过加法器计算的结果数据同时输出到最终数据和写回 SRAM。

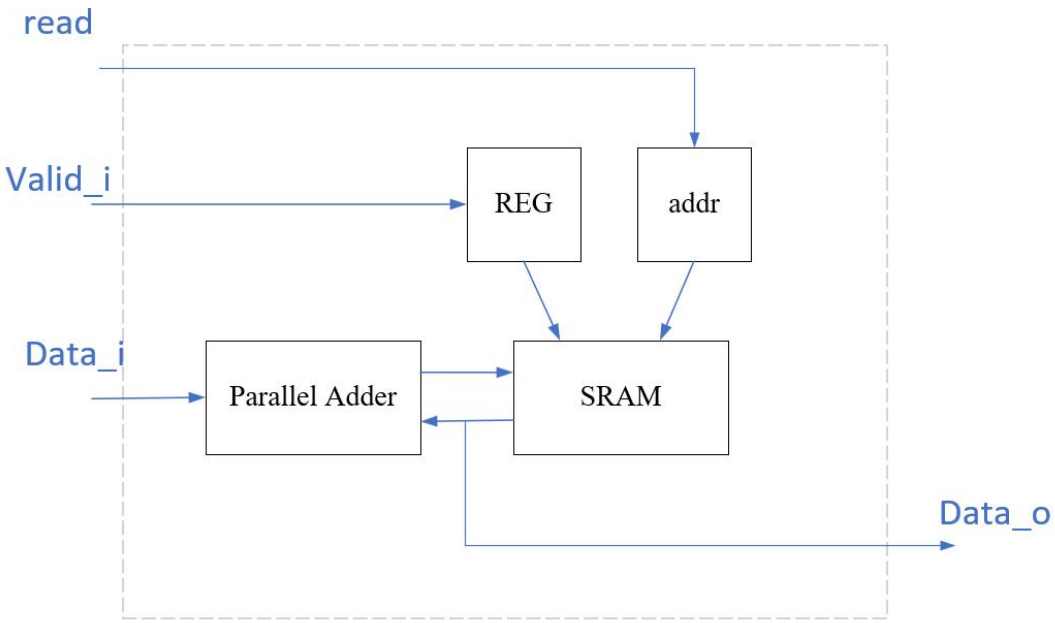


图 2.11 累加模块的结构设计

2.5 数据输出模块设计

在设计的过程中，考虑到总线带宽的限制，很有可能无法做到输入与输出同时进行，也就是说输入数据与输出数据要共用数据通道。因此，数据的输出时序需要单独设计。

由于在数据流的设计上采用输出通道并行的方式，SRAM 在设计时总共使用了 8 块（每块包含 2 个输出通道的内容）。在输出最终结果时，地址信号是广播

给 8 块 SRAM 的，8 块 SRAM 各自输出自己这一地址的数据。8 选 1 的操作是通过一个块通选信号配合一个 MUX 完成的。数据输出模块的结构示意图如图 2.12 所示。图 2.12 是一个概念图，并非实际的结构。实际的结构中 SRAM 是被包裹在累加器内部的，图 2.12 省略了累加器的外壳。在输出过程中，ADDR 信号是不断循环遍历的，多选器 MUX 的 8 个输入也在一遍一遍循环，利用块通选信号 BLOCK_SELECT 完成输出数据的选择，在 8 选 1 的选择操作完成以后，需要对 64 比特的数据拆分成 2 部分，依次输出这两部分数据。BLOCK_SELECT 信号初始化为 0，当 ADDR 完成一次遍历后，BLOCK_SELECT 自加，即选择了下一块 SRAM。

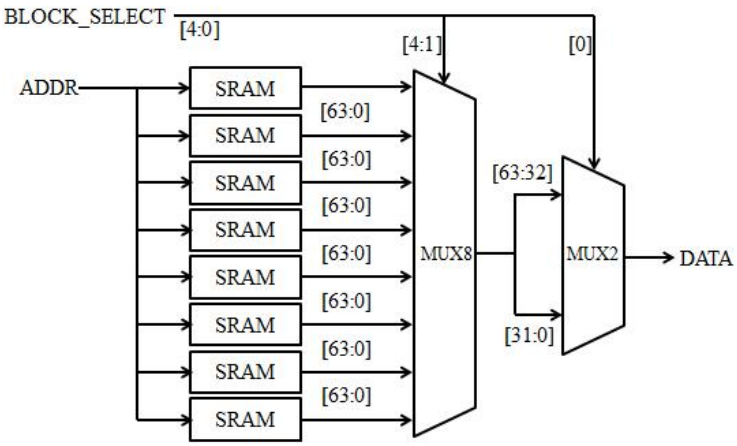


图 2.12

2.6 顶层模块设计

将 3×3 卷积模块做 4 并行后，将各子模块连接，得到 TOP 模块的结构设计图如图 2.13 所示。其中特征图数据与权重数据共用数据通路。

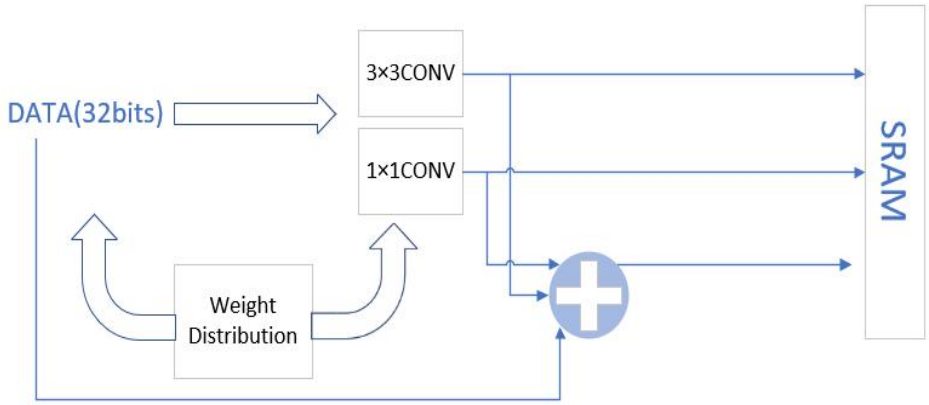


图 2.13 TOP 模块的结构设计

2.7 加速器验证

首先我们用 python 作为编译环境，采用 pytorch 实现输入数据的模拟，用 pytorch 生成 56*56*64 的数组以及 3*3 卷积核，1*1 的卷积核，存在 input.txt、kernel1*1.txt、kernel3*3.txt 中，用 python 完成卷积计算，将输出结果保存在 output.txt 中。本次试验中，将输入数据 input.txt 作为 testbench 的输入，然后验证输出。下面是 testbench 的思路。

C: > Users > huqii > Desktop > SoC课程项目_刘增达_胡起_袁理轶 > ACC设计 > goldenModel_python >

```python
import torch.nn as nn
import torch
import numpy as np
import torch.nn.functional as F

input = torch.randint(-10, 10, [1, 64, 56, 56])
print(input.shape)
print('input', input)
input1 = np.array(input)
input2 = input1.reshape(-1, 1)
np.savetxt('input.txt', input2)

kernel3x3 = torch.randint(-10, 10, [64, 64, 3, 3])
print('kernel3x3', kernel3x3)
a = np.array(kernel3x3)
b = a.reshape(-1, 1)
np.savetxt('kernel3x3.txt', b)

kernel1x1 = torch.randint(-10, 10, [64, 64, 1, 1])
print('kernel1x1', kernel1x1)
c = np.array(kernel1x1)
d = c.reshape(-1, 1)
np.savetxt('kernel1x1.txt', d)

branch3x3 = F.conv2d(input, kernel3x3, stride=1, padding=1)
branch1x1 = F.conv2d(input, kernel1x1, stride=1, padding=0)
output = input + branch3x3 + branch1x1

print(output.shape)
print('output', output)
e = np.array(output)
f = e.reshape(-1, 1)
np.savetxt('output.txt', f)
```

首先仿真权重数据配置功能，在加速器开始工作后，首先利用数据通道传输权重数据。如图 2.14 所示，观察信号 weight_ing 的波形以及信号 valid_i 的波形，可知权重数据的配置功能正确。

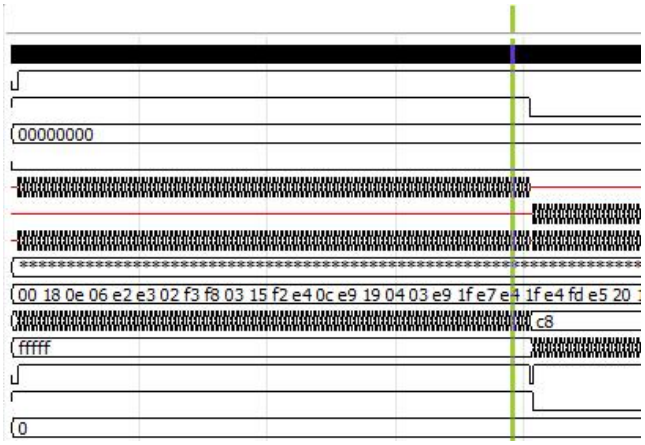
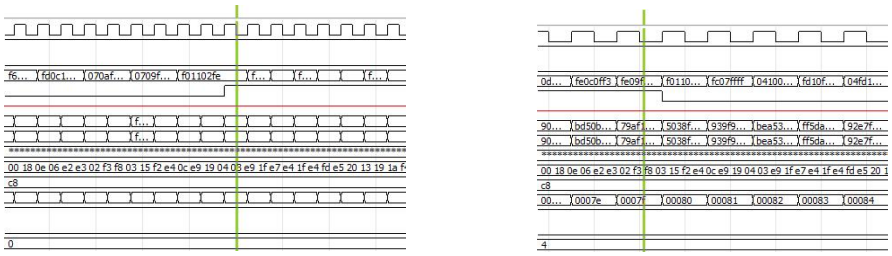


图 2.14 权重数据配置功能仿真

然后仿真加速器的计算过程，时序波形如图 2.15 所示。通过 testbench 将 TOP 的输出数据写入文本文档，结合上文中 python 构造一个 GOLDEN_MODEL，数据对比结果验证输出结果正确。同理，通过 testbench 将 TOP 的输出数据写入文本文档，再利用 System Verilog 构造一个 GOLDEN_MODEL。将 TOP 输出的文本文档与 GOLDEN_MODEL 的结果相对比，其计算功能正确。数据对比结果验证输出结果正确。



（a）　　（b）

图 2.15 仿真结果波形示意。（a）TOP 模块输出首个结果波形，（b）TOP 模块最后结果波形

3 SoC 系统设计

3.1 SoC 整体框架

本实验 SoC 整体框架如图 3.1 所示。其中，蜂鸟 E203 处理器有两个 ICB 接口，分别为私有外设 ICB 接口和存储 ICB 接口；本实验设计的加速器既作为从设备挂载在私有外设 ICB 总线，同时作为主设备通过存储 ICB 总线对 SRAM 进行读写操作。

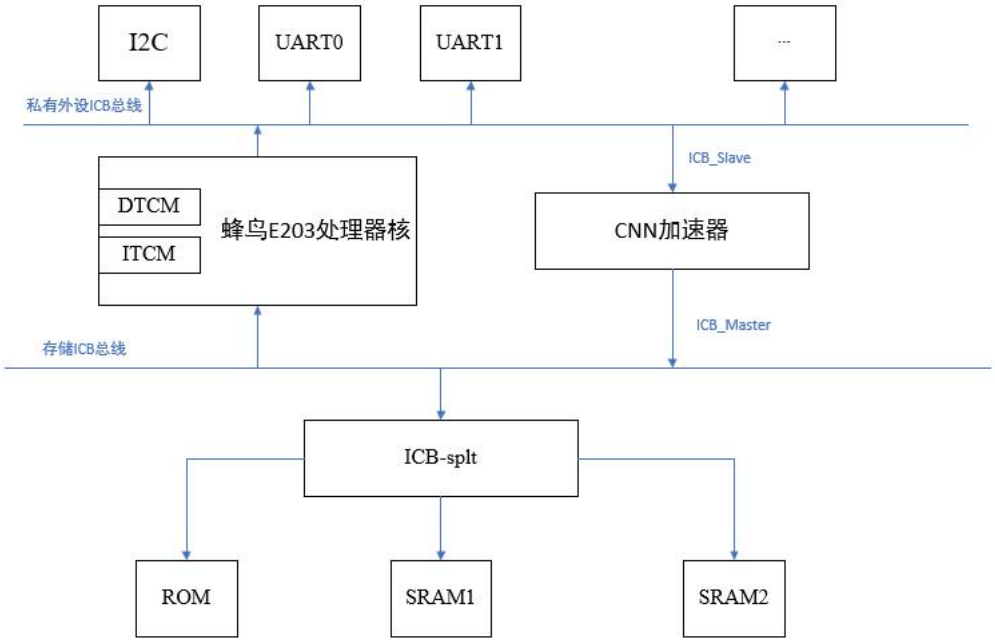


图 3.1 SoC 整体框架

具体地，各模块地址映射情况如表 3.1 所示。

表 3.1 模块地址映射情况

模块名称	起始地址	结束地址
I2C	32'h1004_2000	32'h1004_2FFF
UART0	32'h1001_3000	32'h1001_3FFF
UART1	32'h1002_3000	32'h1002_3FFF

ROM	32'h0000_1000	32'h0000_1FFF
SRAM1	32'h2000_0000	32'h2000_FFFF
SRAM2	32'h3000_0000	32'h3000_FFFF
ACC	32'h1004_1000	32'h1004_1FFF

*I2C、UART0/1 虽然挂载在私有外设总线上，但实际并未使用。

3.2 SoC 软件部分

3.2.1 CPU 配置加速器相关的寄存器

根据表 1 可知，本实验加速器的基地址为 32'h1004_1000，所以将 platform.h 中 SPI0 的控制地址改为 32'h1004_1000 即可借助 SPI0_REG 函数向相应的寄存器写入数据。

```
#define SPI0_CTRL_ADDR          _AC(0x10041000,UL)
#define SPI0_REG(offset)        _REG32(SPI0_CTRL_ADDR, offset)
```

图 3.2

直接对 demo_i2c.c 中的 main 函数进行修改，向 32'h1004_1008 中写入 1 作为启动信号，加速器即可不断地检测相应的寄存器来判断是否启动。

```
SPI0_REG(0x08) = 1; //start signal
```

图 3.3

本实验中，加速器输入特征图和权重的存放地址为 32'h2000_0000，输出特征图的存放地址同样为 32'h2000_0000，类似地 CPU 可配置相应的寄存器。

3.3 SoC 硬件部分

3.3.1 向 SRAM 预存输入特征图及权重数据

实际实验过程中使用 readmem 函数向 SRAM 预存数据。由于输入特征图的数据量较大，本实验使用了 SRAM1 和 SRAM2，直接通过 readmemb，将特征图的数据直接在 initial 模块读入，然后在 e203_subsys_mems.v 进行例化即可。

```
initial begin
$readmemb("/home/riscv/soc/demo_e200/rtl/e203/subsys/data_sram1.txt",mem_r);
```

图 3.4

值得注意的是，SRAM 模型为蜂鸟自带的 Bob Hu 在蜂鸟 E203 中的模型，且传入 SRAM 的地址应为偏移地址，否则不能正确地存入数据。

3.3.2 加速器 ICB_Slave

加速器作为从设备，ICB 接口定义如表 2 所示，代码详见 icb_ctrl_sla.v。

表 2　ICB_Slave 接口定义

信号名	方向	宽度
i_icb_cmd_valid	input	1
i_icb_cmd_ready	output	1
i_icb_cmd_addr	input	32
i_icb_cmd_read	input	1
i_icb_cmd_wdata	input	32
i_icb_cmd_wmask	input	4
i_icb_rsp_valid	output	1
i_icb_rsp_ready	input	1

i_icb_rsp_rdata	output	32
i_icb_rsp_err	output	1

加速器作为从设备时只有写操作，且为下一周期返回结果。具体地，①主设备（CPU）发送 cmd_valid 信号，从设备同时拉高 cmd_ready 信号，此时命令通道握手成功并且写地址、写数据有效。②从设备在下一个周期发送 rsp_valid 信号，在接收到来自主设备的 rsp_ready 信号后返回通道握手成功，表明写操作完成。

在本实验中，当命令通道握手成功并且写地址为 32'h1004_1008，写数据为 1 时加速器将开始工作，实际波形图详见 3.6 节。

3.3.3 加速器 ICB_Master

加速器作为主设备，ICB 接口定义如表 3 所示，代码详见 icb_ctrl_mas.v。

表 3 ICB_Slave 接口定义

信号名	方向	宽度
m_icb_cmd_valid	output	1
m_icb_cmd_ready	input	1
m_icb_cmd_addr	output	32
m_icb_cmd_read	output	1
m_icb_cmd_wdata	output	32
m_icb_cmd_wmask	output	4
m_icb_rsp_valid	input	1
m_icb_rsp_ready	output	1
m_icb_rsp_rdata	input	32

m_icb_rsp_err	input	1

加速器作为主设备时既存在读操作又存在写操作。加速器作为主设备开始工作后 cmd_valid 信号拉高，当接收到从设备（SRAM）发送的 cmd_ready 信号时命令通道握手成功。

（1）读操作。①实验过程中，在加速器运算完成之前 cmd_read 信号始终为高，所以命令通道握手成功的同时读地址有效。②rsp_ready 信号始终置高，当接收到从设备发送的 rsp_valid 信号时读出数据至 rsp_rdata。③实际上读操作经过 3 个周期才能返回结果，如此依次读出存放于 SRAM 中的全部数据并送到加速器进行运算，实际波形图详见 3.6 节。

（2）写操作。①当加速器计算完成后将会把数据写回从设备（SRAM）中，此时 cmd_read 信号始终为低，所以命令通道握手成功的同时写地址、写数据有效。②类似地，此时写操作同一周期返回结果。实际波形图详见 3.6 节。

3.4 挂载加速器

上文详细介绍了本实验 SoC 的软件部分和硬件部分。下面介绍如何将加速器挂载到私有外设总线和存储总线。

3.4.1 挂载到私有外设总线

将包含加速器和 ICB 主从接口的模块在 e203_subsys_perips.v 中进行例化，其中 Slave 接口与私有外设总线进行连接，挂载在 O4 端口上，Master 接口则需从该模块引出。

```
// * Here is an I2C WishBone Peripheral
.O1_BASE_ADDR       (32'h1004_2000),
.O1_BASE_REGION_LSB (3),// I2C only have 3 bits address width
//  * UART0     : 0x1001 3000 -- 0x1001 3FFF
.O2_BASE_ADDR       (32'h1001_3000),
.O2_BASE_REGION_LSB (12),
//  * UART1     : 0x1002 3000 -- 0x1002 3FFF
.O3_BASE_ADDR       (32'h1002_3000),
.O3_BASE_REGION_LSB (12),
//  acc
.O4_BASE_ADDR       (32'h1004_1000),
.O4_BASE_REGION_LSB (12),
```

```
//  acc
  .o4_icb_enable     (1'b1),

  .o4_icb_cmd_valid  (i_icb_cmd_valid),
  .o4_icb_cmd_ready  (i_icb_cmd_ready),
  .o4_icb_cmd_addr   (i_icb_cmd_addr),
  .o4_icb_cmd_read   (i_icb_cmd_read),
  .o4_icb_cmd_wdata  (i_icb_cmd_wdata),
  .o4_icb_cmd_wmask  (i_icb_cmd_wmask),
  .o4_icb_cmd_lock   (),
  .o4_icb_cmd_excl   (),
  .o4_icb_cmd_size   (),
  .o4_icb_cmd_burst  (),
  .o4_icb_cmd_beat   (),

  .o4_icb_rsp_valid  (i_icb_rsp_valid),
  .o4_icb_rsp_ready  (i_icb_rsp_ready),
  .o4_icb_rsp_err    (i_icb_rsp_err),
  .o4_icb_rsp_excl_ok(1'b0   ),
  .o4_icb_rsp_rdata  (i_icb_rsp_rdata),
```

图 3.5

3.4.2 挂载到存储总线

加速器作为主设备挂载到存储总线时直接借用了 DMA 的 Master 接口（便于实现仲裁）。具体地，在 e203_subsys_top.v 中对 e203_subsys_mems 模块进行例化时直接将加速器的 Master 接口连接至 DMA 的相应接口处即可。

```
e203_subsys_mems u_e203_subsys_mems(

    .mem_icb_cmd_valid  (mem_icb_cmd_valid),
    .mem_icb_cmd_ready  (mem_icb_cmd_ready),
    .mem_icb_cmd_addr   (mem_icb_cmd_addr ),
    .mem_icb_cmd_read   (mem_icb_cmd_read ),
    .mem_icb_cmd_wdata  (mem_icb_cmd_wdata),
    .mem_icb_cmd_wmask  (mem_icb_cmd_wmask),

    .mem_icb_rsp_valid  (mem_icb_rsp_valid),
    .mem_icb_rsp_ready  (mem_icb_rsp_ready),
    .mem_icb_rsp_err    (mem_icb_rsp_err  ),
    .mem_icb_rsp_rdata  (mem_icb_rsp_rdata),

    .dma_icb_cmd_valid  (m_icb_cmd_valid),
    .dma_icb_cmd_ready  (m_icb_cmd_ready),
    .dma_icb_cmd_addr   (m_icb_cmd_addr ),
    .dma_icb_cmd_read   (m_icb_cmd_read ),
    .dma_icb_cmd_wdata  (m_icb_cmd_wdata),
    .dma_icb_cmd_wmask  (m_icb_cmd_wmask),

    .dma_icb_rsp_valid  (m_icb_rsp_valid),
    .dma_icb_rsp_ready  (m_icb_rsp_ready),
    .dma_icb_rsp_err    (m_icb_rsp_err  ),
    .dma_icb_rsp_rdata  (m_icb_rsp_rdata),



    .clk            (hfclk  ),
    .bus_rst_n      (main_rst_n),
    .rst_n          (main_rst_n)
);
```

图 3.6

3.5 搭建 testbench

在 tb_top.v 中，将生成的.verilog 文件通过 readmemh 函数读取，并存放在 ITCM 中。初始化后，ROM 中的上电程序执行完成后会跳转到 ITCM 的地址（32'h8000_0000），随后开始执行 ITCM 里面的指令。

如前文所述，当加速器的所有计算结果写回 SRAM 后会产生中断信号，所

以该测试平台不断地检测中断信号（conv_irq）是否为高，如果检测到中断信号为高电平会得到下图所示的结果，这样初步说明输出特征图的数据量是正确的。

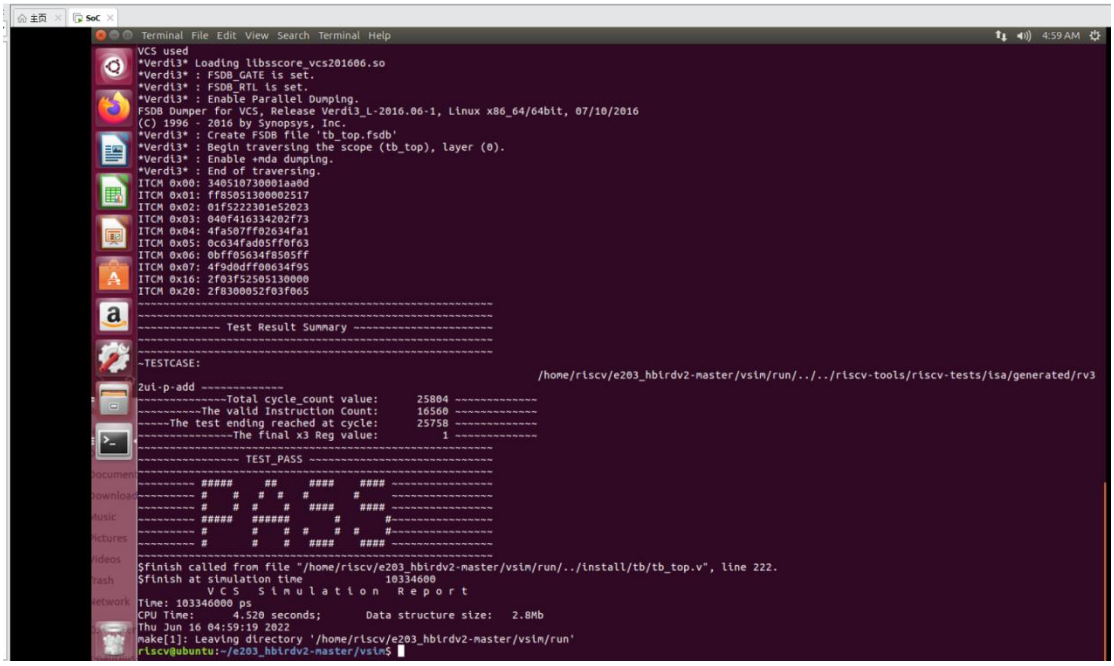


图 3.7　testbench 输出结果

为进一步说明所有写回的计算结果都是正确的，在中断产生后通过 writemem 函数将 SRAM 中的相应结果写入 output_data_ref.txt 文件中，并通过 compare.py 与 GOLDEN_MODEL 的输出结果进行比对。

```python
import sys

with open("./output_data_ref.txt", 'r') as f1:
    ref = []
    for line1 in f1.readlines():
        ref.append(line1.strip())

with open("./output_data_icb.txt", 'r') as f2:
    real = []
    for line2 in f2.readlines():
        real.append(line2.strip())

p = 0
f = 0

if(len(ref)!=len(real)):
    print("*********The number of output is wrong!!!********")
else:
    for i in range(50176):
        if(ref[i]==real[i]):
            p = p + 1
        else:
            f = f + 1
            print("the %s result is wrong" %(i+1))
    if(p==50176):
        print("--------The results are all correct!!!--------")
    else:
        print("********There are %d wrong results********" %f)
```

图 3.8 结果比对代码

由于虚拟机中没有安装 python，将得到的 output_data_icb.txt 和 output_data_ref.txt 导出后在本机上运行，得到结果正确。

```
C:\Users\87490\Desktop\opt_codes_0919\venv\Scripts\python.exe C:/Users/87490/Desktop/opt_codes_0919/compare.py
--------The results are all correct!!!--------

Process finished with exit code 0
```

图 3.9 结果比对代码运行结果

3.6 SoC 整体工作流程（波形结果）

将结果导出在 fsdb/ tb_top.fsdb 中。

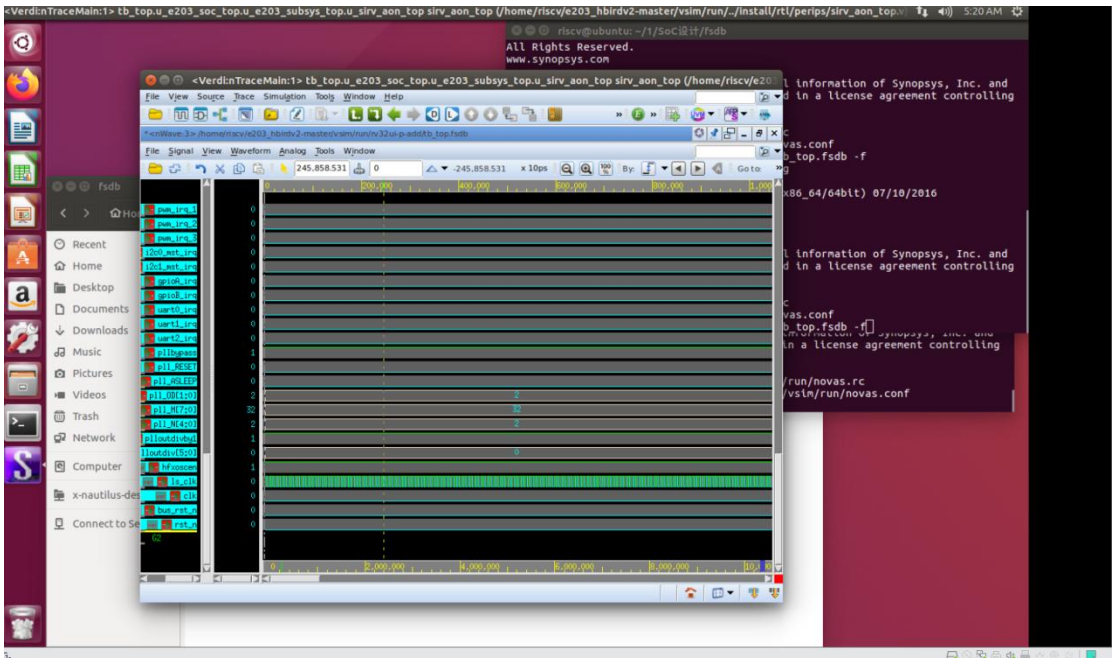

All Rights Reserved.
www.synopsys.com
Recent
Home
Desktop
Documents
Downloads
Music
Pictures
Videos
Trash
Network
Computer

图 3.11 波形示例

4 综合结果及分析

使用课程提供的 Linux 环境下的 Design Compiler 软件对加速器设计进行综合，并根据结果进行优化设计。

需要注意的是，我们组对原有的/home/riscv/library/db 中的 SRAM 模型进行了替换，替换为了助教发的模型。

在项目中，主要针对综合后的时钟频率进行优化。首先以 100MHz 频率进行综合后，时序报告显示时钟裕量仍然充足。经过优化测试后，最终将综合频率设定为 100MHz。以 100MHz 频率进行综合的时序报告如图 4.1 所示。观察报告中的关键路径分析可知，在加速器中 SRAM 的读数据路径是关键路径，由于 SRAM 是使用提供的 IP，因此难以进行进一步优化，除非重新设计整个架构，采用不使用片内 SRAM 的新设计。

```
clock clk (rise edge)                                    2.50       2.50
clock network delay (ideal)                              0.00       2.50
clock uncertainty                                       -0.10       2.40
top_acce_0/conv_2/multiplierBar_0/multiplier8bit_8/out16bit_reg[14]/CLK (DFFARX1_RVT)
                                                         0.00       2.40 r
library setup time                                      -0.06       2.34
data required time                                                  2.34
---------------------------------------------------------------------------
data required time                                                  2.34
data arrival time                                                  -2.34
---------------------------------------------------------------------------
slack (MET)                                                         0.00
```

图 4.1

```
Startpoint: i_icb_cmd_valid
            (input port clocked by clk)
Endpoint: i_icb_cmd_ready
          (output port clocked by clk)
Path Group: in2out
Path Type: max

Des/Clust/Port     Wire Load Model       Library
------------------------------------------------
top                16000                 saed32rvt_ss0p95v125c

Point                                                       Incr       Path
-----------------------------------------------------------------------------
clock clk (rise edge)                                       0.00       0.00
clock network delay (ideal)                                 0.00       0.00
input external delay                                        0.20       0.20 f
i_icb_cmd_valid (in)                                        0.00       0.20 f
icb_ctrl_sla_0/sla_icb_cmd_valid (icb_ctrl_sla)             0.00       0.20 f
icb_ctrl_sla_0/sla_icb_cmd_ready (icb_ctrl_sla)             0.00       0.20 f
i_icb_cmd_ready (out)                                       0.00       0.20 f
data arrival time                                                      0.20

clock clk (rise edge)                                       2.50       2.50
clock network delay (ideal)                                 0.00       2.50
clock uncertainty                                          -0.10       2.40
output external delay                                      -0.20       2.20
data required time                                                     2.20
-----------------------------------------------------------------------------
data required time                                                     2.20
data arrival time                                                     -0.20
-----------------------------------------------------------------------------
slack (MET)                                                            2.00


 data arrival time                                                     2.00

 clock clk (rise edge)                                      2.50       2.50
 clock network delay (ideal)                                0.00       2.50
 clock uncertainty                                         -0.10       2.40
 top_acce_0/conv_4/multiplierBar_0/multiplier8bit_1/out16bit_reg[14]/CLK (DFFARX1_RVT)
                                                            0.00       2.40 r
 library setup time                                        -0.06       2.34
 data required time                                                    2.34
 ----------------------------------------------------------------------------
 data required time                                                    2.34
 data arrival time                                                    -2.00
 ----------------------------------------------------------------------------
 slack (MET)                                                           0.34


 data arrival time                                                     2.11

 clock clk (rise edge)                                      2.50       2.50
 clock network delay (ideal)                                0.00       2.50
 clock uncertainty                                         -0.10       2.40
 output external delay                                     -0.20       2.20
 data required time                                                    2.20
 ----------------------------------------------------------------------------
 data required time                                                    2.20
 data arrival time                                                    -2.11
 ----------------------------------------------------------------------------
 slack (MET)                                                           0.09
```

图 4.2 时序综合报告(100MHz)

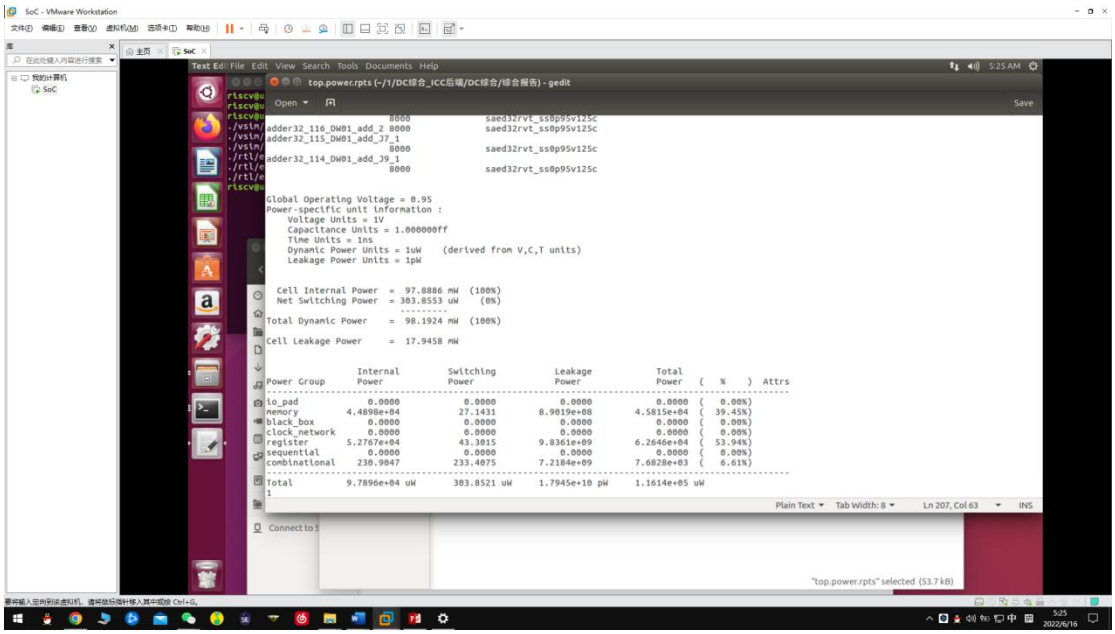


图 4.3 功耗综合报告(100MHz)

以 100MHz 的频率进行综合的功耗报告如图 4.3 所示。其中动态功耗为 98.1924mW，静态功耗为 17.9458mW。

综合的面积报告如图 4.4 所示。观察面积报告可知，最消耗面积的部分是 Macro/Black box area，也就是 SRAM 的面积。在大多数设计中，存储确实是面积的瓶颈，想要优化面积应该针对存储的使用进行优化，对数据流需要重新设计。本加速器使用了 100 个乘法器和 256kB 的 SRAM。

```
Number of ports:                          215
Number of nets:                           247
Number of cells:                            3
Number of combinational cells:              0
Number of sequential cells:                 0
Number of macros/black boxes:               0
Number of buf/inv:                          0
Number of references:                       3

Combinational area:             194751.055739
Buf/Inv area:                    15580.298215
Noncombinational area:          144395.982352
Macro/Black Box area:           799990.187500
Net Interconnect area:          103758.995582

Total cell area:               1139137.225591
Total area:                    1242896.221173
```

图 4.4 综合后的面积报告

由系统综合后的功耗报告和面积报告可以发现，存储器是整个系统的痛点所在，SRAM 既占用面积，还很占用功耗，但是有助于性能的提升。

5 后端物理设计及分析

在完成加速器综合的基础上进一步进行加速器的后端物理设计。基于课程提供的 IC Compiler（ICC）工具进行后端物理设计，物理设计流程如图 5.1 所示。

需要注意的是，我们组对原有的/home/riscv/library/db 中的 SRAM 模型进行了替换，替换为了助教发的模型。

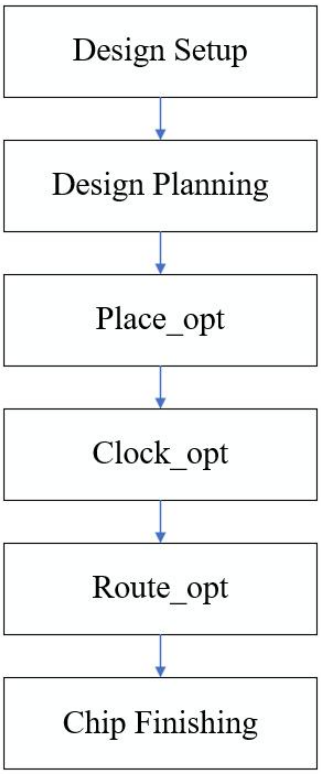


图 5.1 物理设计流程图

整体设计过程中，关键步骤截图如图 5.2 所示，整个设计流程进行中，逻辑门的排布以及芯片版图设计越来越完善，直至最后生成 GDS 文件。

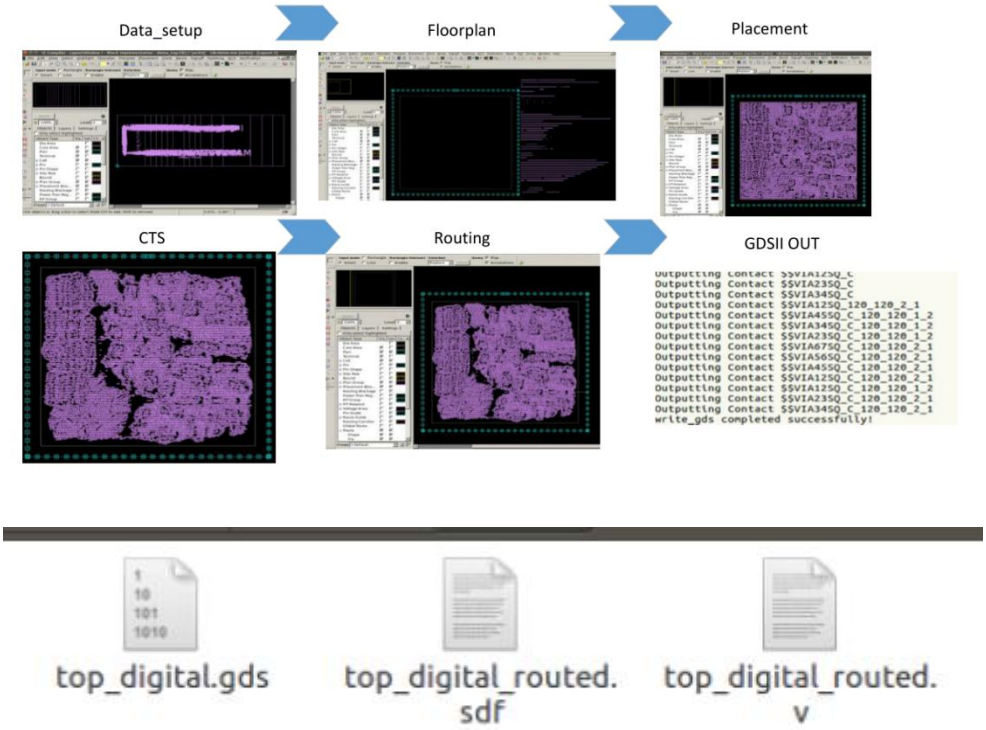


图 5.2 物理关键步骤示意图

最终成功生成了该三个文件。

6 总结

本次课程项目设计的硬件加速器实现了 RepVGG 网络的单个 Block 块，所有硬件设计过程均满足运算、存储资源约束，该加速器具有 100 MHz 的频率，根据综合结果，本项目设计的的加速器具有较高能效，相比起同类型加速器具有的优势，该优势和本项目仅针对卷积计算设计有很大关系。本实验项目用 Golden Model 与 Testbench 验证加速器功能正确。

成功将加速器挂载在蜂鸟 SoC 系统中，并成功利用 ICB 总线实现了 CPU 与加速器、加速器与内存之间的通信。经测试，相比起纯 CPU 计算，利用硬件加速可以实现加速器的加速。在此基础上，完成了加速器的综合、后端物理设计及优化成功生成 GDS 文件。

通过本次课程项目，我们对 IC 设计的前后端设计，从电路设计到系统设计，体验到了 SoC 的整体设计流程和设计细节，学习了基础的软件使用方式，相信学习到的知识对我们未来的科研工作都将起到重要作用

7 小组成员分工

刘增达：加速器电路设计

胡　起：Golden Model

袁理轶：综合、后端物理设计

参考文献

[1] Sandler M, Howard A, Zhu M, et al. Inverted Residuals and Linear Bottlenecks: Mobile Networks for Classification, Detection and Segmentation[J]. 2018.

[2] 蜂鸟 E203 系列 Core & SoC 原型快速使用说明.