# VIVADO HLS 初体验

其实很早就知道 Vivado HLS 这个东西了，但是身边人没有人使用它，只知道它是一个可以把 C/C++ 转化成 HDL 的一个软件。直到这个暑假，有幸在上海的 LLVM 社区中国的线下聚会上，有在北京 Xilinx 工作的工程师分享了一些有关 HLS 的底层实现方面的介绍，也很有趣。而刚好在前段时间的~~划水~~实习中，有要用到一丢丢的 HLS，所以记录一下使用心得。下面要使用的样例是来自 pp4fpgas 中的 Huffman Encoding 章节。源码可以到前面给出的 Github 链接找到。再次推荐一下这本书，从高层到底层的讲解都十分的棒，相见恨晚！

## Start Up

找到仓库中的 examples 文件夹后，里面稍稍显得有点乱，找到里面的

```
examples
  ├── huffman_canonize_tree_alternate.cpp (这个实际是不需要的)
  ├── huffman_canonize_tree.cpp
  ├── huffman_compute_bit_length.cpp
  ├── huffman_create_codeword.cpp
  ├── huffman_create_tree.cpp
  ├── huffman_encoding.cpp                (顶层文件)
  ├── huffman_encoding.tcl                (创建工程的脚本，看一下就知道它做了
什么)
  ├── huffman_encoding_test2.cpp
  ├── huffman_encoding_test.cpp
  ├── huffman_filter.cpp
  ├── huffman.h                           (一些模块的参数定义)
  ├── huffman.random256.golden            (仿真要用到的参考值)
  ├── huffman.random256.txt               (仿真要用到的输入值)
  ├── huffman_sort.cpp
  ├── huffman_truncate_tree.cpp
```

为了让我们的工程看起来简洁一些，把它们拷贝出来，单独放到一个工程文件夹中吧！像这样

```
axi-huffman-encoding-core
├── huffman_encoding.tcl
├── src (编译 IP core 需要的源文件)
|   ├── huffman_canonize_tree.cpp
|   ├── huffman_compute_bit_length.cpp
|   ├── huffman_create_codeword.cpp
|   ├── huffman_create_tree.cpp
|   ├── huffman_encoding.cpp
|   ├── huffman_filter.cpp
|   ├── huffman.h
```

```
|   ├── huffman_sort.cpp
|   └── huffman_truncate_tree.cpp
└── test (放一些测试样例或者测试脚本)
    ├── huffman_encoding_test.cpp
    ├── huffman.random256.golden
    └── huffman.random256.txt
```

现在准备工作就差不多了，现在有两种方式来构建我们的项目，1. 利用 tcl 脚本来帮助我们自动完成所有的步骤。2. 手动创建 HLS 工程。因为原作者提供了写好的 tcl 脚本，所以这样比较方便一些，但我们修改了工程目录结构，需要对 tcl 脚本做一点小小的改动。拿比较顺手的编辑器打开 `huffman_encoding.tcl` ，根据自己的需要做一些调整

```
open_project huffman_encoding.proj -reset  # 创建一个
huffman_encoding.proj 的工程目录，不需要修改
add_files {     # 添加需要的源文件，需要修改路径为
  ./src/huffman_canonize_tree.cpp
  ./src/huffman_create_tree.cpp
  ./src/huffman_filter.cpp
  ./src/huffman_compute_bit_length.cpp
  ./src/huffman_encoding.cpp
  ./src/huffman_sort.cpp
  ./src/huffman_create_codeword.cpp
  ./src/huffman_truncate_tree.cpp
}

add_files -tb { ./test/huffman_encoding_test.cpp } -cflags "-I./src" #
添加测试文件，因为 huffman.h 不在同一个文件夹，这里要设置编译器参数 -I 帮助编译器寻
找头文件，就像平时使用 gcc 或者 clang 一样，因为 HLS 是基于它们的嘛
add_files -tb { ./test/huffman.random256.txt
./test/huffman.random256.golden }
set_top huffman_encoding # 把 huffman_encoding.cpp 中的 huffman_encoding
函数设置为顶层文件
#set_top create_tree
open_solution solution -reset       # 创建一个 solution
set_part xc7z020clg400-1            # 设置器件类型，我自已使用的是 Zynq-7020
所以我修改了这里
create_clock -period 5              # 时钟约束，按需求修改
csim_design -compiler clang         # 选择编译器为 clang，进行仿真
csynth_design                       # 进行 C 综合
# 这里加一句导出 IP 核，后面是一些基本参数，可以参考 ug902 来了解这里的设置
export_design -flow syn -format ip_catalog -rtl verilog -vendor
"com.xilinx.hls" -version "0.0.1"
exit
```

在当前文件夹打开 Vivado HLS 命令行，Linux 下可以直接把 `vivado_hls` 添加到环境变量，Windows 下叫做 `Vivado HLS Command Line Prompt`，输入 `vivado_hls -f huffman_encoding.tcl` 等待脚本运行完毕，就可以看到生成的 IP 核了。

Known issues: 因为原作者的开发环境是 Linux 系统（因为他的 huffman.random256.golden 文件的编码形式是 Linux 下的编码形式，与 Windows 平台是不同的，所以进行 C 仿真或者联合仿真的时候每一行都会报错的，解决办法是注释掉 csim_design 那一行，或者等脚本跑完使用

```
axi-huffman-encoding-
core/huffman_encoding.proj/solution/csim/build/huffman.random256.out
```

替换掉 golden 文件，因为这是由你使用的系统所生成，所以文本的换行符就是根据你的系统来的，谁叫你们不使用 Linux 呢？ :P）

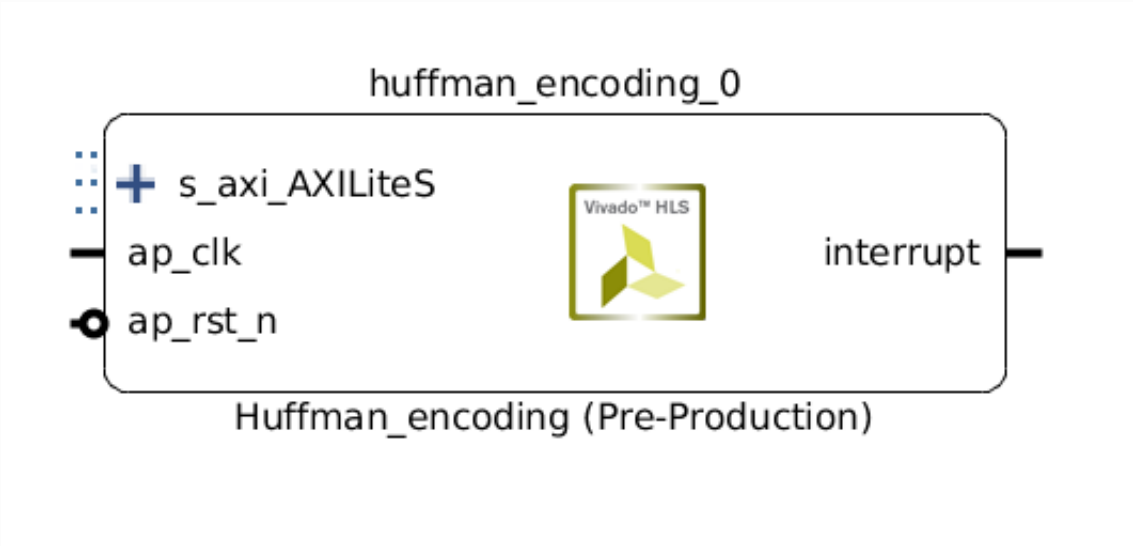
到这里，我们已经可以生成 IP 核了！不过，现在生成的 IP 核的控制信号是很复杂的，一般我们喜欢把控制信号以及速度要求不高的数据信号修改为 AXI-Lite 总线形式的，打开 `huffman_encoding.cpp` 添加几句编译器的编译选项

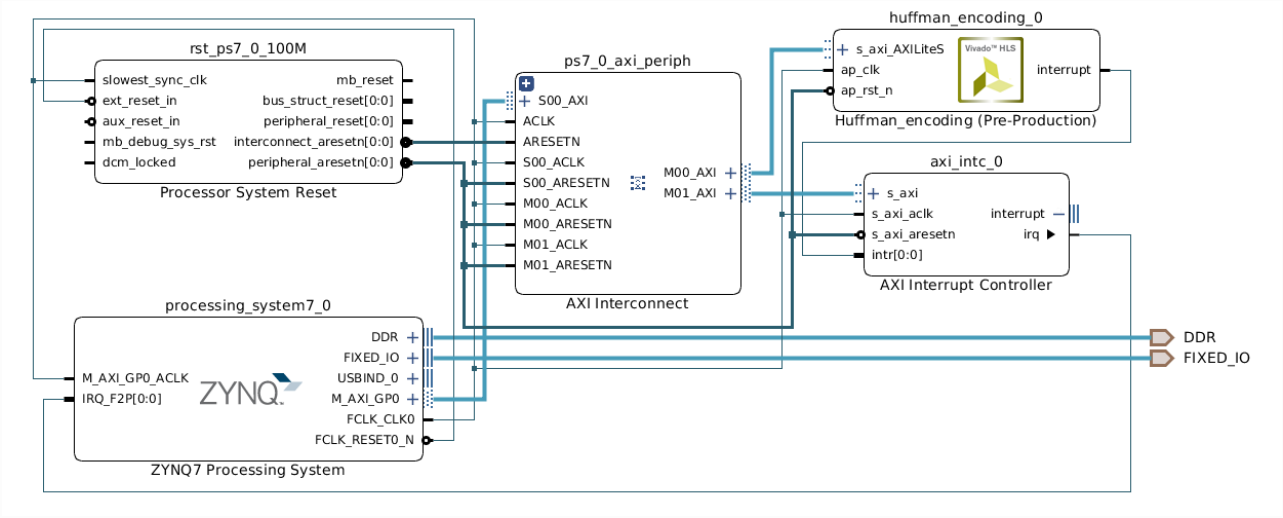
```
void huffman_encoding(
  /* input  */ Symbol symbol_histogram[INPUT_SYMBOL_SIZE],
  /* output */ PackedCodewordAndLength encoding[INPUT_SYMBOL_SIZE],
  /* output */ int *num_nonzero_symbols) {

/*
 * Create an new AXI-LITE bus as control bus
 * */
#pragma HLS INTERFACE s_axilite port=return

/*
 * Create an AXI bus for data transferring
 * */
#pragma HLS INTERFACE s_axilite port=symbol_histogram
#pragma HLS INTERFACE s_axilite port=encoding
#pragma HLS INTERFACE s_axilite port=num_nonzero_symbols
```

由于 Huffman 编码本身是为了无损压缩的，所以对存储空间的占用也比较苛刻，很多情况下，综合出来的存储器往往是每个地址对应两个甚至更多个数据单元(虽然这样并没什么，只是增加了一丢丢计算地址的难度)，为了使后面工作简化一些以及简单起见，我们修改一些头文件里的参数。打开 `huffman.h` 把 `const static int SYMBOL_BITS = 10;` 修改为 `const static int SYMBOL_BITS = 32;` 这样生成的 IP 中，一个地址只对应一个数据，测试时候会方便一些，在对存储要求较高的时候可以按照源代码给的 `log2(字符集大小)` 进行计算。现在重新编译我们的项目即可。

## Block Design

打开 Vivado 的 IP Catalog 可以添加刚刚生成的 IP，像下面这个样子，挂载到 Zynq 芯片周围即可。




## Generate Hardware & Run

这里由于笔者使用的是 PYNQ 板卡，对于不含操作系统裸奔的板卡，笔者还未尝试，请自行查阅资料。有了 Block Design 后，生成 HDL Wrapper 并综合生成 bit 文件。这里，笔者自己写了这个模块的一些单元测试，放到了 Github，按照 PYNQ 的使用方法，下载到板上使用即可。

```
# 这里的寄存器地址注释来自于 HLS 工程自动生成的驱动代码中
# 所在目录为
# (axi-huffman-encoding-
core/huffman_encoding.proj/solution/impl/misc/drivers/huffman_encoding_v
0_0/src/xhuffman_encoding_hw.h)
# AXILiteS
# 0x0000 : Control signals  # 控制信号地址，bit 0 处写 1 即可让模块开始工作
```

```
#          bit 0  - ap_start (Read/Write/COH)
#          bit 1  - ap_done (Read/COR)
#          bit 2  - ap_idle (Read)
#          bit 3  - ap_ready (Read)
#          bit 7  - auto_restart (Read/Write)
#          others - reserved
# 0x0004 : Global Interrupt Enable Register # 允许中断的寄存器，写 1 即可允许产生 interrupt 信号
#          bit 0  - Global Interrupt Enable (Read/Write)
#          others - reserved
# 0x0008 : IP Interrupt Enable Register (Read/Write) # 判断和决定中断种类的寄存器
#          bit 0  - Channel 0 (ap_done)              # bit 0 为 done 信号的中断
#          bit 1  - Channel 1 (ap_ready)             # bit 1 为 ready 信号的中断
#          others - reserved
# 0x000c : IP Interrupt Status Register (Read/TOW)   # 中断状态的寄存器
#          bit 0  - Channel 0 (ap_done)
#          bit 1  - Channel 1 (ap_ready)
#          others - reserved
# 0x1000 : Data signal of num_nonzero_symbols              # 非 0 标志的个数，请参阅 IP core 的使用方法和用途
#          bit 31~0 - num_nonzero_symbols[31:0] (Read)
# 0x1004 : Control signal of num_nonzero_symbols           # 非 0 标志的个数的寄存器的控制信号
#          bit 0  - num_nonzero_symbols_ap_vld (Read/COR)
#          others - reserved
# 0x0400 ~
# 0x07ff : Memory 'symbol_histogram_value_V' (256 * 32b)     # 写入频率表中 symbols 的一片内存
#          Word n : bit [31:0] - symbol_histogram_value_V[n]
# 0x0800 ~
# 0x0bff : Memory 'symbol_histogram_frequency_V' (256 * 32b) # 写入频率表中 frequency 的一片内存
#          Word n : bit [31:0] - symbol_histogram_frequency_V[n]
# 0x0c00 ~
# 0x0fff : Memory 'encoding_V' (256 * 32b)                   # 最后的 encoding 结果，前 27 位为编码结果，后 5 位为编码字长
#          Word n : bit [31:0] - encoding_V[n]
# (SC = Self Clear, COR = Clear on Read, TOW = Toggle on Write, COH = Clear on Handshake)

from pynq import Overlay

overlay = Overlay("../bitstream/huffman-encoding-test.bit")
overlay.download()      # 下载 bit 文件

huffman_encoding = overlay.huffman_encoding_0
```

```
symbol_table = [    # 频率表，这里使用字典进行表示
    { 'symbol': 'a', 'freq': 3 },
    { 'symbol': 'b', 'freq': 1 },
    { 'symbol': 'c', 'freq': 2 },
    { 'symbol': 'd', 'freq': 5 },
    { 'symbol': 'e', 'freq': 5 },
    { 'symbol': 'f', 'freq': 1 },
    { 'symbol': 'g', 'freq': 0 },
]

# 写入频率信息
for idx, sym in enumerate(symbol_table):
    # write symbol & frequency
    huffman_encoding.write(0x0400 + 4*idx, ord(sym['symbol']))
    huffman_encoding.write(0x0800 + 4*idx, sym['freq'])

# 开始计算
huffman_encoding.write(0x0000, 1)

from time import sleep
# 等待一秒进行读取，也可以使用中断，参见笔者代码
(https://github.com/sopynq/huffman-encoding-core/blob/master/tests/irq-
test/notebook/irq-test.ipynb)
sleep(1)

# read number of symbols
num = huffman_encoding.read(0x1000)
print('There are ' + str(num) + ' symbols in huffman tree:')

# read encoding
get_bin = lambda x, n: format(x, 'b').zfill(n)
for idx, sym in enumerate(symbol_table):
    encoding = huffman_encoding.read(0x0c00 + 4*ord(sym['symbol']))
    print('symbol : ' + sym['symbol'] + ', code word : ' +
get_bin(encoding, 32))
```

In [1]:
```
# AXILiteS
# 0x0000 : Control signals
#          bit 0  - ap_start (Read/Write/COH)
#          bit 1  - ap_done (Read/COR)
#          bit 2  - ap_idle (Read)
#          bit 3  - ap_ready (Read)
#          bit 7  - auto_restart (Read/Write)
#          others - reserved
# 0x0004 : Global Interrupt Enable Register
#          bit 0  - Global Interrupt Enable (Read/Write)
#          others - reserved
# 0x0008 : IP Interrupt Enable Register (Read/Write)
#          bit 0  - Channel 0 (ap_done)
#          bit 1  - Channel 1 (ap_ready)
#          others - reserved
# 0x000c : IP Interrupt Status Register (Read/TOW)
#          bit 0  - Channel 0 (ap_done)
#          bit 1  - Channel 1 (ap_ready)
```

```
#          others - reserved
# 0x1000 : Data signal of num_nonzero_symbols
#          bit 31~0 - num_nonzero_symbols[31:0] (Read)
# 0x1004 : Control signal of num_nonzero_symbols
#          bit 0  - num_nonzero_symbols_ap_vld (Read/COR)
#          others - reserved
# 0x0400 ~
# 0x07ff : Memory 'symbol_histogram_value_V' (256 * 32b)
#          Word n : bit [31:0] - symbol_histogram_value_V[n]
# 0x0800 ~
# 0x0bff : Memory 'symbol_histogram_frequency_V' (256 * 32b)
#          Word n : bit [31:0] - symbol_histogram_frequency_V[n]
# 0x0c00 ~
# 0x0fff : Memory 'encoding_V' (256 * 32b)
#          Word n : bit [31:0] - encoding_V[n]
# (SC = Self Clear, COR = Clear on Read, TOW = Toggle on Write, COH = Clear on Handshake)

from pynq import Overlay

overlay = Overlay("../bitstream/huffman-encoding-test.bit")
overlay.download()

huffman_encoding = overlay.huffman_encoding_0

symbol_table = [
    { 'symbol': 'a', 'freq': 3 },
    { 'symbol': 'b', 'freq': 1 },
    { 'symbol': 'c', 'freq': 2 },
    { 'symbol': 'd', 'freq': 5 },
    { 'symbol': 'e', 'freq': 5 },
    { 'symbol': 'f', 'freq': 1 },
    { 'symbol': 'g', 'freq': 0 },
]

for idx, sym in enumerate(symbol_table):
    # write symbol & frequency
    huffman_encoding.write(0x0400 + 4*idx, ord(sym['symbol']))
    huffman_encoding.write(0x0800 + 4*idx, sym['freq'])

# start
huffman_encoding.write(0x0000, 1)

from time import sleep

sleep(1)

# read number of symbols
num = huffman_encoding.read(0x1000)
print('There are ' + str(num) + ' symbols in huffman tree:')

# read encoding
get_bin = lambda x, n: format(x, 'b').zfill(n)
for idx, sym in enumerate(symbol_table):
    encoding = huffman_encoding.read(0x0c00 + 4*ord(sym['symbol']))
    print('symbol : ' + sym['symbol'] + ', code word : ' + get_bin(encoding, 32))
```

```
There are 6 symbols in huffman tree:
symbol : a, code word : 00000000000000000000000000000010
symbol : b, code word : 00000000000000000000000011100100
symbol : c, code word : 00000000000000000000000001100011
symbol : d, code word : 00000000000000000000000001000010
symbol : e, code word : 00000000000000000000000000100010
symbol : f, code word : 00000000000000000000000111100100
symbol : g, code word : 00000000000000000000000000000000
```

## Conclusion

Vivado HLS 是一个十分有趣的工具，很多情况下利用它可以快速生成我们需要的算法，并且我个人认为它在 SoC 之类方面的应用以后一定会十分重要！PYNQ 也是一款十分优秀的板卡，能够快速上手，并可动态的改变 PL 端的逻辑，之前一直在思考为什么不能利用 ARM 核来动态的刷新周围电路，结果 PYNQ 真的可以这样操作，算是让我长了见识！

# Reference

(1)(pp4fpgas)(https://github.com/KastnerRG/pp4fpgas)

(2)(ug902)(https://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_1/ug902-vivado-high-level-synthesis.pdf) HLS 的手册，还是不错的