净荷压缩方案技术文件

技术文件名称：净荷压缩简化方案

技术文件编号：0001

版 本：A.1

拟 制

审 核

会 签

标准化

批 准

修改记录

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 文件编号 | 版本号 | 拟制人/  修改人 | 拟制日期/  修改日期 | 更改理由 | 主要更改内容  （写要点即可） |
| 0001 | A.1 | xx | 2018-12-18 | 初始版本 |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
| 注：文件第一次归档时，“更改理由”、“主要更改内容”栏写“无”。 | | | | | |

目 录

[1. 模板标题1 3](#_Toc1684283382)

[1.1 模板标题2 3](#_Toc1267391699)

[1.1.1 模板标题3 3](#_Toc397310112)

[1.1.1.1 模板标题4 3](#_Toc476051819)

## 模板标题1

见下文

表1.1 References list

|  |  |
| --- | --- |
| Type | Description |
| International standard | IEEE Std 802.3™-2005 |

### 模板标题2

见下文

#### 模板标题3

见下文

##### 模板标题4

见下文

###### 模板标题5

见下文

模板标题6

见下文

模板标题6

见下文

模板标题7

见下文



对于WPS来说，可以从”文件”-->”页面设置”-->”文档网络”，然后选择“无网络”

## 设计目标和设计依据

### 需求目的

通过以太网净荷压缩功能，在某些场景下能够大幅度提升以太网业务的平均传送能力，解决运营商面临的带宽压力，有效提升微波链路承载的以太网业务的带宽利用率。

### 需求描述

支持基于无损数据压缩算法LZW算法的以太网业务净荷压缩。

LZW是基于字典的压缩，它在数据的压缩过程中动态地生成一个串表，以后的数据就可以同串表中的数据相匹配，如果匹配上，则输出的是串表的索引。由于表示串表的索引所用的比特数远小于串的比特数，从而达到压缩的效果。数据的重复性越高，则匹配率越高，压缩效果越明显。

支持压缩/解压缩最高处理能力：1G

PLA下不支持净荷压缩，小于74字节的报文不支持净荷压缩。

支持单板：RMUN

应用场景：移动回传

优 先 级：5-中

使用频度：2-中

特殊需求：无

注释和问题：无

需求来源：06－投标

### 设计依据

无

## 术语、定义和缩略语

### 术语、定义

本文使用的专用术语，见表xx

表1.1 References list

|  |  |
| --- | --- |
| Type | Description |
| International standard | IEEE Std 802.3™-2005 |
| LZW |  |

### 缩略语

本文使用的缩略语，见表1.1

## 基本工作原理

由韦尔奇在1984年开发的LZW算法，是LZ系列码中应用最广、变形最多的。它的标识只有一项，即指向字典的指针，这是它与LZ—77、LZ—78的一个主要不同点。为实现简化标识，它的编码的思想与前述的算法有很大的不同。LZ系列算法的共同点是分解输入流，使其成为长度各异的“短语”，并把它们存入“短语字典”，并给每个“短语”赋予一个码字。只要短语的码字长度短于短语的长度，就达到了压缩的目的。而LZW编码算法则是先建立初始字典，再分解输入流为短语词条，这个短语若不在初始字典内，就将其存入字典，这些新词条和初始字典共同构成编码器的字典。而初始字典可由信源符号集构成，每个符号是一个词条。更一般的，是将扩展的ASCII码存入初始字典，使其成为字典的前256项，即0~255项。这样的初始化字典，在应用中就足够大了。

### 压缩

LZW码的编码原理是：先建立初始化字典，然后将待编码的输入数据流分解成“短语词条”。编码器要逐个输入字符，并累积串联成一个字符串，即“短语词条”I。若I是字典中已有的词条，就输入下一个字符x，形成新词条Ix。当I在字典内，而Ix不在字典内时，编码器首先输出指向字典内词条I的指针；再将Ix作为新词条存入字典，并为其确定顺序号；然后把x赋值给I，当做新词条的首字符。重复上述过程，直到输入流都处理完为止。

面用实例说明LZW码编码过程：

设输入序列为 ababcbabccc

1）先建初始化字典，此处只需将信源符号a,b,c预置为字典的前三项；

2）将首字符a预置为I，即I=a，搜索后知I在字典内，那么继续输入序列的第二项b，即有Ix=ab，搜索后知Ix不在字典内。则编码器先输出指向字典词条I=a相应的码字1，再把Ix=ab作为新词条存入字典，并编码得码字为4。再将x赋值给I，即此时I=b，当作新词条的首字符重复上述做法，得到编码表，如下表所示。

LZW码的编码表：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 码字 | 词条I | 新词条 | 输出码 |
| 初始字典 | 1  2  3 | a  b  c |  |  |
|  | 4  5  6  7  8  9  10 | a  ab  b  ba  a  ab  abc  c  cb  b  ba  bab  b  bc  c  cc  c  cc,eof | NO  Yes  N  Y  N  N  Y  N  Y  N  N  Y  N  Y  N  Y  N  N | 1  2  4  3  5  2  3  10,eof |

最后输出的LZW码字：1，2，4，3，5，2，3，10，eof。其中eof是end-of-file的缩写，是终止符号。

### 解压缩

LZW的解码器也需首先建立初始化字典：字典或是由信源符号集构成，或是由扩展的ASCII码构成。解码的第一步，输入第一个码字，并从字典中取回一个词条I，并将I输出，同时将Ix存入解码字典中。但此时，x是未知的，x将是下一个从字典中读取的词条的首个字符。再输入下一个指针，又从字典中取回词条J，将其输出，并把J的首字符赋予上一步存入字典的词条Ix的x，则此时，Ix已完全确定。重复上述过程，则自动重建了译码表，并将译码输出。

以上题来说明解码过程，其接收码字为1,2,4,3,5,2,3,10，eof：

1）先建解码的初始化字典，此处是信源符号a，b，c，同于编码器；

2）读入第一个接收码字（即第一个指针）“1”，从字典中取出为a，将号a输出，同时将Ix=ax存入字典作为码字4；

3）再读入下一个接受码字“2”，从字典中取出为b，一方面将b输出，同时确定上一步中x=b，得字典中码字4即Ix=ab；然后又将b置于I，又存入新词条Ix=bx作为码字5；

4）读入下一个接受码字“4”，从字典中取出为ab，输出符号ab；一方面确定上一步中x=a，得码字5为ba；一方面又将ab置于I，又存入新词条Ix=abx作为码字6；其中x将由下一个输入码字从字典中读取的词条的首个字符确定；

重复上述过程，则重建了译码表，又译码输出序列为ababcbabccc。

## 设计方案

### 压缩策略

FPGA的压缩策略见下



为了提高压缩率，避免出现越压越大的情况出现，开启数据压缩后，还要区分可压缩报文/不可压缩报文。

可压缩报文/不可压缩报文区分：

可压缩报文：压缩后长度小于原始长度的报文。

不可压缩报文：压缩后长度大于原始长度的报文。这类报文不压缩。

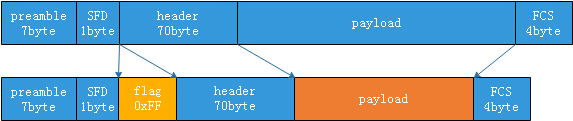
报文经过压缩模块，对报文的净荷开始压缩，同时缓存未压缩的报文净荷。比较压缩后的报文净荷长度和原始报文净荷长度，如果后长度小于原始长度，则输出压缩报文，否则输出原始报文。为了区分报文是否被压缩，在净荷的第一个字节前增加标志位flag。因为要比较压缩后的长度，所以需要缓存一个报文。

#### 报文格式

注：8000上的帧头压缩，最长DA/SA(12)+2VLAN(8)+TYPE(2)+IPV6(40)+UDP(8)=70字节，所以净荷压缩从第71字节开始，flag插入的位置在SFD后面。小于74（其中4byte是FCS）字节的报文不做净荷压缩。

##### 可压缩报文

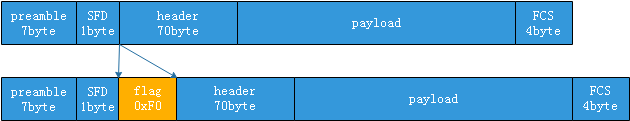
可压缩报文：



FCS重生。

##### 不可压缩报文

不可压缩报文



#### 压缩具体方法

算法主体部分，完成查字典和编码。

压缩启动时，在数据流中插入一个压缩start报文，则在这个start报文后的数据报文都是压缩报文，接收端解析到start报文后，开始解压数据。

压缩终止时，在数据流中插入一个压缩end报文，则在这个end报文后的数据报文都是非压缩报文，接收端解析到end报文后，不解压数据。



Start报文和end报文，选一种实际以太网中不可能用的格式，比如type字段为0x0000。

##### Start报文

Start报文：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Preamble  7byte | SFD  1byte | Flag（2byte）  0x0000 | Start （2byte）  0x0000 | Padding  56byte | FCS  4byte |

##### End报文

End报文：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Preamble  7byte | SFD  1byte | Flag（2byte）  0x0000 | end （2byte）  0xffff | Padding  56byte | FCS  4byte |

##### 字典

2）字典

字典的存储和维护管理。

输入8bit，则在输入任何数据前，先占据了字典的前256项，从257开始存放字典编码。

字典大小：16K

字典指针：14bit（即编码后的数据位宽）

字典模型：

|  |  |  |
| --- | --- | --- |
| 字典编码项 | 前缀编码项 | 当前项x |
| 14bit | 14bit | 8bit |

查字典：

接收到一个字节（即当前码），查字典，目标：当前码+前缀码=字典的当前项+前缀编码项，如果有，则得到一个字典编码项；如果没有，则新建一个新的字典编码项。

每收到一个字节，都要查字典，这就对查字典的速率提出了很高的要求，1clk要完成一次查表。

为了提高查字典速率，把一个字典分成32个小字典，并行查找。

FPGA使用xilinxK7-325，BRAM结构如下表：

XILINX M18K

|  |  |
| --- | --- |
| simple dual-port mode | 32bit\*512 |
| true dual-port mode | 16bit\*1024 |

为了尽可能提高单个BRAM的利用率

Size of dictionary: 16K

Dictionary pointer: 14bit

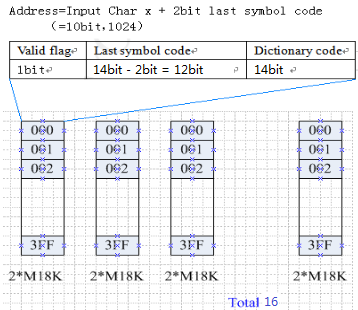
TX Dictionary model:

|  |  |  |
| --- | --- | --- |
| Dictionary code | Last symbol code | Input Char x |
| 14bit | 14bit | 8bit |

Use Input Char x +2bit前缀编码（=10bit，1024）as the address of small dictionaries. The size of each dictionary is 1024\*27bit (2\* M18K)用true dual-port mode

每个字典条目存储的内容：

|  |  |  |
| --- | --- | --- |
| Valid flag | Last symbol code | Dictionary code |
| 1bit | 14bit – 2bit = 12bit | 14bit |



每个小字典大小为1k （tx\_small\_dictionary.v），一共16k，32个M18K。

每收到一个字节，16个小字典并行查找，如果有效标志为1且匹配上，则得到对应的字典编码，如果16个小字典都匹配不上或有效标志为0，则新建一个新的编码项。

字典更新：

字典手动更新，软件配置更新字典才会更新，否则就一直不变。

##### 位宽转换

压缩后的位宽会变成14bit，要再转成8bit才能传输。简化设计，14bit加上2bit填充。

### 解压缩策略



1）位宽转换

16bit转8bit

2）算法实现

完成查字典和解压

3）字典

字典的存储和维护管理。

字典大小：16K

字典指针：14bit

字典模型：

|  |  |  |  |
| --- | --- | --- | --- |
| Valid flag | Dictionary code  （as address） | Last symbol code I | Input Char x |
| 1bit | 14bit | 14bit | 8bit |

Size of RX dictionary：16K\*23bit，一共10\*M36K+1\*M18K

收到待解压数据，直接寻址查字典。

收端字典是发端发过来的。详见2.2

4）解压后的数据反序

当接收到的压缩数据不是字典初始化项时（即能压缩的情况）。假设接收码为6个字节压缩成1个压缩码字的情况，解压后得到的原始数据字节序是反的，顺序为：

A5 A4 A3 A2 A1 A0

输出的时候还需要将字节序反过来。

解压后的数据为正反序交替的格式：（高电平为反序字节部分，低电平为正序字节部分）

\_\_\_\_\_\_-----------------\_\_\_\_\_\_\_\_\_\_------\_\_\_\_\_--------------\_\_\_\_\_\_\_\_\_\_

1 2 3 4 1 2 3 4 5 6 7 8 1 2 3 4 5 6 7 1 2 1 2 3 4 1 2 3 4 5 6 1 2 3 4 5 6 7

数出反序部分的字节数量并记录，写入一个fifo1

所有数据包括正反序flag写入数据fifo2

当整个报文都存入fifo后，开始从fifo2中读数，当flag为低时，一直读，读出的数写入结果RAM，写地址累加，当读到flag为高，则从fifo1中读出一个数，判断反序的字节个数，再从fifo2中读数并写入RAM，此时写地址在前面的地址上+8，+7~+1，通过控制写地址的方式，将写入结果RAM的数据改成正常字节序。

##### 字典同步

软件配置，比如字典学习1min以后，软件配置，字典锁定，不再建立新表项，开始同步。

16个小字典，每个小字典27bit\*1024，每个小字典可以分成512包长（净荷长度）的包，一个小字典分8个包发送，总共32\*8 = 256字典同步报文格式：



Flag：0x0F,字典同步包标志，区别于普通的业务报文。

No：bit[2:0] = 每个小字典中的第几个包

Bit[7:3] = 第几个小字典

Data：虽然小字典的位宽是27bit，为了方便起见，按照32bit传。

同步一次字典，占用空口的数据量为521byte\*256=524.8Kbit

当RX收到所有的字典同步包且无CRC校验错误，则发消息通知TX端可以开始启动压缩。字典同步期间，压缩是关闭的。

##### 字典更新

手动模式：即用户或者软件配置更新，逻辑才会更新，否则就一直用下去。

### 存在风险

#### 字典

字典怎样可靠的建立？

2）发端的字典发给收端

字典同步会引入额外的开销，导致吞吐量降低，字典建立一次需要同步的数据量为0.6Mbit。字典同步报文优先级？

#### RAM资源

字典越大，压缩效果越好。但是字典的大小受限于RAM资源。

|  |  |  |
| --- | --- | --- |
| 字典大小 | 编码后位宽 | RAM资源（M18K） |
| 32K | 15bit | 96（包括压缩和解压端） |
| 16K | 14bit | 33\*M18K+10\*M36k |
| 8K | 13bit |  |

注：K7-325890M18K

#### 查表速率

LZW算法要求1clk完成查表和匹配，对于1G的业务，也就是125MHz\*8bit，要求8ns内完成查表和匹配，这一块的时钟提高到250M，1clk查表，1clk匹配。2clk完成查表和匹配。

#### 解压速率

解压数据反序及逐字节输出耗时翻倍的问题。解压模块时钟也要提速到250M。

### 模块说明

模块说明见下表

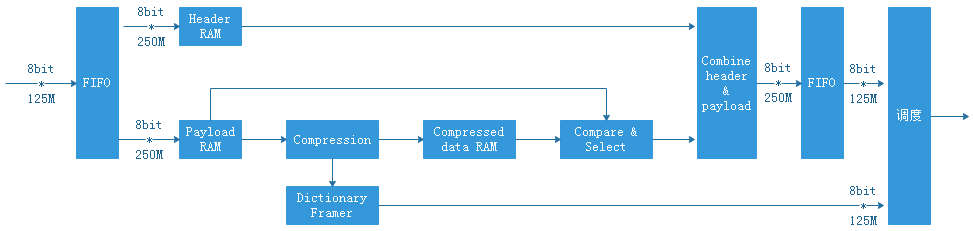
表4.1 模块说明

|  |  |  |  |
| --- | --- | --- | --- |
|  | 对应的模块 | 对应的源文件 | 说明 |
| 顶层模块 |  |  | 顶层模块 |
| 功能块1 |  |  |  |

模块整体结构图，给一个模块的整体结构图，功能框图

#### 压缩方向

压缩处理流程图



##### lzw\_forward\_prepare模块名称

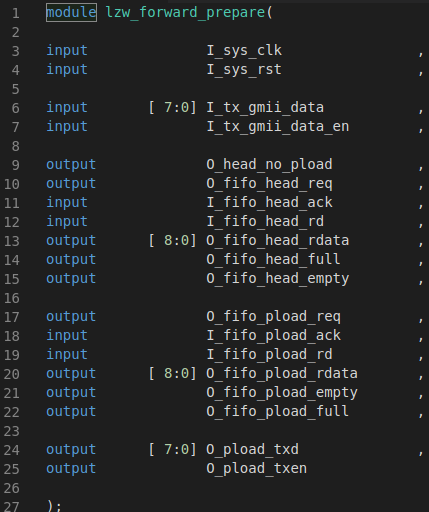
lzw\_forward\_prepare.v文件

###### 功能说明

该模块负责从普通以太网报文中将普通报文的Header头部分与Payload部分剥离开，把Header头部分放到Header RAM里面，剩下部分报文作为Payload送到lzw\_forward\_compress.v压缩模块中

###### I/O说明

顶层I/O接口说明如下：

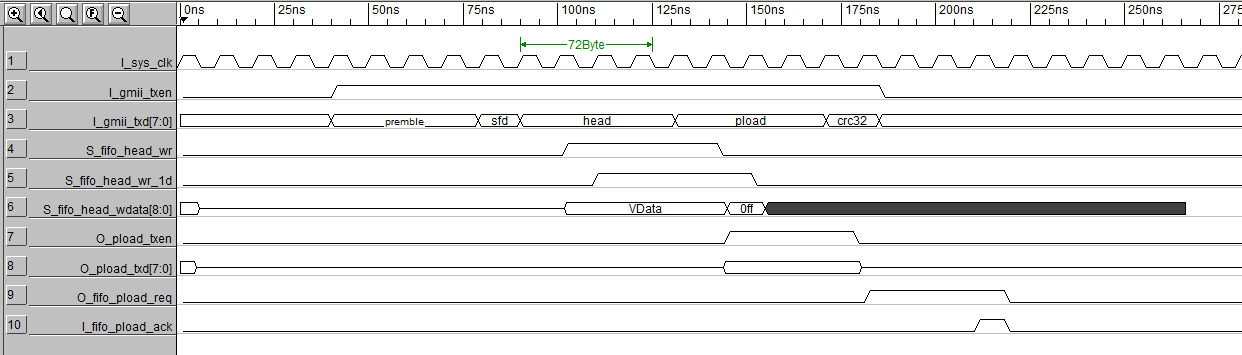


O\_head\_no\_pload信号表示，当检测到的报文小于72Byte时，报文只有head部分，没有pload，需要给出一个pulse指示信号，该信号直接送给后续的lzw\_forward\_framer模块中。

###### 详细说明

时序说明如图所示：

情况1：报文Head部分长度>72Byte，也就是有Pload部分内容



情况2：报文Head长度<=72Byte，没有pload部分

##### lzw\_forward\_compress模块名称

lzw\_forward\_compress.v文件

###### 功能说明

该模块作为前向最重要的模块，负责将payload部分内容进行lzw压缩，

该模块内部又包含lzw\_forward\_dictionary.v模块，负责读写字典的功能。

###### 详细说明

代码已经实现，略。

##### lzw\_forward\_framer模块名称

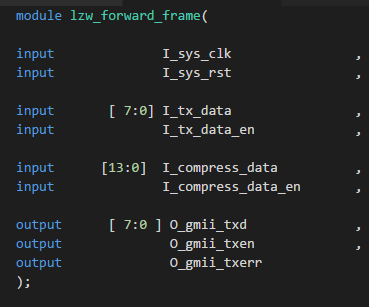
lzw\_forward\_framer.v文件

###### 功能说明

该模块实现lzw前向的组帧功能。

###### I/O说明

顶层I/O接口说明如下：



###### 详细说明

##### 子模块名称

xxx.v文件

###### 功能说明

介绍实现什么功能

###### I/O说明

介绍实现什么功能

###### 详细说明

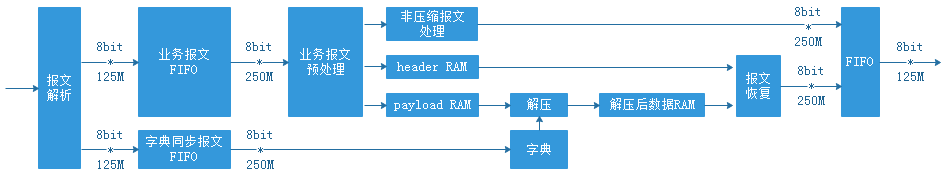
1. 表4.5 接收方向模块接口信号列表

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **名称** | **方向** | **有效电平** | **位宽(bit)** | **说明** |
| I\_reset\_n | I | 低 | 1 | 复位 |
| I\_125m\_clk | I | clock | 1 | 系统时钟 |
| I\_gmii \_dv | I | 高 | 1 | Gmii数据有效 |
| I\_gmii\_d | I | Bus | 8 | Gmii数据 |
|  | | | | |
| O\_gmii\_en | O | 高 | 1 | 内部Gmii接口数据有效，不包含FCS |
| O\_gmii\_d | O | Bus | 8 | 内部Gmii接口数据 |
| O\_gmii\_sfd | O | 高 | 1 | 内部Gmii接口SFD标识 |
| O\_gmii\_fcs | O | 高 | 1 | 内部Gmii接口CRC标识 |
| O\_gmii\_vlantag | O | 高 | 1 | 内部Gmii接口Vlantag检测 |
| O\_gmii\_fcs\_err | O | 高 | 1 | 内部Gmii接口CRC错误检测 |
| O\_gmii\_fcs\_check | O | 高 | 1 | 内部Gmii接口CRC错误检测有效，与O\_gmii\_ fcs\_err配合使用 |
| 与寄存器模块的接口 | | | | |
| O\_state\_machine | O | bus | 3 | 状态机状态 |
| 可测试性 | | | | |
| statistic\_pulse | O | bus | 4 | 性能统计 |

介绍实现什么功能，最好详细到门级。

#### 解压缩方向

解压缩处理流程图



### 模块协作说明

#### LZW在FPGA系统中的位置

LZW压缩模块在RMUN单板上实现，具体位置如图所示

压缩方向 1588---->LZW--->HC

LZW将压缩后的报文送给HC模块即可。

### 关键时序说明

比如可以给出AXI总线的关键时序信号介绍。

## 接口说明

LZW模块具体使用放下说明如下：

1. 用户配置净荷压缩使能
2. 设备基于收到的以太网业务流净荷，学习数据，开始建立压缩字典。
3. 字典建立以后，收发两端进行字典同步。
4. 完成上述步骤后，发送端开始压缩净荷，接收端解压缩。
5. 若软件配置更新字典，则重复2）3）4）。

### 硬件接口

#### DDR3接口

见下文

### 软件接口

#### 地址空间分配

表5.7 地址分配

|  |  |  |
| --- | --- | --- |
| 地址范围 | 含义 | 模块 |
| 0x0000－0x0FFF | 系统全局部分 |  |
| 0x1000－0x2FFF | 统一GPI部分 |  |

#### 寄存器地址列表

#### 寄存器说明

##### 通用寄存器说明

见下文

###### CHIP\_ID(0x0000,R)

见下文

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 寄存器位数 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 寄存器定义 | 逻辑下载成功标识寄存器 | | | | | | | | | | | | | | | |
| 说明 | 只读寄存器，系统复位后为0x55aa | | | | | | | | | | | | | | | |

### 内部接口

比如ap\_none接口，ap\_valid接口等等

## 器件介绍

介绍一些关键芯片，FPGA，DDR，FLASH等等

### FPGA

## 参考资料

[1] Lattice，HB1009 LatticeECP3 Family Handbook v1.7，2010