

除了 UVM, 还有 VMM, OVM。VMM 的英文全称是 Verification Methodology Manual, 由 Synopsys 提出, 集成了寄存器解决方案, 但是它一开始是闭源的。后来, 由 Cadence 和 Mentor 联合发布了 OVM, 它从一开始就是开源的, 并且引入了工厂模式, 但是它一开始并没有寄存器解决方案, 后来才添加, 但是 UVM 从一开始就具有 OVM 和 VMM 的各种优势, 并将它们融合的很好。现在 UVM 得到了 Synopsys、Cadence 和 Mentor 的共同支持, 拥有庞大的用户群, 逐渐成为了验证方法学的主流, 被各大半导体公司采用。

UVM 可以快速搭建验证平台, 灵活的编写测试用例, 而它自身提供的基础类库 (basic class library) 和基本验证结构可以让具有不同程度软件编程经验的数字芯片验证工程师们能够快速构建起一个结构良好可信的验证框架。UVM 自定义的框架构建类和测试类能够帮助数字芯片验证工程师减轻环境构建的负担, 进而将更多的精力集中在如何制定验证计划和创建测试场景当中去^{[2]258}。

2.2 CNN 加速器的介绍

2.2.1 CNN 算法简介

CNN 算法全称是 convolutional neural network, 即卷积神经网络。它主要包括输入层、卷积层、非线性处理、池化层和全连接层, 其中卷积层是实现 CNN 功能的核心层。卷积层是用一个固定长宽高的长方体卷积核在每一层的输入层上面移动做乘加。这个卷积核以一定的“步伐”长度移动, 在每个位置上, 卷积核上的每个元素和输入层上的对应位置的元素相乘并累加, 得到一个输出层的元素, 当卷积核在整个输入上遍历以后, 就得到了完整的输出。通常, 卷积核的正面是正方形的, 具体可分为 3x3、5x5、7x7 和 11x11 四种大小的卷积核; 卷积核移动的“步伐”多数是 1 或者 2 个数字的长度。综上, 可以看出, 一个 CNN 加速器的主要运算是乘和累加。

图 2-1 演示的 CNN 算法中 kernel 是 3x3 的, 每个 kernel 有 3 个 channel, 总共有 4 个 kernel。kernel 可以看作是一个长方体的数据集, 所谓的 3x3 是指这个长方体的长和高都是 3 个数据的长度, 所谓的 channel 是指这个长方体的宽包含多少个数据。input feature 的情况与此类似。上图中的 input feature 是 6x6 的。每个 kernel 和 input feature 对应位置的数据, 即相同颜色的数据, 做乘法, 最后累加得到一个 output feature, 然后 kernel 在 input feature 上面移动一个数据的长度继续上述的运算, 最终得到 4 个 channel 的 output feature, 把 4 个 channel 的 output feature 合在一起就是下一个卷积运算的 input feature。在卷积运算中 input feature 的 channel 和 kernel 的 channel 数目要保持相同, output feature 的 channel 和 kernel 的数目要保持相同。



手机淘宝扫一扫

CNN 的算法图示如下：

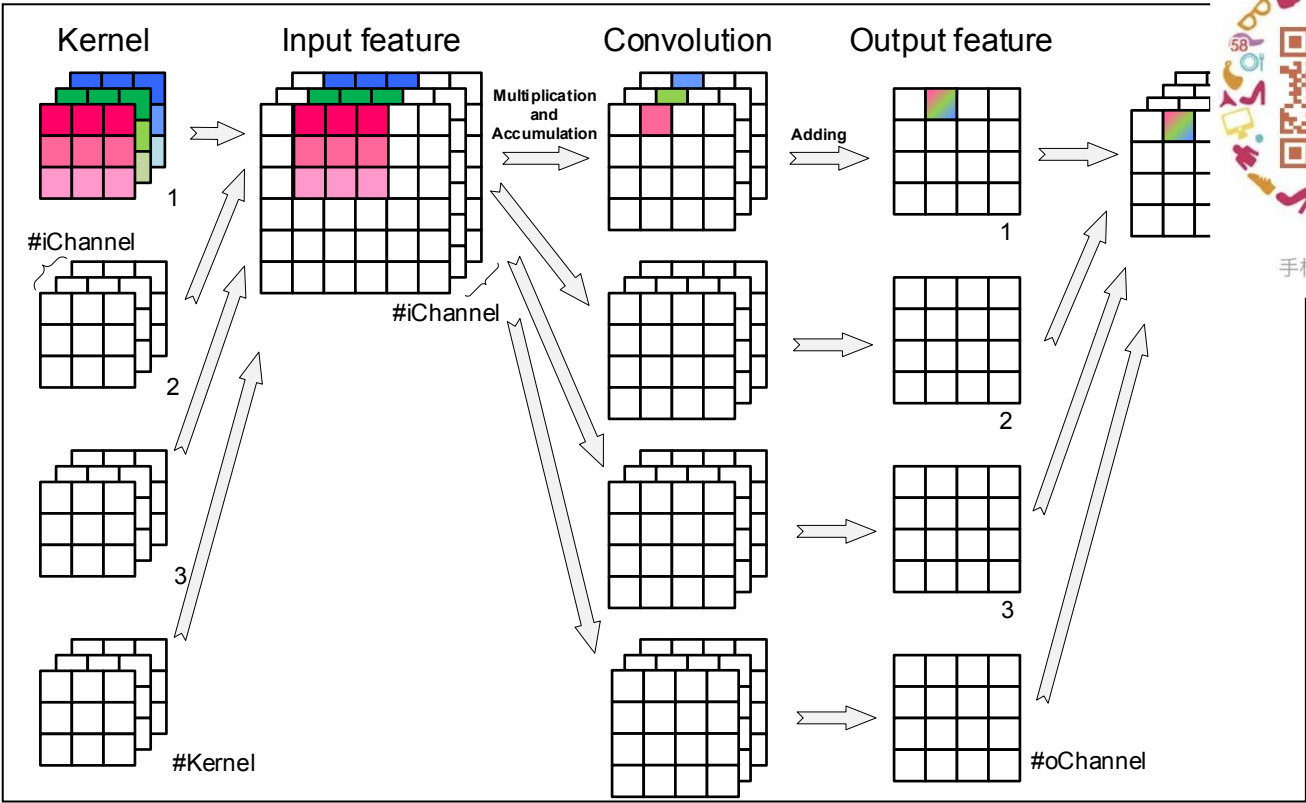


图 2-1：CNN 算法

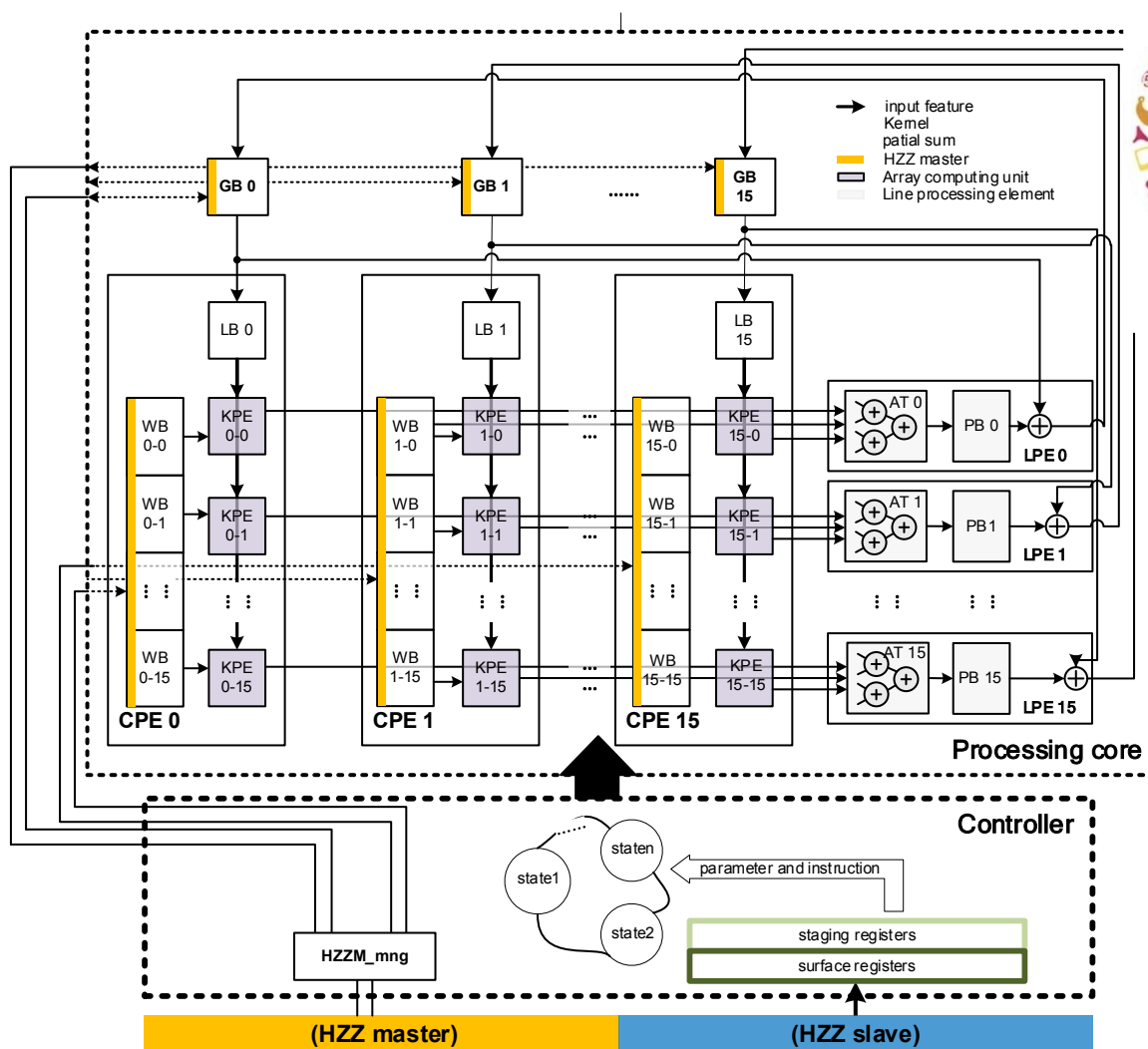
2.2.2 CNN 加速器简介

本论文所验证的加速器来自于实验室的项目，名字是 tanji-3。它可以支持卷积运算、池化运算、全连接运算、RNN 和多种非线性运算。

在本论文所涉及的试验中，我验证了 CNN 加速器中的卷积运算部分，包括数据传输、状态机控制和数据运算。

整个加速器从整体上可分为三部分：第一部分是数据计算部分，由乘法器、加法器和累加器等构成；第二部分是存储器部分，包括片上、片下存储器，片下存储器由 DRAM 构成，在实际的硬件中是 DDR3，片上的存储器都是 SRAM，其中，存放输入、输出层的存储器有两层，第一层是 global buffer，第二层是 local buffer，global buffer 和 SOC 中的其它部分、片外 DDR3 和运算器交换数据，local buffer 存储从 global buffer 中读出的数据，然后再把数据送入卷积运算模块。存放 kernel 数据的存储器只有一层，叫做 weight buffer，它一边直接与 DDR3 交换数据，另一边把 kernel 中的数据传入卷积运算模块。weight buffer 有两组，形成 ping-pong，提高运算器的利用率；第三部分是控制器。这一部分可以更细致的分为两个更小的部分，一部分是控制运

淘宝唯一店铺：数字IC资料店



手机淘宝扫一扫

图 2-2: tanji-3 加速器中有关 CNN 运算的架构简图

算器运算的真正的控制硬件，另一部分是寄存器堆，用来记录系统的配置，从而决定硬件的运行。控制硬件完全都是由状态机实现的，没有指令。局部的控制逻辑中会有微指令。

整个加速器要与 SOC 和 DDR 交换数据，所以特地为此设计了 HZZM-HZZS 接口。HZZM 是主接口，HZZS 是从接口。SOC 端有一个 HZZM 接口，用来和加速器中的 HZZS 接口相匹配，把 SOC 对寄存器的配置参数和启动命令以及 SOC 中其他部分的数据写入加速器中，SOC 也可以通过这个接口读出寄存器的参数和加速器的运行状态。加速器中也有一个 HZZM 接口，它用来和 DDR 中的 HZZS 接口匹配，实现从 DDR 中读数据和写数据的功能。

整个架构的运算部分由 16 个 CPE (channel processing element) 单元构成和 16 个 LPE (line processing element) 单元构成，每个 CPE 单元都包含

淘宝唯一店铺：数字IC资料店

16 个 KPE (kernel processing element), 一个 local buffer, 两组 weight buffer, 每组里面又有 16 个 weight buffer。每一组的 weight buffer 和 KPE 一一对应, local buffer 中的数据被 16 个 KPE 单元共享。每个 CPE 对应一个 global buffer, 总共有 16 个 global buffer, global buffer 中的数据会先写入 CPE 中的 local buffer, 如果 CPE 中的 KPE 工作, 则从 local buffer 中读取数据, 从 weight buffer 中读取权重值, 然后一起送入 KPE 进行运算。KPE 中的运算结果会传入 LPE 单元中。总共有 16 个 LPE 单元, 每个 LPE 单元与 16 个 KPE 单元连接, 可以同时接收 16 个 KPE 的数据, LPE 把接收来的数据相加, 最后再存入 global buffer 中。大致架构如图 2-2。

控制硬件主要控制卷积运算、把数据从 DDR3 读入 global buffer、把数据从 DDR3 读入 weight buffer、把数据从 global buffer 写入 local buffer 和 LPE 的运行。

加速器中的寄存器堆分为两层, 第一层寄存器可以直接由 SOC 中的 CPU 通过 HZSM 接口更改存储值, 叫做 surface registers。第二层寄存器在控制信号的作用下可以把第一层的值复制过来, 在加速器整个运行期间都保持不变, 叫做 staging registers。这样做的好处是: 第二层寄存器相对第一层来说值比较固定, 可以在加速器运行时为控制部分和计算部分的硬件提供相应的参数, 比较稳定可靠, 而第一层由 SOC 更改值比较容易、方便, 而且可以在加速器运行时进行更改, 这样就可以提高加速器的运行效率和利用率, 在加速器运行完毕后, 可以立即将第一层的配置参数写入第二层寄存器, 使加速器不用闲置很久就可以接着进行工作。

加速器由 CPU 控制。每次开始运行, CPU 都会先检查是否已经把相应的参数通过总线写入加速器中, 如果已经写入就立马开始运行加速器, 并且同时把下次加速器运行需要的参数也开始写入加速器的第一层的寄存器中; 如果没有在加速器中写入相应参数, 则先通过总线向加速器写入相应的参数, 写完以后, 再开始启动加速器。每次 CPU 写完相应的参数都要判断加速器是否正在计算, 如果没有, 则立即启动加速器, 如果正在运行, 则使 CPU 等待加速器, 直到加速器完成本次的计算任务再启动。每次加速器完成计算任务后都会发出中断, 以此来通知 CPU。

在本次的验证工作中我首先对简单的 3 个模块进行详细的验证, 然后才对整体进行了几个经典情况的验证。这 3 个模块分别是:

1. dla_kpe
2. dla_lpe
3. dla_regif

淘宝唯一店铺: 数字ic资料店



手机淘宝扫一扫



dla 是 deep learning accelerator 的英文首字母缩写，kpe 是 kernel processing element 的英文首字母缩写，lpe 是 line processing element 的英文首字母缩写，regif 是 register interface 的缩写。

首先介绍 kpe 的功能。kpe 内部主要是采用流水线的乘法和累加运算，它用来完成 CNN 算法中的卷积核中的权重和对应位置上的输入元素的相乘和累加，所以，它主要有两种输入数据：第一种是来自 local buffer 的输入元素，第二种是来自 weight buffer 的权重。它的数据输出就是相乘和累加的结果。kpe 的使能信号 enable 如果为 1，则 kpe 正常运行；enable 的值为 0，则 kpe 的输出 kpe_sum 恒为 0。ctrl_kpe_bypass 如果为 0，则 kpe_sum 的输出为正常的运算结果，如果它的值为 1，则 kpe_sum 的输出直接为 kpe_ifmap 的值。

表 2-1 kpe 的信号列表：

信号	功能	transaction 中的数据 结构
clk	时钟	
rst	异步复位	
enable	模块全局使能	variable(一个固定变量)
kpe_ifmap	ifmap 输入	static_array[18]
kpe_weight	权重输入	static_array[18]
kpe_sum	运算结果输出	static_array[18]
ctrl_kpe_src0_enable	ifmap 输入使能	static_array[18]
ctrl_kpe_src1_enable	权重输入使能	static_array[18]
ctrl_kpe_mul_enable	乘法使能	static_array[18]
ctrl_kpe_acc_enable	累加使能	static_array[18]
ctrl_kpe_acc_rst	累加复位	static_array[18]
ctrl_kpe_bypass	kpe 旁路选择	variable
stgr_precision_kpe_shift	kpe 的移位个数	variable
stgr_precision_ifmap	ifmap 的精度	variable
stgr_precision_weight	权重的精度	variable

ifmap 是 input feature map 的缩写。
transaction 中的数据结构一栏请参阅 3.1.1 节。

淘宝唯一店铺：数字ic资料店

lpe 接受来自 16 个不同 CPE 的 kpe 运算后的数据，它再完成累加、缓存等任务，最后将运算的结果再送入 global buffer。它的输入主要就是 16 个 kpe 的输出数据，除此之外，还有可能是从 global buffer 中读出的数据，因为从 global buffer 中读出的数据也会累加入最终的运算结果。它最终的运算结果会写入 global buffer 中指定地址的位置。它里面也引入了流水线，相应的输入信号中也有流水线的控制信号。lpe 除了累加、缓存的功能外，还实现了 ReLU 函数的功能。ReLU 函数的功能是将输入数据的小于 0 的值都替换为 0，大于等于 0 的值保持不变输出。lpe 里面有一个能放缓存数据的 FIFO，它能够起到缓存的作用。lpe 里面有一个加法树，它的作用就是实现上面的累加功能。lpe 有 3 种工作模式：

1. 16 个 kpe 的数据全部相加，最后运算得一个和。这种模式称为 ALL 模式。
2. 16 个 kpe 的数据每 4 个分为一组进行相加，最后运算得到 4 个和。这种模式称为 IMAGE 模式。
3. 16 个 kpe 的数据不进行相加，直接存放在 lpe 中的 FIFO 里面。这种模式称为 NONE 模式。

表 2-2: lpe 的信号列表

信号	功能	transaction 中的数据 结构
clk	时钟	
rst	异步复位	
stgr_pool_enable	池化使能	variable(一个固定变量)
kpe_sum	kpe 的输出，也是 lpe 的输入	variable
pool_ofmap	池化的输出	variable
lpe_gb_rdata	lpe 从 global buffer 读取的数据	variable
lpe_gb_wdata	lpe 往 global buffer 写的数据	static_array[19]
adt_fifo_set	FIFO 的置位信号	static_array[19]
adt_fifo_ren	FIFO 的读使能	static_array[19]
adt_fifo_empty	FIFO 的空标志	static_array[19]



手机淘宝扫一扫

ctrl_adt_enable	加法树输入数据的使能	static_array[19]
ctrl_lpe_relu	控制累加后的结果是否经过 ReLU 函数处理	variable
ctrl_lpe_acc_bypass	控制是否把运算后的结果和从 global buffer 中读取的数据相加	variable
ctrl_lpe_sprmps	控制 lpe 的模式。	variable
stgr_precision_ifmap	ifmap 的精度	variable

ifmap 是 input feature map 的缩写。

transaction 中的数据结构一栏请参阅 3.1.2 节

regif 模块内部主要就是上述的寄存器堆中的第一层和第二层。该模块在下述列表中的输入、输出信号是中规中矩的，除了下述列表的信号外，剩余的输入输出信号是与加速器内部的其他模块有关，主要用来为硬件提供配置参数和运行参数。这一部分剩余信号中，输出信号主要是第二层寄存器，即 staging register 的输出信号，输入信号主要是来自加速器内部的计算完成时的通知信号。

表 2-3: regif 的部分信号列表

信号	功能	transaction 中的数据 结构
clk	时钟	
rst	异步复位	
regif_addr	寄存器接口的地址	variable(一个固定变量)
regif_wdata	往寄存器中写的数据	variable
regif_wen	写使能信号	variable
regif_rdata	从寄存器中读取的数据	variable
regif_ren	读使能	variable
regif_rvalid	读有效信号，表示读出的数据有效	variable



手机淘宝扫一扫

淘宝唯一店铺：数字ic资料店

第3章 测试点分解和验证环境的搭建

3.1 模块级的测试点分解和验证平台搭建

3.1.1 kpe 模块的测试点分解和验证平台

kpe 是卷积运算的核心模块，它实现了卷积运算的主要部分，即乘加运算。由于 kpe 内部的乘加运算是由流水线实现的，所以在对 kpe 模块施加激励时需要按照一定的时序要求给 kpe 连续输入一定个数的数据。在搭建验证平台的时候为了实现连续给 kpe 输入数据，我在 transaction 中采用了定宽数组的数据结构，并且对于一些使能信号数组，我会根据时序图把每个值都约束为固定的值。根据设计中的时序图，我把定宽数组的大小确定为 18。

表 3-1: kpe 模块分解的测试点

测试点序号	测试点	测试方法
1	使能信号为 0 时，kpe 输出是否为 0。	在用`uvm_do_with 宏发送 transaction 时，约束 enable 信号为 0。其余信号都是随机。
2	ifmap 和 weight 的精度是 16bit 时，kpe_shift 的取值为 0、1、2、3、4、5、6、7、8、9、10、11、12。	在 sequence 中，用`uvm_do_with 宏发送 transaction，enable 信号约束为 1，ctrl_kpe_bypass 信号约束为 0，stgr_precision_ifmap 和 stgr_precision_weight 约束为 16bit 精度，stgr_precision_kpe_shift 用一个 rand 类型变量约束，以便在 virtual sequence 更改 stgr_precision_kpe_shift 的值。其他静态数组的信号按照时序的要求约束，例如，ctrl_kpe_src0_enable 数组元素 0-13 为 1，元素 14-17 为 0。
3	ifmap 和 weight 的精度是 8bit 时，kpe_shift 的取值为 0、1、2、3、4、5、6、7、8。	与测试点 2 的约束方法类似，只不过把 stgr_precision_ifmap 和 stgr_precision_weight 的值约束为 8bit 精度。
4	ctrl_kpe_bypass 信号为 0 时，kpe 的输出	首先约束 enable 信号为 1，然后再约束 ctrl_kpe_bypass 信号为 0。与上面 2 和 3 情



手机淘宝扫一扫

出是否与输入数据 kpe_ifmap 一样。	况不同的是，上面 kpe 运行 18 个时钟周期才会有一个 kpe_sum，在当前这种模式下每个时钟周期都会有一个数据。
------------------------	--

kpe 模块验证平台中规中矩，主要的结构组件是：environment、input_agent、output_agent、reference model、scoreboard 和连接各个组件的 FIFO。其中，reference model 中引入了 C 语言编写的参考模型。它的原型如下：

```
void kpe_algorithm(
    svLogicVecVal * kpe_sum,
    const svLogicVecVal kpe_ifmap[14],
    const svLogicVecVal kpe_weight[14],
    const svLogicVecVal* stgr_precision_kpe_shift,
    const svLogic stgr_precision_ifmap,
    const svLogicVecVal* stgr_precision_weight
)
```

图 3-1: kpe 模块参考模型中的 C 函数头

在 UVM 搭建的验证平台中利用如下语句即可将 C 模型引入验证平台中：

```
import "DPI-C" function void kpe_algorithm(output logic [15 : 0] kpe_sum,
    input logic [15 : 0] kpe_ifmap[14],
    input logic [15 : 0] kpe_weight[14],
    input logic[3 : 0] stgr_precision_kpe_shift,
    input logic stgr_precision_ifmap,
    input logic[1 : 0] stgr_precision_weight);
```

图 3-2: 与 C 函数头对应的 SystemVerilog 函数头

C 模型产生的标准结果和待测硬件产生的输出会在 kpe_scoreboard 中做最终的比较，如果没有错误，kpe_scoreboard 会产生 compare successfully! 的提示，如下图：

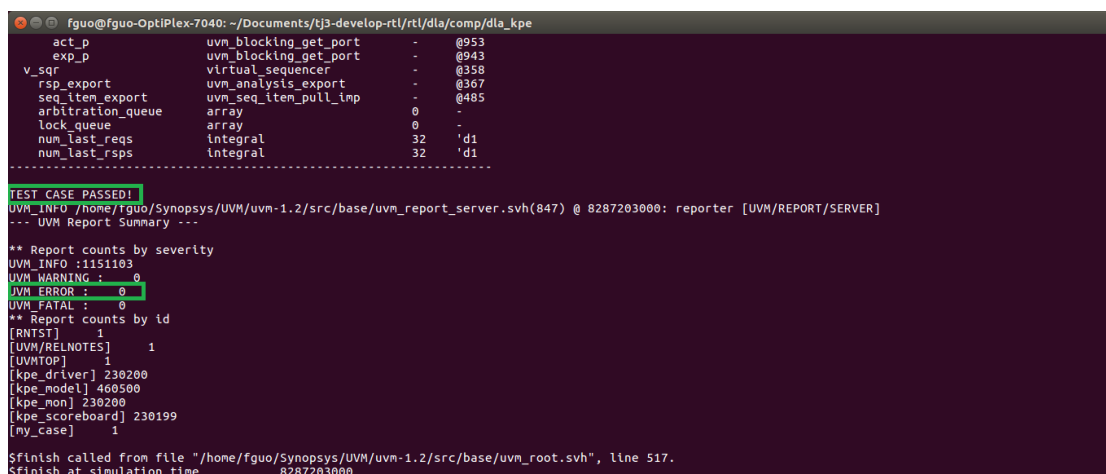
```
fguo@fguo-OptiPlex-7040: ~/Documents/tj3-develop-rtl/rtl/dla/comp/dla_kpe
UVM INFO kpe_driver.sv(53) @ 8287059000: uvm_test_top.env.i.agt.drv [kpe_driver] enter the driver main_phase!
UVM INFO kpe_mon.sv(44) @ 8287061000: uvm_test_top.env.o.agt.mon [kpe_mon] begin to collect a packet!!!
UVM INFO kpe_scoreboard.sv(58) @ 8287061000: uvm_test_top.env.scb [kpe_scoreboard] compare successfully!
UVM INFO kpe_model.sv(46) @ 8287095000: uvm_test_top.env.mdl [kpe_model] kpe_model have got one transaction from driver!!!
UVM INFO kpe_model.sv(54) @ 8287095000: uvm_test_top.env.mdl [kpe_model] ctrl_kpe_bypass is 1 in item!
UVM INFO kpe_model.sv(73) @ 8287095000: uvm_test_top.env.mdl [kpe_model] kpe_model have sent one transaction to scoreboard!!!
UVM INFO kpe_driver.sv(53) @ 8287095000: uvm_test_top.env.i.agt.drv [kpe_driver] enter the driver main_phase!
UVM INFO kpe_mon.sv(44) @ 8287097000: uvm_test_top.env.o.agt.mon [kpe_mon] begin to collect a packet!!!
UVM INFO kpe_scoreboard.sv(58) @ 8287097000: uvm_test_top.env.scb [kpe_scoreboard] compare successfully!
UVM INFO kpe_model.sv(46) @ 8287131000: uvm_test_top.env.mdl [kpe_model] kpe_model have got one transaction from driver!!!
UVM INFO kpe_model.sv(54) @ 8287131000: uvm_test_top.env.mdl [kpe_model] ctrl_kpe_bypass is 1 in item!
UVM INFO kpe_model.sv(73) @ 8287131000: uvm_test_top.env.mdl [kpe_model] kpe_model have sent one transaction to scoreboard!!!
UVM INFO kpe_driver.sv(53) @ 8287131000: uvm_test_top.env.i.agt.drv [kpe_driver] enter the driver main_phase!
UVM INFO kpe_mon.sv(44) @ 8287133000: uvm_test_top.env.o.agt.mon [kpe_mon] begin to collect a packet!!!
UVM INFO kpe_scoreboard.sv(58) @ 8287133000: uvm_test_top.env.scb [kpe_scoreboard] compare successfully!
UVM INFO kpe_model.sv(46) @ 8287167000: uvm_test_top.env.mdl [kpe_model] kpe_model have got one transaction from driver!!!
UVM INFO kpe_model.sv(54) @ 8287167000: uvm_test_top.env.mdl [kpe_model] ctrl_kpe_bypass is 1 in item!
UVM INFO kpe_model.sv(73) @ 8287167000: uvm_test_top.env.mdl [kpe_model] kpe_model have sent one transaction to scoreboard!!!
UVM INFO kpe_driver.sv(53) @ 8287167000: uvm_test_top.env.i.agt.drv [kpe_driver] enter the driver main_phase!
UVM INFO kpe_mon.sv(44) @ 8287169000: uvm_test_top.env.o.agt.mon [kpe_mon] begin to collect a packet!!!
UVM INFO kpe_scoreboard.sv(58) @ 8287169000: uvm_test_top.env.scb [kpe_scoreboard] compare successfully!
UVM INFO kpe_model.sv(46) @ 8287203000: uvm_test_top.env.mdl [kpe_model] kpe_model have got one transaction from driver!!!
UVM INFO kpe_model.sv(54) @ 8287203000: uvm_test_top.env.mdl [kpe_model] ctrl_kpe_bypass is 1 in item!
UVM INFO kpe_model.sv(73) @ 8287203000: uvm_test_top.env.mdl [kpe_model] kpe_model have sent one transaction to scoreboard!!!
UVM INFO /home/fguo/Synopsys/UVM/uvm-1.2/src/base/uvm_root.svh(579) @ 8287203000: reporter [UVMTOP] UVM testbench topology:
Name Type Size Value
```

图 3-3: C 模型和待测硬件输出的比较结果



手机淘宝扫一扫

整个验证平台的消息也会对他们的比较结果有一定的反映。一个是我在验证平台中设置了只要出现 error 类的信息就会显示 TEST CASE FAILED，如果 error 类信息一个也没有，则会显示 TEST CASE PASSED。error 类信息的产生有许多原因。当 C 模型和待测硬件输出的比较结果不吻合时，kpe_scoreboard 会产生 error 类信息，验证平台的其他组件部分也有可能产生 error 类信息。整个验证平台的消息汇总如下图：



```
fguo@fguo-OptiPlex-7040: ~/Documents/tj3-develop-rtl/rtl/dla/comp/dla_kpe
act_p      uvm_blocking_get_port  - @953
exp_p      uvm_blocking_get_port  - @943
v_sqr      virtual_sequencer    - @358
rsp_export uvm_analysis_export    - @367
seq_item_export uvm_seq_item_pull_imp - @485
arbitration_queue array 0 -
lock_queue array 0 -
num_last_reqs integral 32 'd1
num_last_rsps integral 32 'd1
-----
TEST CASE PASSED!
UVM INFO: /home/fguo/Synopsys/UVM/uvn-1.2/src/base/uvn_report_server.svh(847) @ 8287203000: reporter [UVM/REPORT/SERVER]
--- UVM Report Summary ---

** Report counts by severity
UVM INFO: 1151103
UVM WARNING: 0
UVM ERROR: 0
UVM FATAL: 0
** Report counts by id
[QNTST] 1
[UVM/RELNOTES] 1
[UVMTOP] 1
[kpe_driver] 230200
[kpe_model] 460500
[kpe_mon] 230200
[kpe_scoreboard] 230199
[my_case] 1

$finish called from file "/home/fguo/Synopsys/UVM/uvn-1.2/src/base/uvn_root.svh", line 517.
$finish at simulation time 8287203000
```

图 3-4：整个仿真的消息统计

从图中不难看出，TEST CASE PASSED!字样。在打印信息的最后会将各类信息出现的次数进行统计、打印，从图中可以看出 UVM_ERROR 类信息的最后统计次数是 0。这个结果也印证了前面的分析。

如图 3-5，该验证平台的最顶层是 uvm_top（图中未画出），它是 uvm_root 类型的对象，它是整个验证平台中所有组件构成的树结构的根节点。它下面有唯一的一个子节点 uvm_test_top，它是 my_case 类的一个实例化的对象，my_case 类派生自 base_test 类，base_test 类派生自 uvm_test 类。我在 base_test 类中实例化了 kpe_env 类和 virtual_sequencer 类，kpe_env 类派生自 uvm_env 类，virtual_sequencer 类派生自 uvm_sequencer 类，其中 kpe_env 的实例化对象主要用来充当容器，在它里面可以实例化 in_agt 类、out_agt 类、kpe_model 类和 kpe_scoreboard 类。这些在 kpe_env 中实例化的类都是验证平台中的组件。in_agt 和 out_agt 都派生自 uvm_agent 类。in_agt 中又实例化了 kpe_driver、kpe_sequencer 两种类。kpe_driver 派生自 uvm_driver 类，kpe_sequencer 派生自 uvm_sequencer 类。out_agt 类中实例化了 kpe_mon 类，它派生自 uvm_monitor 类。

为了在验证平台中实现各个组件间的通信，首先要在各个组件中定义端口。kpe_driver 中定义了 uvm_analysis_port#(uvm_sequence_item) 类型的端口（图



手机淘宝扫一扫

中 kpe_driver 左侧的小矩形)。这种端口在发起通信时，不用等待其他端口的回应，即该端口是非阻塞的。它与 kpe_env 中定义的 uvm_tlm_analysis_fifo 上的 uvm_analysis_imp 类型端口相连接。uvm_tlm_analysis_fifo 起到了一个中间缓存的功能。然后，uvm_tlm_analysis_fifo 上的 blocking_get_export 类型的端口再与 kpe_model 上的 uvm_blocking_get_port 类型的端口相连接，这样就形成了一个完整的通信通道。同样的在 kpe_model、kpe_mon 组件中也定义了 uvm_analysis_port#(uvm_sequence_item) 类型的端口，它们与 kpe_scoreboard 的连接与上面的连接方式相同。定义好端口以后各组件之间的数据就可以互相流通。

淘宝唯一店铺：数字ic资料店

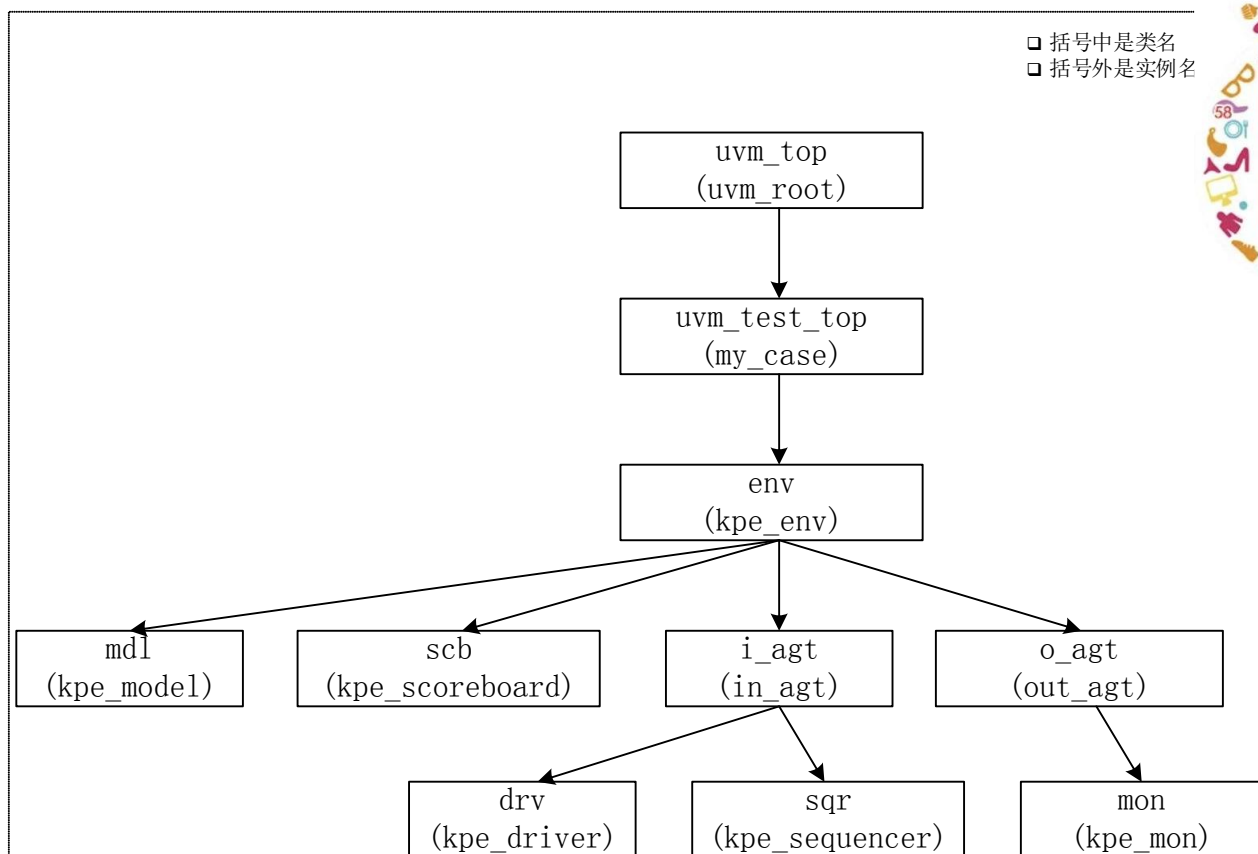


图 3-6: kpe 模块验证平台中各组件构成的树

由于 kpe_mon 要收集 kpe 运行的结果，而 kpe 的运行要靠 kpe_driver 来驱动，所以，kpe_driver 在把数据传输进 kpe 后，要通知 kpe_mon 对 kpe 的结果进行收集。也就是说，它们两个组件之间要有通信来配合完成工作。在 kpe 的验证平台中，我引入了 uvm_event。它可以在全局中共享，而不用那么多繁琐的通信连接。首先，在 kpe_driver 中，定义 uvm_event 类型的变量，然后通过 `uvm_event_pool::get_global()` 将它与全局资源池中的实例化对象对应起来，这样就可以在需要的位置调用 `event_variable_name.trigger()` 函数来触发。在 kpe_mon 一侧，同样要完成定义和与资源池实例化对象的对应，然后，就可以在需要的位置调用 `event_variable_name.wait_trigger()`，kpe_mon 中该调用语句以后的语句要一直等到 kpe_driver 中的事件被触发才可以执行。

kpe 验证平台中覆盖率收集是在 kpe_driver 中，它是利用回调机制实现的。首先，我定义了一个回调基类，在这个基类中定义了两个虚方法：

```
virtual task pre_tx(ref virtual ini_if ii = null);
virtual task post_tx(ref virtual ini_if ii = null);
```

淘宝唯一店铺：数字ic资料店



手机淘宝扫一扫

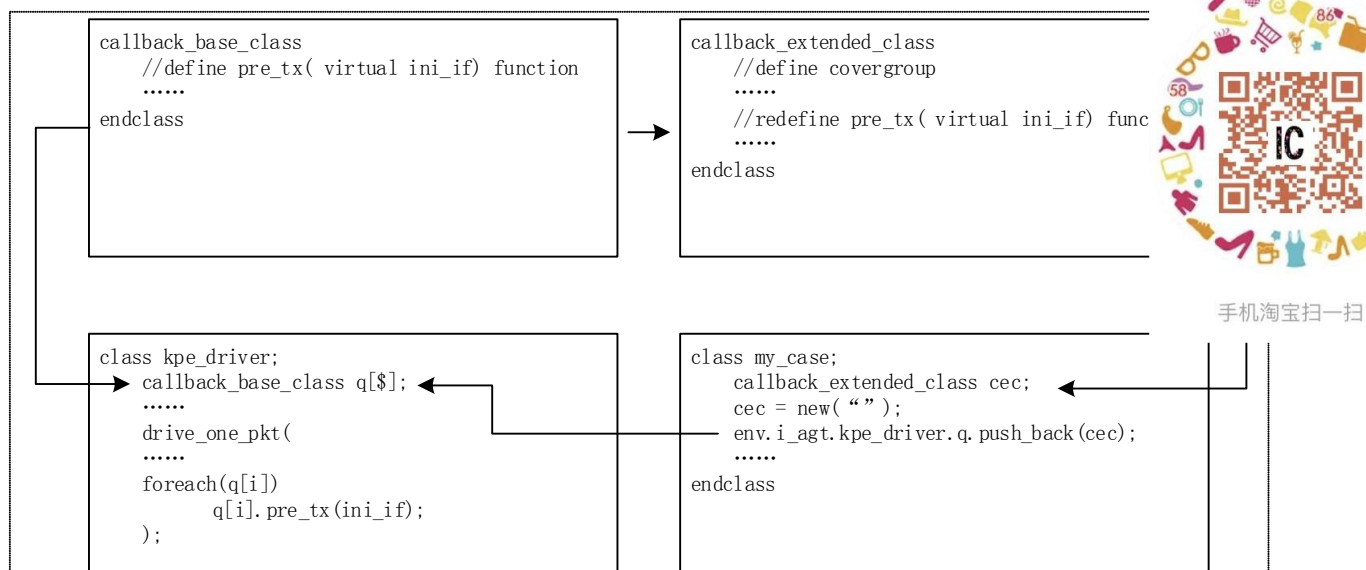


图 3-7：回调收集覆盖率

然后在回调基类上我又写了一个派生类，在这个派生类里面，首先定义了 covergroup，然后定义 pre_tx() 函数的功能，并在 pre_tx() 中调用了采样变量数值的函数——sample() 函数。这样就完成了第一步。接下来，在 kpe_driver 中定义了一个上述的回调基类的队列，并在 kpe_driver 的 drive_one_pkt()（这个函数是专门用来把 transaction 中的信号驱动入待测硬件里面的）函数中，调用队列中的每个对象的 pre_tx() 函数执行。最后，在 my_case 类里面实例化刚才派生自回调基类的类，并把它放入 kpe_driver 中的队列里面，整个过程如图 3-7。

在 kpe 模块中总共写了 4 个 sequence：

case0_sequence: 约束 transaction 类型变量中的 enable 信号为 1，ctrl_kpe_bypass 信号为 0。kpe_ifmap 和 kpe_weight 的后 4 个元素为 0。ifmap 和 weight 的精度为 16bit。kpe_shift 的值通过一个变量约束，然后在 virtual_sequence 中通过 for 循环来遍历 kpe_shift 的不同情况。其余的控制信号按照时序变化的要求即可。

case1_sequence: 约束 transaction 类型变量中的 enable 信号为 1，ctrl_kpe_bypass 信号为 0。kpe_ifmap 和 kpe_weight 的后 4 个元素为 0。ifmap 和 weight 的精度为 8bit。根据精度的值，再确定 kpe_weight 的前 14 个值，约束它们的值在 $[-2^8, 2^8 - 1)$ 范围中。kpe_shift 的值通过一个变量约束，然后在 virtual_sequence 中通过 for 循环来遍历 kpe_shift 的不同情况。其余的控制信号按照时序变化的要求即可。

case2_sequence:约束 transaction 类型变量中的 enable 信号为 0。其余信号随机。

case3_sequence:约束 transaction 类型变量中的 enable 信号为 1，同时约束 ctrl_kpe_bypass 信号为 1。这样就可以验证 kpe 模块的 bypass 功能。

验证平台的编译和运行的命令、C 模型的编译和产生覆盖率报告的命令都是通过 makefile 脚本来控制。

3.1.2 lpe 模块的测试点分解和验证平台

lpe 内部实现两种加法：一个是把来自 kpe 的 16 个数通过加法树加起来，另一个是把加法树的结果和读取自 global buffer 的值相加。由于 lpe 内部的加法树是由流水线实现的，所以在对 lpe 模块施加激励时需要按照一定的时序要求连续给 lpe 输入一定个数的数据。因此，在搭建验证平台的时候为了实现连续给 lpe 输入数据，我在 transaction 中采用了定宽数组的数据结构，这样就可以在某个时间段内，把数组中的数依次取出，送入 lpe，实现连续输入数据。根据设计中 ALL、NONE 和 IMAGE 三种模式下的时序图，ALL 模式需要 6 个时钟周期完成一次完整的流水线运算和结果存储，NONE 模式需要 19 个周期，IMAGE 模式需要 9 个时钟周期，为了用一种 transaction 来把三种情况的仿真都完成，最后我决定把定宽数组的大小确定为 19，并且在 transaction 中定义一个控制时钟周期的变量，这样在用 driver 进行驱动时就可以根据不同模式控制时钟周期。

表 3-2: lpe 模块分解的测试点

测试点序号	测试点	测试方法
1	stgr_pool_enable 信号为 1 时，输出是否与 pool_ofmap 的值一致。	约束 transaction 里面的 stgr_pool_enable 变量为 1，kpe_sum 数组中的数字全部为 0。
2	ctrl_adt_enable 信号为 0 时，输出是否不变	约束 ctrl_adt_enable 为 0。
3	在 stgr_precision_ifmap 为 16bit 精度时，ctrl_lpe_sprmps 为 ALL 模式时，验证 ctrl_lpe_acc_bypass 为 0/1 和 ctrl_lpe_relu 为 0/1 的情况下，输出是否正确。	约束 stgr_precision_ifmap 信号为 16bit，ctrl_lpe_sprmps 信号为 ALL 模式。sequence 中有我定义了 2 个随机变量 A 和 B 用来约束 transaction 中的



手机淘宝扫一扫

		<p>ctrl_lpe_acc_bypass 和 ctrl_lpe_relu。该 sequence 通过`uvm_do_with`宏可以发送 transaction，并通过 A 和 B 两个随机变量来约束 transaction 中的 ctrl_lpe_relu 和 ctrl_lpe_acc_bypass 变量。而在 virtual sequence 中可以通过`uvm_do_on_with`宏启动相应的 sequence，并约束 sequence 中的随机变量（A 和 B），从而约束 transaction 中的变量。这样就能够用 1 个 sequence 来把 4 种情况都测试一遍。</p>
4	<p>在 stgr_precision_ifmap 为 16bit 精度时，ctrl_lpe_sprmps 为 NONE 模式时，验证 ctrl_lpe_acc_bypass 为 0/1 和 ctrl_lpe_relu 为 0/1 的情况下，输出是否正确。</p>	<p>这个与 3 中的约束情况类似，不同之处是把 ctrl_lpe_sprmps 设置成为 NONE 模式。</p>
5	<p>在 stgr_precision_ifmap 为 16bit 精度时，ctrl_lpe_sprmps 为 IMAGE 模式时，验证 ctrl_lpe_acc_bypass 为 0/1 和 ctrl_lpe_relu 为 0/1 的情况下，输出是否正确。</p>	<p>这个与 3 中的约束情况类似，不同之处是把 ctrl_lpe_sprmps 设置成为 IMAGE 模式。</p>
6	<p>在 stgr_precision_ifmap 为 8bit 精度时，ctrl_lpe_sprmps 为 ALL 模式时，验证 ctrl_lpe_acc_bypass 为 0/1 和 ctrl_lpe_relu 为 0/1 的情况下，输出是否正确。</p>	<p>这个与 3 中的约束情况类似，不同之处是把 stgr_precision_ifmap 设置成为 8bit 精度。</p>



手机淘宝扫一扫

7	在 stgr_precision_ifmap 为 8bit 精度时, ctrl_lpe_sprmps 为 NONE 模式时, 验证 ctrl_lpe_acc_bypass 为 0/1 和 ctrl_lpe_relu 为 0/1 的情况下, 输出是否正确。	这个与 4 中的约束情况类似, 不同之处是把 stgr_precision_ifmap 设置成为 8bit 精度。
8	在 stgr_precision_ifmap 为 8bit 精度时, ctrl_lpe_sprmps 为 IMAGE 模式时, 验证 ctrl_lpe_acc_bypass 为 0/1 和 ctrl_lpe_relu 为 0/1 的情况下, 输出是否正确。	这个与 5 中的约束情况类似, 不同之处是把 stgr_precision_ifmap 设置成为 8bit 精度。



手机淘宝扫一扫

lpe 模块验证平台中规中矩, 主要的结构组件是: environment、input_agent、output_agent、reference model、scoreboard 和连接各个组件的 FIFO。其中, reference model 中引入了 C 语言编写的参考模型。它的原型如下:

```
void lpe_algorithm(
    const svLogicVecVal kpe_sum[16],
    const svLogicVecVal* pool_ofmap,
    const svLogic ctrl_lpe_relu,
    const svLogic ctrl_lpe_acc_bypass,
    const svLogic stgr_precision_ifmap,
    const svLogic stgr_pool_enable,
    const svLogicVecVal* lpe_sprmps_e,
    const svLogicVecVal* lpe_gb_rdata,
    svLogicVecVal lpe_gb_wdata[16]
)
```

图 3-8: lpe 模块参考模型中的 C 函数头

在 UVM 搭建的验证平台中利用如下语句即可将 C 模型引入验证平台中:

```
import "DPI-C" function void lpe_algorithm(input logic[15 : 0] kpe_sum[16],
    input logic[15 : 0] pool_ofmap,
    input logic ctrl_lpe_relu,
    input logic ctrl_lpe_acc_bypass,
    input logic stgr_precision_ifmap,
    input logic stgr_pool_enable,
    input logic [1 : 0] lpe_sprmps_e,
    input logic [15 : 0] lpe_gb_rdata,
    output logic [15 : 0] lpe_gb_wdata[16]
);
```

图 3-9: 与 C 函数头对应的 SystemVerilog 函数头

淘宝唯一店铺: 数字IC资料店

C 模型与待测硬件的输出比较与 kpe 模块中的情况类似，在此不再赘述。

lpe 验证平台的树形结构与 kpe 的大致相似，也是有 lpe_driver、lpe_mon、lpe_sequencer、lpe_model、lpe_scoreboard、in_agt、out_agt 等构成。不同之处是在 lpe_driver 和 lpe_mon 的通信中，lpe_mon 的事件调用方法的语句是 event_variable_name.wait_trigger_data()。它与 wait_trigger() 的方法不同之处是它可以从 trigger() 中得到参数，在实际的验证中我把 lpe_driver 从 lpe_sequencer 中获得的 transaction 传递给了 lpe_mon，这样就可以获得 transaction 中控制时钟周期的变量（3.1.2 节有详细说明），从而控制 lpe_mon 收集数据的时钟周期。

lpe 模块的覆盖率的收集也用到了回调，与 kpe 模块中的类似，在此不再赘述。

在 lpe 模块中总共写了 8 个 sequence：

case0_sequence：约束 transaction 类型变量中的 stgr_precision_ifmap 变量为 16bit 精度，ctrl_lpe_sprmps 变量为 ALL 模式，stgr_pool_enable 变量为 0，即没有池化。然后，在 virtual_sequence 中通过`uvm_do_on_with 宏约束 sequence 中的随机控制变量，使这个 sequence 能够把 No-ReLU/ReLU、No-bypass/bypass 四种模式都遍历一遍。

case1_sequence：约束 transaction 类型变量中的 stgr_precision_ifmap 变量为 16bit 精度，ctrl_lpe_sprmps 变量为 IMAGE 模式，stgr_pool_enable 变量为 0，即没有池化。然后，在 virtual_sequence 中通过`uvm_do_on_with 宏约束 sequence 中的随机控制变量，使这个 sequence 能够把 No-ReLU/ReLU、No-bypass/bypass 四种模式都遍历一遍。

case2_sequence：约束 transaction 类型变量中的 stgr_precision_ifmap 变量为 16bit 精度，ctrl_lpe_sprmps 变量为 NONE 模式，stgr_pool_enable 变量为 0，即没有池化。然后，在 virtual_sequence 中通过`uvm_do_on_with 宏约束 sequence 中的随机控制变量，使这个 sequence 能够把 No-ReLU/ReLU、No-bypass/bypass 四种模式都遍历一遍。

case3_sequence：约束 transaction 类型变量中的 stgr_precision_ifmap 变量为 8bit 精度，ctrl_lpe_sprmps 变量为 ALL 模式，stgr_pool_enable 变量为 0，即没有池化。然后，在 virtual_sequence 中通过`uvm_do_on_with 宏约束 sequence 中的随机控制变量，使这个 sequence 能够把 No-ReLU/ReLU、No-bypass/bypass 四种模式都遍历一遍。

case4_sequence：约束 transaction 类型变量中的 stgr_precision_ifmap 变量为 8bit 精度，ctrl_lpe_sprmps 变量为 IMAGE 模式，stgr_pool_enable 变



手机淘宝扫一扫

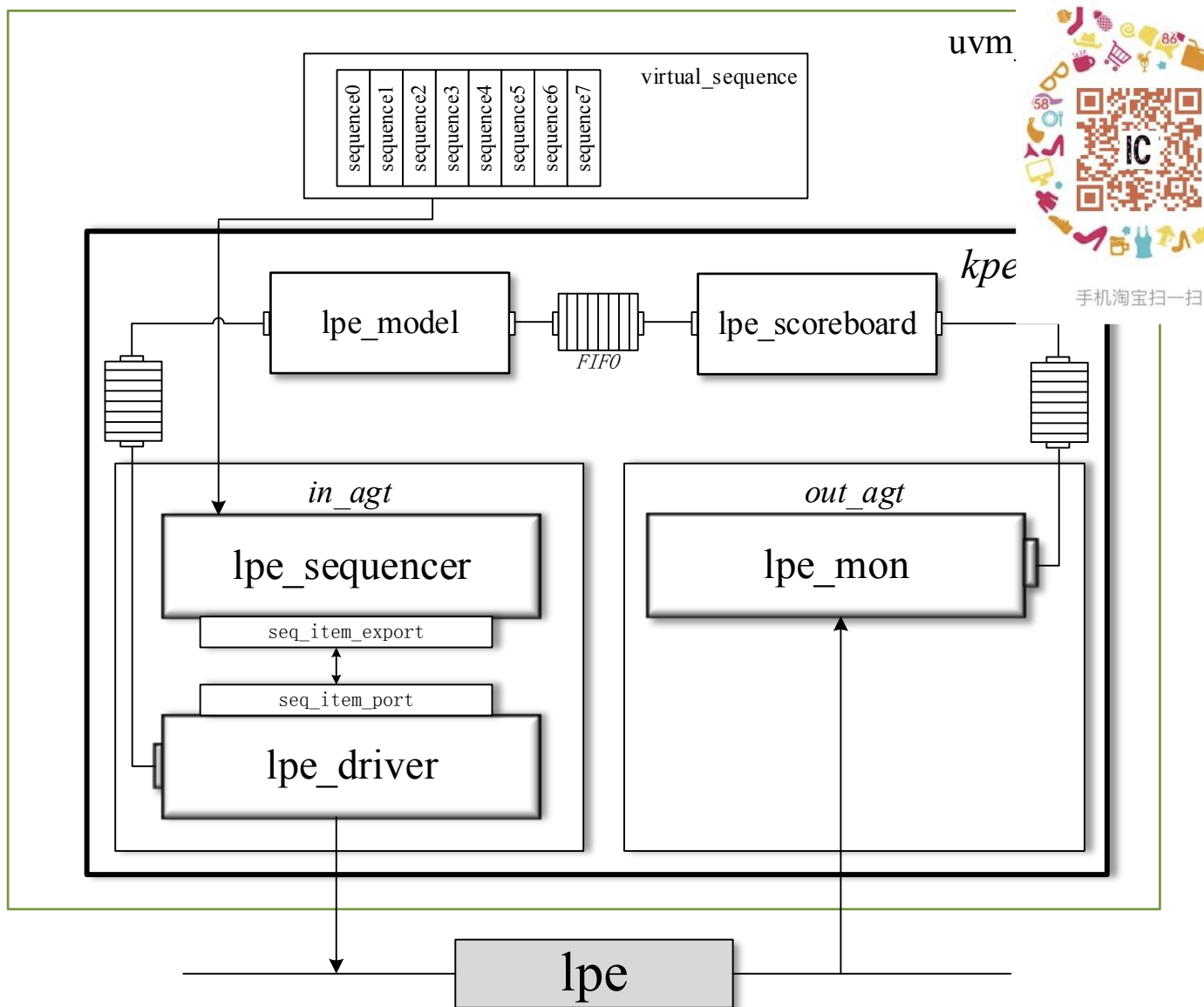


图 3-10: lpe 模块的验证平台

量为 0，即没有池化。然后，在 `virtual_sequence` 中通过 ``uvm_do_on_with` 宏约束 `sequence` 中的随机控制变量，使这个 `sequence` 能够把 No-ReLU/ReLU、No-bypass/bypass 四种模式都遍历一遍。

`case5_sequence`: 约束 `transaction` 类型变量中的 `stgr_precision_ifmap` 变量为 8bit 精度，`ctrl_lpe_sprmps` 变量为 NONE 模式，`stgr_pool_enable` 变量为 0，即没有池化。然后，在 `virtual_sequence` 中通过 ``uvm_do_on_with` 宏约束 `sequence` 中的随机控制变量，使这个 `sequence` 能够把 No-ReLU/ReLU、No-bypass/bypass 四种模式都遍历一遍。

`case6_sequence`: 约束 `transaction` 类型的变量中 `stgr_pool_enable` 变量为 1，使 lpe 模块处于池化模式。上述的其余控制变量随机。

淘宝唯一店铺：数字IC资料店

case7_sequence: 约束 transaction 类型的变量中控制 kpe_sum 数据输入的使能信号 ctrl_adt_enable 为 0, 这样数据就不能输入, 观察 lpe 模块的输出是否一直不变。

验证平台的编译和运行的命令、C 模型的编译和产生覆盖率报告的命令都是通过 makefile 脚本来控制。

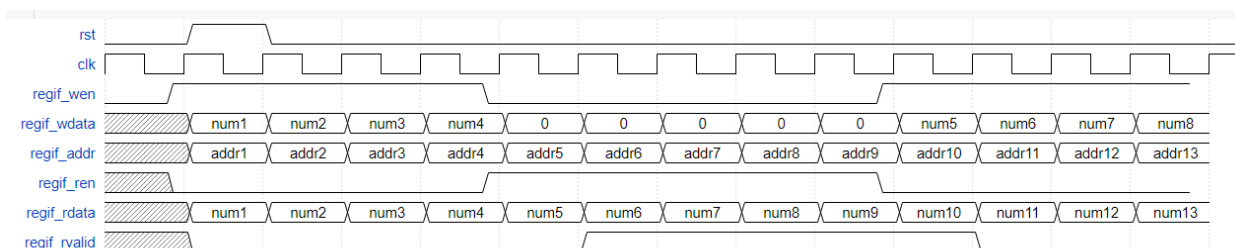
3.1.3 regif 模块的测试点分解和验证平台

regif 模块是整个加速器的“记忆”部分, 它几乎是由寄存器构成, 它要通过 HZS 接口与外部的 SOC 部分连接, 接受外部送来的数据并存储起来。一般来说, 对于正常的项目来说, 寄存器的验证属于验证工作的开头部分, 只有保证寄存器能够发挥正确功能, 才能继而验证整个硬件的其他功能。因此寄存器的验证十分重要。寄存器比较“规整”, 基本的操作主要是“读写”。总之, 寄存器验证有它自身的特点。这一部分的验证我采用了寄存器模型。

表 3-3: regif 模块分解的测试点

测试点序号	测试点	测试方法
1	复位以后寄存器的值是否正确。	复位以后在 sequence 中利用寄存器模型的 read 函数, 通过前门访问的方式读取寄存器复位值。
2	对寄存器连续写。	利用寄存器模型中的 write 函数, 通过前门访问的方式连续写寄存器。
3	对寄存器连续读。	利用寄存器模型中的 read 函数, 通过前门访问的方式连续读寄存器。
4	写一个寄存器后立即读出。	先用 write 函数写, 随后立即用 read 函数读同一个寄存器的值, 然后再写下一个寄存器, 如此往复。

在建立寄存器模型时, 对于访问寄存器的总线模型, 我是基于上述 regif 部分信号列表开发的。时序图如下:



淘宝唯一店铺: 数字ic资料店

图 3-11: regif 模块部分信号的时序图



手机淘宝扫一扫

寄存器模型中的变量有3个级别，分别是：uvm_reg_field、uvm_reg 和 uvm_block。首先，对于寄存器模型中的 uvm_reg 级别，我是按照 register definition 文件中定义的能够通过地址寻址到的最小寄存器组建模的，这组寄存器的最大宽度和数据总线的宽度相等。也就是说，在寄存器模型中，uvm_reg 这个级别是映射为实际寄存器中能够单独寻址的一组寄存器。

uvm_reg 中可以定义 uvm_reg_field 类型的变量。由于一个实际的能够寻址的寄存器组中寄存器的数量并不完全等于数据总线的宽度，总是会分成几个小块，每一块对应数据总线上的某几个连续 bit，例如：数据总线是 32bit 宽的，对应的一个能够寻址的寄存器也应该是 32bit 宽，但是，实际中，可能用不到这么多的寄存器，而只有 3 个，其中，第 1 个寄存器与数据总线的第 0 位相连，后两个与数据总线的第 7、8 位相连。所以，我把一个寄存器组中不同的小块映射为寄存器模型中不同的 uvm_reg_field 类型变量，有几个小块，就映射为几个 uvm_reg_field 类型的变量。



手机淘宝扫一扫

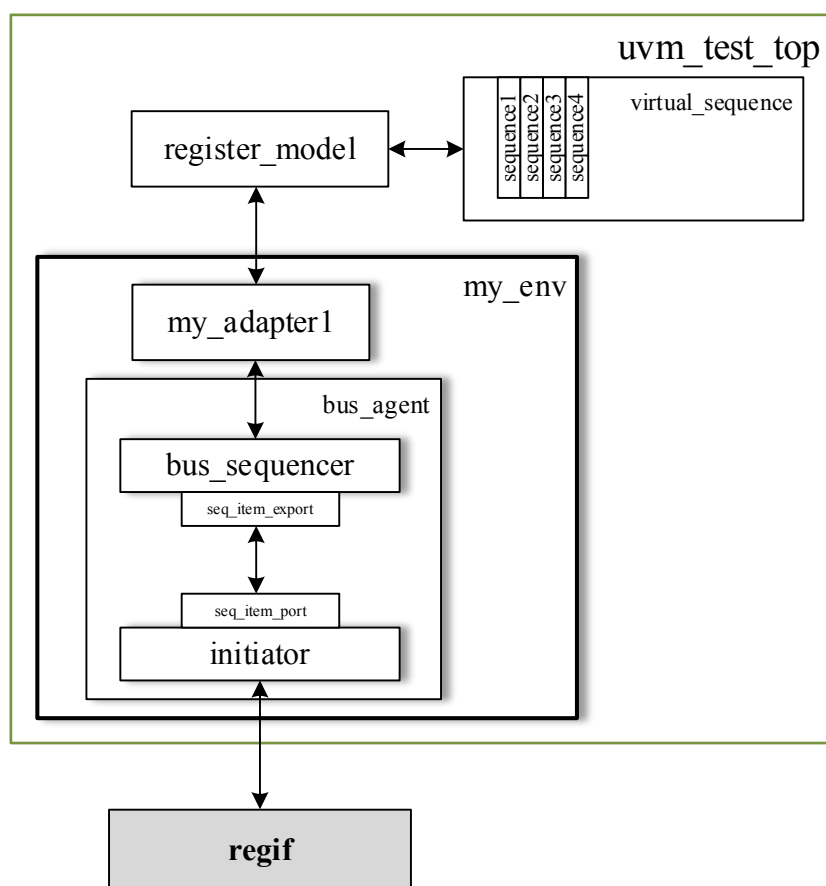


图 3-12: regif 模块的验证平台

淘宝唯一店铺：数字ic资料店



手机淘宝扫一扫

比 `uvm_reg` 类型更高一层的变量类型是 `uvm_reg_block`，在 `uvm_reg_block` 类型的变量中可以定义多个 `uvm_reg` 类型的变量。具体在我建立的寄存器模型中，我把与某一类功能相关的能够寻址的寄存器组都归在了一个 `uvm_reg_block` 中，例如：规定卷积核高、宽，input feature 的高和宽的寄存器组可以归在一个 `uvm_reg_block` 类型变量下，这样的话不同的功能部分就形成了不同的 `uvm_reg_block` 变量。

由于，`uvm_reg_block` 类型的变量可以嵌套自身，所以，为了整个寄存器模型层次分明，我定义了一个最顶层的 `uvm_reg_block` 变量，在它里面将不同功能的 `uvm_reg_block` 实例化。这样一来，整个寄存器模型分为 4 层，最顶层是一个 `uvm_block` 类型的变量，再下一层是根据不同的功能划分的寄存器组，它们也是 `uvm_reg_block` 类型的变量，第 3 层是在第 2 层的 `uvm_reg_block` 变量中实例化的 `uvm_reg` 类型的变量，它们每一个都代表一个可寻址的最小寄存器组，最底层是 `uvm_reg_field` 类型的变量，它们代表一个寄存器组中的部分“片段”，所以它们不可寻址。它们之间的层次关系可以参照图 3-13。

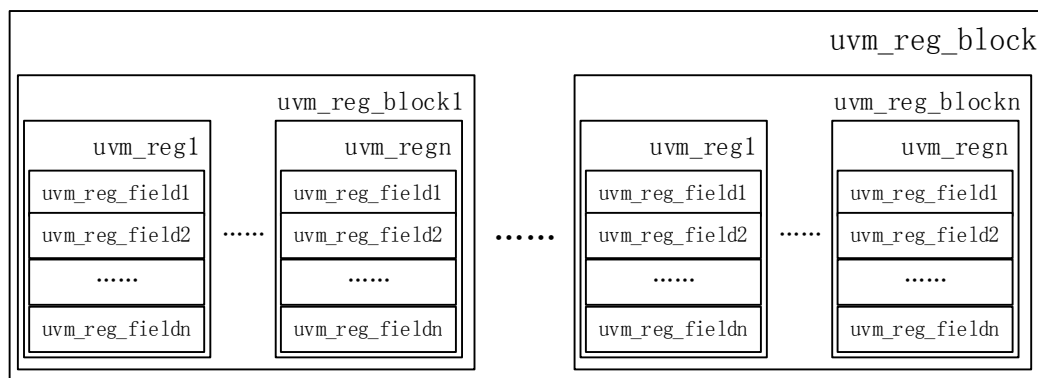


图 3-13：寄存器模型的结构层次

寄存器模型的验证平台也是由基本的组件构成（如图 3-12），与 `kpe`、`lpe` 的验证平台构成大致相同。但是，它的验证环境中会多一个 `adapter` 组件。通常，验证者利用寄存器模型实现前门写或读操作时，都会在寄存器模型内部自动建立 `sequence`，然后在 `sequence` 中产生类型为 `uvm_reg_bus_op` 的数据类型。这个数据类型有它自身的固定的数据成员：

表 3-4： `uvm_reg_bus_op` 类型变量中的成员变量

成员变量	数据类型	变量功能
<code>addr</code>	<code>uvm_reg_addr_t</code>	存储被访问寄存器的地址
<code>data</code>	<code>uvm_reg_data_t</code>	读取或写入的数据

淘宝唯 店铺：数字ic资料店

kind	uvm_access_e	读或者写操作
n_bits	unsigned_int	传输的比特位
byte_en	uvm_reg_byte_en_t	字节操作使能
status	uvm_status_e	操作是否成功



手机淘宝扫一扫

但是不同的寄存器组都有自己的寄存器访问总线，对应的就有不同的 transaction，这些不同类型的 transaction 与 uvm_reg_bus_op 类型变量的信息转换，就需要 adapter 充当转换器。一个正常的 adapter 实现两大功能：一个是当寄存器模型把 uvm_reg_bus_op 类型的变量传递给 sequencer 时，把变量中的信息转换为 transaction 中的信息，另一个是当 sequencer 要把 transaction 送回 sequence 时，把 transaction 中的信息转换为 uvm_reg_bus_op 类型变量中的信息。在本验证平台中，adapter 的主要实现代码如下：

```
function uvm_sequence_item reg2bus(const ref uvm_reg_bus_op rw);
    reg_bus_item item2;
    item2 = new("item2");
    item2.regif_addr = rw.addr;
    item2.regif_wen = (rw.kind == UVM_WRITE) ? 1 : 0;
    item2.regif_ren = (rw.kind == UVM_READ) ? 1 : 0;
    if(item2.regif_wen == 1)
        item2.regif_wdata = rw.data;
    return item2;
endfunction
```

图 3-14: adapter 中 reg2bus 函数

该函数把 uvm_reg_bus_op 类型的变量转变为与相应总线对应的 transaction。

```
function void bus2reg(uvm_sequence_item bus_item, ref uvm_reg_bus_op rw);
    reg_bus_item item3;
    if(!$cast(item3, bus_item)) begin
        `uvm_fatal(tID, "provided bus_item is one of the correct type. Expecting bus_transaction")
        return;
    end

    rw.kind = (item3.regif_ren == 1) ? UVM_READ : UVM_WRITE;
    rw.addr = item3.regif_addr;
    rw.byte_en = 'h3;
    rw.data = (item3.regif_ren == 1) ? item3.regif_rdata : item3.regif_wdata;
    rw.status = UVM_IS_OK;
endfunction
```

图 3-15: adapter 中 bus2reg 函数

该函数把与相应总线对应的 transaction 转变为 uvm_reg_bus_op 类型的变量。

淘宝唯一店铺：数字ic资料店

寄存器模型的覆盖率收集是靠内部定义的 covergroup 收集的。首先，要在每个 uvm_reg 类型的变量中定义 covergroup。然后，UVM 对覆盖率收集会设置双重选择，第一个选择是决定是不是要实例化这个已定义的 covergroup，第二个选择是是否要进行采样。在 uvm_reg 类型变量的 new 函数中，UVM_CVR_ALL 是一个标志符号，意味着允许进行覆盖组的实例化。has_coverage() 函数能够判断是否实例化覆盖组。同时，在 uvm_reg 类型的变量中会定义 sample() 函数，它是 read、write 方法的回调函数。即：每次调用 read、write 函数的最后都会自动调用 sample() 函数。get_coverage 能够判断是否采样覆盖变量。



手机淘宝扫一扫

```
function new(string name = "CONV_K_Size0");
    super.new(name,32,UVM_CVR_ALL);
    set_coverage(UVM_CVR_FIELD_VALS);
    if(has_coverage(UVM_CVR_FIELD_VALS)) begin
        value_cg = new();
    end
endfunction
```

图 3-16: uvm_reg 类型变量中的构造函数

```
function void sample(.....);
    super.sample(.....);
    sample_values();
endfunction

function void sample_values();
    super.sample_values();
    if(get_coverage(UVM_CVR_FIELD_VALS)) begin
        value_cg.sample();
    end
endfunction
```

图 3-17: uvm_reg 类型变量中的采样函数

从上述代码可以得知 sample() 函数内部调用了 sample_values() 函数，sample_values() 函数内部通过 get_coverage() 函数来检测是否允许对覆盖组内定义的变量采样，如果用 set_coverage((UVM_CVR_FIELD_VALS) 设置了，则可以采样。

验证平台的编译和运行的命令和产生覆盖率报告的命令都是通过 makefile 脚本来控制。

淘宝唯一店铺：数字ic资料店

3.1.4 改进生成寄存器模型代码的方式

在我第一次搭建带有寄存器模型的验证平台时，寄存器模型的代码我是通过手写的。由于每个寄存器的建模代码的结构是相同的，只有一小部分代码是根据不同的寄存器而不同，例如：每个寄存器模型的命名。如果用人工一个一个的录入，既麻烦也容易出错。所以，我改进了寄存器模型代码的生成方式。

目前，为了统一不同 EDA 厂商的工具，业内主要采用 IP-XACT 格式来描述寄存器，这个格式在 IEEE 1685 的标准中进行了详细的介绍。它是一种结构化的描述方式，不仅可以用来描述寄存器，还可以描述整个硬件设计 IP 的结构，它使得来自不同 IP 供应商的 IP 能够兼容，尤其在 SOC 的设计环境中，它方便了复杂 IP 包在 EDA 工具中的导入和导出，EDA 在处理通过这种结构描述的硬件设计文档时可以提高自动化的能力。在本实验中我就利用了该语言实现了寄存器模型生成的自动化。

由于 tanji-3 的寄存器描述文档是 docx 格式的 word 文本，为了能够通过 IP-XACT 格式生成寄存器模型的 SystemVerilog 代码，我先通过 python 脚本把 docx 格式文档中的信息进行提取，然后写入 IP-XACT 格式的文档中，最后通过 VCS 的寄存器生成工具生成 SystemVerilog 代码。Python 脚本中使用了 docx 包来处理 word 文件和 xml.dom 包来生成 xml 格式的文件。在提取 word 文件中的信息时是以不同的段落有不同的 style 作为提取特征的。VCS 工具在生成寄存器模型时分为两部：第一步先把 IP-XACT 格式的文件转换为 ralf 格式的文件，第二步再把 ralf 格式的文件转换为最终的代码。VCS 工具能够专门识别 ralf 格式的文件。ralf 格式的文件实际上是用 Tcl 8.5 写成的。

在整个过程中用到的文件、产生的文件如下图：

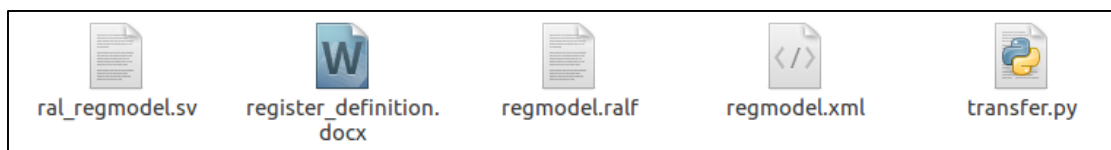


图 3-18：自动化生成寄存器模型所涉及的文件

transfer.py 文件是提取 register_definition.docx 文件信息并创建 regmodel.xml 文件的 python 脚本。regmodel.xml 是以 IP-XACT 格式描述的文件。它以一种树形结构的方式描述寄存器的层次关系。通过 `ralgen -full64 -ipxact2ralf regmodel.xml` 命令将 regmodel.xml 文件转换为 regmodel.ralf 格式的文件，最后通过 `ralgen -full64 -t regmodel -uvm regmodel.ralf` 命令可以生成最终的 ral_regmodel.sv 文档。

淘宝唯一店铺：数字ic资料店



手机淘宝扫一扫

通过上述的步骤只需要几条简单的命令就可以自动化的生成寄存器模型所需的代码，大大减少了人工的工作量，也减少了出错的概率。目前，在工业界采用 IP-XACT 格式描述寄存器的公司已经逐渐多起来。那些原先使用 word、Excel 等文档来描述寄存器的公司也逐渐采用 IP-XACT 格式。这些公司会开发公司内部的脚本，用来把旧的格式转换为 IP-XACT 格式，由于不同的公司原先采用不同的文档格式描述寄存器，所以这些转换的脚本也各不相同。



手机淘宝扫一扫

3.2 子系统级别的验证环境搭建和验证

3.2.1 把数据从 DDR 搬运到 weight_buffer 中

一个子系统通常会包含多个模块且拥有完备的功能，可以执行专门的任务^[5]。现在，我要验证的子系统的功能是能够从 DDR 读出数据，然后经过 HZZM-HZZS 接口，把数据写入加速器中的 weight_buffer 存储器中。它需要的验证环境虽然没有芯片级那么复杂，但是，为了能够实现子系统级到芯片级的验证环境的复用，我在搭建验证环境时考虑了整个加速器 CNN 功能的验证。

我的验证思路是通过文件操作的方式把随机生成的数据写入 DDR 中（此处的 DDR 是仿真用的 dummy_ddr），然后启动待测硬件的数据搬运功能，当待测硬件完成数据搬运后，再从 weight_buffer 存储器中读出数据，然后与刚才写入 DDR 中的文件中的数据做对比，如果完全相同，则说明待测硬件功能正确。

首先，为了待测硬件能够正常工作，需要为它配置相应的寄存器参数。在这个验证平台中我将上次的寄存器模型复用到此处，这样就可以方便的管理、更改寄存器参数。但是，由于上次 regif 模块中的寄存器模型的总线是基于上述 regif 模块的部分输入输出信号列表中的信号，而当前验证平台我要从加速器与 SOC 连接的 HZZS 接口处输入数据，所以要更改寄存器模型的总线以适应 HZZS 接口。

表 3-5: HZZS 接口的信号列表

信号	功能	transaction 中的数据 结构
clk	时钟	
rst	异步复位	
hzzs_mosi	从 SOC 输入加速器的数据输入口	static_array[3]
hzzs_miso	从加速器输入 SOC 的数据输出口	

淘宝唯 一 店铺：数字ic资料店

hzzs_miso_valid	hzzs_miso 数据有效的信号	static_array[3]
hzzs_mosi_valid	hzzs_mosi 数据有效的信号	



手机淘宝扫一扫

在与总线适配的 transaction 中我只定义了两个数组 hzzs_mosi[3] 和 hzzs_mosi_valid[3]。如果操作为 write 的话，三个元素都是有关写操作的信息，如果操作是 read 的话，三个元素中会有一个存放读取的数据。具体会在介绍 adapter 时说明。

DDR 中的数据是随机生成的数据，然后打开文件，通过 \$fscanf() 系统任务将文件中的数据在没有消耗仿真时间的条件下写入 DDR 中。具体代码如图 3-19:

整个验证平台最顶层还是 uvm_top，它的下面是 uvm_test_top，这与以前的模块级的验证平台相同。接下来，在本验证平台中有两层 environment，第一层是 chip_environment，它是一个大的容器。在它里面实例化了一个 environment——env1、寄存器模型的 agent 和寄存器模型的 adapter。env1 中与前述的验证

```

.....
ddr_data_16 = $fopen("ddr_data_16.dat","r");

    if( ddr_data_16 == 0)
        $finish;

    for(int i = 0; i < 32768; ++i) begin
        $fscanf(ddr_data_16,"%h",top1.dummy_ddrif1.ram_data[i]);
        $display("ram_data[%0d] = %0h",i,top1.dummy_ddrif1.ram_data[i]);
    end

    $fclose(ddr_data_16); //this part of code is used to load data from file into dummy_ddr.

```

图 3-19: 把文件中的随机数据写入 dummy_ddr 的代码实现

环境雷同，也是由一些基本的验证组件构成。env1 整体的作用是通过 HZZS 接口与加速器做数据交换。它更多的是为了以后能够实现芯片级的复用而开发的。寄存器模型的 bus_agt 里面包含了 bus_driver 和 bus_sequencer。bus_driver 既可以写数据，也可以读数据，并将读出的数据通过 seq_item_port.item_done(tr) 语句送回寄存器模型中。

寄存器模型的 adapter 比较复杂。由前述知，寄存器模型的 transaction 中有两个数组，每个数组都有 3 个元素。在 adapter 里面就是要将从寄存器模型传来的 uvm_reg_bus_op 类型变量中的信息转换进这两个数组中。首先要将控制读写的信号和地址写入第一个元素，第二个元素根据操作类型会不同。如果

是写操作，则在第二个元素中写入要写的数值，如果是读操作，则什么也不做，直接发送给 bus_driver，它会将读取的数据写入其中。第三个元素是为了满足 HZZS 接口的一次完整的读写操作的需要而添加的。

在验证过程中，为了保持设计的完整性，对待验证硬件部分的某些控制信号，例如：通知数据搬运模块启动的 req 信号、读取 weight_buffer 中数据时用到的使能信号、地址信号等，就直接在 top 模块中进行赋值驱动，而没有通过外部验证环境的 driver 来进行驱动。数据搬运完成后需要从 weight_buffer 硬件中读出，这个也需要从硬件中引出相应的信号，然后再读出它的值并把值存入文件中。在本验证平台中，我先定义了与读出数据端口对应的一组变量，然后利用 assign 语句将它们与设计中的端口连接起来，最后将变量中的值采样并通过 \$fdisplay() 系统任务写入文件中。如果这个文件中的数据与往 DDR 里面写数据的那个文件中的数据完全一样，则说明待测硬件功能正确。这种方法相当于是一种“灰盒”验证方法，它不像黑盒验证，只关注待测硬件的输入输出，而是从硬件内部也引出了信号，这样可以观测待测硬件的内部状况，局部的验证待测硬件的功能和逻辑，从而使验证得到简化，易于理解，而且也便于验证人员逐步理解硬件，形成从局部到整体的理解，为以后全局的验证奠定基础。

3.2.2 把数据从 DDR 搬运到 local_buffer 中

根据硬件的设计规格说明，把数据从 DDR 中搬运到 local_buffer 中，必须先把数据从 DDR 搬运到 global_buffer 中，然后才能从 global_buffer 中搬运到 local_buffer 中。但是，基本的验证思路与上述的子系统级别的验证相同。同样都是通过寄存器模型把运行参数写入待测硬件，然后在 top 模块中给出驱动信号，并监测输出信号，最终将输出信号写入文件中，与原数据做对比。

3.3 芯片级的验证平台搭建和验证

在本论文中的芯片级就是 IP 级，指的是整个加速器。它复用了前面提及的所有验证平台中的部分组件。它是在 3.2.1 节中所介绍的验证平台的框架下搭建的。

它的最顶层的根节点仍然是 uvm_top，uvm_top 的下面是不同的测试用例 uvm_test_top，它是不同 my_case* 类的实例化名字，不同的卷积核大小、不同的步长、不同的精度都会是一个 case，所以会有多个 case，此处的 * 表示 my_case 后面不同的 case 命名不同。uvm_test_top 有唯一一个子节点——chip_env1，它是 chip_env 类的实例化名称。它是一个总的容器类。在它里面实例化了与寄存器模型相关的 bus_agt、reg_adapter 组件、与中断相关的



手机淘宝扫一扫

intr_env 类组件、还有 my_env 类组件，它是整个验证平台的主体部分，intr_env 和 my_env 都派生自 uvm_environment 类。bus_agt 和 reg_adapter 在 3.2.1 节已经详细描述，在此不再赘述。

intr_env 是专门为检测、擦除加速器的中断而编写的测试环境。之所以要单独为中断编写一个环境一是因为需要不断的等待、监测中断，如果中断发生了，还要擦除中断，所以它是一个驱动与检测都需要且比较独立的信号，二是这样使得验证平台更加层次化、结构化，有利于以后的复用。intr_env 类里面实例化了 intr_agt 类组件，它派生自 uvm_agent 类，所以，它的主要作用仍然是充当容器，intr_agt 类里面实例化 interrupt_handler 类和 intr_sequencer 类。它们分别派生自 uvm_driver 类和 uvm_sequencer 类，自此就可以看出 intr_env 也是一个小的验证平台，具有“收发”功能。

my_env 类里面实例化了这个验证平台的主要部分。my_env 类里面实例化了 in_agt 类、out_agt 类、my_scoreboard 类、kpe_model 类和 lpe_model 类。kpe_model 类和 lpe_model 类中引入的 C 语言参考模型来自 kpe、lpe 模块验证时的参考模型。这实现了参考模型的复用。in_agt 类中实例化了 my_driver 类和 my_sequencer 类。out_agt 类中实例化了 my_monitor 类。该实例主要用来从 hzzs_miso 信号中接收数据，即通过 HZZS 接口读出的数据。为了能够接收从 HZZS 接口读出的数据，特地定义了一个 transaction 数据类——my_transaction2，它里面有两个队列：s2t_hzzs_miso[\$]和 s2t_hzzs_miso_valid[\$]，收集的数据放入它们两个队列里面就通过通信管道传递给了 my_scoreboard。在本论文所涉及到的实验中，当卷积运算完成后，运算的结果会存放在 global_buffer 存储器里面，只有通过 s2t_hzzs_mosi 数据口才能读出。

有关卷积运算的所有寄存器参数均通过寄存器模型进行设置。其他的启动命令通过 my_env 中的 my_driver 进行驱动，运行过程中每完成一步操作都会产生相应的中断，可通过 intr_env 中的 interrupt_handler 进行管理。

整个卷积运算是受到一个状态机的控制。加速器中进行卷积运算首先需要把权重从 DDR 搬运到 weight_buffer 里面，然后需要把输入数据从 DDR 搬运到 global_buffer 里面，再从 global_buffer 搬运到 local_buffer 里面。进行运算时，从 weight_buffer 和 local_bffer 里面分别读出数据，然后送入 kpe 模块，总共有 256 个 kpe 模块，kpe 模块完成运算后再把运算结果传递给 lpe 模块，lpe 模块接着进行运算，待 lpe 模块运算完毕后，把最终的结果存放入 global_buffer 的指定位置。这样就完成了一个卷积运算。其中，把数据从 DDR 搬运到 global_buffer 里面的操作，从 global_buffer 搬运到 local_buffer 的



手机淘宝扫一扫

操作和卷积运算的完成都会有相应的中断产生。每产生中断就用 interrupt_handler 擦除掉它，然后开始启动下一个操作。这样一来发送的激励都是顺序进行的。

总共用了 14 个 case_sequence:

case1_sequence: 设置从 DDR 往 global_buffer 里面写数据时 global_buffer 的地址。每次从 DDR 中读取数据的 burst 长度都设置为 256。总共调用 case1_sequence 达 32 次才可以把 global_buffer 写满。这里写操作是调用的寄存器模型。

case2_sequence: 这个 sequence 发出启动从 DDR 到 global_buffer 数据搬运的命令。这个 sequence 把 transaction 发送给 my_env 中的 my_driver，由 my_driver 直接驱动待测硬件。

case3_sequence: 这个 sequence 会通过 interrupt_handler 发送中断的观测，如果中断没有发起，则它会一直等待；一旦中断发起，它会接受中断标志信号并把这个信号通过 intr_sequencer 传回给 sequence。

case4_sequence: 这个 sequence 的目的就是配合 case3_sequence 将已经发起的中断擦除掉。在判断 case3_sequence 中传回的中断标志信号满足中断发起的条件后，就立即发送 case4_sequence，它是通过 my_env 中的 my_driver 直接发送给加速器的 HZM 接口的。

case5_sequence: 这个 sequence 的目的是设置寄存器中的全局使能信号。这个全局使能信号会选中后面卷积运算将会用到的 cpe 单元、kpe 单元、global_buffer 单元和 local_buffer 单元。

case6_sequence: 这个 sequence 的目的是设置与从 global_buffer 搬运数据到 local_buffer 相关的寄存器，它也是调用寄存器模型实现的。它会给出数据在 global_buffer 中的地址和将要在 local_buffer 中写入的地址，同时，它还会给出控制参数用来控制每次连续读写搬运数据的读写长度和每两次连续读写之间的地址间隔。

case7_sequence: 这个 sequence 发出启动从 global_buffer 到 local_buffer 的数据搬运命令。这个 sequence 把 transaction 发送给 my_env 中的 my_driver，由 my_driver 直接驱动待测硬件。

case8_sequence: 这个 sequence 的目的是设置与卷积运算的精度相关的寄存器，有两种可选择的模式：一种是 input feature 是 16bit 精度，weight 也是 16bit 精度；另外一种 input feature 是 8bit 精度，weight 是 8bit 精度。它也是调用寄存器模型实现的。不同的精度用的是同一个 sequence，通过在 my_case* 中用 ``uvm_do on with` 宏进行约束，实现不同的精度设置。为了在



手机淘宝扫一扫

仿真的时候再给出精度的设置，我特地用了条件执行，为不同的精度设置语句定义了 `precision_16` 和 `precision_8` 两个标志，这样在仿真时用 `+label_name` 的仿真命令就可以实现对不同精度的选择，下面会有详细解释。

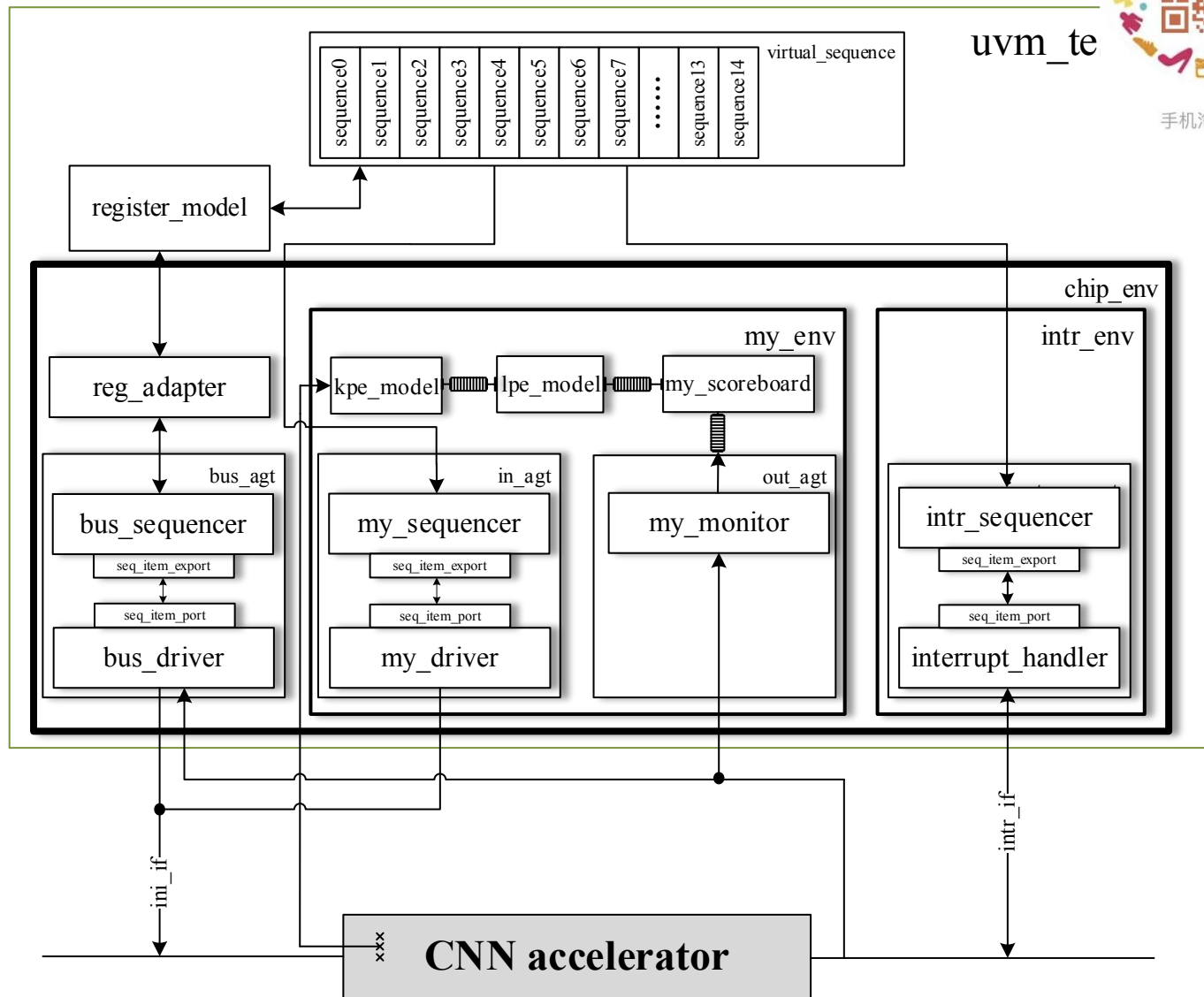


图 3-20: 芯片级的验证平台

case9_sequence: 这个 sequence 的目的是设置与从 DDR 中到 weight_buffer 的权重数据搬运相关的寄存器，它会给出在 DDR 中开始读取权重数据的起始地址参数。它会规定要填写数据的 weight_buffer，它也会规定从 DDR 中读取的数据以怎样的顺序写满选中的 weight buffer。

case10_sequence: 这个 sequence 会设置与 lpe 模块中的运算有关的寄存器。它会提供从 global_buffer 中读取数据的起始地址、每次连续读取数据的长度、需要连续读取数据的次数以及每两次连续读取之间的地址间隔。它会给



手机淘宝扫一扫

读取数据的次数以及每两次连续读取之间的地址

出 lpe 模块最终的运算结果写入 global_buffer 中的地址、连续写的长度、连续写的次数和每两次连续写之间的地址间隔。它会设置 lpe 模块运行的模式——NONE、ALL 或者 IMAGE 模式。它会设置 lpe 模块的 ReLU 功能是否有效，它还有一些其他的参数，在此不一一列举。

case11_sequence: 这个 sequence 设置与卷积运算有关的寄存器参数。它会设置卷积核的大小，包括：卷积核的长和宽，卷积核 padding 的数量等参数。它还会给定 input feature 的大小，包括：长和宽，output feature 的大小，包括：长和宽，还有卷积时移动的“步伐”的大小等参数。

case12_sequence: 这个 sequence 发出启动卷积运算的命令。这个 sequence 把 transaction 发送给 my_env 中的 my_driver，由 my_driver 直接驱动待测硬件。

case13_sequence: 这个寄存器会设置与从 global_buffer 到 HZZS 接口的数据搬运有关的寄存器。在卷积运算完成后会把运算的结果从加速器中读出到 SOC 中的其他组件，这个 sequence 就是配置这个过程。它会提供从 global_buffer 中开始读取的起始地址、一次性连续读取的长度和读取的模式。

case14_sequence: 这个 sequence 发出启动从 global_buffer 往加速器外面进行数据搬运的命令。这个 sequence 把 transaction 发送给 my_env 中的 my_driver，由 my_driver 直接驱动待测硬件。

我在验证的过程中，对于不同的卷积核大小都验证了 16bit 精度、8bit 精度两种情况。对于不同精度的仿真，我利用了系统任务 \$test\$plusargs，系统任务 \$test\$plusargs 可以获取在运行命令中通过 +<string> 定义的参数。这样可以一次性编译然后每次运行不同的精度都定义不同的参数。验证平台中会使用 \$test\$plusargs 来获得运行过程中的参数从而选择不同的执行路径。

在 IP 级我集中验证了加速器支持的 3x3, 5x5, 7x7 和 11x11 的卷积核。每种不同大小的卷积核中又分别验证了 stride 为 1 和 2 的情况。在每种 stride 为 2 时，为了卷积运算的输出长度是整数，我为输入的数据加入了 padding，每次固定 padding 最上面和最左面，且只 padding 一行或者一列，padding 的数也固定全为 0。每种不同大小的卷积核是用不同的 case 实现仿真的，即：一起编译，然后在仿真时用命令行参数 +UVM_TESTNAME = case_name 实现每次运行都用不同的卷积核大小。不同的验证用例定义在不同 package 中，这样一来，不同验证用例中的 sequence 也可以定义相同的名字。不同的验证用例的文件也放在不同的目录中，在编译的命令行中通过 +incdir+<path> 命令将不同的



手机淘宝扫一扫

淘宝唯一店铺：数字ic资料店

目录引入。通过这些方式使得整个验证中用到的文件十分清晰、明了，不容易弄错、混淆，也有利于验证用例的管理和复用。

在大型的验证平台中采用模块化的验证环境搭建方式必不可少，其中，最重要的概念之一就是——package。它能够把相互配合的一组组件封装在一个 package 里面，这样整个验证平台就可以由 package 为单位的结构组件来搭建。具体到本次这个验证平台，我用了一个 package 把与 my_env 相关的组件都通过`include`的方式引入这个 package 中，并把它命名为 testbench_pkga，其中包括 my_transaction1、my_transaction2、k2l_md1_transaction、my_sequencer、lpe_gb_wdata_transaction、my_monitor、in_agt、out_agt、my_scoreboard、kpe_model、lpe_model 和 my_env 组件。它构成一个具有基本功能的验证单位，可以作为更大的验证环境的构成单元。

有关寄存器模型的 package，我在前面的寄存器模型中就已经采用，放在这里进行详细说明。由于本加速器中寄存器数目较多，所以为寄存器建模的文件也比较多。我把 uvm_reg 和 uvm_reg_block 两个类型的变量分别放在两个不同的 package 中，一个命名为 reg_field_pkg 另外一个命名为 reg_block_pkg。定义这两种类型变量的文件也分别放在不同的目录里边。然后，定义第三个 package，用 import 的方式把前两个 package 引入其中，并用`include`把定义最高一级 uvm_reg_block 变量的文件也包括在里面，并命名此 package 为 reg_model_pkg。这样此 package 就代表完整的寄存器模型。在本验证平台中我把 reg_model_pkg 和与寄存器模型相关的组件放到了一个 package 中并命名这个 package 为 testbench_regpkg。

与中断功能相关的验证平台组件我为它们定义了一个 package。该 package 中包括了 intr_transaction、interrupt_handler、intr_sequencer、intr_agt 和 intr_env 组件。它们构成了一个具有基本功能的单位。命名这个 package 为 testbench_intr_pkg。

以上出现了 3 个具有独立、完整功能的、package 封装的验证平台构建单位。最后应该用一个更大一级的 package 把这三个单位引入进去，这个 package 被命名为 testbench_pkg。

在本论文所涉及的验证环境中，在最高的 testbench_pkg 中除了引入上述的 3 个 package 外，还把前述的包括各个用例的 package 也引入进来。同时，一些顶层的组件，例如：virtual_sequencer、chip_env、base_test 等也包括了进来。至此，一个完整的验证平台就基本形成了。

验证平台的编译和运行的命令、C 模型的编译和产生覆盖率报告的命令都是通过 makefile 脚本来实现。通过脚本，不同的测试用例仿真可以一键完成，既



手机淘宝扫一扫

减少了仿真命令的输入，又有利于覆盖率的收集。尤其在仿真平台或者设计经过修改后，通过脚本更方便跑回归测试。



手机淘宝扫一扫

淘宝唯一店铺：数字ic资料店

第4章 验证的结果和分析

4.1 代码覆盖率

4.1.1 kpe 模块的代码覆盖率

kpe 模块的代码覆盖率主要收集了 line、toggle、finite state machine、condition 和 branch 五项。

line 覆盖率指的是待测硬件的 RTL 代码中的每一行是否都被检查过了。代码覆盖率为 100%并不能代表待测硬件的功能正确，只能说明已经写出来的代码是正确的。

toggle 覆盖率是指待测硬件中的端口信号或者模块内部自定义的信号的翻转情况是否都涉及到。例如：一个 1bit 信号从 0 跳变为 1，然后又从 1 跳变为 0。这样算完整的翻转覆盖率。多个位则要求每一个都有 0-1、1-0 翻转。

finite state machine 覆盖率主要是收集状态机中每个状态是否被经过以及状态之间的转换。

condition 覆盖率主要是指：条件运算符中的条件被执行的情况。

branch 覆盖率主要是指在 if、case 语句中的各个分支被执行的情况。

NAME	SCORE	LINE	COND	TOGGLE	FSM	BRANCH
▢ top_tb	98.08	100.00	95.45	96.86		100.00
▢ dla_kpe1	98.02	100.00	95.45	96.62		100.00
⋮ ini_if2	100.00			100.00		
⋮ mon_if2	100.00			100.00		

图 4-1

▢ dla_kpe1	98.02	100.00	95.45	96.62		100.00
⋮ kpe_acc	100.00	100.00	100.00	100.00		100.00
⋮ kpe_acc_r	100.00	100.00	100.00	100.00		100.00
⋮ kpe_mul	92.58	100.00	84.62	85.71		100.00
⊕ kpe_mul_rs	100.00	100.00	100.00	100.00		100.00

图 4-2

上图代表了 kpe 整个验证模块的代码覆盖率，包括 line、condition、toggle 和 branch。从图中可以看出除了 dla_kpe1 模块没有达到 100%覆盖率外

淘宝唯一店铺：数字ic资料店



手机淘宝扫一扫

其余都达到了 100%。所以需要进一步查看 kpe 实例化模块的覆盖率情况。如图 4-2。

从 4-2 图可以看出 kpe 实例化模块的代码覆盖率没有达到 100%是因为 kpe_mul 的实例化模块的 condition 和 toggle 两项覆盖率没有达到 100%，其余模块在各项指标上都达到了 100%的覆盖率。所以需要进一步查看 kpe_mul 的代码覆盖率

Cond Coverage for Module : dla_kpe_mul			
	Total	Covered	Percent
Conditions	13	11	84.62
Logical	13	11	84.62
Non-Logical	0	0	
Event	0	0	

图 4-3

从上图可得 kpe_mul 模块的 condition 覆盖率没有达到 100%。从 4-4、4-5 图可以看出 condition 覆盖率没有达到 100%是因为设计中条件选择语句的两种情况没有遍历到。仔细分析可知：(b == 16'b0) && (a[7] == 1'b1) 这个条件的 1 和 0 的逻辑取值没有实现，实际上是 (b == 16'b0) && (a[7] == 1'b1) 没有实现。

LINE 118		
SUB-EXPRESSION ((b == 16'b0) && (a[7] == 1'b1))		
-----1-----2-----		
-1-	-2-	Status
0	1	Covered
1	0	Not Covered
1	1	Covered

图 4-4

LINE 118		
SUB-EXPRESSION ((b[7] == 1'b1) && (a[7:0] == 8'b0))		
-----1-----2-----		
-1-	-2-	Status
0	1	Not Covered
1	0	Covered
1	1	Covered

图 4-5

为了分析没有覆盖到的原因，现摘取整行代码：



手机淘宝扫一扫

```
dsp_c = (a[7] ^~ b[7]) ? 32'h00000000 :
        (((b == 0) && (a[7] == 1)) || ((b[7] == 1) && (a[7 : 0] == 0))) ?
        32'h0000_0000 : 32'h00010000);
```

图 4-6

从源代码中设计语句可知当 $(b == 16'b0) \&\& (a[7] == 1'b0)$ 满足时根本不会进入 $(b == 16'b0) \&\& (a[7] == 1'b1)$ 这条语句，所以这个条件不可到达。后面的没有覆盖到的那条语句可以进行同样的分析，原因是一样的。

接下来我们分析 toggle 覆盖率。从图 4-7 可知端口部分的输入输出信号的 toggle 覆盖率都达到了 100%。未达到 100% toggle 覆盖率的是模块内部定义的信号(图 4-8)，所以，应该分析模块内部的信号。首先，我们可以从设计的源代码(图 4-9)看出：dsp_c 信号只有可能被赋予两种值：32'h0 和 32'h00010000，所以它除了第 16 位以外。其余所有位的 toggle 的覆盖率为 No。

Toggle Coverage for Module : dla_kpe_mul			
	Total	Covered	Percent
Totals	10	9	90.00
Total Bits	434	372	85.71
Total Bits 0->1	217	186	85.71
Total Bits 1->0	217	186	85.71
Ports	5	5	100.00
Port Bits	154	154	100.00
Port Bits 0->1	77	77	100.00
Port Bits 1->0	77	77	100.00
Signals	5	4	80.00
Signal Bits	280	218	77.86
Signal Bits 0->1	140	109	77.86
Signal Bits 1->0	140	109	77.86

图 4-7

Signal Details			
Name	Toggle	Toggle 1->0	Toggle 0->1
dsp_a[24:0]	Yes	Yes	Yes
dsp_d[24:0]	Yes	Yes	Yes
dsp_b[15:0]	Yes	Yes	Yes
dsp_c[15:0]	No	No	No
dsp_c[16]	Yes	Yes	Yes
dsp_c[31:17]	No	No	No
dsp_p[41:0]	Yes	Yes	Yes

图 4-8



手机淘宝扫一扫

```

always_comb begin
    if ( stgr_precision_weight == PRECISION_WEIGHT_16 ) begin
        dsp_b = b;
        dsp_c = '0;
    end else begin
        dsp_b = {{8{b[7]}}, b[7:0]};
        dsp_c = (a[7] ^~ b[7]) ? 32'h00000000 :
                [(((b == 0) && (a[7] == 1)) || ((b[7] == 1) && (a[7 : 0] == 0))) ?
                32'h0000_0000 : 32'h00010000];
    end
end

```



手机淘宝扫一扫

图 4-9

4.1.2 lpe 模块的代码覆盖率

lpe 模块的代码覆盖率主要收集了 line、toggle、finite state machine、condition、branch 和 path 六项。如下图：

NAME	SCORE	LINE	COND	TOGGLE	FSM	BRANCH	PATH
▢ top_tb	99.90	100.00	100.00	99.50		100.00	100.00
⊕ dla_lpe1	99.90	100.00	100.00	99.48		100.00	100.00
⋮ ini_if2	100.00			100.00			
⋮ rsp_if2	100.00			100.00			

图 4-10

从图中可以看出代码覆盖率没有达到 100%的原因是 dla_lpe1 模块的 toggle 覆盖率没有达到 100%，其余模块的代码覆盖率都达到 100%。我们需要更细致的观察 dla_lpe1 模块的代码覆盖率：

▢ dla_lpe1	99.90	100.00	100.00	99.48		100.00	100.00
⊕ addtree	100.00	100.00	100.00	100.00		100.00	100.00
⋮ lpe_add_signext	100.00	100.00		100.00		100.00	
⋮ lpe_adder	100.00	100.00	100.00	100.00		100.00	
⋮ lpe_bypass_signext	100.00	100.00		100.00		100.00	
⋮ lpe_relu	89.18	100.00		67.54		100.00	
⋮ lpe_signsat	100.00	100.00	100.00	100.00		100.00	100.00

图 4-11

淘宝唯一店铺：数字ic资料店

从图 4-11 中可以看出是 lpe_relu 模块没有达到 100%，其余子模块在各项指标上面都达到了 100%，从而应该分析 lpe_relu 模块的 toggle 覆盖率：

Toggle Coverage for Module : dla_vpcomp

	Total	Covered	Percent
Totals	12	4	33.33
Total Bits	228	154	67.54
Total Bits 0->1	114	77	67.54
Total Bits 1->0	114	77	67.54
Ports	5	2	40.00
Port Bits	100	64	64.00
Port Bits 0->1	50	32	64.00
Port Bits 1->0	50	32	64.00
Signals	7	2	28.57
Signal Bits	128	90	70.31
Signal Bits 0->1	64	45	70.31
Signal Bits 1->0	64	45	70.31

图 4-12

Port Details

Name	Toggle	Toggle 1->0	Toggle 0->1	Direction
a[15:0]	Yes	Yes	Yes	INPUT
b[15:0]	No	No	No	INPUT
y[14:0]	Yes	Yes	Yes	OUTPUT
y[15]	No	No	No	OUTPUT
mode_precision	Yes	Yes	Yes	INPUT
mode_larger	No	No	No	INPUT

图 4-13

这个模块在 lpe 模块中实现 ReLU 函数功能。它的模块端口定义如下：

```
module dla_vpcomp
  import PKG_dla_typedef :: *;
#(
  parameter int GRAN = 8
)(
  input logic [GRAN*2-1:0] a,
  input logic [GRAN*2-1:0] b,
  output logic [GRAN*2-1:0] y,

  input precision_ifmap_e mode_precision,
  input logic mode_larger
);
```

图 4-14



其中有 a, b, y 三个数据信号和 mode_precision、mode_larger 两个控制信号。它在 lpe 模块中的实例化代码如下：

```
dla_vpcmp #(.GRAN(8)) lpe_relu (
    .a          (lpe_res_norelu),
    .b          (16'd0),
    .y          (lpe_res_relu),
    .mode_precision (stgr_precision_ifmap),
    .mode_larger   (1'b1)
);
```

图 4-15

从它的实例化代码可知：b 信号端口被赋予了固定值 16'd0，mode_larger 控制端口被赋予了固定值 1'b1，所以这两个端口的 toggle 覆盖率未能实现全覆盖。对于 ReLU 模块的功能前面有提及，它把输入的负数变为 0，正数和 0 仍按原数据输出，所以 ReLU 函数的输出是非负值。对于 dla_vpcmp 模块的 y 信号来说，它是该模块输出，所以它的输出也是非负值，负数在计算机内以补码表示，最高位为 1，所以 y 信号的最高位 y[15]不可能是 1。

4.1.3 regif 模块的代码覆盖率

regif 模块的代码覆盖率较为简单，主要的是 line 和 toggle 覆盖率。

NAME	SCORE	LINE	COND	TOGGLE
top_tb	92.70	100.00	100.00	70.78
dla_regif1	93.02	100.00	100.00	72.07
regif_ape	94.01	100.00	100.00	76.05
regif_buf	99.55	100.00		98.66
regif_comp	92.83	100.00		78.50
regif_ddr2gb	93.68	100.00		81.05
regif_gb2lb	90.10	100.00		70.31
regif_glb	90.46	100.00		71.37
regif_map	88.79	100.00		66.37
regif_soc2gb	86.76	100.00		60.27

图 4-16

从图中可以看出 regif 模块的 line 代码覆盖率达到 100%，但是 toggle 覆盖率却比较低。这是因为每个被配置的寄存器的实际宽度几乎都不等于读出寄存器的数据总线宽度。所以，每个模块的与读出数据相关的输出端口信号的 toggle 覆盖率都不高。经过对每一个信号的仔细查验，凡是有与数据总线连接的端口或内部信号的 toggle 覆盖率都满足要求。



4.2 功能覆盖率:

4.2.1 kpe 模块的功能覆盖率

```
covergroup covport();
//option.per_instance = 1;
enable:coverpoint enable
{bins enable[] = {[0 : 1]}};
coverpoint kpe_ifmap
{option.auto_bin_max = 16;}
coverpoint kpe_weight
{option.auto_bin_max = 16;}
coverpoint ctrl_kpe_src0_enable;
coverpoint ctrl_kpe_src1_enable;
coverpoint ctrl_kpe_mul_enable;
coverpoint ctrl_kpe_acc_enable;
coverpoint ctrl_kpe_acc_rst;
coverpoint ctrl_kpe_bypass
{bins ctrl_kpe_bypass[] = {[0 : 1]}};
stgr_precision_kpe_shift:coverpoint stgr_precision_kpe_shift
{bins stgr_precision_kpe_shift[] = {[0 : 12]}};
stgr_precision_ifmap:coverpoint stgr_precision_ifmap
{bins stgr_precision_ifmap[] = {[0 : 1]}};
stgr_precision_weight:coverpoint stgr_precision_weight
{bins stgr_precision_weight[] = {[0 : 1]}};

cross stgr_precision_ifmap,stgr_precision_weight;
cross stgr_precision_ifmap,stgr_precision_kpe_shift
{
    ignore_bins l1 = binsof(stgr_precision_ifmap) intersect{1} &&
    binsof(stgr_precision_kpe_shift) intersect{[9 : 12]} ;
}
cross stgr_precision_weight,stgr_precision_kpe_shift
{
    ignore_bins l2 = binsof(stgr_precision_kpe_shift)intersect{[9 : 12]} &&
    binsof(stgr_precision_weight)intersect{1};
}
endgroup
```

图 4-17

从 3.1.1 节的测试点的分解和 case_sequence 的开发可知，针对不同的测试功能点在 sequence 中都覆盖到了。接下来我们可以分析 covergroup 中定义的覆盖点。在 kpe 的 driver_cbs_coverage 类中定义的覆盖组如图 4-17。覆盖组的名称为 covport。在 covport 里面主要为在 3.1.1 节中分解的测试点而自定义了覆盖点的仓。这些信号包括 enable、ctrl_kpe_bypass、stgr_precision_weight、stgr_precision_kpe_shift 和 stgr_precision_ifmap。kpe_ifmap 和 kpe_weight 使用 VCS 自动定义仓的功能，限定自动定义仓的数目为 16 个，便于统计，也可以减少 transaction 的用量。同时又为以下三组信号建立了交叉覆盖组。

1. stgr_precision_ifmap 和 stgr_precision_weight
2. stgr_precision_ifmap 和 stgr_precision_kpe_shift
3. stgr_precision_weight 和 stgr_precision_kpe_shift

stgr_precision_ifmap 和 stgr_precision_weight 交叉覆盖率如图 4-18。



手机淘宝扫一扫

Bins

stgr_precision_ifmap	stgr_precision_weight	COUNT	AT LEAST
stgr_precision_ifmap_PRECISION_IFMAP_8	stgr_precision_weight_PRECISION_WEIGHT_16	198	1
stgr_precision_ifmap_PRECISION_IFMAP_8	stgr_precision_weight_PRECISION_WEIGHT_8	1802142	1
stgr_precision_ifmap_PRECISION_IFMAP_16	stgr_precision_weight_PRECISION_WEIGHT_8	144	1
stgr_precision_ifmap_PRECISION_IFMAP_16	stgr_precision_weight_PRECISION_WEIGHT_16	2340306	1

图 4-18

Bins

stgr_precision_weight	stgr_precision_kpe_shift	COUNT	AT LEAST
stgr_precision_weight_PRECISION_WEIGHT_8	stgr_precision_kpe_shift_5	180000	1
stgr_precision_weight_PRECISION_WEIGHT_8	stgr_precision_kpe_shift_3	180054	1
stgr_precision_weight_PRECISION_WEIGHT_8	stgr_precision_kpe_shift_0	180036	1
stgr_precision_weight_PRECISION_WEIGHT_8	stgr_precision_kpe_shift_6	180018	1
stgr_precision_weight_PRECISION_WEIGHT_8	stgr_precision_kpe_shift_1	181818	1
stgr_precision_weight_PRECISION_WEIGHT_8	stgr_precision_kpe_shift_8	180036	1
stgr_precision_weight_PRECISION_WEIGHT_8	stgr_precision_kpe_shift_2	180036	1
stgr_precision_weight_PRECISION_WEIGHT_8	stgr_precision_kpe_shift_4	360018	1
stgr_precision_weight_PRECISION_WEIGHT_8	stgr_precision_kpe_shift_7	180036	1
stgr_precision_weight_PRECISION_WEIGHT_16	stgr_precision_kpe_shift_b	180018	1
stgr_precision_weight_PRECISION_WEIGHT_16	stgr_precision_kpe_shift_a	180018	1
stgr_precision_weight_PRECISION_WEIGHT_16	stgr_precision_kpe_shift_c	180036	1
stgr_precision_weight_PRECISION_WEIGHT_16	stgr_precision_kpe_shift_0	180018	1
stgr_precision_weight_PRECISION_WEIGHT_16	stgr_precision_kpe_shift_3	180018	1
stgr_precision_weight_PRECISION_WEIGHT_16	stgr_precision_kpe_shift_9	180036	1
stgr_precision_weight_PRECISION_WEIGHT_16	stgr_precision_kpe_shift_6	180018	1
stgr_precision_weight_PRECISION_WEIGHT_16	stgr_precision_kpe_shift_5	180072	1
stgr_precision_weight_PRECISION_WEIGHT_16	stgr_precision_kpe_shift_1	180018	1
stgr_precision_weight_PRECISION_WEIGHT_16	stgr_precision_kpe_shift_2	180018	1
stgr_precision_weight_PRECISION_WEIGHT_16	stgr_precision_kpe_shift_4	180036	1
stgr_precision_weight_PRECISION_WEIGHT_16	stgr_precision_kpe_shift_8	180072	1
stgr_precision_weight_PRECISION_WEIGHT_16	stgr_precision_kpe_shift_7	180036	1

图 4-19

当 stgr_precision_ifmap 和 stgr_precision_weight 为 1 时，即它们为 8bit 精度时，排除掉 9、10、11 和 12bit 的移位操作。stgr_precision_ifmap 和 stgr_precision_kpe_shift 交叉覆盖组的覆盖率收集如图 4-19，从图中可以看出 input feature 为 8bit 精度时，kpe_shift 的值没有 9、10、11 和 12。stgr_precision_weight 和 stgr_precision_kpe_shift 交叉覆盖组的覆盖率如图 4-20，从图中可以看出 weight 为 8bit 精度时，kpe_shift 的值没有 9、10、11 和 12。其余的情况全部覆盖到。

至此可以认为 kpe 模块把分解的测试点全部验证完毕。

淘宝唯一店铺：数字ic资料店



手机淘宝扫一扫

Bins

stgr_precision_ifmap	stgr_precision_kpe_shift	COUNT	AT LEAST
stgr_precision_ifmap_PRECISION_IFMAP_8	stgr_precision_kpe_shift_5	180000	1
stgr_precision_ifmap_PRECISION_IFMAP_8	stgr_precision_kpe_shift_3	180054	1
stgr_precision_ifmap_PRECISION_IFMAP_8	stgr_precision_kpe_shift_0	180036	1
stgr_precision_ifmap_PRECISION_IFMAP_8	stgr_precision_kpe_shift_6	180072	1
stgr_precision_ifmap_PRECISION_IFMAP_8	stgr_precision_kpe_shift_4	360054	1
stgr_precision_ifmap_PRECISION_IFMAP_8	stgr_precision_kpe_shift_1	181890	1
stgr_precision_ifmap_PRECISION_IFMAP_8	stgr_precision_kpe_shift_8	180126	1
stgr_precision_ifmap_PRECISION_IFMAP_8	stgr_precision_kpe_shift_2	180054	1
stgr_precision_ifmap_PRECISION_IFMAP_8	stgr_precision_kpe_shift_7	180036	1
stgr_precision_ifmap_PRECISION_IFMAP_16	stgr_precision_kpe_shift_b	180054	1
stgr_precision_ifmap_PRECISION_IFMAP_16	stgr_precision_kpe_shift_a	180054	1
stgr_precision_ifmap_PRECISION_IFMAP_16	stgr_precision_kpe_shift_c	180054	1
stgr_precision_ifmap_PRECISION_IFMAP_16	stgr_precision_kpe_shift_0	180036	1
stgr_precision_ifmap_PRECISION_IFMAP_16	stgr_precision_kpe_shift_3	180090	1
stgr_precision_ifmap_PRECISION_IFMAP_16	stgr_precision_kpe_shift_9	180072	1
stgr_precision_ifmap_PRECISION_IFMAP_16	stgr_precision_kpe_shift_6	180036	1
stgr_precision_ifmap_PRECISION_IFMAP_16	stgr_precision_kpe_shift_5	180126	1
stgr_precision_ifmap_PRECISION_IFMAP_16	stgr_precision_kpe_shift_1	180036	1
stgr_precision_ifmap_PRECISION_IFMAP_16	stgr_precision_kpe_shift_2	180018	1
stgr_precision_ifmap_PRECISION_IFMAP_16	stgr_precision_kpe_shift_4	180036	1
stgr_precision_ifmap_PRECISION_IFMAP_16	stgr_precision_kpe_shift_8	180108	1
stgr_precision_ifmap_PRECISION_IFMAP_16	stgr_precision_kpe_shift_7	180036	1

图 4-20

4.2.2 lpe 模块的功能覆盖率

从 3.1.2 节的测试点的分解和 case_sequence 的开发可知，针对不同的测试功能点在 sequence 中都覆盖到了。接下来我们可以分析 covergroup 中的覆盖点。

```
covergroup covport();
    coverpoint stgr_pool_enable
    | {bins stgr_pool_enable[] = {[0 : 1]}};
    coverpoint kpe_sum
    | {option.auto_bin_max = 16;}
    coverpoint pool_ofmap
    | {option.auto_bin_max = 16;}
    coverpoint lpe_gb_rdata
    | {option.auto_bin_max = 16;}
    coverpoint adt_fifo_set;

    coverpoint adt_fifo_ren;

    coverpoint ctrl_adt_enable
    | {bins ctrl_adt_enable[] = {[0 : 1]}};
    coverpoint ctrl_lpe_relu
    | {bins ctrl_lpe_relu[] = {[0 : 1]}};
    coverpoint ctrl_lpe_acc_bypass
    | {bins ctrl_lpe_acc_bypass[] = {[0 : 1]}};
    coverpoint ctrl_lpe_sprmps;
    coverpoint stgr_precision_ifmap;

    cross stgr_precision_ifmap , ctrl_lpe_sprmps , ctrl_lpe_acc_bypass , ctrl_lpe_relu;
```

图 4-21



手机淘宝扫一扫

从上图可以看出我为 stgr_pool_enable、ctrl_adt_enable、ctrl_lpe_relu 和 ctrl_lpe_acc_bypass 信号都建立了自定义的仓。由于 stgr_precision_ifmap 和 ctrl_lpe_sprmps 都是枚举类型，VCS 会自动为它们创建仓且仓名是枚举变量的各种取值情况。最后为 stgr_precision_ifmap、ctrl_lpe_sprmps、ctrl_lpe_acc_bypass 和 ctrl_lpe_relu 信号建立了交叉覆盖组，这个交叉覆盖组的覆盖率收集情况如图 4-23。

stgr_pool_enable 和 ctrl_adt_enable 信号的覆盖率如图 4-22：

Bins		
NAME	COUNT	AT LEAST
stgr_pool_enable_0	3000060	1
stgr_pool_enable_1	60000	1

Bins		
NAME	COUNT	AT LEAST
ctrl_adt_enable_0	2790060	1
ctrl_adt_enable_1	270000	1

图 4-22

Bins					
stgr_precision_ifmap	ctrl_lpe_sprmps	ctrl_lpe_acc_bypass	ctrl_lpe_relu	COUNT	AT LEAST
auto_PRECISION_IFMAP_8	auto_LPE_SPRMPS_ALL	ctrl_lpe_acc_bypass_1	ctrl_lpe_relu_1	60000	1
auto_PRECISION_IFMAP_8	auto_LPE_SPRMPS_ALL	ctrl_lpe_acc_bypass_1	ctrl_lpe_relu_0	60000	1
auto_PRECISION_IFMAP_8	auto_LPE_SPRMPS_ALL	ctrl_lpe_acc_bypass_0	ctrl_lpe_relu_1	60000	1
auto_PRECISION_IFMAP_8	auto_LPE_SPRMPS_ALL	ctrl_lpe_acc_bypass_0	ctrl_lpe_relu_0	60000	1
auto_PRECISION_IFMAP_8	auto_LPE_SPRMPS_NONE	ctrl_lpe_acc_bypass_1	ctrl_lpe_relu_1	190000	1
auto_PRECISION_IFMAP_8	auto_LPE_SPRMPS_NONE	ctrl_lpe_acc_bypass_1	ctrl_lpe_relu_0	190000	1
auto_PRECISION_IFMAP_8	auto_LPE_SPRMPS_NONE	ctrl_lpe_acc_bypass_0	ctrl_lpe_relu_1	190000	1
auto_PRECISION_IFMAP_8	auto_LPE_SPRMPS_NONE	ctrl_lpe_acc_bypass_0	ctrl_lpe_relu_0	190000	1
auto_PRECISION_IFMAP_8	auto_LPE_SPRMPS_IMAGE	ctrl_lpe_acc_bypass_1	ctrl_lpe_relu_1	90000	1
auto_PRECISION_IFMAP_8	auto_LPE_SPRMPS_IMAGE	ctrl_lpe_acc_bypass_0	ctrl_lpe_relu_1	90000	1
auto_PRECISION_IFMAP_8	auto_LPE_SPRMPS_IMAGE	ctrl_lpe_acc_bypass_0	ctrl_lpe_relu_0	90000	1
auto_PRECISION_IFMAP_16	auto_LPE_SPRMPS_ALL	ctrl_lpe_acc_bypass_1	ctrl_lpe_relu_1	60000	1
auto_PRECISION_IFMAP_16	auto_LPE_SPRMPS_ALL	ctrl_lpe_acc_bypass_1	ctrl_lpe_relu_0	120060	1
auto_PRECISION_IFMAP_16	auto_LPE_SPRMPS_ALL	ctrl_lpe_acc_bypass_0	ctrl_lpe_relu_1	60000	1
auto_PRECISION_IFMAP_16	auto_LPE_SPRMPS_ALL	ctrl_lpe_acc_bypass_0	ctrl_lpe_relu_0	60000	1
auto_PRECISION_IFMAP_16	auto_LPE_SPRMPS_NONE	ctrl_lpe_acc_bypass_1	ctrl_lpe_relu_1	190000	1
auto_PRECISION_IFMAP_16	auto_LPE_SPRMPS_NONE	ctrl_lpe_acc_bypass_1	ctrl_lpe_relu_0	190000	1
auto_PRECISION_IFMAP_16	auto_LPE_SPRMPS_NONE	ctrl_lpe_acc_bypass_0	ctrl_lpe_relu_1	380000	1
auto_PRECISION_IFMAP_16	auto_LPE_SPRMPS_NONE	ctrl_lpe_acc_bypass_0	ctrl_lpe_relu_0	190000	1
auto_PRECISION_IFMAP_16	auto_LPE_SPRMPS_IMAGE	ctrl_lpe_acc_bypass_1	ctrl_lpe_relu_1	90000	1
auto_PRECISION_IFMAP_16	auto_LPE_SPRMPS_IMAGE	ctrl_lpe_acc_bypass_1	ctrl_lpe_relu_0	90000	1
auto_PRECISION_IFMAP_16	auto_LPE_SPRMPS_IMAGE	ctrl_lpe_acc_bypass_0	ctrl_lpe_relu_1	180000	1
auto_PRECISION_IFMAP_16	auto_LPE_SPRMPS_IMAGE	ctrl_lpe_acc_bypass_0	ctrl_lpe_relu_0	90000	1

图 4-23

至此可以认为 lpe 模块把分解的测试点全部验证完毕。

淘宝唯一店铺：数字ic资料店



手机淘宝扫一扫

4.2.3 regif 模块的功能覆盖率

regif 模块的功能覆盖率就是每个寄存器的取值是否被遍历，结果如下：

Total Groups Coverage Summary										
SCORE		INST SCORE	WEIGHT							
100.00		100.00	1							

NAME	SCORE	INSTANCES	WEIGHT	GOAL	AT LEAST	PER INSTANCE	AUTO BIN MAX	PRINT MISSING	COMMENT
reg_field_pkg::SOC2GB_CONFIG::value_cg	100.00	100.00	1	100	1	1	64	64	
reg_field_pkg::DDR2GB_CTRL::value_cg	100.00	100.00	1	100	1	1	64	64	
reg_field_pkg::DDR2GB_DDR_ADDR0::value_cg	100.00	100.00	1	100	1	1	64	64	
reg_field_pkg::DDR2GB_DDR_ADDR1::value_cg	100.00	100.00	1	100	1	1	64	64	
reg_field_pkg::DDR2GB_GB_ADDR::value_cg	100.00	100.00	1	100	1	1	64	64	
reg_field_pkg::GB2LB_SRC0::value_cg	100.00	100.00	1	100	1	1	64	64	
reg_field_pkg::GB2LB_SRC1::value_cg	100.00	100.00	1	100	1	1	64	64	
reg_field_pkg::GB2LB_DEST::value_cg	100.00	100.00	1	100	1	1	64	64	
reg_field_pkg::APE_CTRL::value_cg	100.00	100.00	1	100	1	1	64	64	
reg_field_pkg::APE_SRC::value_cg	100.00	100.00	1	100	1	1	64	64	
reg_field_pkg::APE_SRC::value_cg	100.00	100.00	1	100	1	1	64	64	
reg_field_pkg::APE_DEST::value_cg	100.00	100.00	1	100	1	1	64	64	
reg_field_pkg::LPE_SRC0::value_cg	100.00	100.00	1	100	1	1	64	64	
reg_field_pkg::LPE_SRC1::value_cg	100.00	100.00	1	100	1	1	64	64	
reg_field_pkg::LPE_DEST0::value_cg	100.00	100.00	1	100	1	1	64	64	
reg_field_pkg::LPE_DEST1::value_cg	100.00	100.00	1	100	1	1	64	64	
reg_field_pkg::LPE_MODE::value_cg	100.00	100.00	1	100	1	1	64	64	
reg_field_pkg::LPE_LEAP0::value_cg	100.00	100.00	1	100	1	1	64	64	
reg_field_pkg::LPE_LEAP1::value_cg	100.00	100.00	1	100	1	1	64	64	
reg_field_pkg::LPE_LEAP2::value_cg	100.00	100.00	1	100	1	1	64	64	
reg_field_pkg::LPE_LEAP3::value_cg	100.00	100.00	1	100	1	1	64	64	
reg_field_pkg::LPE_LEAP4::value_cg	100.00	100.00	1	100	1	1	64	64	
reg_field_pkg::LPE_LEAP5::value_cg	100.00	100.00	1	100	1	1	64	64	
reg_field_pkg::LPE_LEAP6::value_cg	100.00	100.00	1	100	1	1	64	64	
reg_field_pkg::LPE_LEAP7::value_cg	100.00	100.00	1	100	1	1	64	64	
reg_field_pkg::LPE_LEAP7::value_cg	100.00	100.00	1	100	1	1	64	64	
reg_field_pkg::WBLOAD_DDR_ADDR0::value_cg	100.00	100.00	1	100	1	1	64	64	
reg_field_pkg::WBLOAD_DDR_ADDR1::value_cg	100.00	100.00	1	100	1	1	64	64	
reg_field_pkg::WBLOAD_KERNEL::value_cg	100.00	100.00	1	100	1	1	64	64	
reg_field_pkg::WBLOAD_CHANNEL::value_cg	100.00	100.00	1	100	1	1	64	64	
reg_field_pkg::COMP_CTRL::value_cg	100.00	100.00	1	100	1	1	64	64	
reg_field_pkg::PRECISION::value_cg	100.00	100.00	1	100	1	1	64	64	
reg_field_pkg::GLB_INTR::value_cg	100.00	100.00	1	100	1	1	64	64	
reg_field_pkg::GLB_ENABLE_ROW::value_cg	100.00	100.00	1	100	1	1	64	64	
reg_field_pkg::GLB_ENABLE_COL::value_cg	100.00	100.00	1	100	1	1	64	64	



手机淘宝扫一扫

淘宝唯一店铺：数字ic资料店

reg_field_pkg::GLB_ENABLE_ROW::value_cg	100.00	100.00	1	100	1	1	64	64
reg_field_pkg::GLB_ENABLE_COL::value_cg	100.00	100.00	1	100	1	1	64	64
reg_field_pkg::BUF_MASKX::value_cg	100.00	100.00	1	100	1	1	64	64
reg_field_pkg::BUF_MASK0::value_cg	100.00	100.00	1	100	1	1	64	64
reg_field_pkg::BUF_MASK1::value_cg	100.00	100.00	1	100	1	1	64	64
reg_field_pkg::BUF_MASK2::value_cg	100.00	100.00	1	100	1	1	64	64
reg_field_pkg::BUF_MASK3::value_cg	100.00	100.00	1	100	1	1	64	64
reg_field_pkg::BUF_MASK4::value_cg	100.00	100.00	1	100	1	1	64	64
reg_field_pkg::BUF_MASK5::value_cg	100.00	100.00	1	100	1	1	64	64
reg_field_pkg::BUF_MASK6::value_cg	100.00	100.00	1	100	1	1	64	64
reg_field_pkg::BUF_MASK7::value_cg	100.00	100.00	1	100	1	1	64	64



图 4-24

4.2.4 IP 级的功能覆盖率

IP 级的验证我仿真了几种典型的情况。我主要仿真了 3x3、5x5、7x7 和 11x11 的卷积核，每一种长度的卷积核上都会仿真 stride 为 1 和 2 的情况，并且在 stride 为 2 时，会加 padding。所以会有 8 种组合方式，其中每种组合方式下又会分为 16bit 精度和 8bit 精度两种情况。

input feature 和 weight 的两种精度的覆盖情况：

Bins		
NAME	COUNT	AT LEAST
ifmap_precision_16	8	1
ifmap_precision_8	8	1

Bins		
NAME	COUNT	AT LEAST
weight_precision_16	8	1
weight_precision_8	8	1

图 4-25

stride 为 1 和 2 的覆盖情况：

Bins		
NAME	COUNT	AT LEAST
hori_stride_01	8	1
hori_stride_02	8	1

Bins		
NAME	COUNT	AT LEAST
vert_stride_01	8	1
vert_stride_02	8	1

padding 的覆盖情况:

Bins		
NAME	COUNT	AT LEAST
pad_up_0	8	1
pad_up_1	8	1

Bins		
NAME	COUNT	AT LEAST
pad_left_0	10	1
pad_left_1	6	1

图 4-27

不同长度的 kernel 的覆盖情况:

Covered bins		
NAME	COUNT	AT LEAST
kernel_hori_len_03	4	1
kernel_hori_len_05	4	1
kernel_hori_len_07	4	1
kernel_hori_len_0b	4	1

Covered bins		
NAME	COUNT	AT LEAST
kernel_vert_len_03	4	1
kernel_vert_len_05	4	1
kernel_vert_len_07	4	1
kernel_vert_len_0b	4	1

图 4-28



手机淘宝扫一扫

淘宝唯一店铺：数字ic资料店