**机器翻译大作业**

面向机器翻译的RNN加速器的设计与实现

学 生 班 级：计算机硕士2004班

学 生 姓 名：郑涵中

学 生 学 号：2001882

二○二〇年十二月

# **目 录**

[目 录 I](#_Toc23259)

[1 引言 1](#_Toc28387)

[1.1 研究背景和意义 1](#_Toc12479)

[1.2 本次大作业主要工作 1](#_Toc26751)

[1.3 本次大作业结构 2](#_Toc9141)

[2 相关理论技术 3](#_Toc31589)

[2.1 量化神经网络 3](#_Toc1614)

[2.2 Normalization 4](#_Toc26076)

[2.3 语言模型 5](#_Toc25544)

[2.3.1 定义 5](#_Toc31728)

[2.3.2 评价标准 6](#_Toc1694)

[2.4 Vivado HLS 6](#_Toc4687)

[2.4.1 高层次综合HLS 6](#_Toc15488)

[2.4.2 逻辑综合与布局布线 7](#_Toc9424)

[3 加速器结构 8](#_Toc2853)

[3.1 整体结构 8](#_Toc16012)

[3.1.1 处理单元（Processing Element） 8](#_Toc25747)

[3.1.2 隐藏层（HiddenLayer）的配置 9](#_Toc7895)

[3.1.3 存储模块（ROM，FIFO） 10](#_Toc15301)

[3.1.4 输出层（Outputlayer） 12](#_Toc21055)

[3.2计算流程 12](#_Toc29622)

[3.3数据精度 13](#_Toc6799)

[4 具体实现 15](#_Toc5752)

[4.1 顶层模块 15](#_Toc23251)

[4.2 计算模块（DoCompute） 16](#_Toc2355)

[4.2.1 存储模块 19](#_Toc24753)

[4.2.2 隐藏层模块的实现 26](#_Toc29579)

[4.2.3 输出层模块 31](#_Toc27504)

[5 实验分析 35](#_Toc15639)

[5.1实验流程 35](#_Toc31757)

[5.2设计结果分析 37](#_Toc17306)

[6 实验结论 39](#_Toc16145)

[7 课程总结及课堂建议 40](#_Toc30727)

[7.1 课程总结 40](#_Toc9735)

[7.1.1 基于循环神经网络的机器翻译模型 40](#_Toc16528)

[7.1.2 基于卷积神经网络的机器翻译模型 41](#_Toc13769)

[7.1.3 Transformer模型 42](#_Toc12374)

[7.2 课堂建议 43](#_Toc30709)

[附录 44](#_Toc26996)

[附录1 44](#_Toc12620)

[附录2 49](#_Toc25720)

# 1 引言

长短期记忆循环神经网络（LSTM-RNNs）被广泛应用在语言识别，机器翻译等自然语言处理领域中，但是CPU和GPU平台的RNN应用由于架构的局限性，无法实现高效的计算。基于FPGA的循环神经网络加速器凭借其高能效，高性能以及相比于ASIC巨大的灵活性成为最适合实现这些RNN应用的平台。

## 1.1 研究背景和意义

循环神经网络（RNN）尤其是长短期记忆网络（LSTM）在许多应用上取得了相当显著的效果，例如机器翻译语言模型，语音识别以及图像理解等其他应用。然而，卓越的效果是以复杂的计算，巨大的存储空间和位宽为代价的，这就导致这些应用的具体部署变得更加具有挑战性，尤其是在资源受限的平台上例如便携设备等。而且，大部分应用有着很严格的实时性要求例如移动机器人，实时翻译机器以及自动驾驶汽车等。

相较于一般的前馈神经网络，LSTM神经网络更加有挑战性因为需要保存上一程的状态输入到下一个步骤中。这会对架构的设计产生一系列的影响，首先，需要对重复连接以及“前馈”输入激活进行额外的处理。此外，即使所需的状态内存不是特别大，状态保持会创建与前面步骤的数据依赖关系，这将强制地将不同模块的处理步骤序列化并限制了可在可定制体系结构中利用的并行度，这使得多核通用计算平台效率低下。目前已经提出了许多技术来缓解上述的计算和存储挑战。 其中最常见的是剪枝和量化或其组合。所有这些都是基于神经网络中典型的固有冗余，这意味着参数的数量和运算的精度可以显著降低，而不影响精度。

在本次大作业中，我们通过降低数据类型的精度来降低计算成本和存储需求，利用现场可编程门阵列（FPGAs），因为它们是唯一允许以可编程方式在数据类型精度水平上定制流水线计算数据路径和存储器子系统的计算平台。 因此，它们可以最大限度地利用所提出的优化技术。

## 1.2 本次大作业主要工作

为了提高加速器的性能，通常会对模型进行一定程度的压缩，本次大作业采用的量化的方法，然而由于神经网络的固有性质，量化后的模型依然很大。针对这一问题，本次大作业主要研究了从硬件实现的角度，如何用Normalization处理量化的模型以获得更大程度的压缩。为验证这一思路的可行性，本次大作业通过Vivado的HLS高层次综合工具在zynq-7000平台上实现了一个基于LSTM的语言模型的加速器，针对LSTM神经网络的前向传播。本次大作业具体描述了加速器的可配置的并行计算结构，以及为了节省计算资源用混合精度的定点计算代替浮点计算。报告结合具体平台给出了各个模块的具体实现细节。报告最后利用ptb数据集训练了一个量化的语言模型，将模型部署在硬件实现上后，利用Vivado的综合和布局布线工具以及仿真工具验证设计结果的正确性，并对比两者之间的误差。发现对比CPU平台误差完全在可接受的范围内，证明了Normalization应用在硬件实现的可行性，而且功耗较低。

## 1.3 本次大作业结构

本次大作业在第二章中描述了用Normalization处理量化的模型得到更大程度的压缩。为验证这一思路的可行性，通过Vivado的HLS高层次综合工具在zynq-7000平台上实现了一个基于LSTM的语言模型的加速器，针对LSTM神经网络的前向传播。在第三章中描述了加速器的可配置的并行计算结构结构，以及为了节省计算资源用混合精度的定点计算代替浮点计算。在第四章中给出了各个模块的具体实现细节。在第五章中，利用ptb数据集训练了一个量化的语言模型，将模型部署在硬件实现上后，利用Vivado的综合和布局布线工具以及仿真工具验证设计结果的正确性，并对比两者之间的误差。

# 2 相关理论技术

## 2.1 量化神经网络

最极端的量化方法是二值化，其结果是二值化神经网络（BNNs）仅具有二值化的权值，或者具有权值和激活值均被量化为1比特，与32位全精度对等体相比，二进制化权值大大减少了内存大小和访问次数。 同时，BNN通过用廉价的位向XNOR和位计数操作代替代价高昂的实值加权和激活之间的算术操作，大大降低了硬件的复杂度，从而大大提高了硬件的加速性能和功耗。结果表明，在某些应用中，即使是1位二进制化也能获得相当好的性能。然而，在某些应用中，与全精度网络相比，权值和激活值的二值化会导致不希望的和不可接受的精度降低。为了弥补这一差距，最近的工作采用了具有更多比特的量化，如三进制f-1，0，1g和低比特宽度网络，从而实现了更好的性能，在实现成本和精度之间实现了有益的折衷。

在所有已有的关于量化和压缩的工作中，大部分集中在前馈神将网络上，而对RNNs的关注较少。 后者的量化则更具挑战性。众所周知，由于长期依赖关系，RNN存在梯度爆炸和梯度消失问题。 为了避免梯度消失，LSTM采用了self-connection机制计算公式如下。



其中是当前步骤的输入是权重矩阵是偏置向量。和tanh是逐个激活的非线性函数sigmoid和tanh。

然而，LSTM依然存在着梯度爆炸的问题 ，在每一个反向传播步骤中，这个梯度的大小是由一个由LSTM权值的范数所限定的数所控制，假定损失函数为，T为展开的时间步长数，m是当前的步长。在反向传播的过程中，首先会计算和，然后从和反向传播到，考虑相邻两个步长间的，得到以下结论：



 *，*

**

当=0时通过推导，对任意t之前的步长即任意p<t，。因此，当=0，>1时反向传播的梯度将会呈梯度增长。然而，根据经验极少情况下为0所以的上界一般情况下会更大，更容易发生梯度爆炸。然而对于量化后的LSTM，量化的程度越大其对应矩阵的2范数越大，也就越容易发生梯度爆炸。

## 2.2 Normalization

Normalization是为了防止出现梯度爆炸和梯度消失对传播数据的范围进行限定。Normalization的计算公式为 ：



量化后的LSTM为：（N表示Normalization）



E和Var为Normalization方向上的均值和方差γ与β为可学习的仿射变换参数是为防止分母变0而加入的常数通常为1e-5。以上的归一化方法已被发现在改进深度网络训练和性能方面是有用的。 虽然批处理归一化最初仅限于前馈网络，但最近已扩展到LSTMs。最近，Ardakani等人使用这个扩展来训练二值化/三值化的LSTM，并获得与全精度基线相当的性能。 同时weight Normalization化和batch Normalization也被验证可以得到很好的效果。此外，Normalization需要为每个时间步存储全部不的计算结果来进行一系列的求均值和求方差的计算来进行统一的Normalization操作，会导致额外的计算代价，这很有可能比量化的LSTM模型本身花费更多的存储。尤其batch Normalization需要处理每一个步长期间一个batch内的所有数据，而且batch太小Normalization的效果将会很差。然而layer Normalization的实现代价可能会更小因为layer Normalization不是在batch这个维度上进行Normalization，存储的中间计算结果数目因此会是batch Normalization的batch size分之一效率更高。

本次大作业将利用这种方法设计一个量化LSTM神经网络的FPGA平台上的硬件实现以达到加速的目的。但是会在在硬件平台增加额外的计算，本次大作业把该设计应用到[lun]实现的ptb数据集语言模型中去对算法的可行性进行了验证。

## 2.3 语言模型

### 2.3.1 定义

标准定义：对于语言序列 ，语言模型就是计算该序列的概率，即 。从机器学习的角度来看：语言模型是对语句的概率分布的建模。通俗解释：判断一个语言序列是否是正常语句，即是否是更符合对应语言的使用习惯，例如：P(I am right)>P(I right am) 。

语言模型是很多自然语言处理应用的基石，非常多自然语言处理应用的技术都是基于语言模型。语言模型的任务就是预测每个句子在语言中出现的概率。本次大作业实现的语言模型是基于ptb数据集的，而且是基于字符级的语言模型。

### 2.3.2 评价标准

1.混淆度 (Perplexity) 用来衡量一个语言模型在未见过的的字符串S上的表现。对于一个长度为N的字符串S，语言模型给出概率P(S)，对应的混淆度 (Perplexity)为 2^{-(1/N) log2 P(S)}。其中字符串长度单位可以是字符 (characters) 也可以是单词 (words)。字符串的内容是用来测试的用例。

2.bits-per-character (bpc)，当计算基于字符长度单位的混淆度 (Perplexity)时，Perplexity = 2^bpc.bpc和ppl值越小越好。Perplexity和bpc本质上都是对比模型计算出来的预测序列的概率分布和实际的概率分布，并通过交叉熵来衡量这两个分布的相似程度

## 2.4 Vivado HLS

设计定义阶段主要进行方案验证、系统设计和FPGA芯片选型等准备工作。根据任务要求，评估系统的指标和复杂度，对工作速度和芯片本身的资源、成本等方面进行权衡，选择合理的设计方案和合适的器件类型。代码实现阶段是将划分好的各功能模块用硬件描述语言表达出来，常用的硬件描述语言有Verilog和VHDL。HLS工具可以在一定程度上把c语言代码映射成硬件语言实现来降低开发周期。

### 2.4.1 高层次综合HLS

HLS（High Level Synthesis，高层次综合）是一种代码的综合技术，可以把特定编程风格的c语言代码映射成rtl级的设计。开发进度快适用于设计架构的快速验证。因为从C代码中，编译器往往得不到足够的信息来完成优化工作，譬如数据的输入输出采用何种接口，程序内部的计算并行度是多少以及计算过程中是否需要进行流水线设计。这一些无法在C代码中指定的内容，可以采用Pragma进行指定，以对综合过程进行更高的控制。然而也并不是所有的C语言代码都可以被设计成rtl的设计，使用C代码时依然要有硬件的实现考虑到硬件的实现，并且符合rtl设计时的基本逻辑和思想。

### 2.4.2 逻辑综合与布局布线

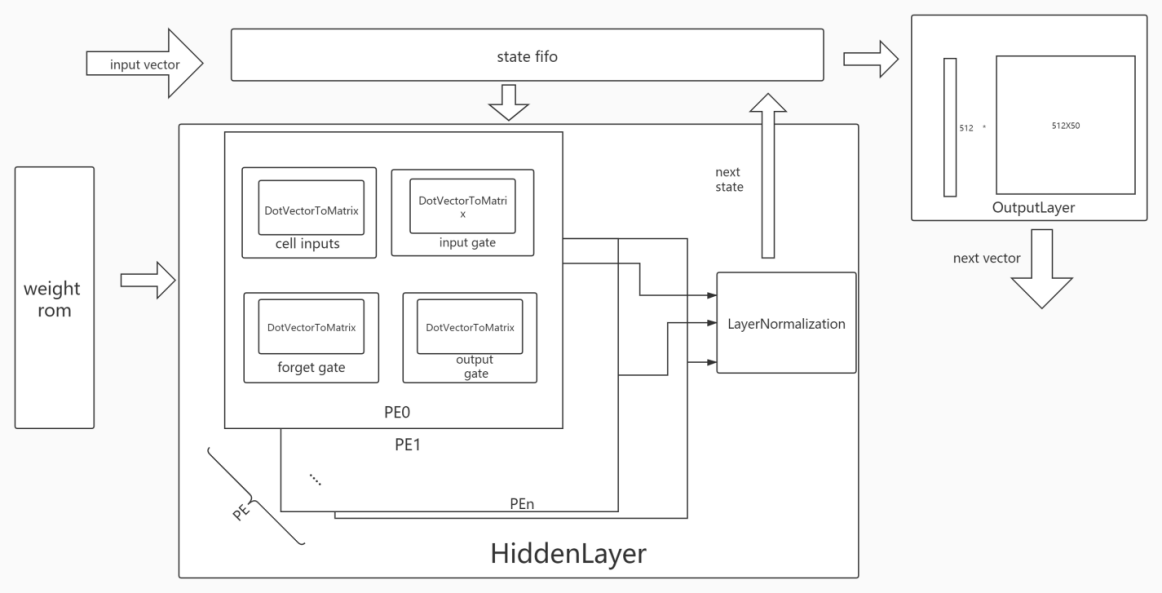
逻辑综合的概念是：将高级抽象层次的语言描述转化成较低层次的电路结构。也就是说将硬件描述语言描述的电路逻辑转化成与门、或门、非门、触发器等基本逻辑单元的互连关系，也就是我们常说的门级网表。利用Verilog 做电路设计时需要强调可综合的概念，可综合代码可以被翻译成门级电路，不可综合就是代码不能被翻译成与之对应的门级电路。综合是创造性的转化过程，它不但能翻译电路，还能够根据我们的需求优化电路，比如去除电路描述中冗余的电路结构，或者复用功能相同的电路结构。Verilog的逻辑综合和HLS的综合本质上市一样的都是从一个高层级映射到低层级的实现。

综合后生成的门级网表只是表示了门与门之间虚拟的连接关系，并没有规定每个门的位置以及连线长度等。布局布线就是一个将门级网表中的门的位置以及连线信息确定下来的过程。 综合后生成的门级网表只是虚拟的连接关系，并没有反应到实物上去。布局布线有时也叫作实现与布局布线，布局布线将逻辑网表中的门级连接关系配置到FPGA芯片内部的固有硬件结构上。布局布线过程将每一个门实际配置到固定位置的可编程逻辑块（CLB）中。是相对于综合来说更加具体的实现，布局布线后经过仿真验证就可以经行板级调试了。

# 3 加速器结构

## 3.1 整体结构

下图展示了LSTM加速器的硬件实现结构，主要模块有隐藏层模块，输出层模块，存放权重和状态的存储器。隐藏层计算模块主要用来实现神经网络中隐藏层的计算，以及Normalization处理。归一化处理前的计算以PE（processing element）为单位进行展开，PE内部的计算时并行执行的，通过调整PE的大小即可调整计算的并行程度，增加PE会增加并行性，导致更高的硬件使用量和更低的延迟，这种灵活性允许根据目标设备上的可用资源调整并行性。PE并行计算得到所有矩阵计算的结果，经过Normalization处理后整理成HiddenLayer的计算结果即，将其都写入FIFO中，输出层读取后进行计算，输出层内部的计算依然采用并行计算的结构。本次大作业实现的语言模型只有一层隐藏层，若是更复杂的结构可以根据需求增加隐藏层模块的数量或者共享一个模块。



### 3.1.1 处理单元（Processing Element）

PE内部的计算是由4个DotVextorToMatrix模块完成的，每个模块负责LSTM结构中的一个门，DotVextorToMatrix模块的实现也是采用可参数化并行的结构实现的，输入的状态和输入的字符向量分别按SIMD\_INPUT和SIMD\_RECURRENT用pragma HLS ARRAY展开，每次并行计算SIMD\_INPUT和SIMD\_RECURRENT个乘法，SIMD\_INPUT和SIMD\_RECURRENT都是可以通过隐藏层进行配置。称一次这样的计算为SIMD，同时每次SIMD计算操作之间采用pipeline进行优化。如下图所示

### P

流水线用HLS pragma pipeline来实现，HLS会默认选择时延最少的结构去实现。

### 3.1.2 隐藏层（HiddenLayer）的配置

通过模板函数的形式实现该模块的配置，相关属性有：

#### 并行度

unsigned int PE, Number of neurons to be executed in parallel  
unsigned int SIMD\_INPUT, Number of parallel MAC performed in the gates on input pixels  
unsigned int SIMD\_RECURRENT, Number of parallel MAC performed in the gates on recurrent path

#### 类型和位宽

typename Vector\_t, Type of the char vector  
unsigned int VectorWidth, number of bits of each char

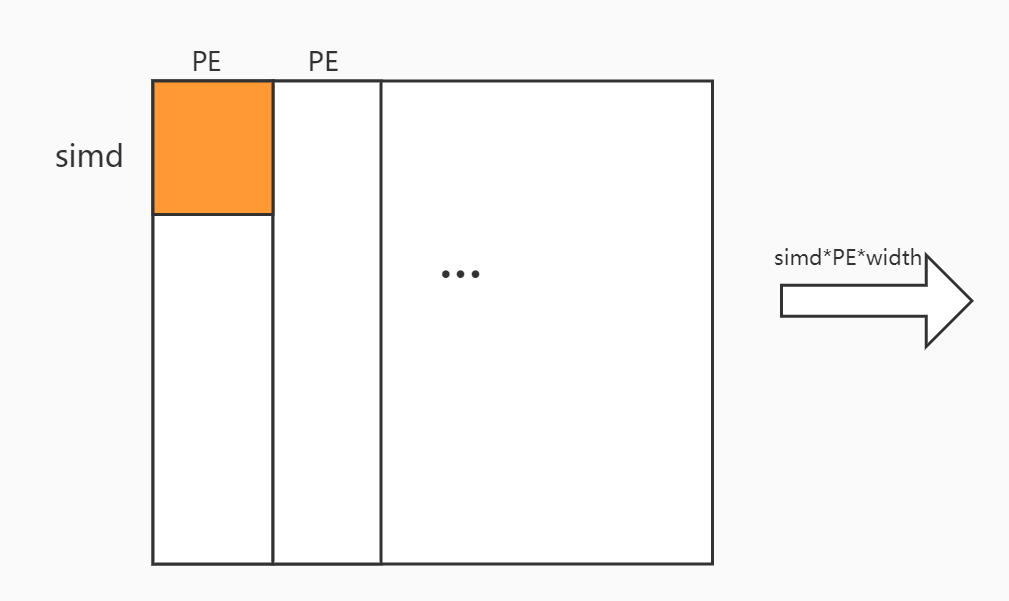
typename Bias\_t\_gi, Type of the bias for gate i  
unsigned int BiasWidth\_gi, number of bits of each   
typename Weight\_t\_gi, Type of the weights for   
unsigned int WeightWidth\_gi, number of bits of each   
typename DotProductResult\_t\_gi, type of the result for MAC with weight of gate i   
typename gix\_accumulator\_t,

typename Bias\_t\_gf, Type of the bias for gate f  
unsigned int BiasWidth\_gf, number of bits of each bias (gate f)  
typename Weight\_t\_gf, Type of the weights for gate f  
unsigned int WeightWidth\_gf, number of bits of each weight (gate f)  
typename DotProductResult\_t\_gf, type of the result for MAC with weight of gate f  
typename gfx\_accumulator\_t,  
typename Bias\_t\_go, Type of the bias for gate o  
unsigned int BiasWidth\_go, number of bits of each bias (gate o)  
typename Weight\_t\_go, Type of the weights for gate o  
unsigned int WeightWidth\_go, number of bits of each weight (gate o)  
typename DotProductResult\_t\_go, type of the result for MAC with weight of gate o  
typename gox\_accumulator\_t,  
typename Bias\_t\_ci, Type of the bias for gate ci  
unsigned int BiasWidth\_ci, number of bits of each bias (gate ci)  
typename Weight\_t\_ci, Type of the weights for gate ci  
unsigned int WeightWidth\_ci, number of bits of each weight (gate ci)  
typename DotProductResult\_t\_ci, type of the result for MAC with weight of gate ci  
typename gi\_ci\_accumulator\_t,  
typename OutputActivation\_t,  
unsigned int OutputActivationWidth,  
typename VectorSize\_t,  
unsigned int VectorSize,  
typename NumberHiddenUnits\_t,  
unsigned int NumberHiddenUnits,  
unsigned int MaxNumberColumns,  
typename State\_t,  
typename Sigmoid\_out\_t, 查找表内数据的类型，即查找表的输出类型  
unsigned int Lut\_Entries\_Sigmoid, SIGMOID 查找表的长度  
typename Sigmoid\_limit\_t, 查找表的limit的数据类型  
typename Sigmoid\_step\_t, 查找表的recip\_step的数据类型  
typename Tanh\_out\_t,   
unsigned int Lut\_Entries\_Tanh, Tanh 查找表的长度  
typename Tanh\_limit\_t,   
typename Tanh\_step\_t,  
typename t\_norm\_weights,  
typename t\_norm\_bias

### 3.1.3 存储模块（ROM，FIFO）

存储模块用来存放权重参数和传输结果，参数是通过数组的形式来实现，加了const关键字以后，HLS进行综合的时候会吧权重和偏置等参数被放到ROM中。输入和状态都被综合成FIFO用于数据传输，通过hls::stream类来实现。

要使得并行的计算结构有效，相应的存储器的位宽也应该满足并行计算是的读取要求，否则数据的读取会成为限制计算速度的瓶颈。因此ROM的输出位宽如下图：



用到的pragma有：

##### ****ARRAY\_PARTITION****

将数组分块，存储数组时将数组存储在不同的memory（RAM/ROM/FIFO）中，提供有不同维度的的不同划分方式，增加了吞吐量但是花费更多资源。

##### ****ARRAY\_MAP****

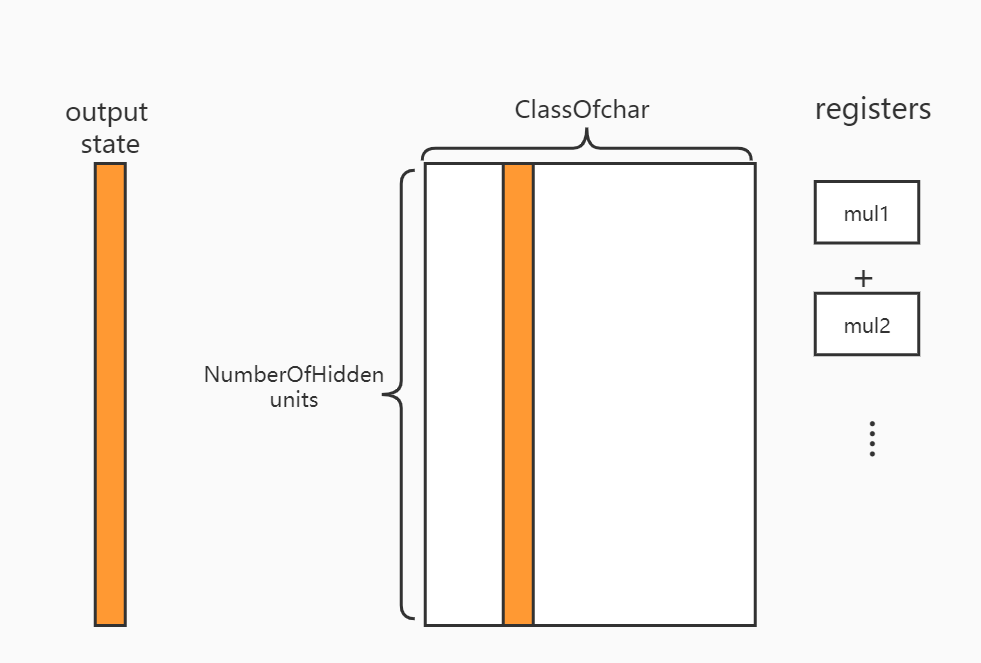
将不同的数组的元素进行拼接，横向拼接保持位宽不变所以对数据的吞吐率并没有改善纵向映射就是把对应位置的元素做位拼接，最终的数组元素个数等于之前各个小数组中最长的数组的元素个数，但是位宽是对应位置各个数组元素位宽之和。

##### ****ARRAY\_RESHAPE****

ARRAY\_RESHAPE 相当于是将 ARRAY\_PARTITION 和纵向的 ARRAY\_MAP 结合在一起的，这样能降低资源消耗，同时也有利于数据的并行访问。

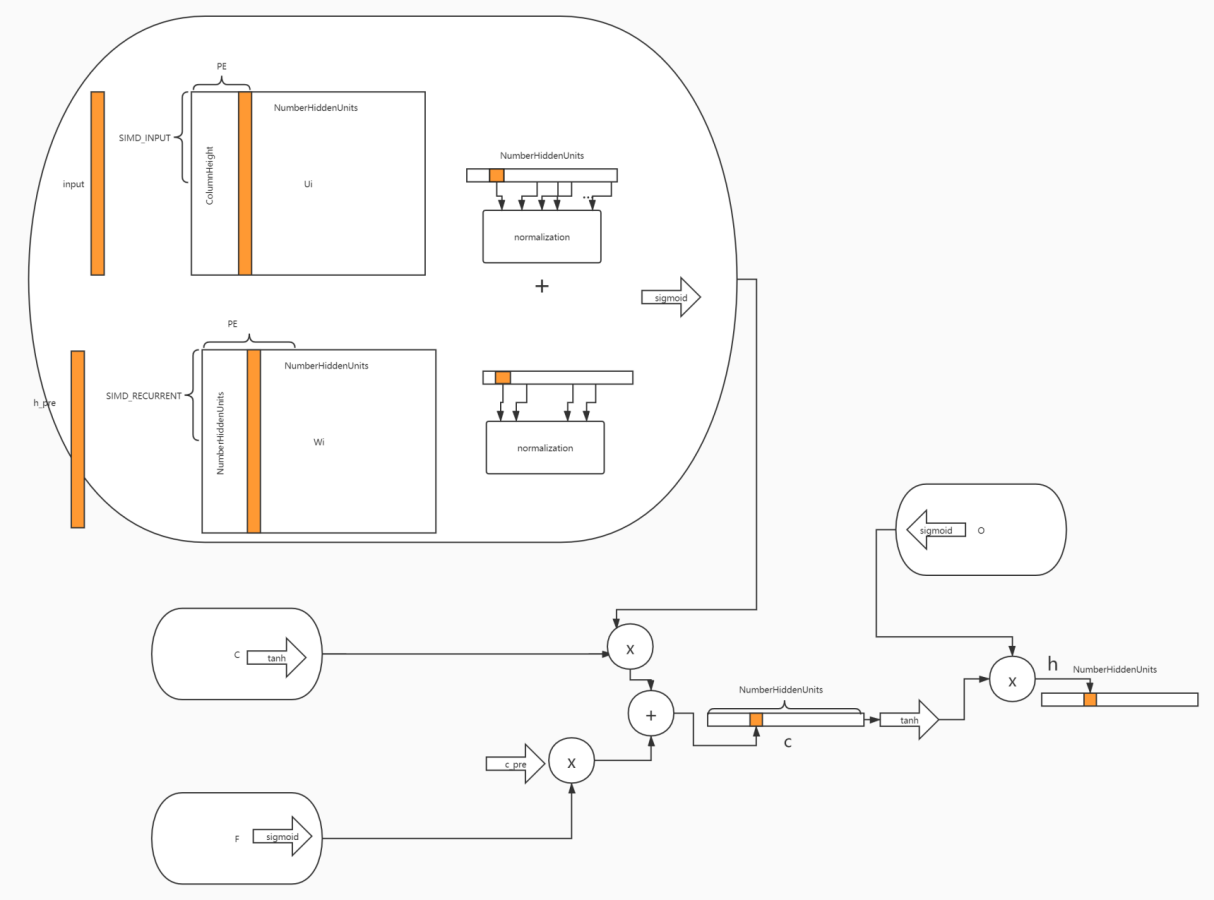
### 3.1.4 输出层（Outputlayer）

输出向量长度为512，全连接矩阵为512x50。计算过程被展开为50个并行的计算进程（CE），用mul来存储乘法结果向量长度为512，同时为了加快并行计算速度，将mul用512个寄存器来实现。



## 3.2计算流程

整体的计算流程如下图所示，LSTM内部的计算都是一些矩阵乘法的计算，矩阵乘法得到的结果向量经过Normalization处理后再执行进行激活再经过一系列的运算后得到下一个状态作为循环。4个门内部的计算比较相似用input gate举例，并行计算的部分就是SIMD\_INPUT\*PE，得到输出结果的一部分，并行的程度越大即SIMD\_INPUT\*PE越大计算的速度就越快同时消耗的资源也就越大，目前4个门的计算也是并行的，这些计算占总使用资源的85%以上，这也是是导致设计所消耗的资源较大的最主要的原因。



## 3.3数据精度

最初的语言模型采用的是浮点数据类型的计算，然而在硬件实现的时候会消耗很多计算资源和存储空间，所以选择用定点数据类型来替代，相应的计算也用定点计算来实现，但是这会不可避免的造成精度上的损失，经过实验发现采用定点数据类型造成的误差在可以接受的范围内，而且，softmax得到的结果有微小的误差并不会对模型的性能有巨大的影响。对于不同的数据会有不同的精度，没有采用统一的精度。例如输出层的计算比较总要对精度的要求比较高，输出层内部的计算就采用较高的精度，隐藏层中的矩阵计算机采用相对低的精度。同时，由于激活函数的计算比较复杂，直接用查找表来近似实现。经过不停的实验后，在保证无溢出的情况下采用的方案如下表所示：

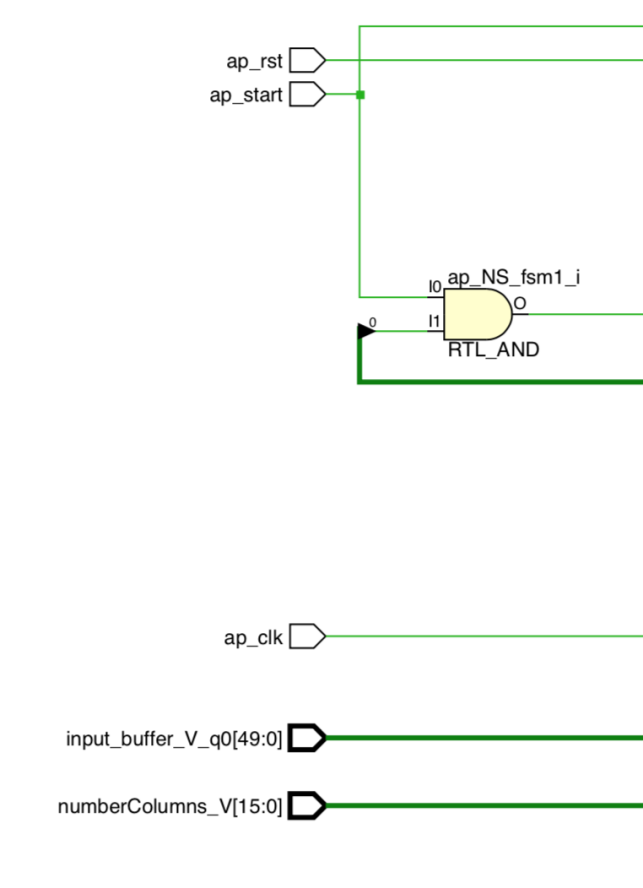
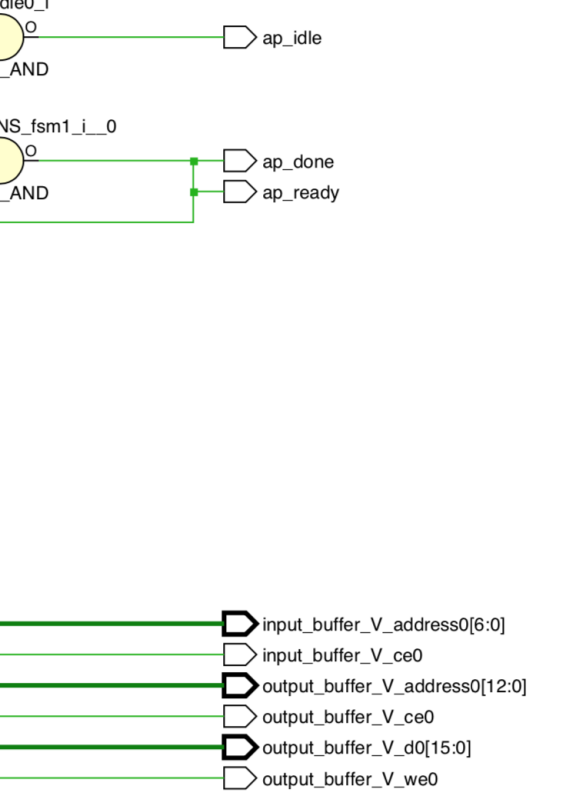
| 数据 | 位宽 | 数据类型 |
| --- | --- | --- |
| gate权重矩阵 | 2bit | 整型 |
| 偏置 | 8bit | 定点数（1位整数位） |
| Normalization权重 | 8bit | 定点数（2位整数位） |
| Normalization偏置 | 8bit | 定点数（2位整数位） |
| gate内矩阵乘法的乘积中间结果 | 输入：3bit ，h状态：14bit | 整数，定点数（7位整数位） |
| gate内矩阵乘法的求和中间结果 | 14bit | 定点数（7位整数位） |
| h状态 | 9bit | 定点数（3位整数位） |
| c状态 | 9bit | 定点数（3位整数位） |
| 字符向量 | 1bit | 无符号整数 |
| 输出层 | 16bit | 定点数（8位整数位） |
| sigma激活函数 | 8bit | 无符号定点数（1位整数位） |
| tanh激活函数 | 8bit | 定点数（2位整数位） |
| 输出层的权重和偏置 | 8bit | 定点数（1位整数位） |
| Normalization均值 | 16bit | 定点数（8位整数位） |
| Normalization标准差 | 16bit | 定点数（8位整数位） |

# 4 具体实现

## 4.1 顶层模块

输入信号包括时钟和控制信号和数据输入，input\_buffer输入字符向量，numberColumns输入要计算的序列长度，数据的输入方式采取从RAM读取的方式，这是因为c代码的顶层函数用数组做参数会默认综合成这种形式，因此输出信号中有对应的input\_buffer地址线和使能信号。除此之外还有输出的控制信号（空闲，完成和就绪），同样，输出也是以写RAM的方式输出，所以输出有使能信号和写信号以及输出数据。输出的数据是预测的概率分布，对应一个字符的概率位宽为16。

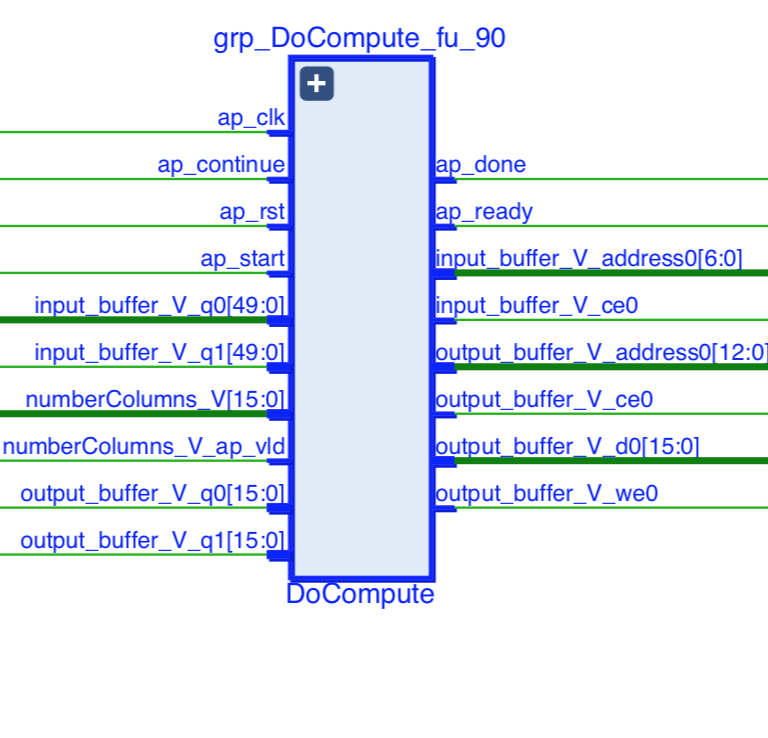
顶层模块是对计算模块的一个简单封装，没有实际功能。

输入信号 输出信号

## 

## 4.2 计算模块（DoCompute）

接口类型和顶层模块基本一致，但是计算模块支持双端口的数据输入和输出，这里只用到了单端口。对应结构图中的存储模块，隐藏层模块和输出层模块均在这里实现。对应的c代码实现：



**void** DoCompute(ap\_uint<16> numberColumns,  
 ap\_uint<32> numberBytesRead,  
 ap\_uint<DATAWIDTH> \*input\_buffer,  
 t\_fixed\_sum\_fc \*output\_buffer)  
  
{  
#pragma HLS ALLOCATION instances=DSP48 limit=160 core  
 **const unsigned int** StreamPerColumn = (SIZE\_OF\_VECTOR\*VECTORWIDTH) / DATAWIDTH + (((SIZE\_OF\_VECTOR\*VECTORWIDTH) % DATAWIDTH)>0); *// CEILING* **const unsigned int** BitPadding = StreamPerColumn\*DATAWIDTH - SIZE\_OF\_VECTOR\*VECTORWIDTH;  
 **const unsigned int** LastStreamBits = DATAWIDTH - BitPadding;  
 #pragma HLS DATAFLOW  
 HLS::stream<ap\_uint<DATAWIDTH> >output\_stream\_dma\_input(**"output\_stream\_dma\_input"**);  
#pragma HLS STREAM variable=output\_stream\_dma\_input depth=2  
  
 HLS::stream<ap\_uint<SIZE\_OF\_VECTOR\*VECTORWIDTH> > output\_stream\_columns(**"output\_stream\_columns"**);  
#pragma HLS STREAM variable=output\_stream\_columns depth=2  
  
 HLS::stream<ap\_uint<DATAWIDTH \* StreamPerColumn> > stream\_column\_padded(**"stream\_column\_padded"**);  
#pragma HLS STREAM variable=stream\_column\_padded depth=2  
  
 HLS::stream<ap\_uint<OUTPUTACTIVATIONHIDDENLAYERWIDTH\*NUMBER\_OF\_NEURONS/PE> > output\_stream\_hidden\_layer(**"output\_stream\_hidden\_layer"**);  
#pragma HLS STREAM variable=output\_stream\_hidden\_layer depth=2  
  
 HLS::stream<ap\_uint<OUTPUTACTIVATIONHIDDENLAYERWIDTH \* NUMBER\_OF\_NEURONS> > output\_stream\_input\_streamer(**"output\_stream\_input\_streamer"**);  
#pragma HLS STREAM variable=output\_stream\_input\_streamer depth=2  
  
 HLS::stream<t\_fixed\_sum\_fc> output\_stream\_mac(**"output\_stream\_mac"**);  
#pragma HLS STREAM variable=output\_stream\_mac depth=2  
  
 HLS::stream<t\_fixed\_sum\_fc> output\_stream\_concatenator(**"output\_stream\_concatenator"**);  
#pragma HLS STREAM variable=output\_stream\_concatenator depth=2  
  
 HLS::stream<maxx> output\_stream\_div\_max\_per\_column(**"output\_stream\_div\_max\_per\_column"**);  
#pragma HLS STREAM variable=output\_stream\_div\_max\_per\_column depth=2  
  
 HLS::stream<ap\_uint<8> > output\_stream\_final\_labeling(**"output\_stream\_final\_labeling"**);  
#pragma HLS STREAM variable=output\_stream\_final\_labeling depth=2

ap\_uint<5056>in\_tf;  
 **for**(**int** i=0;i<numberBytesRead/8;i++){  
 in\_tf=in\_tf >> 64;  
 in\_tf(5055,5055-63)=input\_buffer[i];  
  
 }  
  
  
 **for**(**int** i=0;i<numberColumns;i++){  
 output\_stream\_columns.write(in\_tf(49+i\*50,i\*50));  
  
 }  
 HiddenLayer\_noPH  
 <PE, SIMD\_INPUT, SIMD\_RECURRENT, t\_fixed\_image, VECTORWIDTH,  
 t\_fixed\_bgi, BIASWIDTH, t\_fixed\_wgi, WEIGHTWIDTH, t\_fixed\_sum\_wgi, t\_fixed\_gix\_sum,  
 t\_fixed\_bgf, BIASWIDTH, t\_fixed\_wgf, WEIGHTWIDTH, t\_fixed\_sum\_wgf, t\_fixed\_gfx\_sum,  
 t\_fixed\_bgo, BIASWIDTH,t\_fixed\_wgo, WEIGHTWIDTH, t\_fixed\_sum\_wgo, t\_fixed\_gox\_sum,  
 t\_fixed\_bci, BIASWIDTH, t\_fixed\_wci, WEIGHTWIDTH, t\_fixed\_sum\_wci, t\_fixed\_ci\_gi\_mul,  
 t\_fixed\_recurrent, OUTPUTACTIVATIONHIDDENLAYERWIDTH,  
 ap\_uint<SIZE\_OF\_VECTOR\_TYPEWIDTH>, SIZE\_OF\_VECTOR,  
 ap\_uint<NUMBER\_OF\_NEURONS\_TYPEWIDTH>, NUMBER\_OF\_NEURONS,  
 MAX\_NUMBER\_COLUMNS\_TEST\_SET,  
 t\_fixed\_state,  
 t\_fixed\_sigma\_o, NUMBER\_OF\_LUT\_ETRIES\_SIGMOID\_1, t\_fixed\_lut\_sigmoid\_limit, t\_fixed\_lut\_sigmoid\_recip\_step,  
 t\_fixed\_tanh\_o, NUMBER\_OF\_LUT\_ETRIES\_TANH\_1, t\_fixed\_lut\_tanh\_limit, t\_fixed\_lut\_tanh\_recip\_step,t\_norm\_weights,t\_norm\_bias  
 >  
 (numberColumns, output\_stream\_columns, output\_stream\_hidden\_layer,bgi,iih\_norm\_w,iih\_norm\_b,wgi\_ih,ihh\_norm\_w,ihh\_norm\_b,wgi\_hh,bgf,fih\_norm\_w,fih\_norm\_b,wgf\_ih,fhh\_norm\_w,fhh\_norm\_b,wgf\_hh, bgo,oih\_norm\_w,oih\_norm\_b,wgo\_ih,ohh\_norm\_w,ohh\_norm\_b,wgo\_hh,bci,cih\_norm\_w,cih\_norm\_b,wci\_ih,chh\_norm\_w,chh\_norm\_b,wci\_hh, lut\_sigmoid\_1, lut\_tanh\_1);  
 StreamingDataWidthConverter\_Batch<OUTPUTACTIVATIONHIDDENLAYERWIDTH\*(NUMBER\_OF\_NEURONS/PE), OUTPUTACTIVATIONHIDDENLAYERWIDTH \* NUMBER\_OF\_NEURONS, PE>(output\_stream\_hidden\_layer, output\_stream\_input\_streamer, numberColumns);  
  
 OutputLayer  
 <  
 t\_fixed\_bfc, FCBIASWIDTH,  
 t\_fixed\_wfc, FCWEIGHTWIDTH,  
 t\_fixed\_recurrent, OUTPUTACTIVATIONHIDDENLAYERWIDTH,  
 t\_fixed\_sum\_fc, OUTPUTACTIVATIONOUTPUTLAYERWIDTH,  
 ap\_uint<NUMBER\_OF\_NEURONS\_TYPEWIDTH>, NUMBER\_OF\_NEURONS,  
 ap\_uint<NUMBER\_OF\_CLASSES\_TYPEWIDTH>, NUMBER\_OF\_CLASSES,  
 ap\_uint<MAX\_NUMBER\_COLUMNS\_TEST\_SET\_TYPEWIDTH>  
 >  
 (bfc, wfc, numberColumns, output\_stream\_input\_streamer, output\_stream\_mac);  
  
 **for**(**int** a=0;a<numberColumns\*SIZE\_OF\_VECTOR;a++){

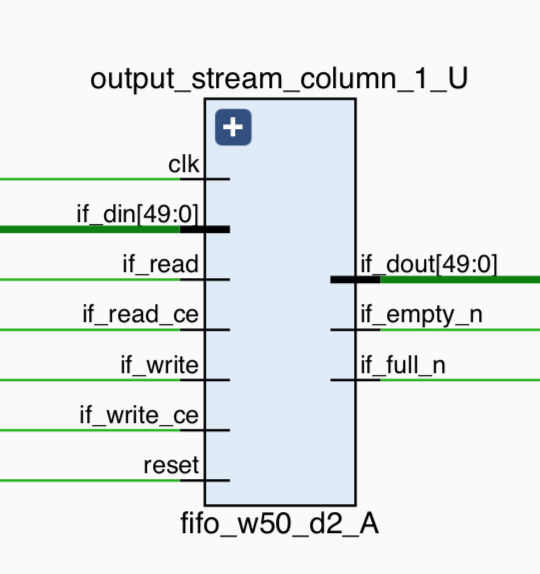
output\_buffer[a]=output\_stream\_mac.read();  
 }  
  
  
}

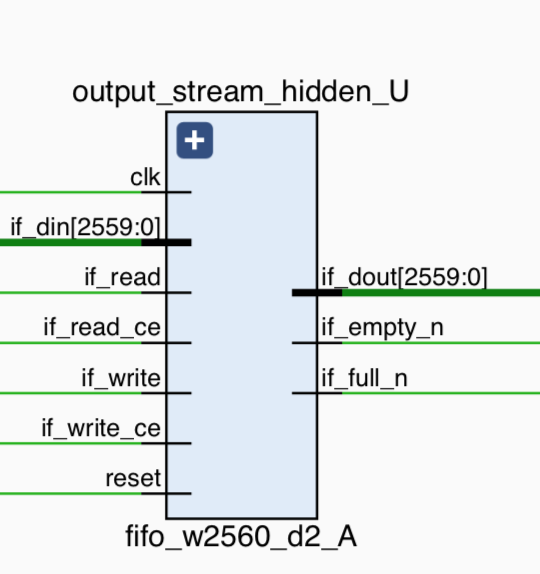
### 4.2.1 存储模块

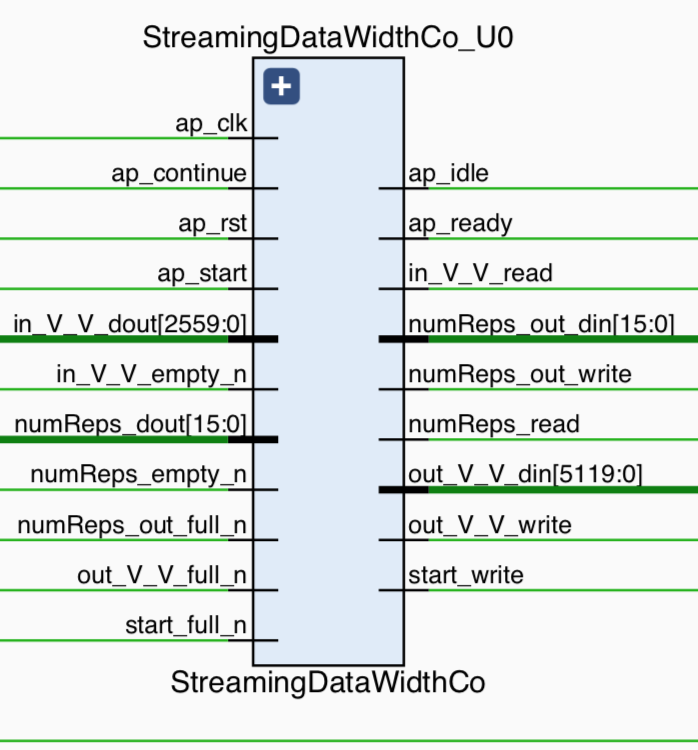
#### 1.用于数据传输的FIFO

存储模块的实现方式主要有ROM,RAM和FIFO RAM（randomaccessmemory）随机存储器。存储单元的内容可按需随意取出或存入（ROM只支持读取），且存取的速度与存储单元的位置无关的存储器。FIFO（FirstInFirstOut）是一种先进先出的数据缓存器，他与普通存储器的区别是没有外部读写地址线，这样使用起来非常简单，但缺点就是只能顺序写入数据，顺序的读出数据，其数据地址由内部读写指针自动加1完成，不能像普通存储器那样可以由地址线决定读取或写入某个指定的地址。在系统设计中，以增加[数据](http://baike.baidu.com/view/38752.htm" \t "https://blog.51cto.com/lihaichuan/_blank)传输率、处理大量数据流、匹配具有不同传输率的系统为目的而广泛使用[FIFO](http://baike.baidu.com/view/132385.htm" \t "https://blog.51cto.com/lihaichuan/_blank)存储器，从而提高了系统性能。然而，规模较大的FIFO一般都是用RAM实现的（规模特别小的FIFO才会使用寄存器实现）。

output\_stream\_column用来存放输入的字符向量，用位宽为50深度为2的FIFO来实现，数据的输入来自DoCompute的字符向量输入接口。输出到隐藏层模块中。



output\_stream\_hidden用来存放h状态，用位宽为2560深度为2的FIFO来实现，数据的输入来自隐藏层的计算结果，本次大作业实现的结构中，PE=2所以用深度为2的FIFO就足够，输出结果的数据类型位宽为10位，一个PE负责256个结果，所以位宽设置为256\*10。FIFO的输出不直接连接到隐藏层中，先将不同PE得到的结果拼接到一起。StreamingDataWidth模块负责这一功能的实现。

对应的c代码实现：

void StreamingDataWidthConverter\_Batch(stream<ap\_uint<InWidth> > & in, stream<ap\_uint<OutWidth> > & out, const unsigned int numReps)

{

    if (InWidth > OutWidth)

    {

        // emit multiple output words per input word read

        CASSERT\_DATAFLOW(InWidth % OutWidth == 0);

        const unsigned int outPerIn = InWidth / OutWidth;

        const unsigned int totalIters = NumInWords \* outPerIn \* numReps;

        unsigned int o = 0;

        ap\_uint<InWidth> ei = 0;

        for (unsigned int t = 0; t < totalIters; t++)

        {

    #pragma HLS PIPELINE II=1

            // read new input word if current out count is zero

            if (o == 0)

                ei = in.read();

            // pick output word fROM the rightmost position

            ap\_uint<OutWidth> eo = ei(OutWidth - 1, 0);

            out.write(eo);

            // shift input to get new output word for next iteration

            ei = ei >> OutWidth;

            // increment written output count

            o++;

            // wraparound indices to recreate the nested loop structure

            if (o == outPerIn)

            {

                o = 0;

            }

        }

    }

    else if (InWidth == OutWidth)

    {

        // straight-through copy

        for (unsigned int i = 0; i < NumInWords \* numReps; i++)

        {

    #pragma HLS PIPELINE II=1

            ap\_uint<InWidth> e = in.read();

            out.write(e);

        }

    }

    else

    { // InWidth < OutWidth

        // read multiple input words per output word emitted

        CASSERT\_DATAFLOW(OutWidth % InWidth == 0);

        //const unsigned int inPerOut = OutWidth / InWidth;

        const ap\_uint<8> inPerOut = OutWidth / InWidth;

        const unsigned int totalIters = NumInWords \* numReps;

        //unsigned int i = 0;

        ap\_uint<8> i = 0;

        ap\_uint<OutWidth> eo = 0;

        for (unsigned int t = 0; t < totalIters; t++)

        {

    #pragma HLS PIPELINE II=1

            // read input and shift into output buffer

            ap\_uint<InWidth> ei = in.read();

            eo = eo >> InWidth;

            eo(OutWidth - 1, OutWidth - InWidth) = ei;

            // increment read input count

            i++;

            // wraparound logic to recreate nested loop functionality

            if (i == inPerOut)

            {

                i = 0;

                out.write(eo);

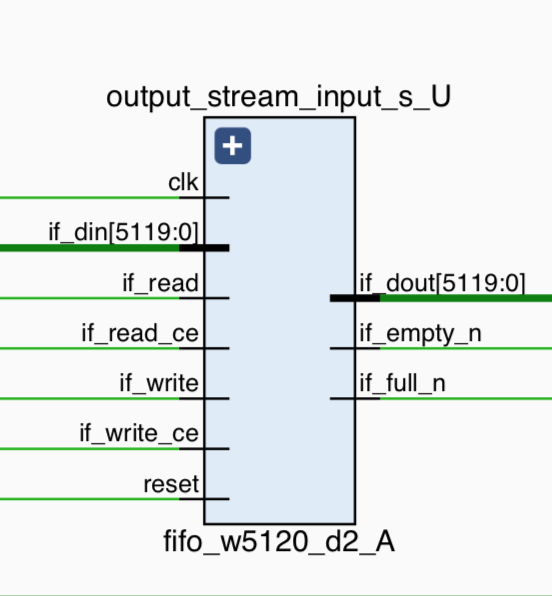
            }

        }

    }

}

output\_stream\_input



拼接完成后的位宽为5120，结果保存到output\_stream\_input中，由于拼接模块的输出，一次仅能输出一个结果所以深度为2不会影响数据传输延时，数据的输出端口连接输出层的输入接口，可以进入到输出层中进行计算。

同样，由于本次大作业的模型输出和h状态一样，所以h状态不仅要传输到输出层区进行计算，还要循环到下一个不长中作为输入，循环输入通过recurrent\_stream FIFO来实现。

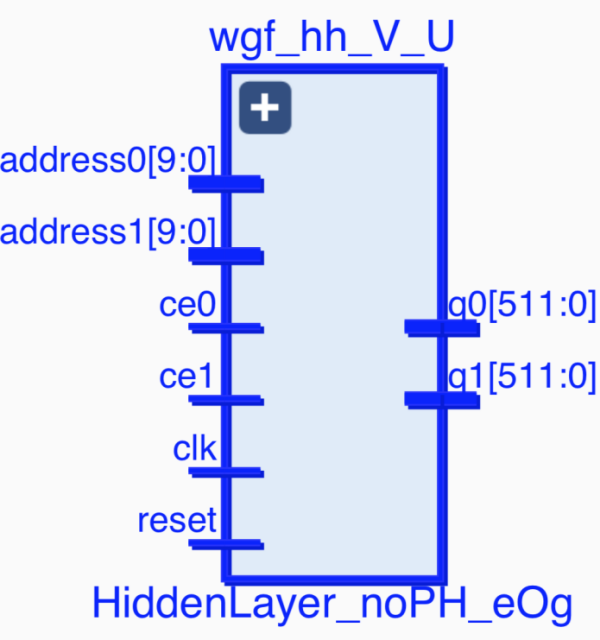
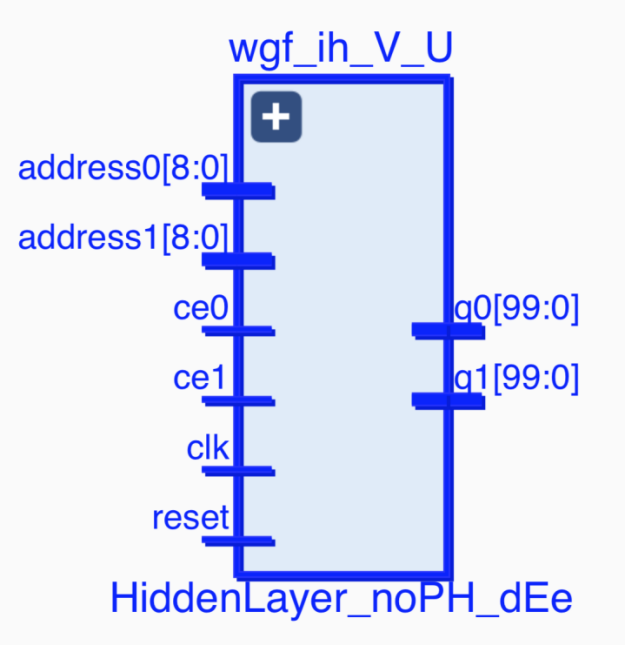
#### 

#### 2.权重参数存储模块的实现（ROM）

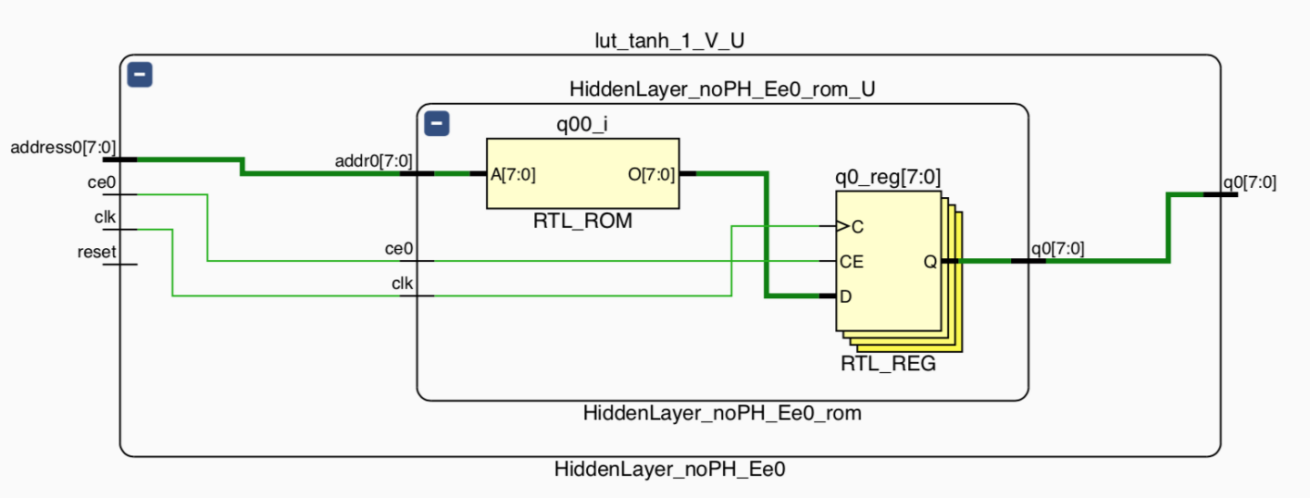
权重存储模块由两块ROM实现，同时支持双端口以满足高性能的pipeline优化要求，每一个参数矩阵都被存储在一个ROM中。每个门有两个维度不同的参数矩阵，分别对应输入(i)和隐藏状态(h)，以输入门为例，两个例子化的ROM为i矩阵的位宽为：100(PE\*SIMD\_INPUT\*WIDTH)

h矩阵的位宽为：

100(PE\*SIMD\_RECURRENT\*WIDTH)



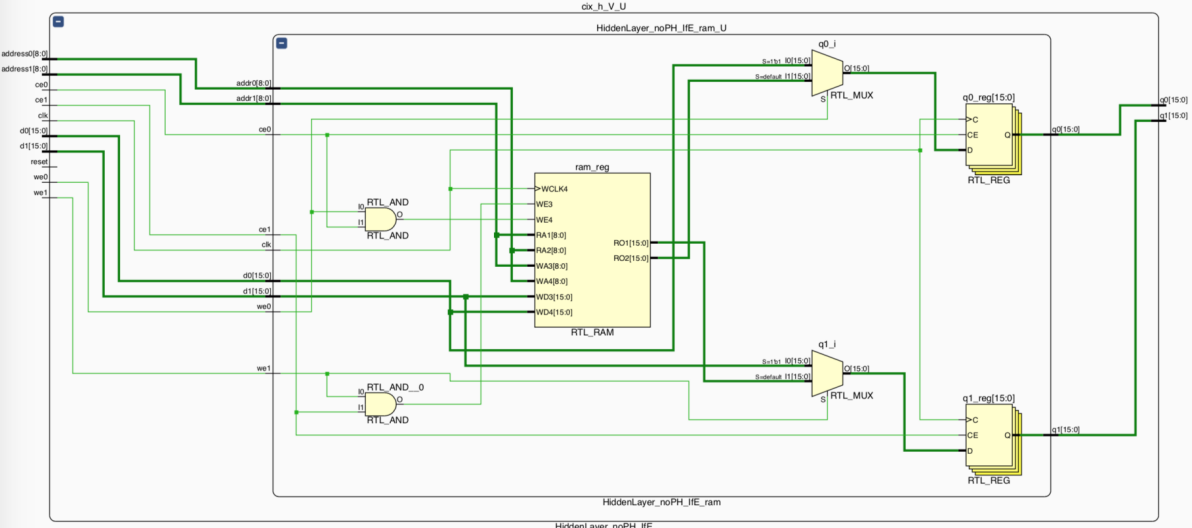
#### 3.激活函数查找表（ROM）

激活函数查找表的值也被放入ROM中，下图为lut查找表的实现：

#### 4.c状态的存储（RAM）

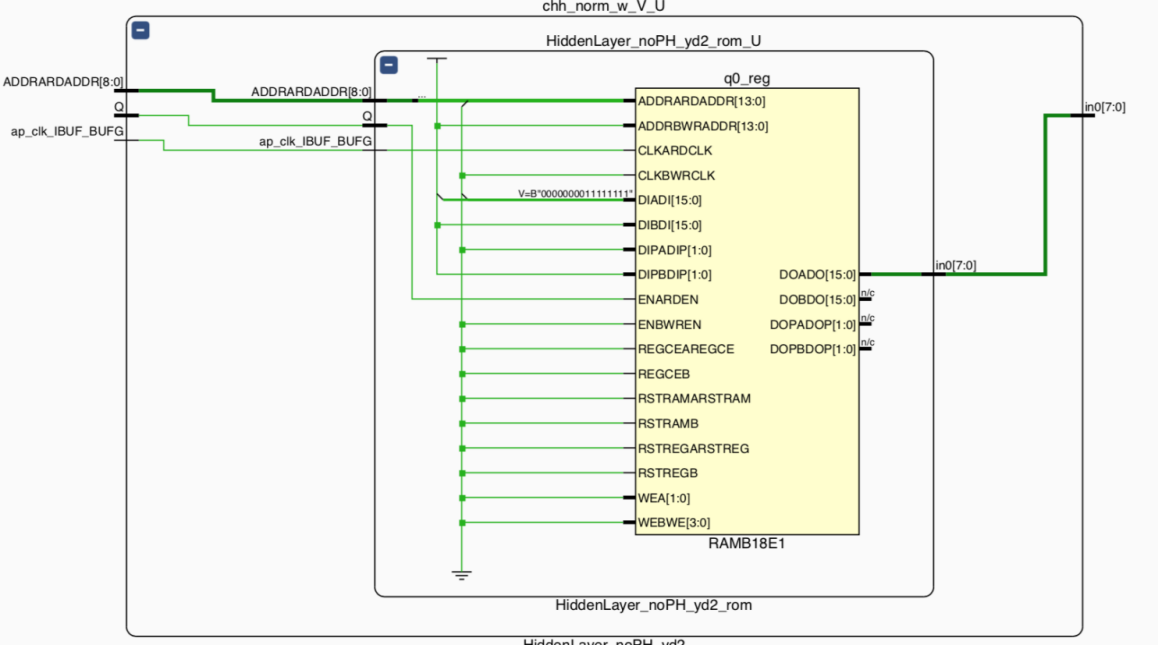
#### 

#### 5.矩阵计算中间结果（RAM）

进过矩阵计算后的结果需要进行Normalization，而Normalization要求所有值都求出来以后，才可以进行处理，因此，需要将中间结果存储起来，直到所有计算结束后在读取出来计算。同样的每个门有两个矩阵，每个门对应两个RAM，共有8个。以其中一个门为例。

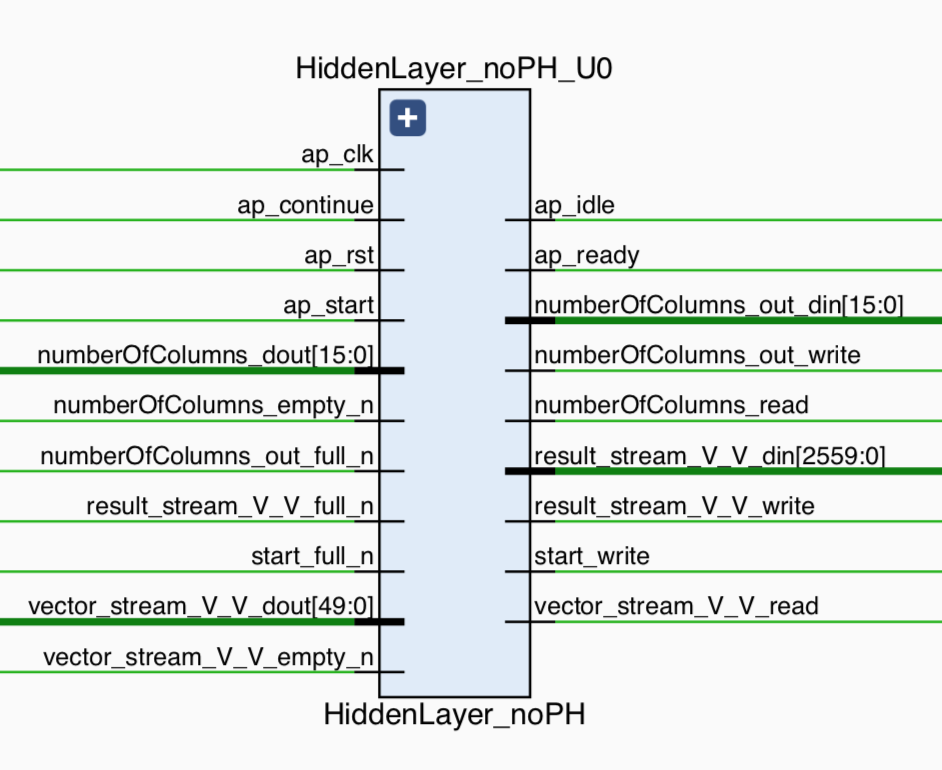
#### 6.Normalization参数

除此之外，还有Normalization的仿射参数，与权重参数不同因为没有对Normalization加流水线优化，所以被综合成RAMB18E1。下图模块对应于放射参数γ：



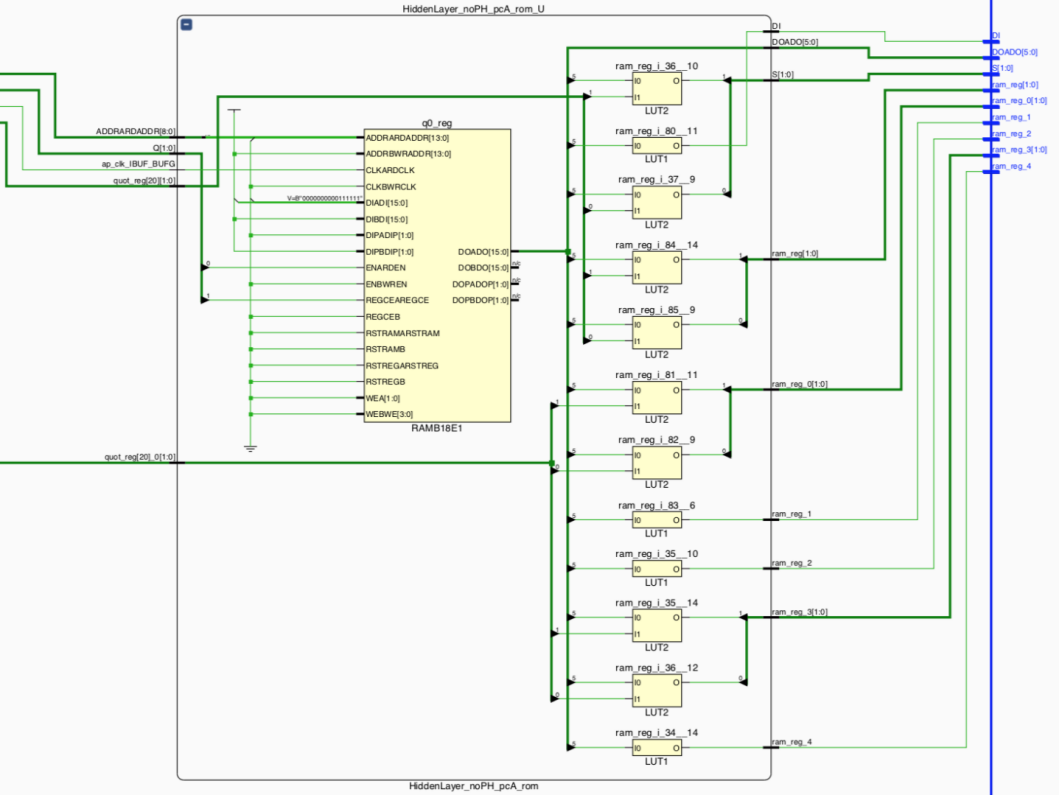
仿照参数对应不同的门都有两块ROM（γ和β）β的ROM实现如下

如图，与γ的ROM相比，内部除了RAMB18E1块以为还加入了20位的输入数据，和一些lut。这是用来实现加法操作的，输入的数据和存储块读取出的数据直接想加。由于该参数用于加法，在实现的时候直接把对应的加法和存储模块放到一起有利于降低时延。

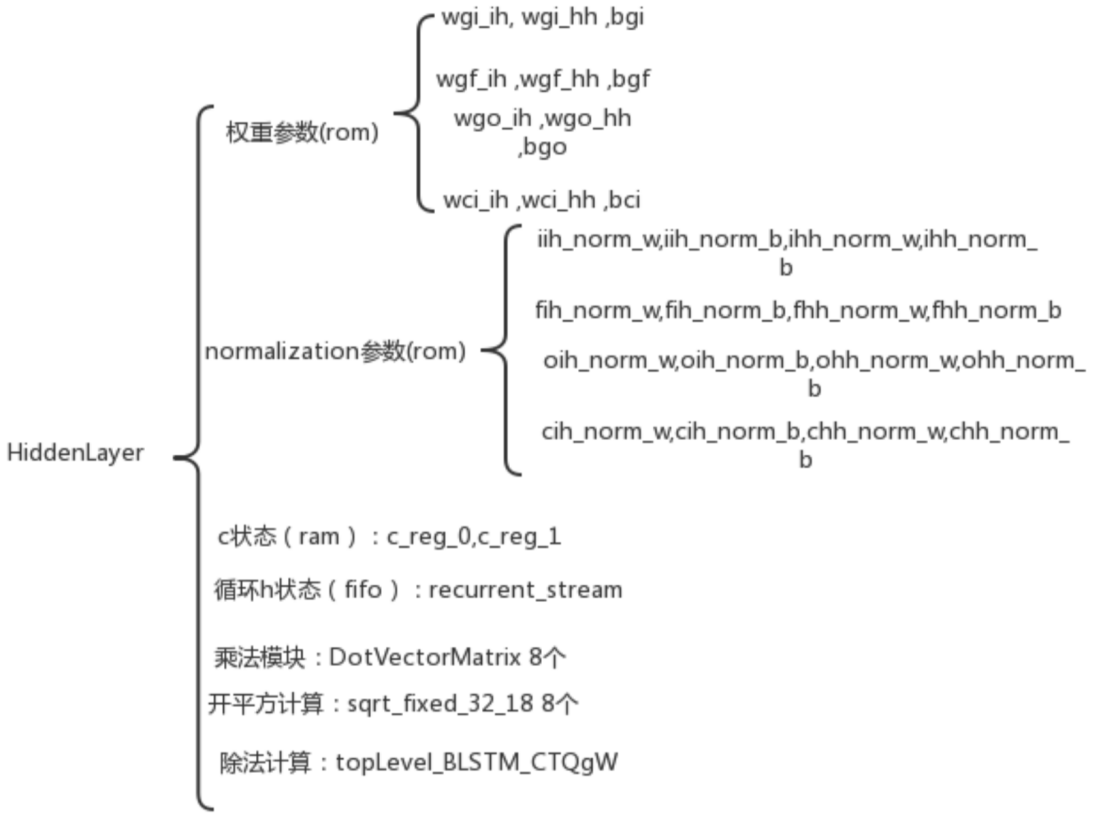


### 

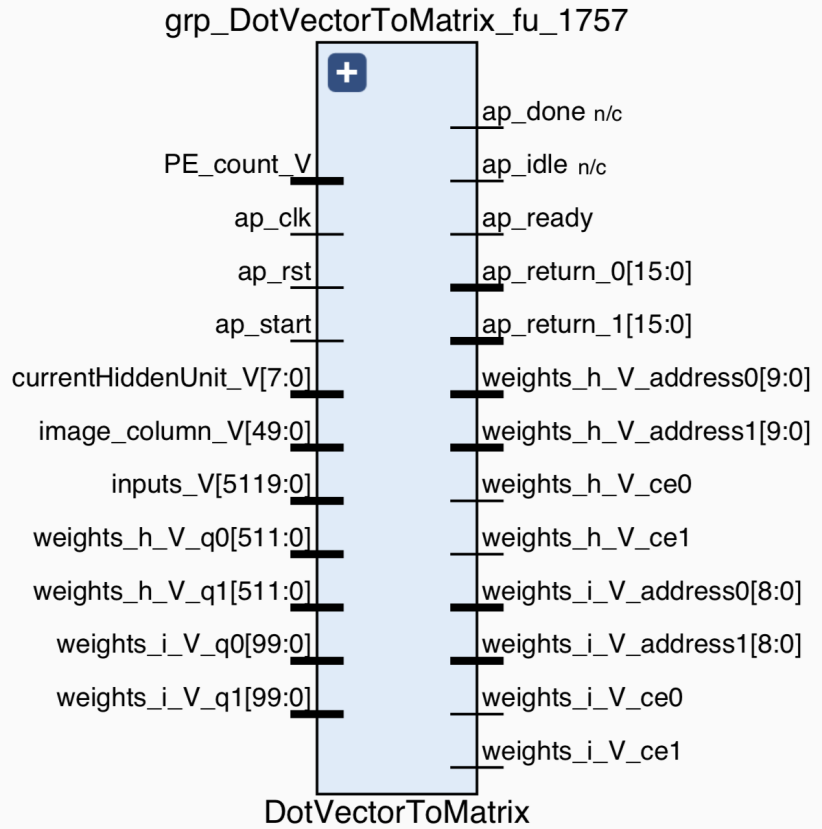
### 4.2.2 隐藏层模块的实现



numberOfColumns是序列的长度，也是通过FIFO传输的。vector\_stream与output\_stream\_column（字符向量输入）的输出及相关控制信号相连。输出的结果送到output\_stream\_hidden中，其余的信号都是数据接口相应的控制信号。



##### 1.DotVectorMatrix

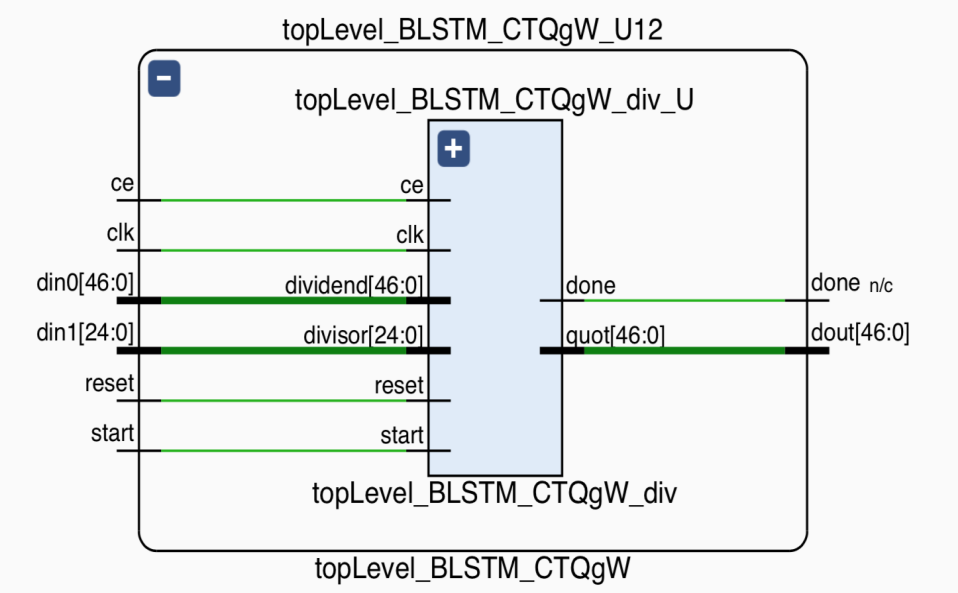


实现代码：

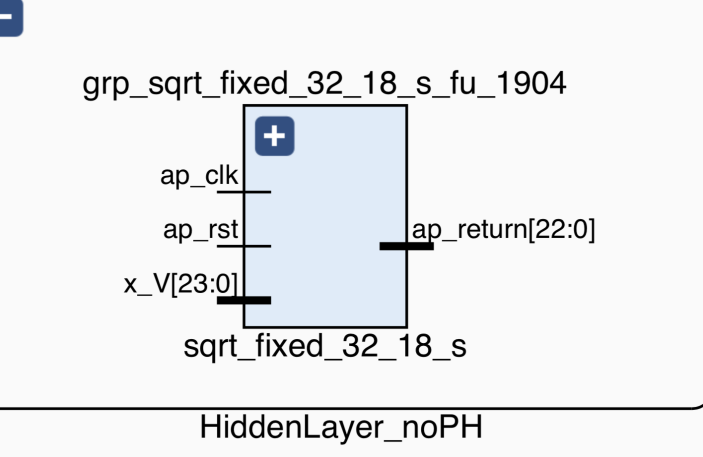
template  
<  
**unsigned int** PE,  
**unsigned int** SIMD\_INPUT, *// Number of parallel MAC performed in the gates on input pixels***unsigned int** SIMD\_RECURRENT, *// Number of parallel MAC performed in the gates on recurrent path***typename** Pixel\_t,  
**unsigned int** PixelWidth,  
**typename** OutputActivation\_t,  
**unsigned int** OutputActivationWidth,  
**typename** Bias\_t,  
**unsigned int** BiasWidth,  
**typename** Weight\_t,  
**unsigned int** WeightWidth,  
**typename** DotProductResult\_t,  
**typename** ColumnHeight\_t,  
**unsigned int** ColumnHeight,  
**typename** NumberHiddenUnits\_t,  
**unsigned int** NumberHiddenUnits  
>  
**void** DotVectorToMatrix(*//const Bias\_t biases[PE][NumberHiddenUnits/PE],***const** Weight\_t weights\_i[SIMD\_INPUT][ColumnHeight/SIMD\_INPUT][PE][NumberHiddenUnits/PE],  
**const** Weight\_t weights\_h[SIMD\_RECURRENT][NumberHiddenUnits/SIMD\_RECURRENT][PE][NumberHiddenUnits/PE], ap\_uint<ColumnHeight \* PixelWidth> image\_column,  
ap\_uint<OutputActivationWidth \* NumberHiddenUnits> inputs,NumberHiddenUnits\_t currentHiddenUnit,  
 NumberHiddenUnits\_t PE\_count,  
 DotProductResult\_t &sum\_pix1,  
 DotProductResult\_t &sum\_neuron1)  
{  
 **const unsigned int** FoldingInput = ColumnHeight / SIMD\_INPUT;  
 **const unsigned int** FoldingRecurrent = NumberHiddenUnits / SIMD\_RECURRENT;  
 DotProductResult\_t mul\_pix[SIMD\_INPUT][FoldingInput];  
 DotProductResult\_t mul\_neuron[SIMD\_RECURRENT][FoldingRecurrent];  
 DotProductResult\_t sum\_pix = 0.0;  
 DotProductResult\_t sum\_neuron = 0.0;  
 DotProductResult\_t sum = 0.0;  
 *//ap\_int<BiasWidth>bias\_temp=biases[PE\_count][currentHiddenUnit];  
 //Bias\_t bias = biases[PE\_count][currentHiddenUnit];  
 //Bias\_t bias = \*reinterpret\_cast<Bias\_t \*>(&bias\_temp);*#pragma HLS ARRAY\_PARTITION variable=mul\_pix complete dim=1  
#pragma HLS ARRAY\_PARTITION variable=mul\_neuron complete dim=1  
#pragma HLS ARRAY\_RESHAPE variable=weights\_i complete dim=1  
#pragma HLS ARRAY\_RESHAPE variable=weights\_h complete dim=1  
#pragma HLS ARRAY\_RESHAPE variable=weights\_i complete dim=3  
#pragma HLS ARRAY\_RESHAPE variable=weights\_h complete dim=3  
 **for**(ColumnHeight\_t j = 0; j < FoldingInput; j++) {  
#pragma HLS PIPELINE II=1 rewind  
 **for**(ColumnHeight\_t i = 0; i < SIMD\_INPUT; i++)  
 {  
 #pragma HLS UNROLL  
 **unsigned int** PixelInColumn = i\*FoldingInput+j;  
 ap\_uint<PixelWidth> pixel\_temp = image\_column((PixelInColumn+1)\*PixelWidth-1, PixelInColumn\*PixelWidth);  
 Pixel\_t pixel = \***reinterpret\_cast**<Pixel\_t \*>(&pixel\_temp);  
  
  
 Weight\_t weigth = weights\_i[i][j][PE\_count][currentHiddenUnit];  
 */\* if(currentHiddenUnit==0&&PE\_count==0){  
 //std::cout<<"FoldingInput:"<<j<<"SIMD\_INPUT"<<i<<  
 }\*/* mul\_pix[i][j] = pixel \* weigth;  
 }  
 }  
 **for**(NumberHiddenUnits\_t j = 0; j < FoldingRecurrent; j++) {  
#pragma HLS PIPELINE II=1 rewind  
 **for**(NumberHiddenUnits\_t i = 0; i < SIMD\_RECURRENT; i++)  
 {  
 #pragma HLS UNROLL  
 **unsigned int** ActivationInRecurrent = i\*FoldingRecurrent+j;  
 ap\_int<OutputActivationWidth> input\_temp = inputs((ActivationInRecurrent + 1) \* OutputActivationWidth - 1, ActivationInRecurrent \* OutputActivationWidth);  
 OutputActivation\_t input = \***reinterpret\_cast**<OutputActivation\_t \*>(&input\_temp);  
 *//std::cout<<"h\_prev"<<i\*FoldingRecurrent+j<<":"<<input<<",";* Weight\_t weigth = weights\_h[i][j][PE\_count][currentHiddenUnit];  
  
 mul\_neuron[i][j] = input \* weigth;  
 }  
 }  
 *//std::cout<<std::endl;* **for**(ColumnHeight\_t j = 0; j < FoldingInput; j++) {  
#pragma HLS PIPELINE II=1 rewind  
 **for**(ColumnHeight\_t i = 0; i < SIMD\_INPUT; i++)  
 {  
 #pragma HLS UNROLL  
 sum\_pix += mul\_pix[i][j];  
 }  
 }  
 **for**(NumberHiddenUnits\_t j = 0; j < FoldingRecurrent; j++) {  
#pragma HLS PIPELINE II=1 rewind  
 **for**(NumberHiddenUnits\_t i = 0; i < SIMD\_RECURRENT; i++)  
 {  
 #pragma HLS UNROLL  
 sum\_neuron += mul\_neuron[i][j];  
 }  
 }  
  
 sum\_pix1=sum\_pix;  
 sum\_neuron1=sum\_neuron;  
}

##### 计算Normalization的sqrt和divide模块

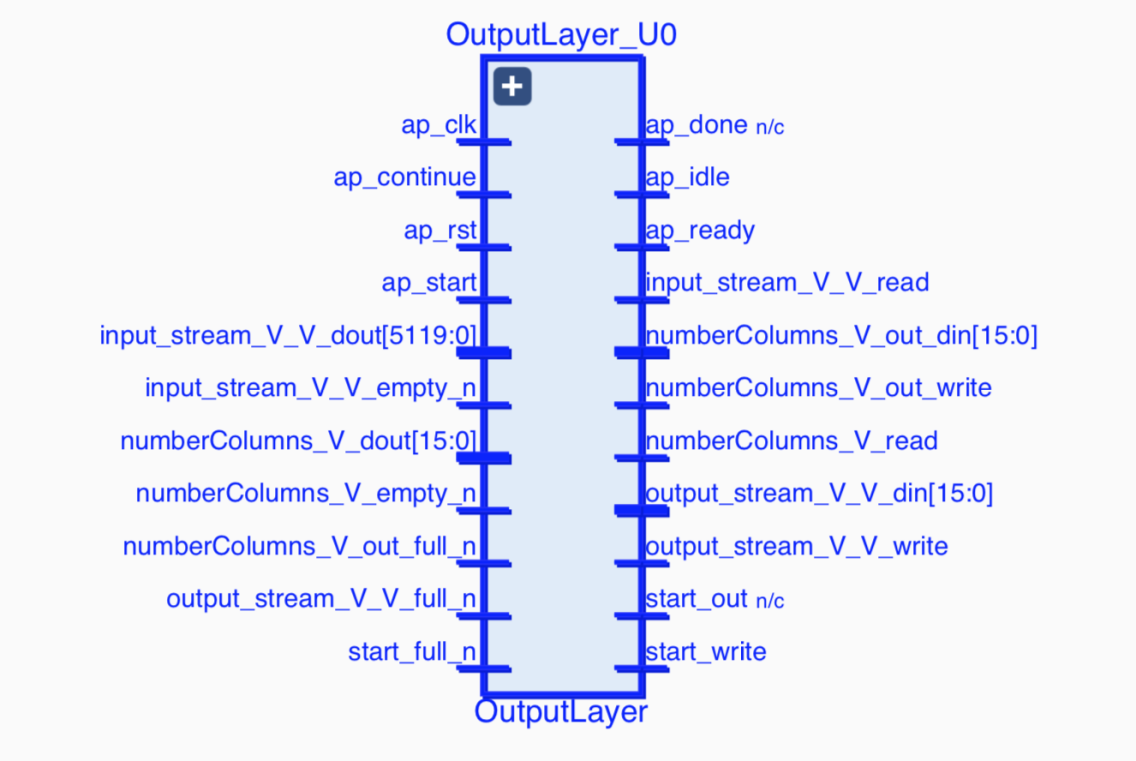
这一部分的实现由HLS工具根据涉及到的运算（开根号和除法）自行生成的设计，内部结构可读性不高只做简单的介绍。除法模块：din0和din1输入除数和被除数，dout输出结果：



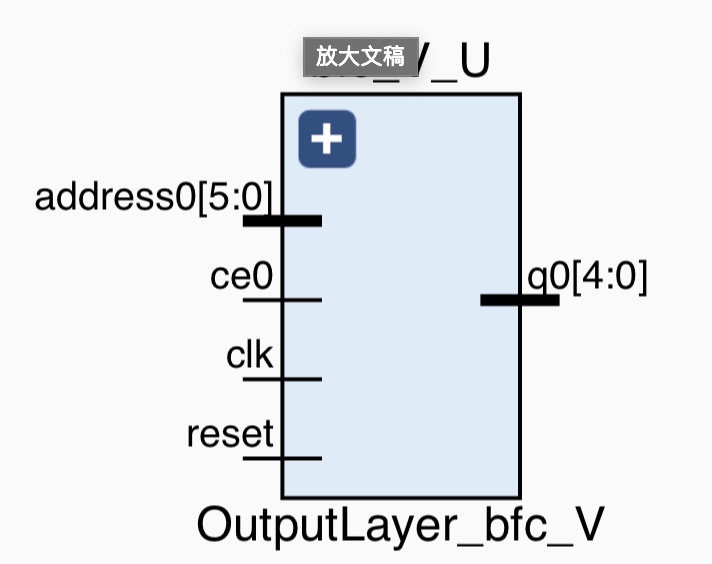
开根号模块：



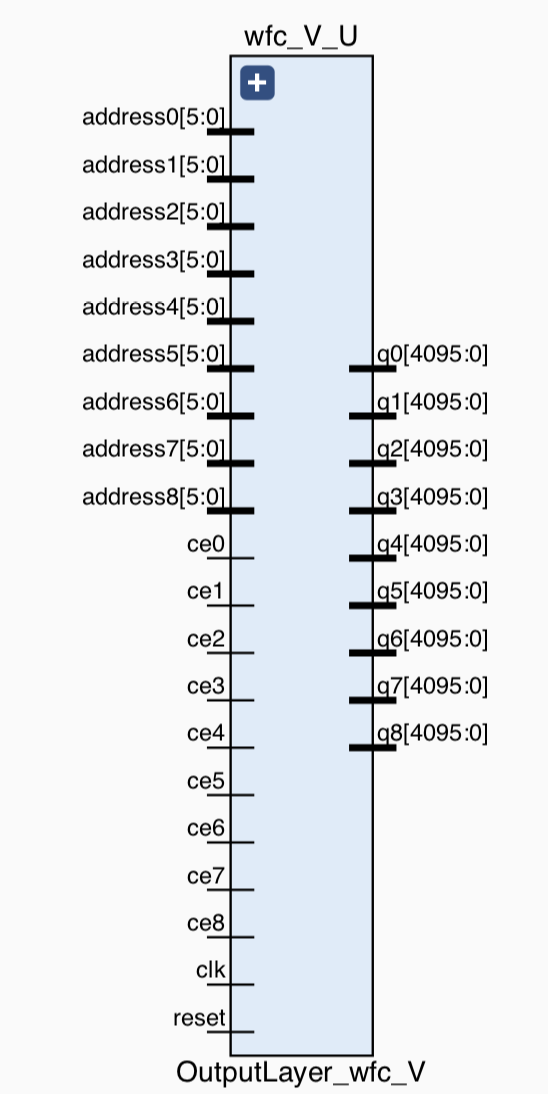
### 4.2.3 输出层模块



输出口是从input\_stream FIFO读取到的，numberColumns是输入的序列长度，会直接输出，而输出的结果和上一层模块DoCompute的输出相连。内部还有相应的存储模块来保存全连接层的权重参数和偏置。分别为bfc和wfcbfc：存储偏置



如3.1.4部分介绍，每次并行计算的乘法次数为numberHiddenUnits,所以一次计算需要numberHiddenUnits个权重参数，本次大作业权重参数为8，numberHiddenUnits=512所以输出端口的位宽为4096。为了加快速度，对每一次乘法之间进行流水线优化，所以相应的权重参数ROM也支持多端口，如下图：



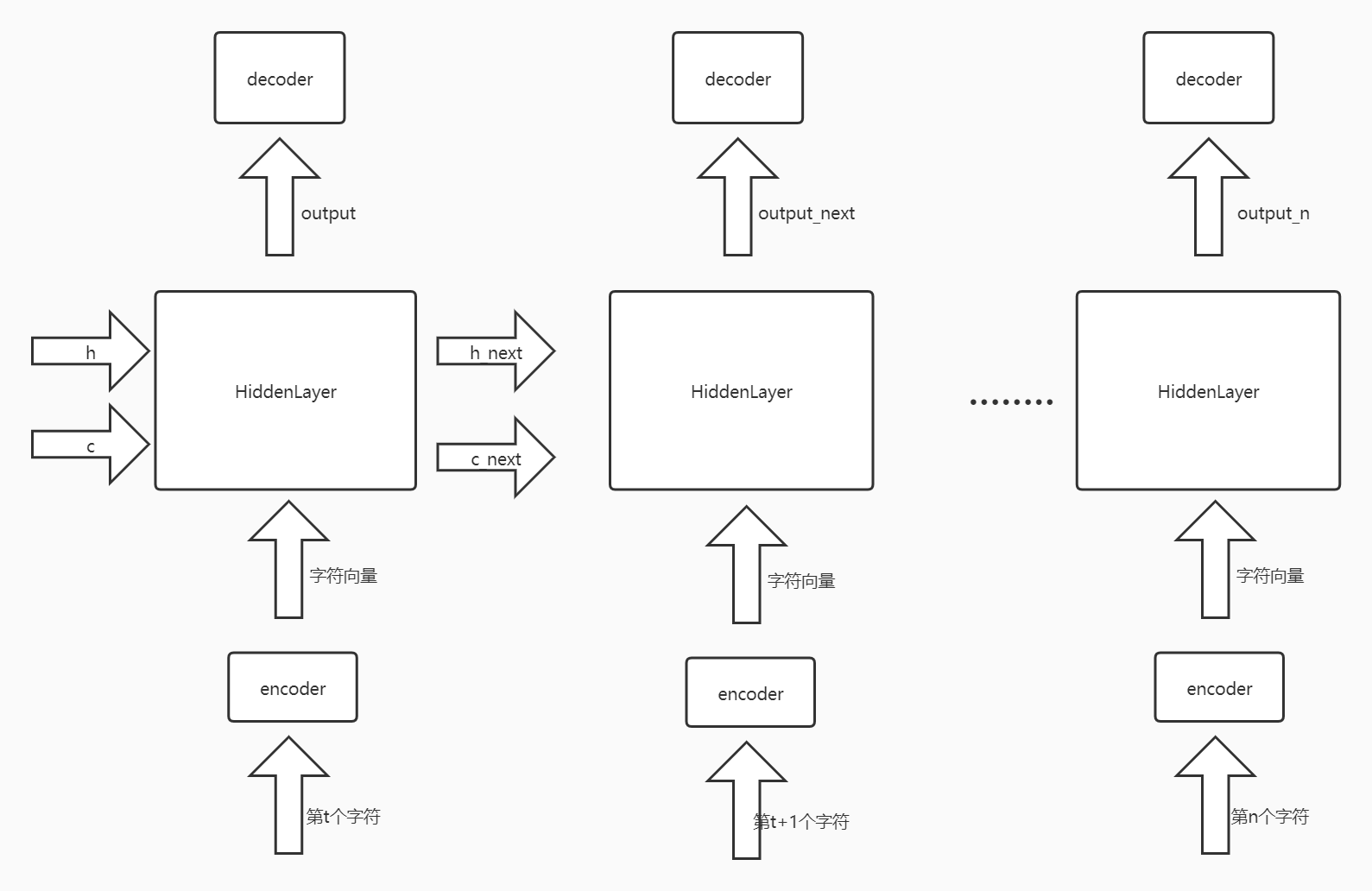
输出层的代码实现：

template  
<  
**typename** Bias\_fc\_t,  
**unsigned int** BiasWidth\_fc,  
**typename** Weight\_fc\_t,  
**unsigned int** WeightWidth\_fc,  
**typename** OutputActivationHiddenLayer\_t,  
**unsigned int** OutputActivationHiddenLayerWidth,  
**typename** OutputActivationOutputLayer\_t,  
**unsigned int** OutputActivationOutputLayerWidth,  
**typename** NumberHiddenUnits\_t,  
**unsigned int** NumberHiddenUnits,  
**typename** NumberOutputUnits\_t,  
**unsigned int** NumberOutputUnits,  
**typename** NumberColumns\_t  
>  
**void** OutputLayer(**const** Bias\_fc\_t biases[NumberOutputUnits],  
 **const** Weight\_fc\_t weights[NumberHiddenUnits][NumberOutputUnits],  
 NumberColumns\_t numberOfColumns,  
 HLS::stream<ap\_uint<OutputActivationHiddenLayerWidth \* NumberHiddenUnits> > &input\_stream,  
 HLS::stream<OutputActivationOutputLayer\_t> &output\_stream)  
{  
 ap\_uint<OutputActivationHiddenLayerWidth \* NumberHiddenUnits> input\_stream\_temp;  
 OutputActivationOutputLayer\_t mul[NumberHiddenUnits];  
 OutputActivationOutputLayer\_t sum;  
  
 #pragma HLS ARRAY\_PARTITION variable=mul complete dim=1  
  
 **for**(NumberColumns\_t currentColumn = 0; currentColumn < numberOfColumns; currentColumn++)  
 {  
 input\_stream.read(input\_stream\_temp);  
  
 **for**(NumberOutputUnits\_t currentClass = 0; currentClass < NumberOutputUnits; currentClass++)  
 {  
 #pragma HLS PIPELINE II=1  
  
 Bias\_fc\_t bias = biases[currentClass];*//[count \* NumberOutputUnits + currentClass];  
 //Bias\_fc\_t bias = \*reinterpret\_cast<Bias\_fc\_t \*>(&bias\_temp);* sum = (OutputActivationOutputLayer\_t)bias;  
  
 **for**(NumberHiddenUnits\_t currentHiddenUnit = 0; currentHiddenUnit < NumberHiddenUnits; currentHiddenUnit++)  
 {  
 #pragma HLS UNROLL  
  
 ap\_int<OutputActivationHiddenLayerWidth> input\_temp = input\_stream\_temp((currentHiddenUnit + 1) \* OutputActivationHiddenLayerWidth - 1, currentHiddenUnit \* OutputActivationHiddenLayerWidth);  
 OutputActivationHiddenLayer\_t input = \***reinterpret\_cast**<OutputActivationHiddenLayer\_t \*>(&input\_temp);  
 Weight\_fc\_t weigth = weights[currentHiddenUnit][currentClass];*//[count \* NumberOutputUnits + currentClass];  
 //Weight\_fc\_t weigth = \*reinterpret\_cast<Weight\_fc\_t \*>(&weigth\_temp);  
 /\*if(currentClass==0)  
 {std::cout<<currentHiddenUnit<<":"<<input<<std::endl;}\*/* mul[currentHiddenUnit] = input \* weigth;  
 }  
  
 **for**(NumberHiddenUnits\_t currentHiddenUnit = 0; currentHiddenUnit < NumberHiddenUnits; currentHiddenUnit++)  
 {  
 #pragma HLS UNROLL  
  
 sum += mul[currentHiddenUnit];  
 }  
  
 output\_stream.write(sum);  
  
 }  
*// }* }  
}

# 5 实验分析

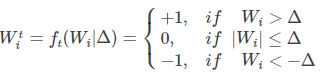
## 5.1实验流程

语言模型的结构如下图：



隐藏层的大小为512，字符向量长度为50字符经过预处理后得到对应的编号并且记录在一个字典中，encoder是将输入的字符编号转换成字符向量，由于字符的种数即字典的大小较小所以encoder直接采用one-hot的编码方式，而对于单词级的语言模型或其他字典长度比较大的模型，一般会采用其他更加高效的编码方式。转换得到的字符向量和上一个步长传递下来的隐藏状态h，c一同输入到隐藏层中去，隐藏层中只有一个LSTM cell即上文所介绍的四个门的结构，经过隐藏层的计算得到下一个状态和输出结果，得到的状态传递到下一个步长中去，而输出经过decoder得到预测的下一个字符的结果，decoder在这里的作用相当于一个全连接层。字符向量的长度为50，h和c的长度为512，decoder是一个512\*50的矩阵。

量化方案是只在前向和后向过程中使用使用权值简化，但是在参数更新仍然是使用连续的权值。简单的说就是先利用公式计算出三值网络中的阈值：将每一层的权值绝对值求平均值乘以0.7算出一个Δ作为三值网络离散权值的阈值，具体的离散过程如下：



使用随机梯度下降（SGD）方法来训练TWN。 Batch size为64，learning rate为0.002在前向传播和反向传播期间使用三值权重，但不在参数更新期间使用。 此外，还加入了上文提到的layer Normalization应用在每次矩阵乘法后。训练后将模型保存，读取模型里的参数。

用c语言来实现上文的设计结构，用Vivado HLS 高层次综合工具（2017.4）对代码进行仿真验证以后，把c语言的设计综合成verilog设计，综合策略以降低时延为目标。将生成的verilog设计导入到Vivado设计套件（2017.4）中对verilog代码进行仿真验证，通过以后选择zynq-7000系列的xc7z100ffg1156-2硬件平台,工作频率设置为40MHZ执行综合和布局布线后将设计布局到具体的FPGA硬件平台上，再次对布局步线后的结果进行仿真验证得到结果。

结果精度对比：

由于实验设备的限制，为了节省时间语言模型的训练只用了数据集中的一部分训练数据，将训练好的模型保存，并得到模型的所有权值和偏置包括Normalization中用到的可学习仿射参数，将参数放到存储模块中后，给加速器和在CPU上运行的模型同样的输入，对比得到的结果如下：加速器得到的结果（只截取了一部分，对应的布局布线后仿真波形图见附录2）

[[ 2.1719, 3.1055, -3.8047, ..., -0.9531, -0.7734, -0.9062],

[-2.0938, -2.4102, 1.4453, ..., -1.8672, -1.2617, -1.1406],

[ 1.5742, -0.3828, 1.7617, ..., -3.7812, -4.6875, -4.0117],

...,

[ 2.1055, 1.8398, 1.5547, ..., -5.7930, -6.1562, -5.8594],

[ 5.5664, 7.0547, 4.3398, ..., -4.0977, -3.7891, -4.3320],

[-0.7969, 1.3008, 2.0117, ..., -3.0273, -3.5312, -4.1445]]

平均交叉熵为1.084928。

原模型得到的结果：

[[ 2.2337, 3.1726, -3.7849, ..., -1.0029, -0.8817, -0.9617],

[-2.0151, -2.3914, 1.5612, ..., -1.9734, -1.3528, -1.2464],

[ 1.6059, -0.3899, 1.7694, ..., -3.8662, -4.7884, -4.1135],

...,

[ 2.1965, 1.8386, 1.6789, ..., -5.8842, -6.2326, -5.9827],

[ 5.7009, 7.1357, 4.3823, ..., -4.2067, -3.8891, -4.5028],

[-0.7945, 1.3751, 1.9812, ..., -3.1976, -3.6468, -4.2624]]

平均交叉熵为1.083791。

得到的交叉熵可能较大，由于模型没有进行充分训练，只采用了一小部分数据进行了训练，目的是对比在同样模型的情况下加速器的结果和CPU上计算的结果。采用上文提到的训练方法可以训练到一个相对理想的效果（bpc=1.43左右）

## 5.2设计结果分析

从资源利用的角度来看，由于计算结构的并行程度较大所以小耗的资源较多，如下图只适用于中高端FPGA平台，具体结果见附录1。

性能方面，通过过布局布线后的仿真得到在输入时钟为40MHZ的情况下，输入100个连续的字符序列的计算时间为0.4s左右，在CPU平台（型号为Xeon E5-2670 2.6GHZ，用python实现）的计算时间约为0.19s，[22]实现了一个想当规模的LSTM加速器用于语言识别，连续输入1000个字符的计算时间为0.11s，[23]同样实现了一个字符级的语言模型，但是该设计的权值和中间结果都放在FPGA片外，运行时间只考虑了FPGA上的计算时间，同样由于需要从外部读取参数所以FPGA的资源占用也很少。运行时间的对比证明该设计在运行速度方面合格，同时该加速器的功耗估算仅有1.69w对比于其他实现功耗效率得到很大的提升。

|  |  | 频率 |
| --- | --- | --- |
| 本次大作业 | 0.4~0.5s | 40MHZ |
| [22]LSTM#1,#2,#3 | 0.11s | 100MHZ |
| Xeon E5-2670 | 0.19s | 2.6GHZ |
| Zynq zc7020 CPU | 21.756s | 667MHZ |
| [23]LSTM | 1.017s | 142MHZ |

|  | lut | FF | bRAM | DSP |
| --- | --- | --- | --- | --- |
| 本次大作业中的开平方 | 1667\*8 | 141\*8 | 0 | 0 |
| 本次大作业 | 210778 | 167962 | 395 | 31 |
| [22] | 198280 | 183646 | 1072 | 1176 |
| [23] | 7627 | 12960 | 16 | 50 |

# 6 实验结论

本次大作业把Normalization运用到了RNN的硬件实现上，得以在现有的量化RNN的硬件实现基础上加大了量化的程度并且模型的性能精度损失很小，通过实现以后的精度对比，运行时间的比较以及资源和功耗的对比等，证明了Normalization在运用在硬件平台上的可行性。并且该硬件加速器可以进行高层次的配置，通用性好可以应用在其他LSTM应用中。但是，由于该实现是从高层次的角度去实现，抽象层级太高，实现过程中的很多细节部分无法控制。

因此，该设计可以作为一个先行的验证步骤，后续的实现可以在此基础上进行底层的细节优化来提高性能或者应用在其他相关的应用上去。

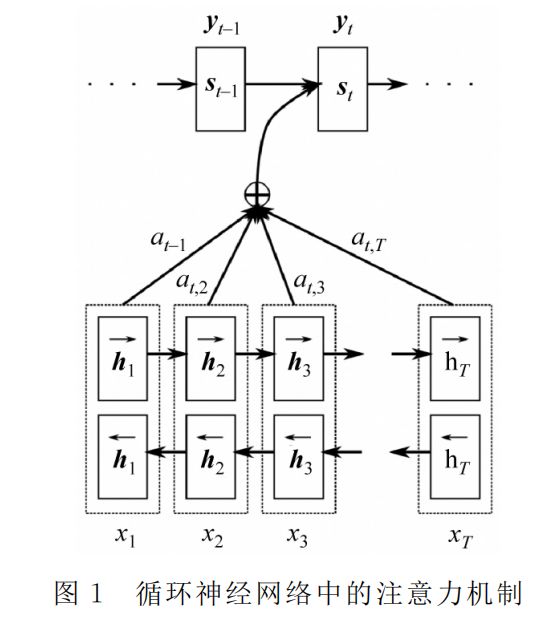
# 7 课程总结及课堂建议

## 7.1 课程总结

机器翻译是指通过计算机将源语言句子翻译到与之语义等价的目标语言句子的过程，是自然语言处理领域的一个重要研究方向。神经机器翻译仅需使用神经网络就能实现从源语言到目标语言的端到端翻译，目前已成为机器翻译研究的主流方向。机器翻译主要可以分为三种方法：基于规则的机器翻译、基于统计的机器翻译和基于神经网络的机器翻译。

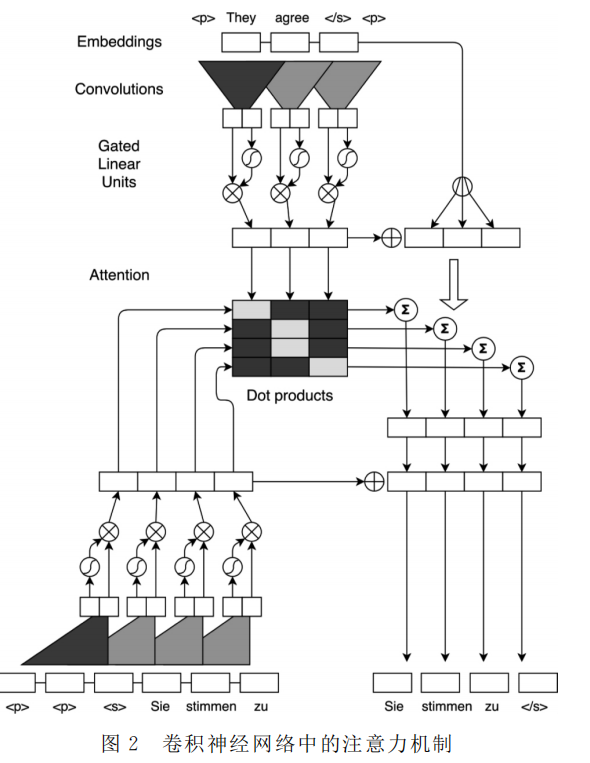
### 7.1.1 基于循环神经网络的机器翻译模型

基于循环神经网络和编码器—解码器结构的神经机器翻译模型在很长一段时间内都是神经机器翻译的主流模型。其中，在编码器—解码器框架的基础上，引入了注意力机制，使得生成每个目标端词语时，解码器可以将“注意力”集中到源端的几个相关的词语上，并从中获取有用的信息，从而获得更好的翻译表现。注意力机制使得翻译模型能够更好地处理长距离的依赖关系，解决了在循环神经网络中信息在长距离的传输中容易被丢失、遗忘的问题，其注意力机制如图1所示：



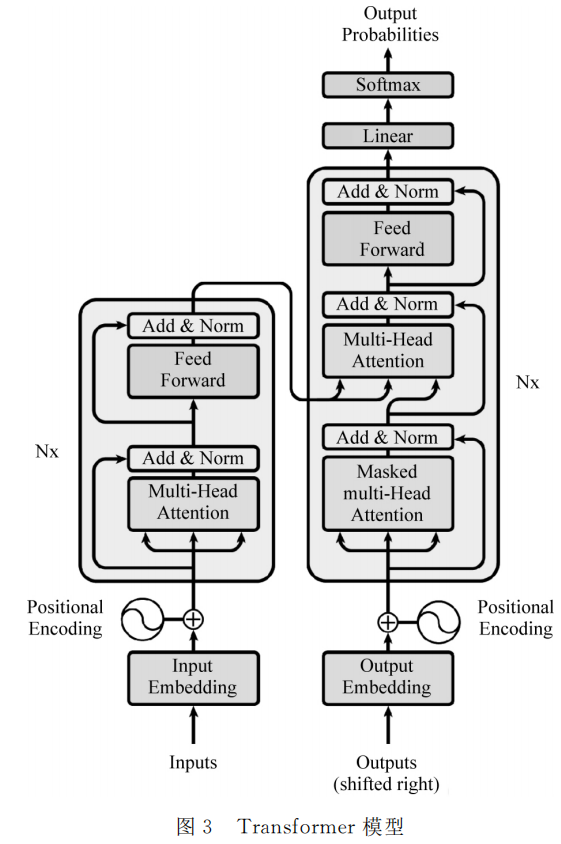
### 7.1.2 基于卷积神经网络的机器翻译模型

在基于循环神经网络的神经机器翻译模型展现出强大的能力后，研究者们也开始探索基于卷积神经网络的神经翻译模型。用卷积神经网络来编码源端语言，并将其集成到统计机器翻译模型中，将神经网络翻译模型的源端编码器替换成了基于卷积神 经网络的结构，随后提出了完全基于卷积神经网络的机器翻译模型，如图2所示：



### 7.1.3 Transformer模型

2017年提出的完全基于注意力机制的Transformer模型创新性地使用自注意力机制来对序列进行编码，其编码器和解码器均由注意力模块和前向神经网络构成。Transformer模型具有高度并行化的模型结构，因此在训练速度上远超循环神经网络，且在翻译质量上也有大幅提升。近期，Transformer已成为神经机器翻译研究中的主流模型，且在自然语言处理的其他领域中也有广泛应用。Transformer的模型结构如图３所示：



## 7.2 课堂建议

1. 希望老师以后加入观看当今机器翻译前沿技术视频的授课形式，感觉这种方式能让大家了解到现如今机器翻译领域已经做到什么程度了，瓶颈和需要克服的困难、改进的方向在哪里，能更好的培养大家的学习兴趣，给大家一个更好的机器翻译的启蒙教育。
2. 希望老师以后上课能继续多多与学生进行互动，比如以课堂提问或者师生交流的形式来互动，这方面做的已经非常棒了，继续保持这样的风格。
3. 在保证授课进度的前提下，可以适当增加课堂展示交流学习来让实验室的师兄师姐分享对机器翻译的理解以及对如何学好机器翻译给出一定的建议，毕竟同龄人对彼此的了解能更透彻一些。
4. 继续多安排一些上机实验的课程，感觉这部分对我们的收益和提高会非常大。
5. 希望老师增加对同声传译、非自回归模型、多模态机器翻译、篇章翻译、领域自适应、多语言翻译等方向的前沿发展前景进行介绍。

# 附录

## 附录1

Copyright 1986-2017 Xilinx, Inc. All Rights Reserved.  
---------------------------------------------------------------------------------------------------------------------------------  
| Tool Version : Vivado v.2017.4 (lin64) Build 2086221 Fri Dec 15 20:54:30 MST 2017  
| Date : Sat May 16 20:26:17 2020  
| Host : knight-desktop running 64-bit Ubuntu 18.04.4 LTS  
| Command : report\_utilization -file topLevel\_BLSTM\_CTC\_utilization\_placed.rpt -pb topLevel\_BLSTM\_CTC\_utilization\_placed.pb  
| Design : topLevel\_BLSTM\_CTC  
| Device : 7z100ffg1156-2  
| Design State : Fully Placed  
---------------------------------------------------------------------------------------------------------------------------------  
  
Utilization Design Information  
  
Table of Contents  
-----------------  
1. Slice Logic  
1.1 Summary of Registers by Type  
2. Slice Logic Distribution  
3. Memory  
4. DSP  
5. IO and GT Specific  
6. Clocking  
7. Specific Feature  
8. Primitives  
9. Black Boxes  
10. Instantiated Netlists  
  
1. Slice Logic  
--------------  
  
+----------------------------+--------+-------+-----------+-------+  
| Site Type | Used | Fixed | Available | Util% |  
+----------------------------+--------+-------+-----------+-------+  
| Slice LUTs | 210778 | 0 | 277400 | 75.98 |  
| LUT as Logic | 210738 | 0 | 277400 | 75.97 |  
| LUT as Memory | 40 | 0 | 108200 | 0.04 |  
| LUT as Distributed RAM | 0 | 0 | | |  
| LUT as Shift Register | 40 | 0 | | |  
| Slice Registers | 167962 | 0 | 554800 | 30.27 |  
| Register as Flip Flop | 167962 | 0 | 554800 | 30.27 |  
| Register as Latch | 0 | 0 | 554800 | 0.00 |  
| F7 Muxes | 926 | 0 | 138700 | 0.67 |  
| F8 Muxes | 459 | 0 | 69350 | 0.66 |  
+----------------------------+--------+-------+-----------+-------+  
  
  
1.1 Summary of Registers by Type  
--------------------------------  
  
+--------+--------------+-------------+--------------+  
| Total | Clock Enable | Synchronous | Asynchronous |  
+--------+--------------+-------------+--------------+  
| 0 | \_ | - | - |  
| 0 | \_ | - | Set |  
| 0 | \_ | - | Reset |  
| 0 | \_ | Set | - |  
| 0 | \_ | Reset | - |  
| 0 | Yes | - | - |  
| 0 | Yes | - | Set |  
| 37 | Yes | - | Reset |  
| 86 | Yes | Set | - |  
| 167839 | Yes | Reset | - |  
+--------+--------------+-------------+--------------+  
  
  
2. Slice Logic Distribution  
---------------------------  
  
+-------------------------------------------+--------+-------+-----------+-------+  
| Site Type | Used | Fixed | Available | Util% |  
+-------------------------------------------+--------+-------+-----------+-------+  
| Slice | 65664 | 0 | 69350 | 94.68 |  
| SLICEL | 40118 | 0 | | |  
| SLICEM | 25546 | 0 | | |  
| LUT as Logic | 210738 | 0 | 277400 | 75.97 |  
| using O5 output only | 13 | | | |  
| using O6 output only | 155588 | | | |  
| using O5 and O6 | 55137 | | | |  
| LUT as Memory | 40 | 0 | 108200 | 0.04 |  
| LUT as Distributed RAM | 0 | 0 | | |  
| LUT as Shift Register | 40 | 0 | | |  
| using O5 output only | 3 | | | |  
| using O6 output only | 5 | | | |  
| using O5 and O6 | 32 | | | |  
| LUT Flip Flop Pairs | 42622 | 0 | 277400 | 15.36 |  
| fully used LUT-FF pairs | 9865 | | | |  
| LUT-FF pairs with one unused LUT output | 26730 | | | |  
| LUT-FF pairs with one unused Flip Flop | 32396 | | | |  
| Unique Control Sets | 2009 | | | |  
+-------------------------------------------+--------+-------+-----------+-------+  
\* Note: Review the Control Sets Report for more information regarding control sets.  
  
  
3. Memory  
---------  
  
+-------------------+------+-------+-----------+-------+  
| Site Type | Used | Fixed | Available | Util% |  
+-------------------+------+-------+-----------+-------+  
| Block RAM Tile | 395 | 0 | 755 | 52.32 |  
| RAMB36/FIFO\* | 304 | 0 | 755 | 40.26 |  
| RAMB36E1 only | 304 | | | |  
| RAMB18 | 182 | 0 | 1510 | 12.05 |  
| RAMB18E1 only | 182 | | | |  
+-------------------+------+-------+-----------+-------+  
\* Note: Each Block RAM Tile only has one FIFO logic available and therefore can accommodate only one FIFO36E1 or one FIFO18E1. However, if a FIFO18E1 occupies a Block RAM Tile, that tile can still accommodate a RAMB18E1  
  
  
4. DSP  
------  
  
+----------------+------+-------+-----------+-------+  
| Site Type | Used | Fixed | Available | Util% |  
+----------------+------+-------+-----------+-------+  
| DSPs | 31 | 0 | 2020 | 1.53 |  
| DSP48E1 only | 31 | | | |  
+----------------+------+-------+-----------+-------+  
  
  
5. IO and GT Specific  
---------------------  
  
+-----------------------------+------+-------+-----------+-------+  
| Site Type | Used | Fixed | Available | Util% |  
+-----------------------------+------+-------+-----------+-------+  
| Bonded IOB | 154 | 0 | 400 | 38.50 |  
| IOB Master Pads | 74 | | | |  
| IOB Slave Pads | 75 | | | |  
| Bonded IPADs | 0 | 0 | 50 | 0.00 |  
| Bonded OPADs | 0 | 0 | 32 | 0.00 |  
| Bonded IOPADs | 0 | 0 | 130 | 0.00 |  
| PHY\_CONTROL | 0 | 0 | 8 | 0.00 |  
| PHASER\_REF | 0 | 0 | 8 | 0.00 |  
| OUT\_FIFO | 0 | 0 | 32 | 0.00 |  
| IN\_FIFO | 0 | 0 | 32 | 0.00 |  
| IDELAYCTRL | 0 | 0 | 8 | 0.00 |  
| IBUFDS | 0 | 0 | 384 | 0.00 |  
| GTXE2\_COMMON | 0 | 0 | 4 | 0.00 |  
| GTXE2\_CHANNEL | 0 | 0 | 16 | 0.00 |  
| PHASER\_OUT/PHASER\_OUT\_PHY | 0 | 0 | 32 | 0.00 |  
| PHASER\_IN/PHASER\_IN\_PHY | 0 | 0 | 32 | 0.00 |  
| IDELAYE2/IDELAYE2\_FINEDELAY | 0 | 0 | 400 | 0.00 |  
| ODELAYE2/ODELAYE2\_FINEDELAY | 0 | 0 | 150 | 0.00 |  
| IBUFDS\_GTE2 | 0 | 0 | 8 | 0.00 |  
| ILOGIC | 0 | 0 | 400 | 0.00 |  
| OLOGIC | 0 | 0 | 400 | 0.00 |  
+-----------------------------+------+-------+-----------+-------+  
  
  
6. Clocking  
-----------  
  
+------------+------+-------+-----------+-------+  
| Site Type | Used | Fixed | Available | Util% |  
+------------+------+-------+-----------+-------+  
| BUFGCTRL | 2 | 0 | 32 | 6.25 |  
| BUFIO | 0 | 0 | 32 | 0.00 |  
| MMCME2\_ADV | 0 | 0 | 8 | 0.00 |  
| PLLE2\_ADV | 0 | 0 | 8 | 0.00 |  
| BUFMRCE | 0 | 0 | 16 | 0.00 |  
| BUFHCE | 0 | 0 | 168 | 0.00 |  
| BUFR | 0 | 0 | 32 | 0.00 |  
+------------+------+-------+-----------+-------+  
  
  
7. Specific Feature  
-------------------  
  
+-------------+------+-------+-----------+-------+  
| Site Type | Used | Fixed | Available | Util% |  
+-------------+------+-------+-----------+-------+  
| BSCANE2 | 0 | 0 | 4 | 0.00 |  
| CAPTUREE2 | 0 | 0 | 1 | 0.00 |  
| DNA\_PORT | 0 | 0 | 1 | 0.00 |  
| EFUSE\_USR | 0 | 0 | 1 | 0.00 |  
| FRAME\_ECCE2 | 0 | 0 | 1 | 0.00 |  
| ICAPE2 | 0 | 0 | 2 | 0.00 |  
| PCIE\_2\_1 | 0 | 0 | 1 | 0.00 |  
| STARTUPE2 | 0 | 0 | 1 | 0.00 |  
| XADC | 0 | 0 | 1 | 0.00 |  
+-------------+------+-------+-----------+-------+  
  
  
8. Primitives  
-------------  
  
+----------+--------+---------------------+  
| Ref Name | Used | Functional Category |  
+----------+--------+---------------------+  
| FDRE | 167839 | Flop & Latch |  
| LUT6 | 76337 | LUT |  
| LUT5 | 51989 | LUT |  
| LUT2 | 51036 | LUT |  
| LUT3 | 45109 | LUT |  
| LUT4 | 37434 | LUT |  
| CARRY4 | 27040 | CarryLogic |  
| LUT1 | 3970 | LUT |  
| MUXF7 | 926 | MuxFx |  
| MUXF8 | 459 | MuxFx |  
| RAMB36E1 | 304 | Block Memory |  
| RAMB18E1 | 182 | Block Memory |  
| IBUF | 112 | IO |  
| FDSE | 86 | Flop & Latch |  
| SRL16E | 69 | Distributed Memory |  
| OBUF | 42 | IO |  
| FDCE | 37 | Flop & Latch |  
| DSP48E1 | 31 | Block Arithmetic |  
| SRLC32E | 3 | Distributed Memory |  
| BUFG | 2 | Clock |  
+----------+--------+---------------------+  
  
  
9. Black Boxes  
--------------  
  
+----------+------+  
| Ref Name | Used |  
+----------+------+  
  
  
10. Instantiated Netlists  
-------------------------  
  
+----------+------+  
| Ref Name | Used |  
+----------+------+

## 附录2

仅展示了前3行向量

