

目录

1	背景介绍.....	3
2	设计流程.....	4
3	设计目标.....	5
3.1	目标配置.....	5
3.2	目标位流.....	6
4	参考软件设计与验证.....	7
4.1	参考软件结构.....	7
4.2	软件验证方法.....	8
4.3	硬件验证方法.....	10
5	RTL 设计与验证.....	11
5.1	顶层模块.....	11
5.1.1	设计.....	11
5.1.2	接口.....	12
5.1.3	验证.....	12
5.2	滤波模块.....	14
5.2.1	原理.....	14
5.2.2	设计.....	15
5.2.3	接口.....	16
5.2.4	验证.....	17
5.3	LZ77 模块.....	18
5.3.1	原理.....	18
5.3.2	设计.....	19
5.3.3	接口.....	21
5.3.4	验证.....	22
5.4	哈夫曼 (huffman) 模块.....	23
5.4.1	原理.....	23
5.4.2	设计与接口.....	24
5.4.3	验证.....	24
5.5	ADLER32 模块.....	25
5.5.1	原理与设计.....	25
5.5.2	接口.....	25
5.5.3	验证.....	25
5.6	CRC32 模块	26
5.6.1	原理与设计.....	26
5.6.2	接口.....	26
5.6.3	验证.....	27
5.7	位流 (bs) 模块.....	27
5.7.1	原理.....	27
5.7.2	设计与接口.....	27
5.7.3	验证.....	28
6	门级设计与验证.....	28
6.1	Designer Compiler.....	28

6.1.1	流程.....	28
6.1.2	关键路径优化.....	29
6.1.3	综合结果.....	30
6.2	Primetime.....	30
6.3	Formality	31
7	物理级设计与验证.....	33
7.1	IC Compiler.....	33
7.1.1	Setup.....	34
7.1.2	Read Design.....	34
7.1.3	Floorplan.....	35
7.1.4	Preroute.....	36
7.1.5	Place.....	37
7.1.6	Clock Tree Synthesis.....	37
7.1.7	Route.....	38
7.1.8	Fix & DRC & LVS.....	38
7.1.9	Export Design and Reports.....	40
7.2	IO Pad	42
7.3	Primetime.....	43
7.4	Formality	43
8	总结.....	43

1 背景介绍

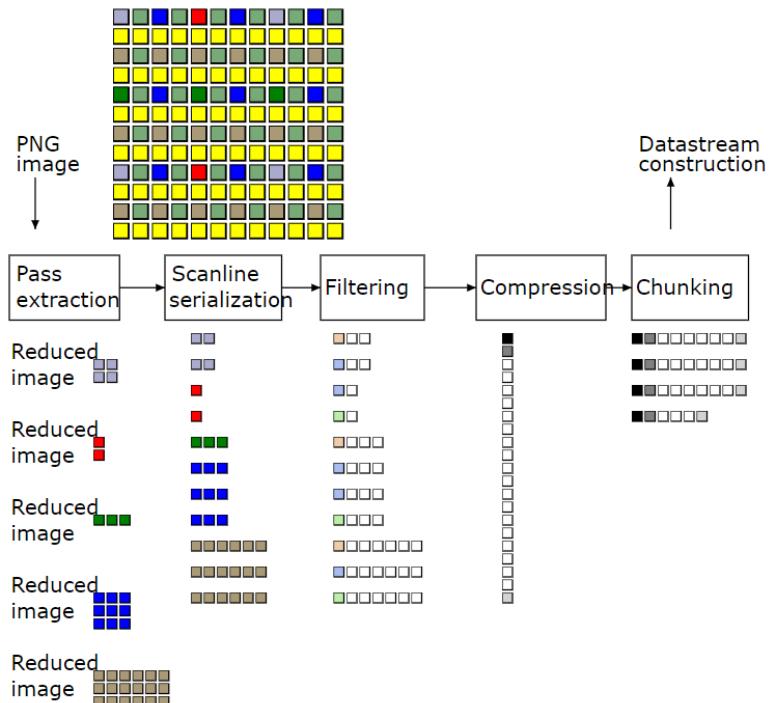
PNG (Portable Network Graphics, 便携式网络图形) 是一种支持无损压缩的位图图形格式。与 RAW、JPEG、GIF 等其他常见图形格式相比，PNG 具有无损压缩、支持多种颜色类型、支持透明特性、优化的网络传输显示、避免专利影响等特性，因此成为了目前主流的图像格式之一，在一些场合中得到了非常广泛的应用。

PNG 图像压缩标准即是描述了 PNG 位流要求符合的格式（语法语义）以及从位流还原出原始图像像素（解析解码）的过程的规范。PNG 的 1.0 版本规范于 1996 年发布，在这之后经过了几次修改。现行版本是国际标准（ISO/IEC 15948:2003），并在 2003 年作为 W3C 建议发布，该版本可以在网络上得到。

PNG 标准的实现即主要是 PNG 编码器和 PNG 解码器，前者负责将原始图像像素编码成符合 PNG 标准的位流，后者负责将符合 PNG 标准的位流解码成原始图像像素。根据我们小组目前能搜索到的资料，PNG 标准主要的开源软硬件参考如下所示：

类型	项目名	项目简介
软件	libpng	官方 PNG 开源参考库，支持 PNG 标准的几乎所有的特性 (www.libpng.org/pub/png/libpng.html)
	lodepng	无依赖的 PNG 开源编解码库，支持 PNG 标准的大部分的特性 (lodev.org/lodepng/)
硬件	无	

综合以上背景和课程设计要求，我们小组选题计划采用 verilog 语言实现一个简化的 PNG 硬件编码器芯片。PNG 编码器功能是将原始图像像素编码成符合 PNG 标准的位流，该位流的大小要比原始图像像素小得多，从而达到节省存储空间和传输带宽等的目的。根据 PNG 标准，图像的压缩过程主要可以分为六个步骤，如下所示：



(1) Pass extraction: 如果需要支持渐进式显示, 原始图像像素将按照 Adam7 顺序被重新排列成几个更小的子图, 称为 reduced images 或 passes; 如果不需要支持渐进式显示, 这一步被跳过。

(2) Scanline serialization: 图像像素数据(包括了各通道或索引)按行排列形成 scanline, 然后由一行行的 scanline 排列形成图像像素数据的字节流。

(3) Filtering: 分别对每行 scanline 使用 PNG 标准规定的五种滤波类型之一进行滤波, 以提高后续的压缩效率。由滤波后的值形成每行 scanline 数据的新的字节流, 并在每行前添加一个字节来指示该行的滤波类型。

(4) Compression: 对滤波后的字节流进行 deflate 压缩。deflate 压缩是 PNG 的核心算法, 是同时使用了 LZ77 和 Huffman Coding 的一种无损压缩算法: LZ77 是通过“滑动窗”使用已经出现过的相应匹配信息替换当前数据, Huffman Coding 是将 LZ77 压缩后的符号结果进行编码压缩。deflate 压缩后得到的位流需要封装成符合 zlib 标准的位流。

(5) Chunking: 压缩后的数据和图像的其他信息按照 PNG 标准分别封装成对应 chunk。

(6) Datastream construction: 各 chunk 和 PNG signature 按照 PNG 标准组织形成最终的 PNG 图像文件/位流。即最终的 PNG 图像文件/位流包括了一个 PNG signature 和其后的一系列 chunk。

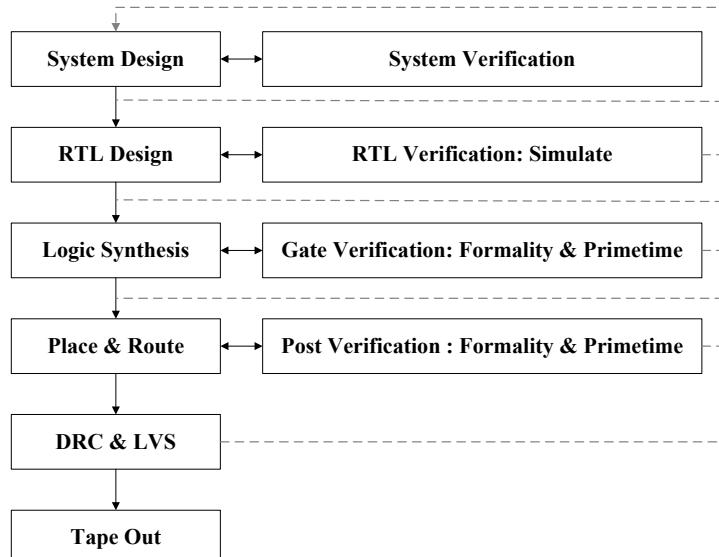
以上的详细过程和具体细节可参见 PNG 标准 (ISO/IEC 15948:2003)、zlib 标准 (RFC 1950) 和 deflate 标准 (RFC 1951), 限于篇幅这里不做过多描述。

我们小组的选题, 从选题内容角度而言, 与本课程和专业都紧密相关; 从选题意义角度而言, 硬件编码器往往比软件编码器具有更好的性能, 具有很强的实用性; 另外还可作为 PNG 编码甚至其他音视频图像信号处理芯片在具体实现和流程等方面上的一个参考, 具有很强的学习性。

2 设计流程

数字芯片设计的一般流程如下所示, 主要包含了 5 个层次及其对应层次的设计与验证。若在后续步骤中发现设计不满足某约束条件, 可能需要回退到之前的步骤进行迭代。

在本设计中, 系统级对应于参考软件设计与验证, RTL 级对应于 RTL 设计与验证, 门级对应于门级设计与验证, 物理级和工艺级对应于物理级设计与验证。



层次	步骤	描述
系统级	系统设计 System Design	确定系统的功能和性能指标，设计芯片的总体架构、模块划分和算法选择等。
	系统验证 System Verification	使用 C、C++、Matlab 等高级语言对系统、算法进行建模实现和验证。
RTL 级	RTL 设计 RTL design	使用 Verilog HDL、VHDL 等硬件描述语言对电路进行 RTL 级代码描述。
	RTL 验证 RTL Verification	使用 Modelsim、VCS 等工具通过仿真对 RTL 设计进行功能验证
门级	逻辑综合 Logic Synthesis	使用 DC 等工具将 RTL 设计转换成特定工艺下的、符合约束条件的门级网表。
	门级验证 Gate Verification	使用 Primetime 和 Formality 等工具对逻辑综合后的门级网表进行时序和等效性的验证。
物理级	布局布线 Place & Route	使用 ICC 等工具对门级网表的标准单元模块等确定其版图上的位置和完成节点的连接等。
	后验证 Post Verification	使用 Primetime 和 Formality 等工具对布局布线后的网表进行时序和等效性的验证
工艺级	版图检查 DRC & LVS	使用 Calibre 等工具对版图进行设计规则检查、逻辑图网表和版图网表比较。
	流片 Tape Out	将 GDSII 格式的版图交付晶圆代工厂进行制造。

3 设计目标

3.1 目标配置

PNG 编码器只要求将原始图像像素编码成符合 PNG 标准的位流，这样即保证了可以根据 PNG 标准，从编码后的 PNG 位流解码还原出原始图像像素。至于需要支持的特性和实现的方法等，则可以根据编码器在实现时的性能、成本、应用场合等目标来自行决定。

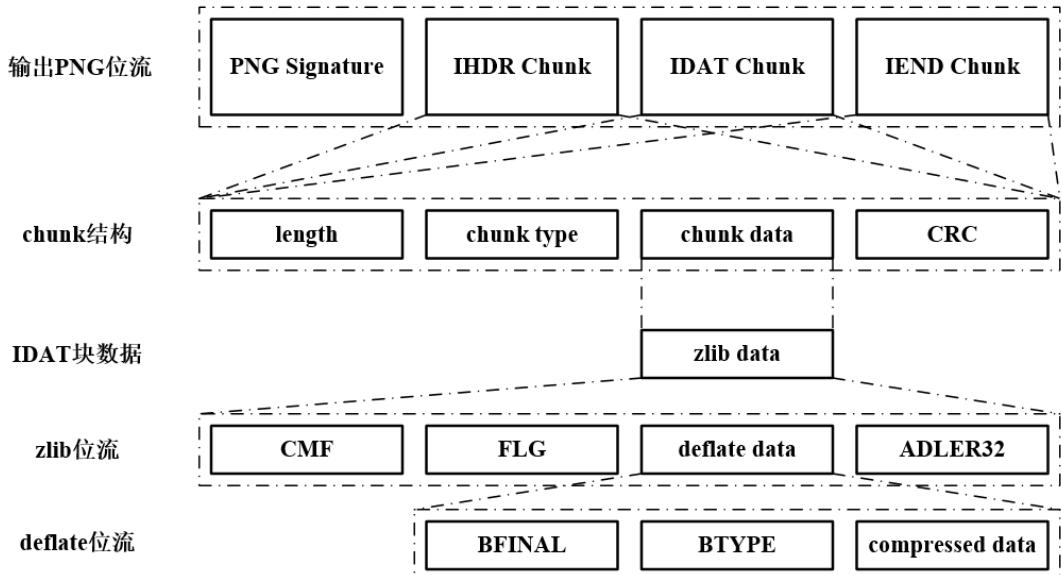
例如，PNG 标准规定在滤波过程中分别对每行 scanline 使用标准规定的五种滤波类型之一进行滤波，但没有限制如何确定要使用哪一种滤波类型，编码器可以使用固定的方法，这样实现简单但压缩效率较低，也可以使用根据图像自适应的方法，这样压缩效率高但实现较复杂。再例如，PNG 标准规定对滤波后的字节流进行 deflate 压缩，而 deflate 压缩包括了无压缩、LZ77 并以固定哈夫曼编码压缩、LZ77 并以动态哈夫曼编码压缩三种类型，对于要使用哪一种 deflate 压缩类型、如果使用了 LZ77 那么要如何实现 LZ77 等方面，PNG 标准没有作出限制，而应该根据编码器的实现目标来权衡决定。

在本设计中，考虑到时间有限、工作量大、没有现有的开源硬件资料可以参考等因素，最终对支持的配置的选择如下所示，以降低设计复杂度和硬件实现代价，但同时也保持有较好的实用性能。此外，目标时钟频率定为 125MHz，以保证较好的实时性。

类型	配置变量	支持的配置项
原始	图像像素	RGBA 四通道, 每通道 8 比特
	图像宽高	不超过 511 像素
输出	色彩类型	RGBA
	样本位宽	8
	块类型	1 个 IHDR、1 个 IDAT、1 个 IEND
	渐进式显示	无渐进式显示
	滤波策略	每行计算最小和, 自适应选择该行滤波类型
	压缩策略	1 个 LZ77 并以固定哈夫曼编码的压缩块

3.2 目标位流

目标位流即对于编码器的输出 PNG 位流的目标配置。本设计中编码器目标的输出 PNG 位流包括了一个 PNG signature 和其后的一个 IHDR chunk、一个 IDAT chunk 和一个 IEND chunk，如下所示：



(1) PNG signature 为固定的八个字节“137 80 78 71 13 10 26 10”，标识了当前位流是 PNG 位流。

(2) chunk 结构由 length、chunk type、chunk data、CRC 四个域依次组成，如下所示：

域	字节数	内容
length	4	chunk data 域的字节数
chunk type	4	块的类型，如 IHDR、IDAT、IEND 等
chunk data	0~ 2^{31} -1	块的数据，可以为空
CRC	4	chunk type 和 chunk data 域的四字节循环冗余校验码

(3) IHDR chunk 的块数据为固定的十三个字节，包括了图像的宽高、样本位宽、色彩类型、渐进式显示等信息。

(4) IDAT chunk 包括了压缩后的图像数据，块数据为生成的字节数不固定的 zlib 位流。而 zlib 位流结构由一个字节 CMF、一个字节 FLG、字节数不固定的 deflate 位流、四个字节

ADLER32 依次组成；deflate 位流则由一个比特 BFINAL、一个比特 BTYPE、比特数不固定的压缩后位流依次组成。

(5) IEND chunk 标识了 PNG 位流的结束，块数据为空。

关于输出 PNG 位流中 chunk 结构、zlib 位流、deflate 位流、压缩后位流等的更详细具体的定义规范可参见 PNG 标准、zlib 标准和 deflate 标准，限于篇幅这里不做过多描述。

4 参考软件设计与验证

4.1 参考软件结构

参考软件设计即采用 C、C++、Matlab 等语言对系统算法进行建模实现，是设计流程中系统算法级设计的一个重点。一方面，参考软件能够快速测试和验证方案，以指导硬件设计工作；另一方面，参考软件能够作为设计的一个黄金模型，以加快硬件设计中的测试和验证过程。

本设计中基于开源 PNG 编解码软件库 lodepng 搭建硬件对应的参考软件（c model）。lodepng 的代码文件主要包括 lodepng.h 和 lodepng.cpp 两个文件、共 8443 行代码，还有其余几个代码文件提供了一些编程使用示例。搭建参考软件设计的过程中进行了大量多方面的工作。

首先，lodepng 支持 PNG 规范的大部分的特性，例如编码和解码、辅助块、多种色彩类型和样本位宽、多种压缩策略、渐进式显示等，本设计中对不需要支持的配置进行了裁剪。

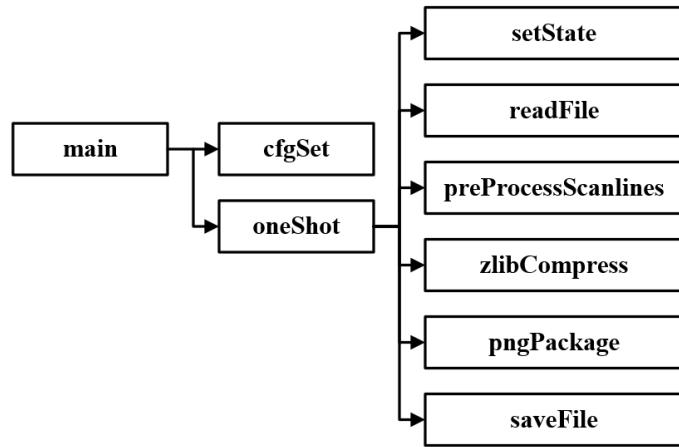
其次，lodepng 本身作为一个应用程序编程库，留出了大量接口，同时调用关系繁杂，也没有给出主处理过程，本设计中进行了编写主处理函数、明确程序流程、简化使用接口和调用。

最后，lodepng 原始的结构、划分和算法都是面向软件的，并不能对应于硬件设计和用于硬件的测试验证，本设计中进行了大量层次化、模块化的重构和修改，使参考软件能适应于硬件的实现工作，同时具有较好的可维护性、可扩展性。

本设计中最终设计的参考软件的文件结构如下所示：

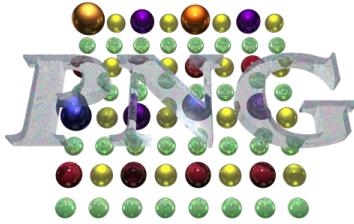
文件名	文件主要功能描述
mypng.hpp	头文件，存放公共的结构和函数
mypng.cpp	主处理
mypng_parse.cpp	解析编码参数，读写文件
mypng_filter.cpp	执行 PNG 图像压缩的滤波过程
mypng_deflate.cpp	执行 PNG 图像压缩的 deflate 压缩过程
mypng_chunk.cpp	执行 PNG 图像压缩的 PNG 位流组织封装过程
mypng_dump.cpp	导出数据用于硬件测试验证

本设计中参考软件的主要运行流程为，首先解析编码参数，然后根据编码参数读入原始图像，并按照 PNG 标准执行 PNG 图像压缩过程，最后将生成的 PNG 位流写入到文件中。主要函数调用关系如下所示，其中 preProcessScanlines()、zlibCompress()、pngPackage() 三个函数即执行 PNG 图像压缩过程的主要函数：



4.2 软件验证方法

为了验证参考软件的正确性，本设计中选择了五幅 PNG 图像，将其解码得到的原始图像像素的 RGBA 值按照参考软件的目标输入格式保存到文件中，作为参考软件的测试验证数据集，如下所示：

图像信息	内容	RGBA 值（截取部分）																		
AlphaBall.png		<table> <tr><td>262136</td><td>3</td></tr> <tr><td>262137</td><td>0</td></tr> <tr><td>262138</td><td>0</td></tr> <tr><td>262139</td><td>254</td></tr> <tr><td>262140</td><td>3</td></tr> <tr><td>262141</td><td>0</td></tr> <tr><td>262142</td><td>0</td></tr> <tr><td>262143</td><td>255</td></tr> <tr><td>262144</td><td>3</td></tr> </table>	262136	3	262137	0	262138	0	262139	254	262140	3	262141	0	262142	0	262143	255	262144	3
262136	3																			
262137	0																			
262138	0																			
262139	254																			
262140	3																			
262141	0																			
262142	0																			
262143	255																			
262144	3																			
AlphaEdge.png		<table> <tr><td>262136</td><td>255</td></tr> <tr><td>262137</td><td>0</td></tr> <tr><td>262138</td><td>0</td></tr> <tr><td>262139</td><td>254</td></tr> <tr><td>262140</td><td>255</td></tr> <tr><td>262141</td><td>0</td></tr> <tr><td>262142</td><td>0</td></tr> <tr><td>262143</td><td>255</td></tr> <tr><td>262144</td><td>255</td></tr> </table>	262136	255	262137	0	262138	0	262139	254	262140	255	262141	0	262142	0	262143	255	262144	255
262136	255																			
262137	0																			
262138	0																			
262139	254																			
262140	255																			
262141	0																			
262142	0																			
262143	255																			
262144	255																			
black817-480x360-3.5.png		<table> <tr><td>691192</td><td>0</td></tr> <tr><td>691193</td><td>0</td></tr> <tr><td>691194</td><td>0</td></tr> <tr><td>691195</td><td>0</td></tr> <tr><td>691196</td><td>0</td></tr> <tr><td>691197</td><td>0</td></tr> <tr><td>691198</td><td>0</td></tr> <tr><td>691199</td><td>0</td></tr> <tr><td>691200</td><td>0</td></tr> </table>	691192	0	691193	0	691194	0	691195	0	691196	0	691197	0	691198	0	691199	0	691200	0
691192	0																			
691193	0																			
691194	0																			
691195	0																			
691196	0																			
691197	0																			
691198	0																			
691199	0																			
691200	0																			

globe-scene-fish-bowl-pngcrush .png		787504 0 787565 0 787566 0 787567 0 787568 0 787569 0 787570 0 787571 0 787572 0
imgcomp-440x330.png		580792 0 580793 0 580794 0 580795 0 580796 0 580797 0 580798 0 580799 0 580800 0

参考软件依次以测验证数据集中的 RGBA 数据为输入，编码得到对应的输出 PNG 图像文件。然后采用 python 中的 PNG 编解码库 pypng，对参考软件输出的 PNG 图像文件进行解码，并将解码得到的 RGBA 数据与输入的 RGBA 数据进行对比，完成验证过程。完整的软件验证脚本如下所示：

```

1  #!/bin/bash
2  # build
3  ./build.sh
4
5  #           file_name           width   height
6  LIST_INFO=(                                )
7  |   AlphaBall          256    256
8  |   AlphaEdge           256    256
9  |   black817-480x360-3.5 480    360
10 |  globe-scene-fish-bowl-pngcrush 393    501
11 |  imgcomp-440x330       440    330
12 |  test                 512    512
13 )
14 PATH_RGBA_ANCHOR="./pic_rgba_anchor"
15 PATH_RGBA="./pic_rgba"
16 PATH_PNG="./pic_png"
17
18 # main loop
19 cnt=0
20 num=${#LIST_INFO[*]}
21 while [ $cnt -lt $num ]
22 do
23     # get info
24     file=${LIST_INFO[$cnt]} ; cnt=$((cnt+1))
25     width=${LIST_INFO[$cnt]} ; cnt=$((cnt+1))
26     height=${LIST_INFO[$cnt]} ; cnt=$((cnt+1))
27
28     echo "-----"
29     echo "$cntFile processing $file"
30
31     # encode rgba to png
32     ./mypng -c ./png.cfg \
33             -i $PATH_RGBA_ANCHOR/$file.txt \
34             -o $PATH_PNG/$file.png \
35             -w $width \
36             -h $height
37
38     # decode png to rgba
39     python3 ./png2rgba.py $PATH_PNG/$file.png $PATH_RGBA/$file.txt
40
41     # compare rgba_anchor with rgba
42     diff -q $PATH_RGBA_ANCHOR/$file.txt $PATH_RGBA/$file.txt
43 done

```

另外，为了保证参考软件以及目标硬件实现的实用性，本设计中还在测试验证数据集上对参考软件的压缩效率进行了测试，结果如下所示：

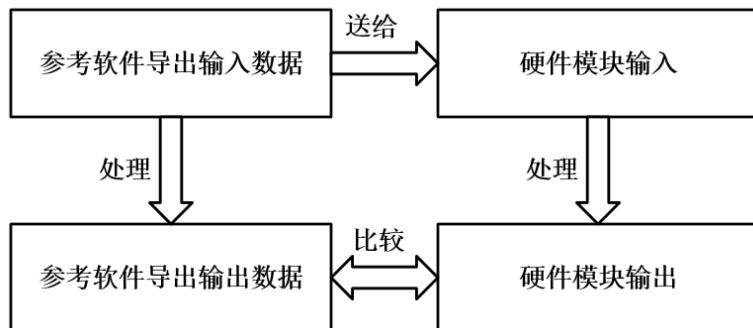
图像名	原始大小	压缩后大小	压缩效率	基准
AlphaBall.png	262144 bytes	58574 bytes	22.34%	9.26%
AlphaEdge.png	262144 bytes	58678 bytes	22.38%	9.46%
black817-480x360-3.5.png	691200 bytes	294488 bytes	42.61%	26.54%
globe-scene-fish-bowl-pngcrush.png	787572 bytes	380152 bytes	48.27%	28.90%
imgcomp-440x330.png	580800 bytes	60704 bytes	10.45%	4.51%

其中基准一栏中是没有采用面向硬件的 PNG 编码器编码出的 PNG 图像的压缩效率，要达到这样的压缩效率必然会增加设计复杂度和硬件实现代价。因此可见，目标实现的编码器确实在降低设计复杂度和硬件实现代价的前提下保持了较好的实用性能。

4. 3 硬件验证方法

参考软件的重要作用之一是作为硬件设计的一个黄金模型，加快硬件设计中的测试验证过程。由于已经采用软件验证方法验证了参考软件的正确性和实用性，因此硬件验证方法主要就是验证硬件设计的顶层模块和核心子模块的功能是否与参考软件设计一致。

本设计中在参考软件中添加了专用代码（dump），用于导出硬件设计的对应顶层模块和核心子模块的输入输出数据。在硬件模块编写的 testbench 中，从导出的数据中读取模块的输入数据，并配合其他控制信号和时序送到模块输入工作，然后检测模块输出，并与导出的参考软件输出数据比较是否一致，完成验证过程，如下所示：



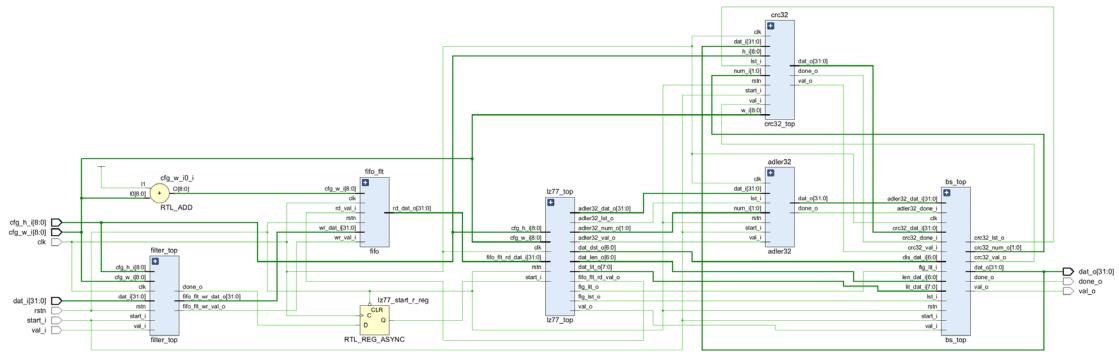
verilog 中的 \$fscanf 系统函数用于从参考软件导出的数据文件读取数据。本设计中使用 modelsim 进行 testbench 的仿真功能验证。

5 RTL 设计与验证

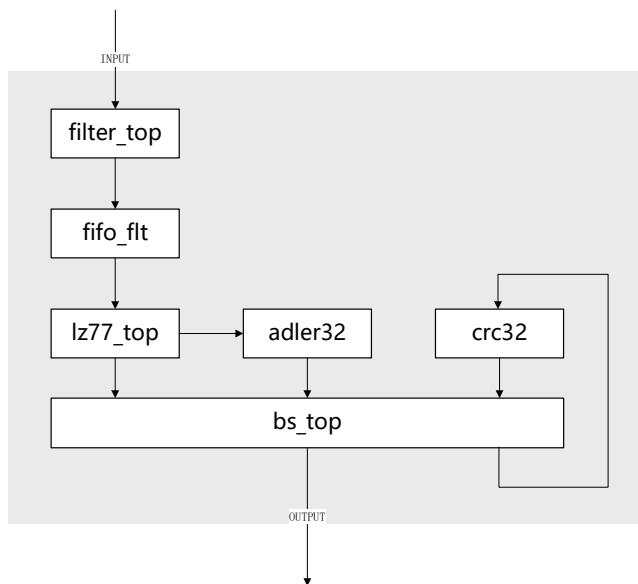
5.1 顶层模块

5.1.1 设计

本设计顶层（png_top）中包含了滤波模块（filter_top），压缩模块(lz77_top)，数据缓存模块（fifo_flt），校验码模块（crc32, adler32）和码流拼接模块（bs_top）共 6 个模块。在 Vivado2019.1 中，通过 RTL ANALYSIS 得出的电路框图如下所示，详尽地展示了模块间的互连关系。



简化的电路框图如下所示，输入原始像素首先经过 filter_top 模块进行滤波，去除相邻像素之间的空间冗余；再通过 lz77_top 模块进行基于字典的滑动窗无损压缩，去除滤波后数据间的信息熵冗余。考虑到滤波模块和压缩模块的吞吐率不同，在两模块间使用尺寸为 32x512 的自定义 FIFO 即 fifo_flt 进行数据缓冲。压缩后的数据一部分送至 adler32 模块计算校验值，一部分送至 bs_top 进行 huffman 编码以及拼接码流，bs_top 输出的部分码流送至 crc32 进行校验值计算，输出码流中包含了压缩数据与校验码。



需注意的是，在 PNG 标准中，位流中的 `length` 在 `data` 之前输出；但压缩数据是与图像内容相关的变量，压缩长度无法提前预知。如果需要硬件端输出符合标准的码流，则需要将压缩数据全部缓存起来，直至压缩结束得到长度后再依次写出。考虑到极端情况下，压缩率为 100%（无压缩）时，需要的 RAM 尺寸（ $32 \times 512 \times 512 = 1\text{MB}$ ）过大，因此最终采取的方案是在压缩过程中输出压缩数据以及各个校验码，用外部缓存存储（仿真时打印到文件中），并用软件脚本拼接成符合标准的码流。

5.1.2 接口

输入输出信号如下所示，包含了常规的时钟信号、复位信号、启动和结束信号、输入输出数据及其对应使能信号，和图像宽高的配置信号。输入输出的数据位宽均为 32bits；输入数据是 1 个包含 RGBA4 通道（每通道 8bits）的原始像素，输出数据是拼接后的码流。数据位宽越高，并行度越大，吞吐率越大，硬件面积也越大。在吞吐率与面积之间进行折中后，选取 32 作为数据位宽。图像宽高均为 9bits，最大值为 511。图像宽度越大，内部缓存使用的 RAM 面积越大，因此需要根据应用场景进行选取，此处选择 511 以压缩小尺寸的图片。

输入信号	位宽	描述	输出信号	位宽	描述
<code>clk</code>	1	时钟	<code>done_o</code>	1	结束，高有效，脉冲
<code>rstn</code>	1	复位，低有效	<code>val_o</code>	1	输出使能，高有效
<code>start_i</code>	1	启动，高有效，脉冲	<code>dat_o</code>	32	输出数据，码流
<code>val_i</code>	1	输入使能，高有效			
<code>dat_i</code>	32	输入数据，RGBA			
<code>cfg_w_i</code>	9	图像宽度			
<code>cfg_h_i</code>	9	图像高度			

5.1.3 验证

正如 4.3 介绍的硬件验证方法所言，采用基于文件的自动对比软硬件输出，便于 debug 以及批量测试。若软硬件数据不匹配则报错，并停止仿真，如下图所示。

```
// check
if( sim_flg_lit ) begin
    if ( sim_flg_lit != dut_flg_lit || sim_flg_lst != dut_flg_lst || sim_dat_lit != dut_dat_lit ) begin
        $display ("\\nLZ77_TOP ERROR: at %08d ns, lz77_data(scanline %02d-%02d) should be %01x-%02x-%02x, however is %01x-%02x-%02x \\n"
                  , $time
                  , cnt_h_r
                  , cnt_r
                  , sim_flg_lst
                  , sim_flg_lit
                  , sim_dat_lit
                  , dut_flg_lst
                  , dut_flg_lit
                  , dut_dat_lit
                );
    end
    // log
    #( 10 * `CLK_FULL );
    $stop;
end
```

对 5 张不同尺寸的测试图片都进行了验证，debug 修正后均通过，以 AlphaBall 为例进行展示。

该模块的输入数据为 RGBA，`RGBA.dat` 每行中存储了图像一行的数据。

```

rtl> sim> sim.png> check_data > 选 RGBA.dat
1  ffff0003 feff0000 feff0003 feff0003 fdff0003 fcff0003 fbff0003 faff0003 f9ff0003 f8ff0003
2  fffe0003 fefe0003 fefe0003 fefe0003 fdfe0003 fcfe0003 fbfe0003 fafe0003 f9fe0003 f8fe0003
3  fffe0003 fefe0003 fefe0003 fefe0003 fdfe0003 fcfe0003 fbfe0003 fafe0003 f9fe0003 f8fe0003
4  fffe0003 fefe0003 fefe0003 fefe0003 fdfe0003 fcfe0003 fbfe0003 fafe0003 f9fe0003 f8fe0003
5  fffd0003 fefd0003 fefd0003 fdfd0003 fcf0003 fbf0003 faf0003 f9fd0003 f8fd0003
6  fffc0003 fecf0003 fecf0003 fd0003 fcfc0003 fbf0003 faf0003 f9fc0003 f8fc0003
7  ffb0003 fefb0003 fefb0003 fefb0003 fd0003 fcf0003 fbf0003 faf0003 f9fb0003 f8fb0003
8  fffa0003 fefa0003 fefa0003 fefa0003 fd0003 fcfa0003 fbf0003 faf0003 f9fa0003 f8fa0003

```

该模块的输出数据为 zlib 码流 + zlib 长度 + IDAT / IHDR / IEND 的 CRC32 校验值。

```

rtl> sim> sim.png> check_data > 选 bs_o.dat

1  780163fc
2  ff9f81f9
3  3f03c35f
4  06060666
5  206600b2
6  411846a3
7  f3f1c931
8  a2f38198
9  05889981
10 980984a1

```

仿真开始截图

```

# vsim -voptargs="+acc" work.sim_png_top
# Start time: 16:39:10 on Jun 06,2021
# ** Note: (vsim-3813) Design is being optimized due to module recompilation...
# Loading work.sim_png_top(fast)
# Loading work.png_top(fast)
# Loading work.filter_top(fast)
# Loading work.filter(fast)
# Loading work.filter_paeth(fast)
# Loading work fifo(fast)
# Loading work.SRAM32x512_lrw(fast)
# Loading work.SRAM32x512_lrw_lbit(fast)
# Loading work.lz77_top(fast)
# Loading work.lz77_detect_one(fast)
# Loading work.adler32(fast)
# Loading work.bs_top(fast)
# Loading work.huffman_fixed(fast)
# Loading work.bs_output(fast)
# Loading work.crc32_top(fast)
# Loading work.crc32_core(fast)
#
# *** CHECK STARTS ***
#
#           function check to png is on!
#           function check to filter is on!
#           function check to lz77 is on!
#           dump to png is on!
#
#       at    100 ns, starting png  0 ... (delta cycle 3254)
#       at   32745 ns, starting png  1 ... (delta cycle 3087)
#       at   63725 ns, starting png  2 ... (delta cycle 3138)
#       at   95215 ns, starting png  3 ... (delta cycle 3200)
#       at  127325 ns, starting png  4 ... (delta cycle 3325)
#       at  160685 ns, starting png  5 ... (delta cycle 3299)
#       at  193785 ns, starting png  6 ... (delta cycle 3331)
#       at  227205 ns, starting png  7 ... (delta cycle 3423)
#       at  261545 ns, starting png  8 ... (delta cycle 3386)
#       at  295515 ns, starting png  9 ... (delta cycle 3453)

```

仿真结束截图

```

#       at 8997985 ns, starting png  254 ... (delta cycle 3493)
#       at 9033025 ns, starting png  255 ... (delta cycle 3361)
#
#
# *** CHECK DONES ***
#
# (delta cycle 903849)
# ** Note: $stop : ../../sim/sim_png/sim_png_top.v(142)
#   Time: 9067275 ns Iteration: 0 Instance: /sim_png_top
# Break in Module sim_png_top at ../../sim/sim_png/sim_png_top.v line 142

```

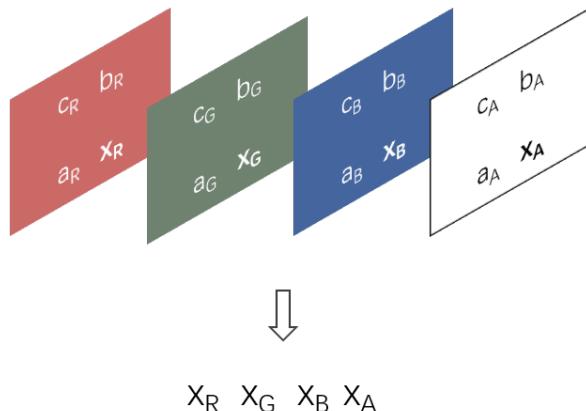
总体周期数如下表所示，其中优化指 lz77 中提前结束，详见 5.3.2。

图像名	尺寸 (WxH)	原始 周期数	优化后 周期数	优化后 / 原始
AlphaBall.png	256x256	2914422	903849	31%
AlphaEdge.png	256x256	2860292	899870	31.5%
black817-480x360-3.5.png	480x360	19671018	3519172	17.9%
globe-scene-fish-bowl-pngcrush.png	393x501	23433514	4142049	17.7%
imgcomp-440x330.png	440x330	3819733	734903	19.2%

5.2 滤波模块

5.2.1 原理

本设计支持标准中所规定的所有滤波类型，共 5 种，根据图片内容自适应选择每行的最优滤波类型。滤波时使用到的字节如下图所示，其中 x 为当前待滤波字节， a 为与 x 同行的前一字节， c 和 b 为上一行对应位置的字节。参考字节 a 、 b 和 c 为同通道的字节，通道间可并行滤波。若 x 在左上边界，则参考字节均为无效字节，置为 0。



5 种滤波类型如下表所示。

其中 Orig 表示原始数据，Filt 表示滤波后数据，floor 为向下取整。

类型	名称	函数
0	None	Filt(x) = Orig(x)
1	Sub	Filt(x) = Orig(x) - Orig(a)
2	Up	Filt(x) = Orig(x) - Orig(b)
3	Average	Filt(x) = Orig(x) - floor((Orig(a) + Orig(b)) / 2)
4	Paeth	Filt(x) = Orig(x) - PaethPredictor(Orig(a), Orig(b), Orig(c))

PaethPredictor 的伪代码如下所示，选取梯度最小的点作为预测点以计算残差。

```
p = a + b - c
pa = abs(p - a)
pb = abs(p - b)
```

```

pc = abs(p - c)
if pa <= pb and pa <= pc then Pr = a
else if pb <= pc then Pr = b
else Pr = c
return Pr

```

以行为单位累加滤波后数据的绝对值作为误差和，选取最小误差和所对应的滤波类型作为本行的滤波类型。将滤波类型（0-4），与本行滤波类型对应的滤波后数据一同送至后续模块进行数据压缩。

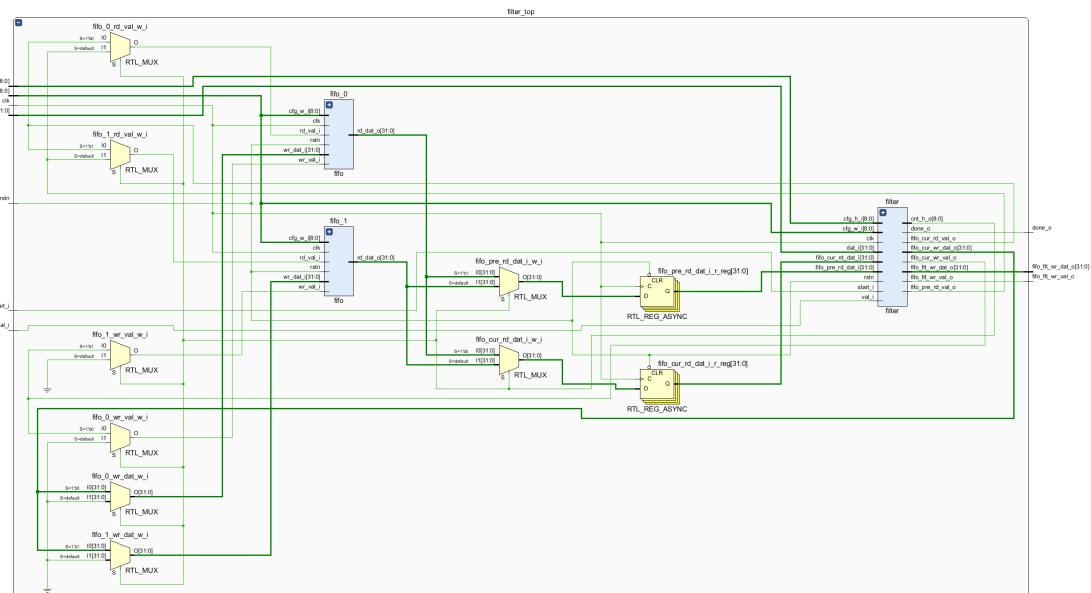
5. 2. 2 设计

正如上文提到，对一行数据进行 5 种类型的滤波并比较最小误差和后才能确定本行的滤波类型，进而选取对应的滤波后数据。在软件中，可以把 5 种滤波类型对应的滤波后数据都存下来，最后进行选择输出。但在硬件中，如果将中间数据都存下，则需要 5 块 $32 \times 512 = 2\text{KB}$ 大小的 RAM，面积过大。为此，以行为单位进行滤波，对每行进行两次滤波。第一次滤波的同时累加误差和，滤波结束后计算得出最小误差和对应的滤波类型，第二次滤波时根据最优滤波类型输出滤波后数据。考虑到第一次和第二次滤波计算相同，复用了计算逻辑以减小面积，即此处选择用时间换面积。后续可根据不同场景的需求进行修改，比如实时性要求高，则用面积换时间，只进行一次滤波，但需要用 5 块 2KB 的 RAM。

以行为单位进行两次滤波时，在两次滤波中均需上一行原始像素作为某些滤波类型的参考像素，第二次滤波时还需本行原始像素作为待滤波像素。因此，至少需要 2 个 $32 \times 512 = 2\text{KB}$ 的 RAM 缓存数据。考虑滤波过程中，上一次的本行数据变成了下一次的上一行数据，fifo 需要轮转以避免数据搬运。

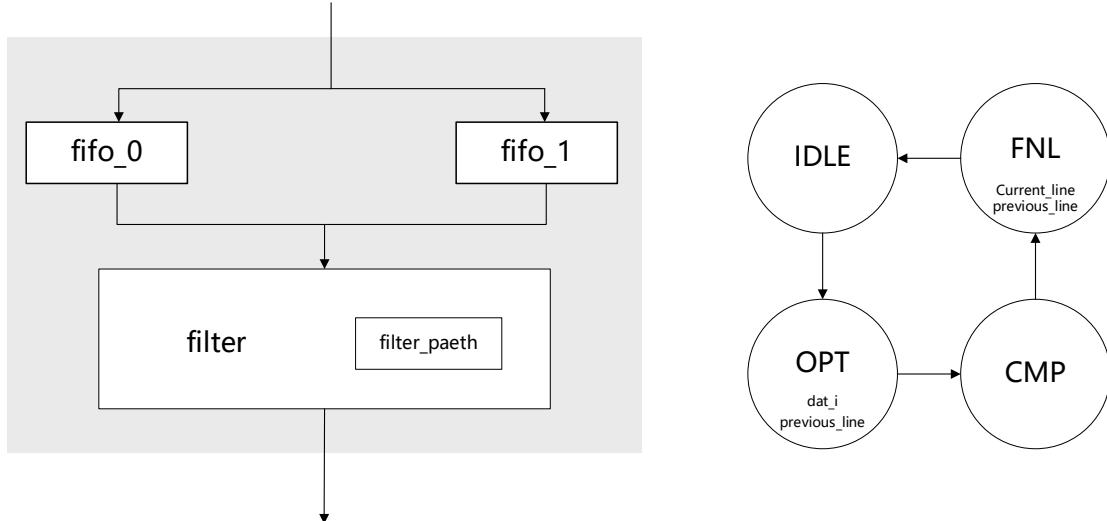
综上，本滤波模块包含 2 个 2KB 的自定义 fifo(fifo_0, fifo_1)用于缓存当前行和前一行的原始像素，以及滤波子模块 filter。

在 Vivado2019.1 中，得出的电路框图如下所示，详尽地展示了模块间的互连关系。



简化后的框图如下所示。其中 fifo_0 和 fifo_1 交替存入当前行的原始像素，具体关系如

下表所示。filter 子模块完成滤波，其状态转换图如下所示，各状态的描述如表所示。
filter_paeth 子模块专用于计算 PaethPredictor 滤波。



名称	T0	T1	T2	T3
fifo_0	ROW_0 写入 ROW_0 的当前行	ROW_0 的前一行	ROW_2 写入	ROW_3 的前一行
fifo_1		ROW_1 写入	ROW_1 的前一行	ROW_3 写入

状态	描述
IDLE	空闲态
OPT	第一次滤波，并累加各误差和
CMP	将 5 种滤波类型对应的误差和进行逐次比较，以求得最小误差和，及其对应的滤波类型
FNL	第二次滤波，并输出滤波后数据。

5.2.3 接口

输入输出信号如表 x 所示，与顶层的输入输出信号类似。在 `start_i` 高有效脉冲后，`val_i` 与 `dat_i` 可有效输入，默认是连续送入一行的 RGBA 原始像素，第二次滤波时连续输出 `val_o` 和 `dat_o`，一行结束后输出高有效脉冲 `done_o`。

输入信号	位宽	描述	输出信号	位宽	描述
clk	1	时钟	done_o	1	结束，高有效，脉冲
rstn	1	复位，低有效	fifo_flt_wr_val_o	1	输出使能，高有效
start_i	1	启动，高有效，脉冲	fifo_flt_wr_dat_o	32	输出数据，码流
val_i	1	输入使能，高有效			
dat_i	32	输入数据，RGBA			
cfg_w_i	9	图像宽度			
cfg_h_i	9	图像高度			

自定义 fifo 的接口如下表所示。与普通 fifo 一致的是，自定义 fifo 也是在内部读写时自动将读写地址加 1。自定义 fifo 与普通 fifo 有区别的三处，一是在地址等于 cfg_w_i 时进行地址复位，二是去掉了标志信号 empty 和 full 以精简端口，三是去掉了 rd_val_o 信号以精简端口（默认 1 个周期后给输出读数据）。

输入信号	位宽	描述	输出信号	位宽	描述
clk	1	时钟	rd_dat_o	1	读数据
rstn	1	复位，低有效			
cfg_w_i	9	图像宽度			
wr_val_i	1	写使能，高有效			
wr_dat_i	32	写数据			
rd_val_i	1	读使能，高有效			

5.2.4 验证

对 5 张测试图片都进行了验证，debug 修正后均通过，以 AlphaBall 为例进行展示。

输入数据：RGBA，RGBA.dat 每行中存储了图像一行的数据。

```
rtl > sim > sim_filter > check_data > E RGBA.dat
1 ffff0003 feff0000 feff0003 feff0003 fdff0003 fcff0003 fbff0003 faff0003 f9ff0003 f8ff0003 f7ff0003
2 fffe0003 fefe0003 fefe0003 fefe0003 fdfe0003 fcfe0003 fbfe0003 fafe0003 f9fe0003 f8fe0003 f7fe0003
3 fffe0003 fefe0003 fefe0003 fefe0003 fdfe0003 fcfe0003 fbfe0003 fafe0003 f9fe0003 f8fe0003 f7fe0003
4 fffe0003 fefe0003 fefe0003 fefe0003 fdfe0003 fcfe0003 fbfe0003 fafe0003 f9fe0003 f8fe0003 f7fe0003
5 fffd0003 fefd0003 fefd0003 fefd0003 fcf0003 fbf0003 fatd0003 f9fd0003 f8fd0003 f7fd0003
6 fffc0003 fecf0003 fecf0003 fecf0003 fdfc0003 fcfc0003 fbfc0003 ffc0003 f9fc0003 f8fc0003 f7fc0003
7 fffb0003 fefb0003 fefb0003 fefb0003 fdfb0003 fcfb0003 ffb0003 fatb0003 f9fb0003 f8fb0003 f7fb0003
```

输出数据：滤波类型 + 滤波后数据，Filtered.dat 每行中存储了图像一行的数据。

```
rtl > sim > sim_filter > check_data > E Filtered.dat
1 01000000 ffff0003 ff0000fd 00000003 00000000 ff000000 ff000000 ff000000 ff000000 ff000000
2 04000000 00ff0000 00000003 00000000 00000000 00000000 00000000 00000000 00000000 00000000
3 04000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
4 04000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
5 01000000 fffd0003 ff000000 00000000 00000000 ff000000 ff000000 ff000000 ff000000 ff000000
6 01000000 fffc0003 ff000000 00000000 00000000 ff000000 ff000000 ff000000 ff000000 ff000000
7 01000000 fffb0003 ff000000 00000000 00000000 ff000000 ff000000 ff000000 ff000000 ff000000
8 01000000 fffa0003 ff000000 00000000 00000000 ff000000 ff000000 ff000000 ff000000 ff000000
9 01000000 ffff90003 ff000000 00000000 00000000 ff000000 ff000000 ff000000 ff000001 ff000000
```

仿真开始截图

```
# vsim -voptargs="+acc" work.sim_filter_top
# Start time: 16:24:41 on Jun 06, 2021
# ** Note: (vsim-3813) Design is being optimized due to module recompilation...
# ** Note: (vopt-143) Recognized 1 FSM in module "filter(fast)".
# Loading work.sim_filter_top(fast)
# Loading work.filter_top(fast)
# Loading work.filter(fast)
# Loading work.filter_paeth(fast)
# Loading work.fifo(fast)
# Loading work.SRAM32x512_lrw(fast)
# Loading work.SRAM32x512_lrw_lbit(fast)
#
#
# *** CHECK STARTS ***
#
#       function check to filter is on!
#
#      at    100 ns, starting scanline  0 ... (delta cycle  525)
#      at   5455 ns, starting scanline  1 ... (delta cycle  525)
#      at  10815 ns, starting scanline  2 ... (delta cycle  525)
#      at  16175 ns, starting scanline  3 ... (delta cycle  525)
#      at  21535 ns, starting scanline  4 ... (delta cycle  525)
#      at  26895 ns, starting scanline  5 ... (delta cycle  525)
```

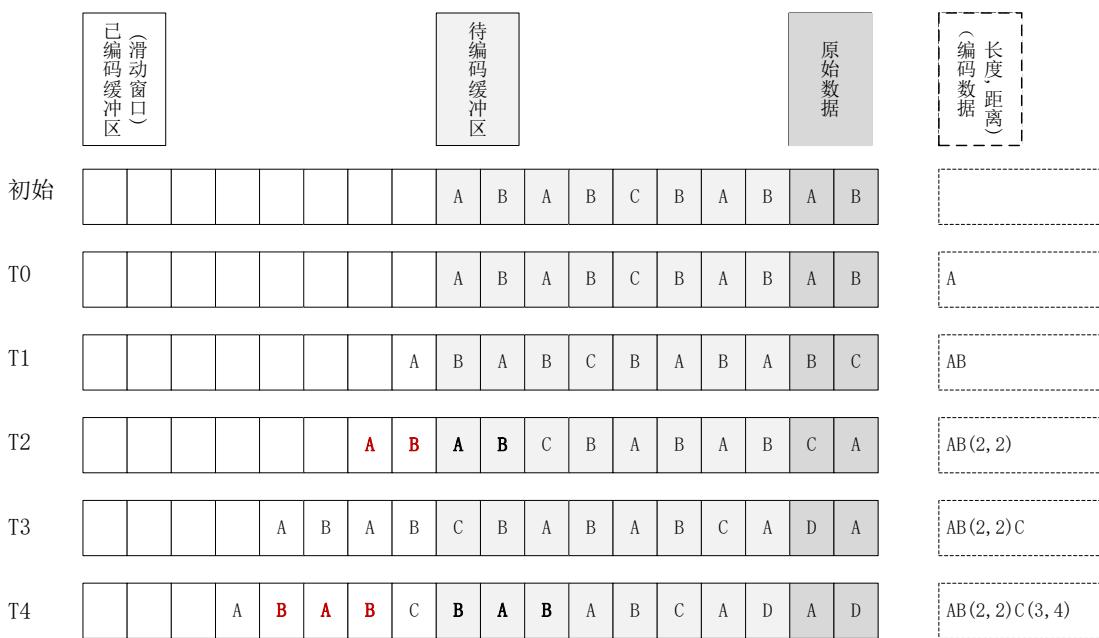
仿真结束截图

```
#      at 1361535 ns, starting scanline 254 ... (delta cycle 525)
#      at 1366895 ns, starting scanline 255 ... (delta cycle 525)
#
#
# *** CHECK DONES ***
#
# ** Note: $stop : ../../sim_filter_top.v(137)
#   Time: 1372355 ns Iteration: 0 Instance: /sim_filter_top
# Break in Module sim_filter_top at ../../sim_filter_top.v line 137
```

5.3 LZ77 模块

5.3.1 原理

LZ77 是一种基于字典的滑动窗无损压缩算法。首先，可以将所有数据二分到待编码区与已编码区。在编码时，搜索待编码区的数据是否出现在已编码区：如果是，则仅需记录下<长度，距离>，进而去除序列中的信息冗余；如果否，则记录下原始数据，无压缩；持续进行此编码过程直到编码结束。考虑到实现代价，仅缓存某固定长度的已编码区数据供搜索，每次编码后需要更新已编码缓冲区数据，已编码缓冲区即为滑动窗。同时，对待编码数据也设置最大长度，即为待编码缓冲区，编码后也需要更新缓冲区中的数据。



示意图如上图所示，假设滑动窗口的大小为 8 个字节，待编码缓冲区的大小为 8 个字节，待编码数据为 ABABCBABABCADAD，编码过程如下所示：

初始时，滑动窗口中无数据，待编码缓冲区中为 ABABCBAB。

T0 时，在滑动窗口中未找到待编码缓冲区中的数据，记录原始数据 A。

T1 时，在滑动窗口中未找到待编码缓冲区中的数据，记录原始数据 B。

T2 时，在滑动窗口中找到待编码缓冲区中的数据 AB，长度为 2，距离为 2，记为<2,2>。

T3 时，在滑动窗口中未找到待编码缓冲区中的数据，记录原始数据 C。

T4 时，在滑动窗口中找到待编码缓冲区中的数据 BAB，长度为 3，距离为 4，记为<3,4>。

显然，记录<长度，距离>也需要若干字节；若匹配长度过短，可能出现记录长度-距离

对所需字节数比记录原始数据所需字节数更大的情况，不是有效的压缩。因此，需要设置最小匹配长度。例如，滑动窗尺寸为 64，待编码缓冲区大小为 64，则长度和距离各需 9bits，共 18bits，向上取整为 3 个字节，即最小匹配长度为 3。

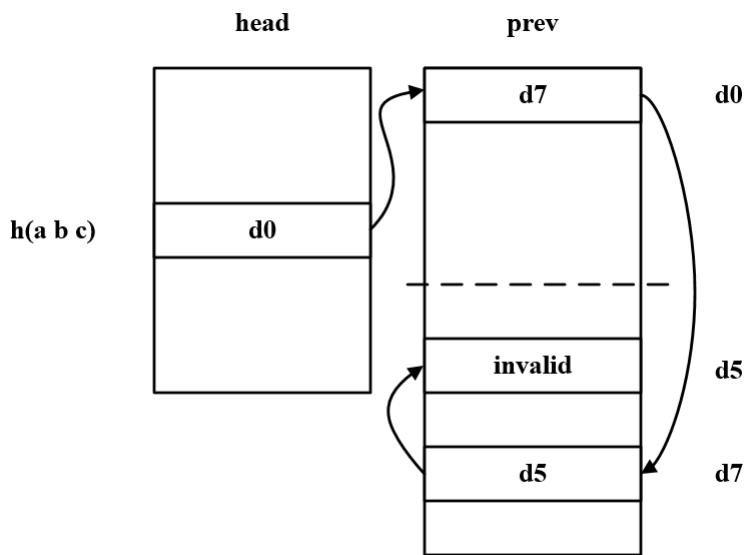
在具体匹配过程中，有逐次比较型和基于哈希表的快速比较型。

例如在 T2 时，若进行逐次比较，则首先对 A 在滑动窗口中进行匹配，再对 AB、ABC、ACB、ABCBA、ABCBAB、ABCBABA 和 ABCBABAB 进行匹配，计算最佳匹配长度及其对应的距离。若最佳匹配长度小于 3，则记录原始数据，反之，则记录<长度，距离>。

基于哈希表的快速比较型主要应用于滑动窗口尺寸较大的情况。该方法旨在建立数据到存放地址的映射关系，通过复杂的映射关系控制来避免逐次在滑动窗口中进行比较，从而大大加快搜索的速度。

由于哈希映射关系可能将需要缓存的相同或不同的数据映射到同一个地址，这种现象称为冲突。映射关系选择的要求之一就是减少在滑动窗口中出现冲突的可能性。由于最小匹配长度为 3，因此通常用待编码数据的前 3 个字节运算产生一定位数的映射地址。映射地址数目过多则存放单元的空间利用率很低，数目过少则出现冲突的可能性增加，通常映射地址数目取少于 3 个字节数据的所有可能性（即 $2^{32}-1$ ），例如 65535（即 $2^{15}-1$ ）。

由于将相同或不同数据映射到同一个地址的冲突很难避免，需要用一定方法处理冲突。通常用链地址法，即将所有映射到同一个地址的数据的存放单元链接成一个线性链表，然后将其链表头存放在相应的映射地址对应的存放单元中。例如，使用一个大小等于映射地址数目的数组 head 存放链表头，使用一个大小等于滑动窗口尺寸的循环数组 prev 存放链表中后一个链接地址，prev 的位置则直接表示滑动窗口中数据的位置。如图所示，3 个字节数据映射到 head 中的位置为 $h(a\ b\ c)$ ，该位置存放了滑动窗口中首个得到相同映射地址的数据在 prev 中的位置 d0，则可以到滑动窗口中 d0 所对应位置进行匹配；同时 d0 存放了滑动窗口中下一个得到相同映射地址的数据在 prev 中的位置 d7，依此类推。链表尾用一定方法表示无效。



5.3.2 设计

本设计采用的滑动窗口尺寸较小且为了控制复杂度，放弃了基于哈希表的快速比较型 LZ77，而选择实现逐次比较型的 LZ77。此外，对于滑动窗和待编码缓冲区的实现方式有 RAM 和寄存器两种方式可选，寄存器堆的匹配速度更快，面积也更大。在面积与速度的综合考虑

中，我们选择用寄存器实现，并将滑动窗和待编码缓冲区的大小都设置为 64，在保证一定压缩效率下尽可能减小面积。

在实现 LZ77 时，首先需要更新滑动窗和待编码缓冲区。其中待编码数据为 filter_top 模块输出的滤波类型与滤波后数据，缓存于 filter_top 与 lz77_top 之间的 32x512 自定义 fifo(fifo_flt)中。LZ77 以字节 (8bits) 为单位进行编码，而 fifo 的位宽是 32bits，即在从 fifo 中取数据时需要向上取整到 4 的倍数，因此待编码缓冲区的大小需要由 64bits 扩展到 67bits，如图 x 所示。为了实现滑动窗和待编码缓冲区之间的数据滑动，首先将滑动窗和待编码缓冲区数据作为整体用寄存器变量 dat_all_r 缓存，每次从 fifo 中取数据后进行移位寄存到 dat_all_r 中。假设某次编码前，待编码缓冲区中数据有 66 个，本次编码后无匹配或最佳匹配长度小于 3，则直接记录滤波后数据 A，待编码缓冲区中的数据变为 65。在进行下一次编码前，无需从 fifo 中取新数据，即不更新 dat_all_r，但需要将 A 从待编码缓冲区移入滑动窗中。

滑动窗 64 bytes	待编码缓冲区 64+3 bytes	滤波后数据 4n bytes
-----------------	----------------------	-------------------

最后，需要在滑动窗中搜索是否有待编码缓冲区中的数据。以滑动窗和待编码缓冲区尺寸均为 8 字节进行说明，设最小匹配长度为 2，如图下所示。

滑动窗 win		待编码缓冲区 inp	
N0	7 6 5 4 3 2 1 0	7 7 7 7 7 7 7 7	D B C C D D B C D D D D D D D D
N1	7 6 5 4 3 2 1 0	6 6 6 6 6 6 6 6	D B C C D D B C D D D D D D D D
N2	7 6 5 4 3 2 1 0	5 5 5 5 5 5 5 5	D B C C D D B C B B B B B B B B

在 N0 时，比较 inp[7] (inp -> input) 与 win[7]~win[0] (win -> sliding window) 是否相等，得到 8 位的 flag 信号（若相等则为 1，反之则为 0）。若 flag 某位为 1，代表对应位置的匹配长度为 1。

在 N1 时，比较 inp[6] 与 win[6]~win[0] 是否相等，低位 flag 补零，将此时得到的 flag 与 N0 得到的 flag 进行逻辑与操作。若 flag 某位为 1，代表对应位置的匹配长度为 2。

以此类推，在 N2 时，比较 inp[5] 与 win[5]~win[0] 是否相等。若 flag 某位为 1，代表匹配长度为 3。

例如 win = DBCCDDBC，inp = DDDBAAAA，则

flag(N0) = 10001100，在 2、3、7 处均有匹配，最小匹配处为 2，匹配长度为 1；

flag(N1) = 00011000 & 10001100 = 00001000，在 3 处有匹配，匹配长度为 2；

flag(N2) = 00001000 & 00001000 = 00001000，在 3 处有匹配，匹配长度为 3；

flag(N3) = 00001000 & 00000000 = 00001000，在 3 处有匹配，匹配长度为 4；

flag(N4) = 00000000 & 00000000 = 00000000，无匹配。

flag(N5) = 00000000 & 00000000 = 00000000，无匹配。

flag(N6) = 00000000 & 00000000 = 00000000，无匹配。

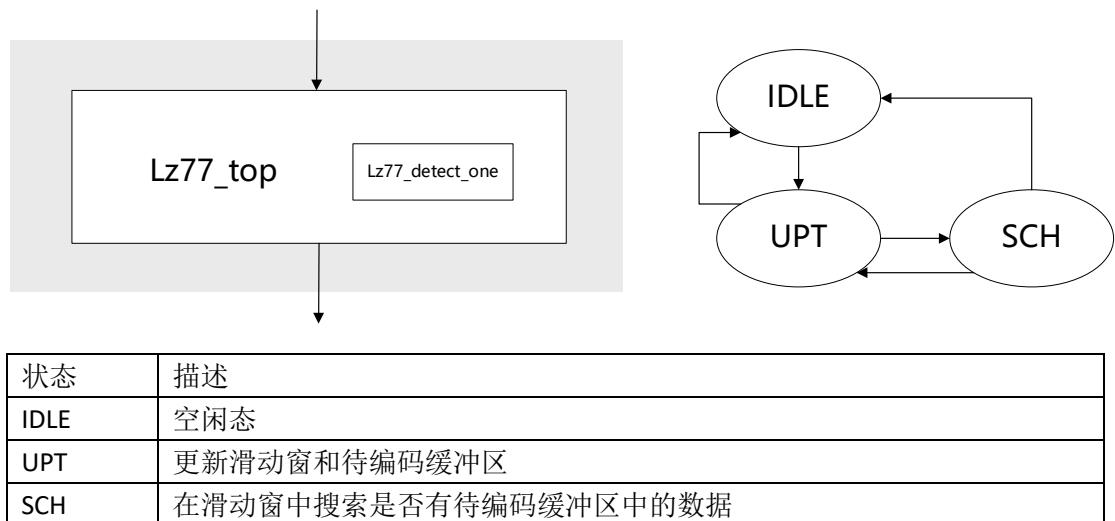
`flag(N7)= 00000000&00000000=00000000`, 无匹配。

综述, 最佳匹配长度为 4, 距离为 4 (地址从 0 开始, 距离从 1 开始, 距离=地址+1)。

若某次有多个位能匹配, 则记录最小匹配处, 即最短距离。此时, 需要一个检测编码串中最低位 1 所在位置的模块, 即 `lz77_detect_one`。例如对一个 8bits 的编码串进行检测: 当输入为 `11110100` 时, 它含 1 的最小位置为 3; 但输入为 `00000000` 时, 无 1, 设置为最大值+1 即 9。

在每次匹配过程中, 更新最短距离和最长匹配长度, 若最长匹配长度超过[3,64], 则视为无效匹配, 直接输出滤波后数据。从例子中可以看出, 若某次不匹配, 则后续均不匹配, 可提前结束匹配过程以提升实时性。

简化的电路框图如图所示, 其中 `lz77_detect_one` 模块是检测编码串中最低位 1 所在位置的模块。状态转换如图所示, 状态描述如表所示。



5.3.3 接口

除压缩功能外, `lz77_top` 还会将读取的滤波后数据传一份给 `ADLER32` 模块使用, 这是因为在实现自定义 `fifo` 时使用了单端口的 `RAM`, 只支持一读。同时为了节约读取时的动态功耗, 避免将相同数据读取两次。考虑到 `ADLER32` 每处理 32bits 输入需要 4 个周期, 在读取 `fifo_flt` 数据时, 也每隔 4 周期读取一次。

输入输出信号如表所示。在 `start_i` 高有效脉冲后, 输出 `fifo_flt_rd_val_o` 以读取 `fifo_flt`。一行压缩结束后输出高有效脉冲 `done_o`。需注意的是, 滤波模块是基于行进行的, 在对某行尾部数据进行压缩时, 剩余待编码数据可能不足 64 个, 此时需结束该行的压缩, 先进行下一行的滤波以产生数据填满待压缩数据缓冲区。若在此时继续压缩, 则计算得到的最大匹配长度一定小于 64 且逐渐减小, 会带来编码效率的损失。只有当该行已经是图片的最后一行, 后续确实已无数据时, 才继续压缩该行。

输入信号	位宽	描述	输出信号	位宽	描述
<code>clk</code>	1	时钟	<code>done_o</code>	1	结束
<code>rstn</code>	1	复位, 低有效	<code>fifo_flt_rd_val_o</code>	1	读使能
<code>start_i</code>	1	启动	<code>val_o</code>	1	输出使能

fifo_flt_rd_dat_i	32	读数据	flg_lit_o	1	滤波后数据标志
cfg_w_i	9	图像宽度	dat_lit_o	8	滤波后数据
cfg_h_i	9	图像高度	dat_len_o	9	长度
			dat_dst_o	9	距离
			flg_lst_o	1	最后标志
			adler32_val_o	1	adler32 输出使能
			adler32_dat_o	32	adler32 输出数据
			adler32_num_o	2	adler32 输出字节数
			adler32_lst_o	1	adler32 最后标志

5.3.4 验证

对 5 张测试图片都进行了验证，debug 修正后均通过，以 AlphaBall 为例进行展示。

输入数据：滤波类型 + 滤波后数据，Filtered.dat 每行中存储了图像一行的数据。

```
rtl > sim > sim_filter > check_data > Filtered.dat
1 01000000 ffff0003 ff0000fd 00000003 00000000 ff000000 ff000000 ff000000 ff000000 ff000000
2 04000000 00ff0000 00000003 00000000 00000000 00000000 00000000 00000000 00000000 00000000
3 04000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
4 04000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
5 01000000 fffd0003 ff000000 00000000 00000000 ff000000 ff000000 ff000000 ff000000 ff000000
6 01000000 fffc0003 ff000000 00000000 00000000 ff000000 ff000000 ff000000 ff000000 ff000000
7 01000000 fffb0003 ff000000 00000000 00000000 ff000000 ff000000 ff000000 ff000000 ff000000
8 01000000 fffa0003 ff000000 00000000 00000000 ff000000 ff000000 ff000000 ff000000 ff000000
9 01000000 ffff9003 ff000000 00000000 00000000 ff000000 ff000000 ff000000 ff000000 ff000001 ff000000
```

输出数据：最后标志 + 滤波后数据标志 + 滤波数据 / <长度，距离>，Lz77.dat 每行存储了一次匹配结果。

```
rtl > sim > sim_lz77 > check_data > Lz77.dat
1 00 01 01
2 00 01 ff
3 00 01 ff
4 00 01 00
5 00 01 03
6 00 01 ff
7 00 01 00
8 00 01 00
9 00 01 fd
10 00 01 00
11 00 01 00
12 00 01 00
13 00 01 03
14 00 00 03 04
```

仿真开始截图

```

# vsim -voptargs="+acc" work.sim_lz77_top
# Start time: 21:14:13 on Jun 01,2021
# ** Note: (vsim-3813) Design is being optimized due to module recompilation...
# ** Note: (vopt-143) Recognized 1 FSM in module "lz77_top(fast)".
# Loading work.sim_lz77_top(fast)
# Loading work.lz77_top(fast)
# Loading work.lz77_detect_one(fast)
# ** Warning: (vsim-WLF-5000) WLF file currently in use: vsim.wlf
#           File in use by: tiitili267 Hostname: DESKTOP-ETCMRJ ProcessID: 29756
#           Attempting to use alternate WLF file "./wlifthe007f".
# ** Warning: (vsim-WLF-5001) Could not open WLF file: vsim.wlf
#           Using alternate file: ./wlifthe007f
#
#
# *** CHECK STARTS ***
#
#       function check to lz77 is on!
#
#       at 100 ns, starting scanline 0 ... (delta cycle 2728)
#       at 27485 ns, starting scanline 1 ... (delta cycle 2561)
#       at 53205 ns, starting scanline 2 ... (delta cycle 2612)
#       at 79435 ns, starting scanline 3 ... (delta cycle 2674)
#       at 106285 ns, starting scanline 4 ... (delta cycle 2799)
#       at 134385 ns, starting scanline 5 ... (delta cycle 2773)

```

仿真结束截图

```

#       at 7630875 ns, starting scanline 253 ... (delta cycle 3096)
#       at 7661945 ns, starting scanline 254 ... (delta cycle 2967)
#       at 7691725 ns, starting scanline 255 ... (delta cycle 2835)
#
#
# *** CHECK DONES ***
#
# ** Note: $stop : .../sim_lz77_top.v(136)
# Time: 7720285 ns Iteration: 0 Instance: /sim_lz77_top

```

5.4 哈夫曼 (huffman) 模块

5.4.1 原理

LZ77 压缩后的数据需要经过哈夫曼编码才形成 deflate 压缩数据块位流。根据 deflate 标准，在固定哈夫曼编码方式下，字面、长度数据和块结束标记会被映射为值 0-285 以及若干额外位，然后值 0-285 使用一张标准规定的固定哈夫曼表进行编码，如下所示：

Extra			Extra			Extra		
Code	Bits	Length(s)	Code	Bits	Lengths	Code	Bits	Length(s)
257	0	3	267	1	15,16	277	4	67-82
258	0	4	268	1	17,18	278	4	83-98
259	0	5	269	2	19-22	279	4	99-114
260	0	6	270	2	23-26	280	4	115-130
261	0	7	271	2	27-30	281	5	131-162
262	0	8	272	2	31-34	282	5	163-194
263	0	9	273	3	35-42	283	5	195-226
264	0	10	274	3	43-50	284	5	227-257
265	1	11,12	275	3	51-58	285	0	258
266	1	13,14	276	3	59-66			

Lit	Value	Bits	Codes
0 - 143		8	00110000 through 10111111
144 - 255		9	110010000 through 111111111
256 - 279		7	0000000 through 0010111
280 - 287		8	11000000 through 11000111

距离数据则会被映射为值 0-29 以及若干额外位，然后值 0-29 使用五位定长码进行编码，如下所示：

Extra			Extra			Extra		
Code	Bits	Dist	Code	Bits	Dist	Code	Bits	Distance
0	0	1	10	4	33-48	20	9	1025-1536
1	0	2	11	4	49-64	21	9	1537-2048
2	0	3	12	5	65-96	22	10	2049-3072
3	0	4	13	5	97-128	23	10	3073-4096
4	1	5,6	14	6	129-192	24	11	4097-6144
5	1	7,8	15	6	193-256	25	11	6145-8192
6	2	9-12	16	7	257-384	26	12	8193-12288
7	2	13-16	17	7	385-512	27	12	12289-16384
8	3	17-24	18	8	513-768	28	13	16385-24576
9	3	25-32	19	8	769-1024	29	13	24577-32768

deflate 压缩的另外两种类型为无压缩、LZ77 并以动态哈夫曼编码压缩。可见，与采用无压缩方式相比，固定哈夫曼编码能达到一定的压缩率；而与采用动态哈夫曼编码方式相比，固定哈夫曼编码则以损失一定的压缩率为代价，省去了复杂的动态哈夫曼表生成和传输过程。

5.4.2 设计与接口

本设计中哈夫曼模块采用查表方式实现固定哈夫曼编码。模块以 LZ77 每次压缩后的字面数据或长度、距离数据对作为输入，按照标准规定索引出对应的固定哈夫曼编码输出。由于是纯组合逻辑，输入输出在一个时钟周期内完成。

另外，由于已经在 LZ77 中对长度和距离数据值进行了限制，这同时也减少了哈夫曼编码输入值的可能性，极大地简化了查表索引过程。

5.4.3 验证

哈夫曼模块的仿真示例部分如下所示，与设计比较可以看出模块符合设计的电路功能：



5.5 ADLER32 模块

5.5.1 原理与设计

zlib 位流中最后四个字节固定为 ADLER32 校验和。该 ADLER32 校验和按字节处理滤波后的字节流，用类似 C 语言的语法描述的典型算法如下：

```
s1 = 0x1;  
s2 = 0x0;  
for(i = 0; i < bytelength; ++i) {  
    s1 = s1 + bytedata[i];  
    s2 = s2 + s1;  
    s1 = s1 % 65521;  
    s2 = s2 % 65521;  
}  
r = (s2 << 16) | s1;
```

可见该算法实现的主要难点在于每次处理需要两个取模运算。软件实现通常用大位宽类型的变量来寄存 s1 和 s2，由于每次处理的求和的增量是有上限的，因此可以只在求和要达到预先计算出的上溢出次数、或字节流处理结束时才进行取模运算，从而大大节省了取模运算的次数。

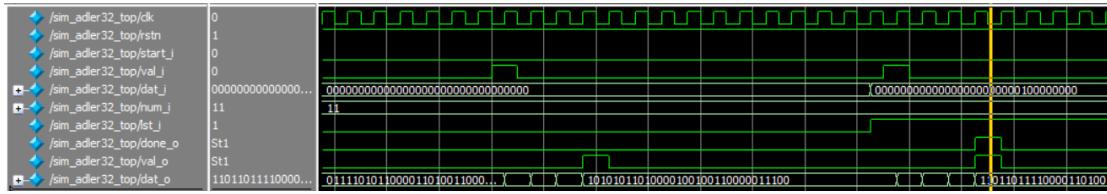
对于硬件实现，如果除数是 2 的正整次幂的取模运算，通常采用直接取被除数的低位来实现，但是该算法的除数（65521）不是 2 的正整次幂。而如果采用类似前述软件实现的方法，则需要大位宽的寄存器和额外的控制逻辑，消耗较大；并且每个时钟周期处理一个字节输入，而每个字节都可能是最后一个字节而需要紧接的取模运算，也就是说取模运算路径始终存在，因此该方法不适用于硬件实现。本设计中考虑到每次取模运算后 s1 和 s2 的值都不大于 65520，并且每次处理一个字节，则求和后 s1 的最大值不大于 65520+255，s2 的最大值不大于 65520+65520+255，因此紧接的取模运算可以简单地通过比较和减法运算来实现，从而避免了直接使用取模运算逻辑。

5.5.2 接口

本设计中 ADLER32 模块使用一个状态机控制处理过程。启动信号后初始化 s1 和 s2 寄存器。然后以每次 FIFO 读出的 4 字节（32 位）滤波后数据作为输入，并使用一个信号指示输入中的有效字节数，四个时钟周期后输出本次处理结果值。使用一个信号指示输入是否为最后一次输入，如果是最后一次输入，则本次处理完成后输出最终 zlib 位流的 ADLER32 值，并产生一个完成信号。

5.5.3 验证

ADLER32 模块的仿真示例部分如下所示，与设计比较可以看出模块符合设计的电路功能：



5.6 CRC32 模块

5.6.1 原理与设计

PNG 位流中每个块的最后四个字节固定为循环冗余校验域，即该块的类型域和数据域位流的循环冗余校验。循环冗余校验的思想是在要传输的 k 比特数据后添加 c 比特冗余位形成 $(k+c)$ 比特的传输帧，同时提供一个预先设定的 $(c+1)$ 比特整数，并且要求添加后的传输帧能被该整数模 2 整除。

PNG 位流中使用的循环冗余校验是一种 CRC32，用类似 C 语言的语法描述的典型算法如下：

```
r = 0xffff_ff;
for(i = 0; i < bitlength; ++i) {
    r = (r >> 1) ^ (((r & 0x1) ^ bitdata[i]) ? 0xedb8_8320 : 0);
}
r = r ^ 0xffff_ff;
```

可见与一般的循环冗余校验相比具有以下特点：（1）位流的输入顺序在字节间是先低地址字节后高地址字节、在字节内是先低有效位后高有效位；（2）32 位循环冗余校验的初始值是十六进制数 FFFFFFFF，顺序是先低有效位后高有效位；（3）每次处理时依据当前移入输入位和移出循环冗余校验位二者的异或值，来判断本次处理值是否要异或上预先设定的多项式；（4）最终值要再异或上十六进制数 FFFFFFFF。

软件实现通常采用查表法，每次并行处理位流中一个字节输入。对于硬件实现，如果采用类似的查表法需要存储一共 256 个表项、每个表项 32 比特的复杂数据，消耗较大；并且考虑到硬件又具有位操作方便灵活的优点。因此本设计中采用位操作来并行处理 CRC32 的位输入，需要的位操作运算可以很容易地由前述算法推导得出；另外 easicis 网站上也提供了自动化生成循环冗余校验位操作运算的硬件描述语言代码的工具，不过需要根据 PNG 规范的特点额外地将输入字节内反序、将输出反序并异或十六进制数 FFFFFFFF。

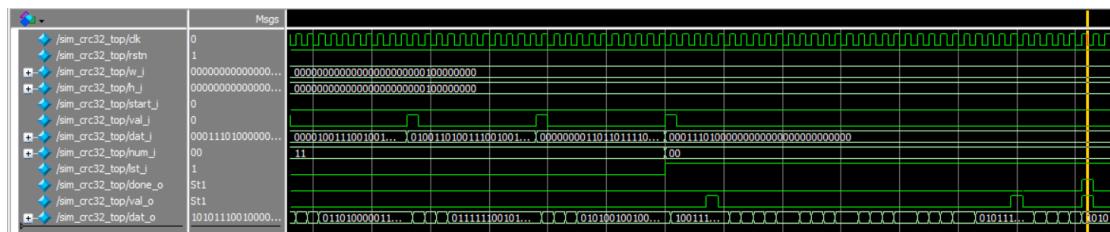
5.6.2 接口

本设计中需要计算 IHDR、IDAT、IEND 三个块的 CRC32 值，实现时将前述 CRC32 计算作为一个核心模块以复用，然后在外层使用控制逻辑来控制计算核心模块的输入数据。CRC32 计算核心模块的处理过程类似于 ADLER32 模块，即使用状态机控制，然后每次处理 32 位数据输入，并使用一个信号指示输入中的有效字节数，四个时钟周期后输出本次处理结果值。外层控制逻辑使用一个状态机。启动信号后先利用计算核心模块计算 IDAT 块的 CRC32 值，需要的输入数据为 IDAT 块的类型域和数据域，而其中 IDAT 块的数据域就使用位流模块输出的 zlib 位流。输出 IDAT 块的 CRC32 值后重新设置计算核心模块，然后再依

次计算 IHDR、IEND 两个块的 CRC32 值，需要的输入数据按照标准。全部输出后产生一个完成信号。

5.6.3 验证

CRC32 模块的仿真示例部分如下所示，与设计比较可以看出模块符合设计的电路功能：



5.7 位流 (bs) 模块

5.7.1 原理

位流模块负责将一系列分离的输出数据拼接形成硬件最终的输出位流。根据本设计中的顶层模块设计，硬件最终的输出位流依次为 zlib 位流（按照 32 位补齐）、zlib 位流实际长度（32 位）、IDAT 块的 CRC32 值、IHDR 块的 CRC32 值、IEND 块的 CRC32 值，如下所示：



5.7.2 设计与接口

实现时本设计中使用一个拼接模块将一系列分离的输出数据拼接形成硬件最终的输出位流，然后在外层使用控制逻辑来控制拼接模块的输入数据。外层控制逻辑使用一个状态机，不同状态下送到拼接模块的输入数据如下所示：

状态	送到拼接模块的输入数据	备注
IDLE	\	\
CMF_FLG	zlib 位流的 CMF 和 FLG	\
BLK_0	zlib 位流中 deflate 位流的 BFINAL 和 BTTYPE	\
BLK_1	LZ77 并且哈夫曼编码后的数据	使用一个信号指示输入是否为最后一次输入
BLK_2	哈夫曼编码后的块结束标记	\
BLK_3	填充 0 以满足字节对齐	如已满足字节对齐则可跳过
ADLER32	ADLER32 模块输出的 ADLER32 值	\
FLUSH	填充 0 以满足四字节对齐	如已满足四字节对齐则可跳过
ZLIB_LEN	统计的 zlib 位流实际长度	\

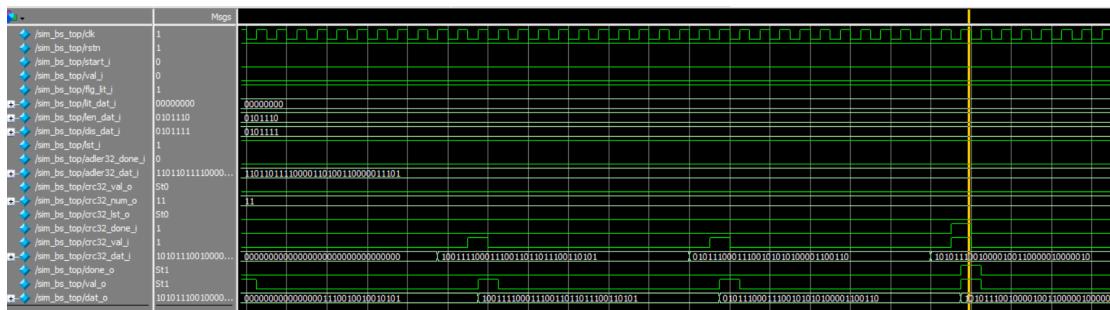
CRC32	CRC32 模块输出的 IDAT、IHDR、IEND 块的 CRC32 值	全部输出后产生一个完成信号
-------	---------------------------------------	---------------

拼接模块将一系列分离的输出数据拼接形成硬件最终的输出位流。分离的输出数据将在不同的时钟周期到来，并且不会超过 32 位，使用一个信号指示输入中的有效位数。拼接模块内部使用一个长度为 64 位的寄存器缓冲，使用一个计数器指针标记缓冲中的未输出数据位数。每次分离的输出数据到来时，将分离的输出数据从缓冲的低有效位移入，同时更新指针。然后检查指针；如果缓冲中的未输出数据不少于 32 位，则输出缓冲中未输出数据的高有效 32 位并产生一个时钟周期的输出有效信号，同时将指针减去 32。

另外按照标准，deflate 压缩数据块位流在形成字节流时，填充到字节的顺序是逐位先字节的低有效位后高有效位；而对于其他的字节数据，位的顺序则保持不变。因此为了简化拼接逻辑，本设计中将所有送到拼接模块的输入数据先转换成先低有效位后高有效位的形式，即对于字节数据要先额外进行一次字节内反序再送到拼接模块，这样拼接模块就可以以统一的形式处理所有送来的输入数据。

5.7.3 验证

位流模块的仿真示例部分如下所示，与设计比较可以看出模块符合设计的电路功能：



6 门级设计与验证

6.1 Designer Compiler

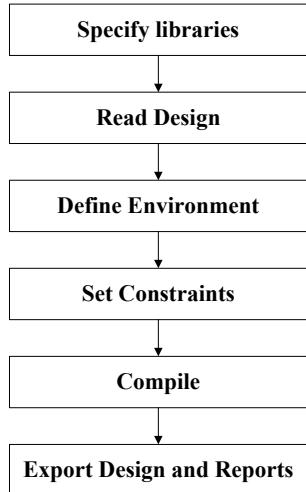
6.1.1 流程

逻辑综合主要包含三个步骤：

流程	说明
转换 (Translation)	读取并分析 RTL 文件，进行结构级的优化（包含对设计结构的选择，数据通路的优化，资源共享和重排序运算符号等），将其翻译为独立于工艺库（使用 synopsis 内部的 GETCH 库）的门级网表。
映射 (Mapping)	使用具体工艺库的标准单元，进行逻辑级的优化（包含结构优化和展平优化等），将上一步的门级网表映射到具体工艺库中的门

	级网表。
优化 (Optimization)	进行延迟优化, 设计规则修复和面积优化

详细步骤共 6 步, 顺序及其说明如下:



步骤	说明
Specify libraries	指定包含标准单元的工艺库以及所用到宏单元的链接库
Read Design	读入和分析 RTL 设计, 并进行链接 (link) 和唯一化 (uniquify)
Define Environment	设置输入驱动和输出负载 (<code>clk</code> 和 <code>rst</code> 均设为无限驱动, 因为后续在 <code>icc</code> 中会进行时钟树综合和大扇出信号的修复)
Set Constraints	设计规则: 最大扇出、转换时间和电容 时钟的周期、延迟、不确定度 (时钟和复位均为理想网络) 输入输出延时 面积
Compile	设置映射强度为 <code>high</code>
Export Design and Reports	写出网表和 <code>sdc</code> 约束文件

6. 1. 2 关键路径优化

整体和各模块优化前后关键路径延时和最高频率如下表所示。

显然, 对于前三个模块 Adler32, bs 和 Crc32, 其内部的计算量小, 关键路径延时小, 频率高, 均满足目标频率 125MHz, 无需优化。

对于 Filter_top 和 Lz77_top, 内部计算大, 关键路径延时大, 频率低于 125MHz。对这两个模块, 需要根据时序报告拆分组合逻辑, 并入若干级寄存器以缓存中间数据。

需注意, 关键路径缩短后, 次关键路径又变成了新的关键路径, 需要继续迭代直至频率在 125MHz 左右。

经过若干次迭代, 将 filter_top 频率提升至 123MHz, 基本满足设计目标。Lz77_top 频率仅提升至 87MHz, 此时的关键路径已不便拆分, 选择在 ICC 阶段进行后续时序优化直至满足设计目标。

模块	优化前延时 / ns	优化前频率 / MHz	优化后延时 / ns	优化后频率 / MHz
adler32	5.2	192		

bs	3.4	294		
crc32	2.4	416		
filter_top	24.5	40	8.1	123
lz77_top	41.6	24	11.38	87
png_top	42.3	23	12	83

6. 1. 3 综合结果

面积	1.26 mm ² (SRAM 面积: 55%)
频率	83 MHz
功耗	2.58 mW

6. 2 Primetime

Primetime 是最常用的静态时序分析工具，通过对路径延时的静态估计来快速计算关键路径，从而保证时序的满足。静态的特性，能极大提高了验证的速度，以及摆脱了工艺约束和仿真 testbench 的不完备性，更全面地检查电路的时序。步骤主要是设置顶层设计和路径，设置库，读入网表和 sdc 进行分析以及导出时序报告这 4 步，简短的脚本如下：

```
rtl > primetime > E primetime.dc.tcl
1  # dc
2
3  ===== set variables =====
4  set top          "png_top"
5  set design_path  "/net/dellr940d/export/ybfan2/ttli/VLSI/mypng/dc_high/netlist"
6  set search_path  "/net/dellr940d/export/ybfan2/ttli/VLSI/mypng/ref/db"
7
8
9  ===== set library =====
10 set target_library "saed90nm_typ.db SRAM32x512_1rw_typ.db"
11 set link_library   "* $target_library" ; # * is a must, or submodule will not be found
12
13
14 ===== read netlist and sdc =====
15 read_verilog $design_path/${top}.dc.v
16 current_design $top
17 link
18
19 source $design_path/${top}.dc.sdc
20
21
22 ===== report =====
23 report_timing > primetime.dc.log
24 check_timing -verbose > primetime.violations.dc.log
25
26 quit
```

结果如下，时钟为 12ns， dc 后建立时间满足，即最高频率可达 83MHz。

```

17 Startpoint: lz77_top/cnt_sch_r_reg[0]
18             (rising edge-triggered flip-flop clocked by clk)
19 Endpoint: lz77_top/flg_mat_r_reg[50]
20             (rising edge-triggered flip-flop clocked by clk)
21 Path Group: clk
22 Path Type: max
23
24 Point                                     Incr      Path
25 -----
26 clock clk (rise edge)                     0.00      0.00
27 clock network delay (ideal)              0.10      0.10
28 lz77_top/cnt_sch_r_reg[0]/CLK (DFFARX1)  0.00      0.10 r
29 lz77_top/cnt_sch_r_reg[0]/QN (DFFARX1)   0.14      0.24 r
30 lz77_top/U15910/QN (NOR2X0)              0.93      1.17 f
31 lz77_top/U11971/Q (AND2X4)               0.55      1.72 f
32 lz77_top/U12225/Z (DELLN1X2)             0.67      2.39 f
33 lz77_top/U14374/QN (NAND2X0)             0.69      3.08 r
34 lz77_top/U9999/Z (DELLN1X2)              0.81      3.89 r
35 lz77_top/U4737/Z (NBUFFX2)               2.05      5.93 r
36 lz77_top/U7822/Q (OA222X1)               1.71      7.64 r
37 lz77_top/U7818/QN (NAND4X0)              0.59      8.24 f
38 lz77_top/U63/Q (OR4X2)                   0.79      9.02 f
39 lz77_top/U9389/Q (XNOR2X1)              0.75      9.77 r
40 lz77_top/U6930/Q (AND2X1)                0.47      10.24 r
41 lz77_top/U6921/Q (AND4X1)                0.38      10.62 r
42 lz77_top/U7356/QN (AOI21X1)              0.46      11.09 f
43 lz77_top/U7333/QN (OAI21X1)              0.47      11.55 r
44 lz77_top/flg_mat_r_reg[50]/D (DFFARX1)  0.36      11.92 r
45 data arrival time                         11.92
46
47 clock clk (rise edge)                     12.00     12.00
48 clock network delay (ideal)              0.10      12.10
49 clock reconvergence pessimism           0.00      12.10
50 clock uncertainty                        -0.10     12.00
51 lz77_top/flg_mat_r_reg[50]/CLK (DFFARX1) 12.00      12.00 r
52 library setup time                      -0.08     11.92
53 data required time                      11.92
54 -----
55 data required time                      11.92
56 data arrival time                       -11.92
57 -----
58 slack (MET)                           0.00

```

6. 3 Formality

Formality 是最常用的形式验证工具，它能通过比较两个设计在逻辑功能上是否等同的方法来验证电路的功能。与静态时序一致一致，能极快地全面检查电路的功能。步骤主要是设置顶层设计和路径，设置库，设置参考设计，设置实现设计，以及 `match` 和 `verify`。

```

rtl> formality > formal.dc.tcl
1  # rtl vs dc
2
3  ===== set variables =====
4  set top          "png_top"
5  set ref_design_path      "/net/dellr940d/export/ybfan2/ttli/VLSI/mypng/rtl"
6  set imp_design_path      "/net/dellr940d/export/ybfan2/ttli/VLSI/mypng/dc_high/netlist"
7  set search_path        "/net/dellr940d/export/ybfan2/ttli/VLSI/mypng/ref/db"
8
9
10 ===== set library =====
11 set target_library "saed90nm_typ.db SRAM32x512_1rw_typ.db"
12 read_db -tech $target_library
13
14
15 ===== set reference =====
16 ~ read_verilog -r "    $ref_design_path/common/defines.vh \
17 |           $ref_design_path/common/ram.v \
18 |           $ref_design_path/common/fifo.v \
19 |           $ref_design_path/filter/filter.v \
20 |           $ref_design_path/filter/filter_paeth.v \
21 |           $ref_design_path/filter/filter_top.v \
22 |           $ref_design_path/adler32/adler32.v \
23 |           $ref_design_path/bs/bs_output.v \
24 |           $ref_design_path/bs/bs_top.v \
25 |           $ref_design_path/bs/huffman_fixed.v \
26 |           $ref_design_path/crc32/crc32_core.v \
27 |           $ref_design_path/crc32/crc32_top.v \
28 |           $ref_design_path/lz77/lz77_detect_one.v \
29 |           $ref_design_path/lz77/lz77_top.v \
30 |           $ref_design_path/$top.v \
31 "
32 set_top r:/WORK/${top}
33 current_design $top
34
35
36 ===== set implementation =====
37 read_verilog -i $imp_design_path/${top}.dc.v
38 set_top i:/WORK/${top}
39 current_design $top
40
41
42 ===== compare =====
43 match
44 verify
45
46
47 ===== report =====
48 report_status > formality.dc.log
49
50 quit

```

结果如下，可看到 dc 前后的设计通过了形式验证的对比。

```

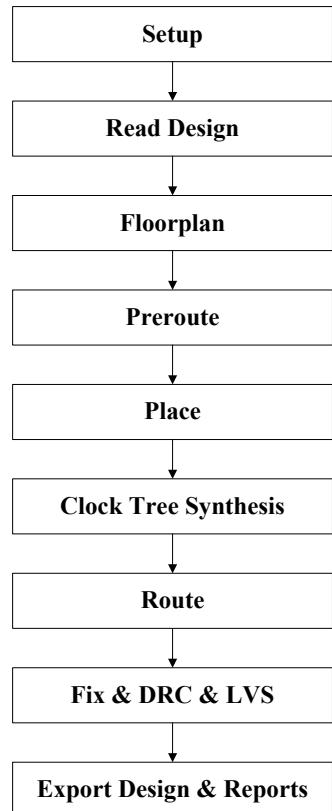
1 ****
2 Report      : status
3
4 Reference   : r:/WORK/png_top
5 Implementation : i:/WORK/png_top
6 Version     : Q-2019.12-SP5
7 Date        : Thu Jun  3 12:47:28 2021
8 ****
9
10
11 ***** Verification Results *****
12 Verification SUCCEEDED
13 -----
14 Reference design: r:/WORK/png_top
15 Implementation design: i:/WORK/png_top
16 3486 Passing compare points
17 -----
18 Matched Compare Points    BBPin    Loop    BBNet    Cut    Port    DFF    LAT    TOTAL
19
20 Passing (equivalent)    135      0       0       0      34     3317     0     3486
21 Failing (not equivalent) 0       0       0       0      0      0       0      0
22 Not Compared
23 Unread                  0       0       0       0      0       2       0      2
24 ****
25 1

```

7 物理级设计与验证

7.1 IC Compiler

物理设计的基本步骤和说明如下，在详细步骤中标红的步骤尤其值得注意：



7.1.1 Setup

1. 设置输出 log 文件，便于复盘查看与迭代设计。
2. 设置库文件。除 dc 中用到的标准单元和宏单元逻辑库外，还需要其对应的物理库、工艺文件和 RC 参数文件等等。此外，还需设置 max_library 和 min_library 分别用于修复建立时间和保持时间。
3. 设置顶层设计与 sdc 文件路径。
4. 设置输出文件路径。
5. 设置最大可用核数以加速设计。

```
1 #-----#
2 #                               Log
3 #
4 define_name_rules BORG -allowed {A-Za-z0-9_} -first_restricted "_" -last_restricted "_" -max_length 30
5 set timestamp [clock format [clock scan now] -format "%Y-%m-%d %H-%M"]
6 set log_path      "../log"
7 set sh_output_log_file "$log_path/log.[pid].$timestamp"
8 set sh_command_log_file "$log_path/cmd.[pid].$timestamp"
9
10 #-----
11 #                               Variables
12 #
13 #   library
14 set lib_name      "saed90nm_typ"
15 set target_library "saed90nm_typ.db          \
16 |           SRAM32x512_1rw_typ.db"
17 set link_library  "* $target_library"
18 set lib_path      "/net/dellr940d/export/ybfan2/ttli/VLSI/mypng/ref/db"
19 set script_path   "../script"
20 set search_path   "$lib_path $script_path"
21
22 set file_path     "/net/dellr940d/export/ybfan2/ttli/VLSI/mypng/ref/"
23 set tech_file     "$file_path/tf/saed90nm_icc_1p9m.tf"
24 set mw_path       "$file_path/mw_std          \
25 |           $file_path/mw_sram_32x512 \
26 |           $file_path/mw_io"
27 set tlup_map      "$file_path/tlup/tech2itf.map"
28 set tlup_max      "$file_path/tlup/saed90nm_1p9m_1t_Cmax.tluplus"
29 set tlup_min      "$file_path/tlup/saed90nm_1p9m_1t_Cmin.tluplus"
30
31 # using max to fix setup time, min to fix hold time
32 set_min_library saed90nm_max.db      -min_ver saed90nm_min.db
33 set_min_library SRAM32x512_1rw_max.db -min_ver SRAM32x512_1rw_min.db
34 set_min_library saed90nm_io_max.db   -min_ver saed90nm_io_min.db
35
36 #   design
37 set top_name       "png_top"
38 set rtl_path       "/net/dellr940d/export/ybfan2/ttli/VLSI/mypng/rtl"
39 set dc_netlist_path "/net/dellr940d/export/ybfan2/ttli/VLSI/mypng/dc_high/netlist"
40 set verilog_file   "$dc_netlist_path/png_top.dc.v"
41 set sdc_file       "$dc_netlist_path/png_top.dc.sdc"
42
43 set netlist_path   "../netlist"
44 set report_path    "../report"
45 set gds_path       "../gds"
46
47 # speed up
48 set_host_options -max_core 16
```

7.1.2 Read Design

1. 根据标准单元的物理库和工艺库建立本设计的物理库。
2. 读入综合后的网表文件，并进行 uniquify 和 link。
3. 读入 sdc 文件
4. 设置 RC 参数文件
5. 检查错误、保存设计以及报告时序。每步结束后都会进行此操作，后续省略该步的介绍。

```

#-----
#                               1  setup
#-----
gui_start

create_mw_lib -technology           $tech_file  \
               -mw_reference_library $mw_path    \
               -bus_naming_style   {[%d]}       \
               -open                 $top_name

read_verilog $verilog_file \
             -dirty_netlist \
             -top $top_name \
             -cel $top_name
current_design $top_name
uniqueify_fp_mw_cel
link

read_sdc $sdc_file

set_tlu_plus_files -max_tluplus $tlup_max \
                   -min_tluplus $tlup_min \
                   -tech2itf_map $tlup_map

# if {[check_error -verbose] != 0} { exit 1 }
check_error -verbose
save_mw_cel    -as $top_name\_1_setup
report_timing > $report_path/timing_1_setup.rpt

```

7.1.3 Floorplan

1. 读入 pin.tcl，其中规划了 pin 的具体位置，如图所示。总体而言，共 88 个，四边每边各 22 个，输入 pin 在左上角，输出 pin 在右下角。其中 side 和 order 的对应关系已标注。

```

# side:
#      2
#      1     3
#      4

# order:
#      1 2 3
#      -----
#      ^          dat_i[0-21]          ^
#      |  val_i           dat_i[22-31] |
#      |  cfg_h[0-8]       done          |
#      |  cfg_w[0-8]       val_o         |
#      |  start           dat_o[0-9]    |
#      |  rst            io[vdd_r]    |
#      3  |  clk           | 3
#      2  |              | 2
#      1  |          dat_o[31-10] | 1
#      -----
#      1 2 3

# 1: left
set_pin_physical_constraints -pin_name { clk } -side 1 -order 1
set_pin_physical_constraints -pin_name { rstn } -side 1 -order 2
set_pin_physical_constraints -pin_name { start_i } -side 1 -order 3

set_pin_physical_constraints -pin_name { cfg_h_i[0] } -side 1 -order 4
set_pin_physical_constraints -pin_name { cfg_h_i[1] } -side 1 -order 5

```

2. 初始化 floorplan，根据 dc 面积预估芯片面积、设置核利用率和标准单元放置方式。

```

create_floorplan -control_type width_and_height \
    -core_width 1300 \
    -core_height 1300 \
    -core_utilization 0.7 \
    -left_io2core 10 \
    -top_io2core 10 \
    -right_io2core 10 \
    -bottom_io2core 10 \
    -start_first_row

```

3. 设置宏块间的间隔为 20，可根据具体设计进行迭代，留出 power ring 和布线的余量。

```

set_fp_placement_strategy -macros_on_edge on \
    -min_distance_between_macros 20 \
    -auto_grouping high

```

4. 预放置标准单元和宏块，并将其连接至 power 和 ground。每次设计网表发生变化或有单元移动都要重新连接 power 和 ground。

```

create_fp_placement
source ../../script/connect_pg.tcl

```

```

icc > connect_pg.tcl
  √ derive_pg_connection -power_net "VDD" \
    -ground_net "VSS" \
    -power_pin VDD \
    -ground_pin VSS
  √ derive_pg_connection -power_net "VDD" \
    -ground_net "VSS" \
    -tie

```

7.1.4 Preroute

1. 创建核的电源环，RAM 的电源环和电源条以降低电源电压的损失。

```

# power ring
create_rectangular_rings -nets {VDD VSS} \
    -left_offset 0.5 -left_segment_layer M8 -left_segment_width 1.5 \
    -right_offset 0.5 -right_segment_layer M8 -right_segment_width 1.5 \
    -bottom_offset 0.5 -bottom_segment_layer M9 -bottom_segment_width 1.5 \
    -top_offset 0.5 -top_segment_layer M9 -top_segment_width 1.5

# power ring around ram
create_rectangular_rings -nets {VDD VSS} \
    -around specified \
    -cells {fifo_flt/ram_90nm \
        filter_top/fifo_1/ram_90nm \
        filter_top/fifo_0/ram_90nm } \
    -left_offset 0.5 -left_segment_layer M6 -left_segment_width 1.5 \
    -right_offset 0.5 -right_segment_layer M6 -right_segment_width 1.5 \
    -bottom_offset 0.5 -bottom_segment_layer M7 -bottom_segment_width 1.5 \
    -top_offset 0.5 -top_segment_layer M7 -top_segment_width 1.5

# power straps
create_power_straps -direction horizontal \
    -do_not_route_over_macros \
    -nets {VSS VDD} \
    -layer M5 \
    -width 2 \
    -configure step_and_stop \
    -start_at 30 \
    -step 60 \
    -stop 1300
create_power_straps -direction vertical \
    -nets {VDD VSS } \
    -layer M4 \
    -width 2 \
    -configure step_and_stop \
    -start_at 30 \
    -step 60 \
    -stop 1300

```

2. 将标准单元、宏单元预布线至 power 和 ground。

```
preroute_standard_cells -nets {VDD VSS} \
    -port_filter_mode off \
    -cell_master_filter_mode off \
    -cell_instance_filter_mode off \
    -voltage_area_filter_mode off
preroute_instances -ignore_pads \
    -ignore_cover_cells \
    -select_net_by_type \
        specified -nets {VDD VSS}
```

3. 设置 M1、M2 下禁止电源地网络（标准单位使用了底层金属，避免短路），M3-M9 视情况走电源地网络。

```
set_pnet_options -complete {M1 M2}
set_pnet_options -partial {M3 M4 M5 M6 M7 M8 M9}
```

4. 进行增量预放置，快速布线，并连接 power 和 ground。

```
create_fp_placement -incremental all
route_fp_proto
source ../script/connect_pg.tcl
```

5. 固定宏单元位置，即 3 个 RAM。

```
set_dont_touch_placement {fifo_flt/ram_90nm \
    filter_top/fifo_1/ram_90nm \
    filter_top/fifo_0/ram_90nm }
```

7.1.5 Place

1. 去除 dc 中对于 rstn 设置的理想网络。
2. 一键布局。
3. 连接 power 和 ground。

```
remove_ideal_net [get_nets rstn]
place_opt
source ../script/connect_pg.tcl
```

7.1.6 Clock Tree Synthesis

1. 去除 dc 中对于 rstn 设置的理想网络。
2. 设置时钟树的 skew 和 transition 以及可布线的高层金属 M3-M9。
3. 一键进行时钟树综合。
4. 修复保持时间。
5. 优化面积。
6. 连接 power 和 ground。

```
remove_ideal_net [get_ports clk]
check_physical_design -stage pre_clock_opt

set_clock_tree_options -target_skew 0.1 \
    -max_transition 0.5 \
    -layer_list {METAL3 METAL4 METAL5 METAL6 METAL7 METAL8 METAL9}

clock_opt
report_constraint

set_fix_hold [all_clocks]
clock_opt -fix_hold_all_clocks
report_constraint

set_max_area 0
set physopt_area_critical_range 1.0
psynopt -area_recovery
source ../script/connect_pg.tcl
report_constraint
```

7.1.7 Route

1. 一键布线。
2. 基于 DRC 问题修复的布线。

```
route_opt
route_zrt_eco
```

7.1.8 Fix & DRC & LVS

1. 插入标准单元 **filler**, 金属 **filler** 以达到 DRC 所规定的密度以及避免天线效应。
2. 连接 **power** 和 **ground**。
3. 将标准单元、宏单元预布线至 **power** 和 **ground**。
4. 基于 DRC 问题修复的布线。
5. 运行 DRC & LVS 检查。

```

insert_stdcell_filler -cell_without_metal "SHFILL1 SHFILL2 SHFILL3 SHFILL64" \
                     "SHFILL128 DHFILLHLH2 DHFILLLHL2 DHFILLHLHLS11" \
                     -connect_to_power {VDD} \
                     -connect_to_ground {VSS}
insert_metal_filler -bounding_box [get_placement_area] \
                     -from_metal 1 -to_metal 9 \
                     -tie_to_net none \
                     -fill_poly \
                     -timing_driven

source ../script/connect_pg.tcl

preroute_standard_cells -nets {VDD VSS} \
                         -port_filter_mode off \
                         -cell_master_filter_mode off \
                         -cell_instance_filter_mode off \
                         -voltage_area_filter_mode off
preroute_instances -ignore_pads \
                   -ignore_cover_cells \
                   -select_net_by_type \
                   specified -nets {VDD VSS}

report_constraints

route_zrt_eco
report_constraints
verify_zrt_route

verify_lvs

```

DRC 和 LVS 结果显示如下：

Error Browser						
ErrorSet ▾	Total	Visible	Fixed	Ignored	NULL Net	
png_top.CEL;1	3172	3172	0	0	3172	
png_top_lvs.err;1	3172	3172	0	0	3172	
Floating Port	2464	2464	0	0	2464	
Min Area	708	708	0	0	708	

共 2464 个 Floating Port，发现其中 2264 个是寄存器输出端的 QN doesn't connect to any net，199 个是寄存器输出端的 Q doesn't connect to any net，剩余 1 个是加法器的进位输出 CO doesn't connect to any net，均可忽略。

```

rtl>icc > floating.rpt
1  (#)
2  png_top.CEL;1 (#3172)
3  png_top_lvs.err;1 (#3172)
4  png_top_lvs.err;1 (#3172)
5 #####
6 Num ErrId Type Layer Info BBBox Status | > QN doesn't connect to any net Aa Ab * 2264 中的 2 ↑ ↓ = ×
7 0 10041 Floating Port OUTPUT PortInst filter_top/filter/sum_abs_2_r_reg[3] QN doesn't connect to any net. (920.760 10.550) (921.110 12.570)
8 1 10042 Floating Port OUTPUT PortInst filter_top/filter/sum_abs_2_r_reg[4] QN doesn't connect to any net. (931.960 10.550) (932.310 12.570)
9 2 10043 Floating Port OUTPUT PortInst bs_top/b5s_output/dat_out_buf_r_reg[60] QN doesn't connect to any net. (1224.010 13.190) (1224.360 13.190)
10 3 10044 Floating Port OUTPUT PortInst bs_top/b5s_output/dat_out_buf_r_reg[61] QN doesn't connect to any net. (1230.730 10.550) (1231.880 12.570)

```

```

rtl>icc > floating.rpt
342  335 17428 Floating Port OUTPUT PortInst lz77_top/dat_inp_r_reg[499] QN doesn't connect to any net. (1007.800 134.150) (1008.150 136.170) Error
343  336 17429 Floating Port OUTPUT PortInst lz77_top/dat_inp_r_reg[87] QN doesn't connect to any net. (1007.800 134.150) (1008.150 136.170) Error
344  337 17430 Floating Port OUTPUT PortInst lz77_top/dat_inp_r_reg[82] QN doesn't connect to any net. (1007.800 134.150) (1008.150 136.170) Error
345  338 17431 Floating Port OUTPUT PortInst lz77_top/dat_lit_r_reg[1] Q doesn't connect to any net. (1038.475 134.110) (1038.785 136.020) Error
346  339 17432 Floating Port OUTPUT PortInst lz77_top/dat_lit_r_reg[3] Q doesn't connect to any net. (1062.795 131.660) (1063.105 133.570) Error

```

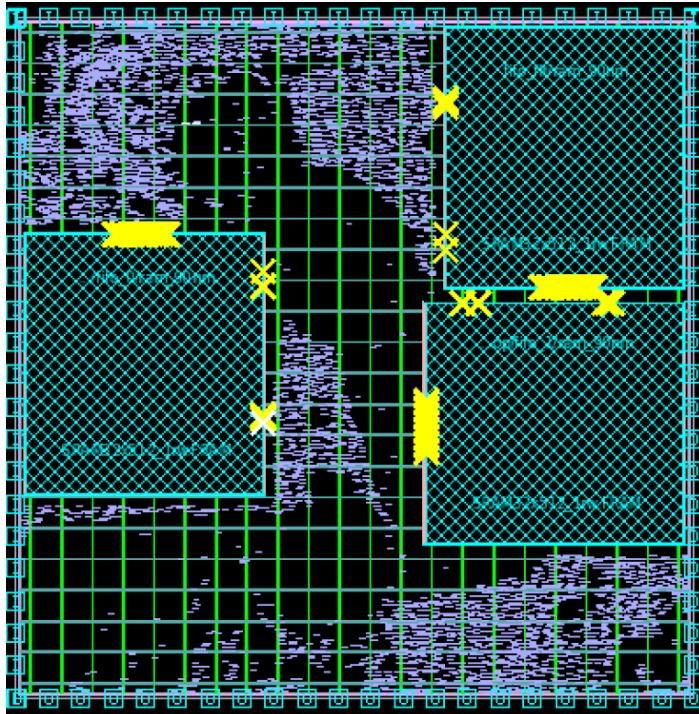
```

rtl>icc > floating.rpt
1507  1508 41798 Floating Port OUTPUT PortInst filter_top/filter/dat_c_r_reg[24] Q doesn't connect to any net. (94.495 1090.270) (94.895 1092.180) Error
1508  1501 41798 Floating Port OUTPUT PortInst filter_top/filter/dat_c_r_reg[31] Q doesn't connect to any net. (94.495 1090.270) (94.895 1092.180) Error
1509  1502 41799 Floating Port OUTPUT PortInst filter_top/filter/dat_c_r_reg[29] Q doesn't connect to any net. (94.495 1090.270) (94.895 1092.180) Error
1510  1503 41800 Floating Port OUTPUT PortInst filter_top/filter/add_0_root_add_0_root_add_469_3/U1_8 CO doesn't connect to any net. (260.985 16.150) Error

```

共 708 个 Min Area，从图中可以看出均在 SRAM 附近，进一步分析发现，SRAM 的 pin 面积仅为 $0.0256\mu\text{m}^2$ ，pin 含多层金属，未连出去的其他金属层均存在面积过小的问题，暂时未修复该问题，请教学长后得知可以导入到 virtuoso 中通过指令强行添加金属以修复此问题。

short 和 open 问题已被修复，基本可以认为通过了 ICC 内部的 DRC 和 LVS。



7.1.9 Export Design and Reports

1. 写出 gds 用于生产制造。
2. 写出 verilog 网表、sdc 文件和寄生参数文件用于进行静态时序分析和形式验证。
3. 报告建立时间约束、保持时间约束，面积和功耗。

在 dc 中将时钟设为 12ns，即 83MHz。在 icc 后建立时间满足，且关键路径裕量为 5.35ns，以此推断最高频率能达到 150MHz；

adler32/sub_188_aco/U2_14/CO (FADDX1)	0.09 &	7.16 r
adler32/U14/Q (XOR3X1)	0.06 &	7.22 r
adler32/U135/Q (A022X1)	0.07 &	7.29 r
adler32/adler32_s1_cur_r_reg[15]/D (DFFARX1)	0.00 &	7.29 r
data arrival time		7.29
clock clk (rise edge)	12.00	12.00
clock network delay (propagated)	0.80	12.80
clock uncertainty	-0.10	12.70
adler32/adler32_s1_cur_r_reg[15]/CLK (DFFARX1)	0.00	12.70 r
library setup time	-0.06	12.63
data required time		12.63
data required time	12.63	
data arrival time	-7.29	
slack (MET)		5.35

保持时间满足：

Point	Incr	Path
clock clk (rise edge)	0.00	0.00
clock network delay (ideal)	0.10	0.10
input external delay	-0.10	0.00 f
cfg_h_i[1] (in)	0.00 @	0.00 f
crc32/h_i[1] (crc32_top)	0.00	0.00 f
crc32/U87/Q (A021X1)	0.11 @	0.11 f
crc32/U86/Q (A022X1)	0.07 &	0.18 f
crc32/chunk_crc32_core/dat_i[1] (crc32_core)	0.00	0.18 f
crc32/chunk_crc32_core/U16/Q (A022X1)	0.08 &	0.26 f
crc32/chunk_crc32_core/U77/Z (DELLN3X2)	0.58 &	0.83 f
crc32/chunk_crc32_core/dat_i_buf_r_reg[1]/D (DFFARX1)	0.00 &	0.83 f
data arrival time		0.83
clock clk (rise edge)	0.00	0.00
clock network delay (propagated)	0.74	0.74
clock uncertainty	0.10	0.84
crc32/chunk_crc32_core/dat_i_buf_r_reg[1]/CLK (DFFARX1)	0.00	0.84 r
library hold time	-0.01	0.83
data required time		0.83

data required time		0.83
data arrival time		-0.83

slack (MET)		0.00

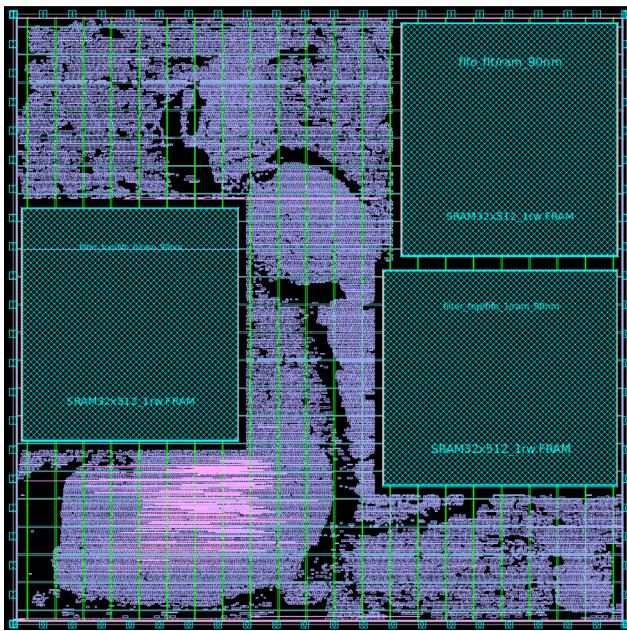
总面积为 1.69mm²:

Combinational area:	387835.085077
Buf/Inv area:	59258.879780
Noncombinational area:	107137.844963
Macro/Black Box area:	694548.562500
Net Interconnect area:	208622.373329
Total cell area:	1189521.492540
Total area:	1398143.865869
Core Area:	1688336
Aspect Ratio:	0.9993
Utilization Ratio:	1.0000

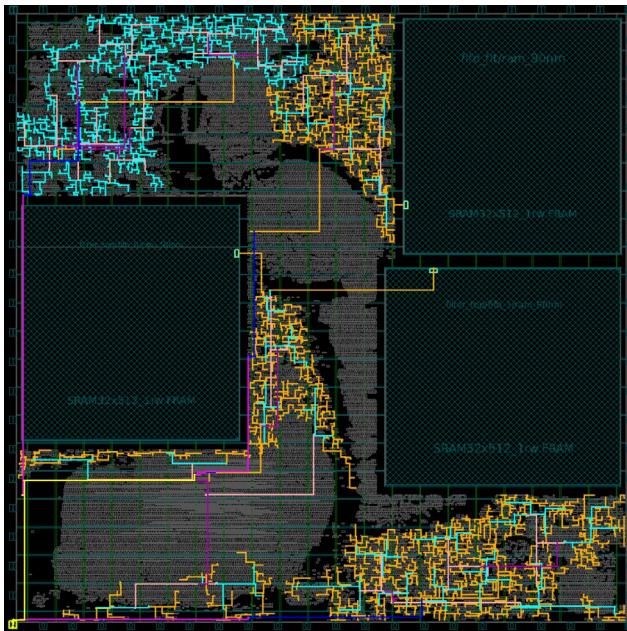
功耗为 4.95mW:

Power Group	Internal Power	Switching Power	Leakage Power	Total Power (%)	Attrs
io_pad	0.0000	0.0000	0.0000	0.0000 (0.00%)	
memory	-7.7033e+00	24.4438	0.0000	16.7405 (0.34%)	
black_box	0.0000	0.0000	0.0000	0.0000 (0.00%)	
clock_network	376.4254	1.3366e+03	4.5616e+06	1.7176e+03 (34.71%)	
register	47.2114	53.2320	3.7765e+08	478.0908 (9.66%)	
sequential	0.0000	0.0000	0.0000	0.0000 (0.00%)	
combinational	517.3493	558.3039	1.6596e+09	2.7352e+03 (55.28%)	
Total	933.2828 uW	1.9725e+03 uW	2.0418e+09 pW	4.9476e+03 uW	

最终的版图如下，其中三块宏单元是 32X512 的 SRAM；核外围有电源环，宏单元周围有电源环，宏单元外部布了很多电源条以减少 IR drop。



时钟树如下，组合逻辑和 filler 处无时钟布线。



7.2 IO Pad

与前述不加 IO Pad 的流程相比，额外所需步骤如下：

1. 在 dc 产生的网表后包一层 IO pad 作为新的顶层文件，输入 pad 选择 I1025，输出 pad 选择 B12I1025（无输出 pad，因此选择双向 pad 作为输出 pad）。
2. 载入 IO pad 相关的逻辑库和物理库，新的顶层文件。
3. 创建核电源 pad 和 IO 电源 pad，corner，并设置各 pad 的位置。
4. 插入 pad filler。

遇到的问题如下：

1. 使用 wire bond 类型 pad 库时，发现 corner 的大小明显小于 pad。此外，有同学发现 B12I1025 的 VDD/VSS(理论上应该朝向芯片内部)与 PADIO (理论上应该朝向芯片外部)在同一侧，导致 pad 处无法布线成功。
2. 经同学建议，使用 flip chip 类型的工艺库制作 pad 物理库并旋转 270° 后，corner 的大小正常，pad 的朝向正常，但在 place_opt 时遇到找不到 pad 物理库的问题，暂未解决，可能是制作库过程有误。

Error: No physical cell pin PADIO for logical cell I1025. ([PSYN-410](#))
Severe Error: Fatal error: Placer did not complete. ([PSYN-375](#))

7.3 Primetime

icc 后验证结果如下，时钟周期为 12ns，，时序满足，推断最高频率能达到 150MHz 左右：

adler32/U14/Q (XOR3X1)	0.06 +	7.15 r
adler32/U135/Q (A022X1)	0.07 +	7.21 r
adler32/adler32_s1_cur_r_reg[15]/D (DFFARX1)	0.00 +	7.21 r
data arrival time		7.21
clock clk (rise edge)	12.00	12.00
clock network delay (propagated)	0.77	12.77
clock reconvergence pessimism	0.00	12.77
clock uncertainty	-0.10	12.67
adler32/adler32_s1_cur_r_reg[15]/CLK (DFFARX1)		12.67 r
library setup time	-0.06	12.60
data required time		12.60
data required time		12.60
data arrival time		-7.21
slack (MET)		5.39

7.4 Formality

形式验证通过。

```
***** Verification Results *****
Verification SUCCEEDED
-----
Reference design: r:/WORK/png_top
Implementation design: i:/WORK/png_top
3486 Passing compare points
-----
Matched Compare Points BBPin Loop BBNet Cut Port DFF LAT TOTAL
-----
Passing (equivalent) 135 0 0 0 34 3317 0 3486
Failing (not equivalent) 0 0 0 0 0 0 0 0
Not Compared
  Unread 0 0 0 0 0 1 0 1
*****
```

8 总结

本小组基本完成了 PNG 编码器芯片的设计任务，其支持的配置以及性能如下，基本完成了设计目标。在 150MHz 频率下，对 393x501 大小的测试图片 globe-scene-fish-bowl-pngcrush.png 进行编码仅需 27.6ms (4142049cycle/150000000cycle)。

类型	配置变量	支持的配置项
原始	图像像素	RGBA 四通道，每通道 8 比特
	图像宽高	不超过 511 像素
输出	色彩类型	RGBA
	样本位宽	8
	块类型	1 个 IHDR、1 个 IDAT、1 个 IEND
	渐进式显示	无渐进式显示
	滤波策略	每行计算最小和，自适应选择该行滤波类型
	压缩策略	1 个 LZ77 并以固定哈夫曼编码的压缩块

面积	1.69 mm ²
最高频率	150 MHz
功耗	4.95 mW