单 LUT 刷新

1、研究背景

1.1 相关基础知识

FPGA 是实现高性能计算与网络的重要工具，得益于其高度的并行性与用户可编程的特性，FPGA 得到了越来越广泛的应用。FPGA 由 CLB（Configurable Logic Block，可编程逻辑单元）、BRAM（Block RAM，块 RAM）、DSP48E1（专用数字处理单元）、可编程布线资源、可编程 IO 资源等部分组成，其中，CLB 是实现逻辑功能的基础，Xilinx 7 系列 FPGA CLB 内部结构如图 1.1 所示[1]。

从图 1.1 中我们可以看到，一个 CLB 由两个 SLICE 组成，而 SLICE 可以分为 SLICEL 和 SLICEM，SLICEL 仅可以实现逻辑功能，而 SLICEM 除了可以实现逻辑功能外，还可以实现位宽为 1bit、深度为 64bit 的 RAM， SLICEL 和 SLICEM 的结构如图 1.2 和图 1.3 所示[1]，SLICEL 和 SLICEM 都是由 4 个 6 输入 LUT（Look up table，查找表）、3 个 MUX（数据选择器）、1 个 Carry Chain（进位链）和 8 个 Flip-Flop（触发器）组成，其中，LUT 本质是一块 6 地址输入 1 数据输出的存储器（不考虑 5 输入 LUT 的情况），通过改变其内容，可以得到各种各样的真值表，也就得到了各种各样的逻辑功能，这是 FPGA 可编程性的基础之一。

对比图 1.2 和图 1.3 可以发现，SLICEL 的 4 个 LUT，只有 6bit 地址输入和 1bit 数据输出（不考虑 5 输入 LUT 的情况），当 bitstream 文件配置下去后，用户是无法对 LUT 内容进行更改的，可以将之看作是一块 ROM；而 SLICEM 除 6bit 读地址输入和 1bit 数据输出外，还有 6bit 的写地址输入（图 1.2 中 W6:W1）和 1bit 的写数据使能（图 1.2 中 WEN），2bit 的写数据输入（DI1 和 DI2），这也是 SLICEM 的 LUT 可以被配置为位宽为 1bit、深度为 64bit 的 RAM 的原因。

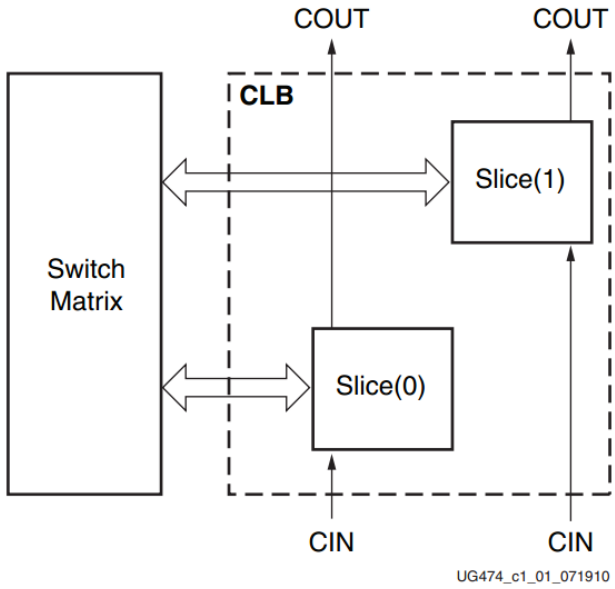

图 1.1 CLB 内部结构

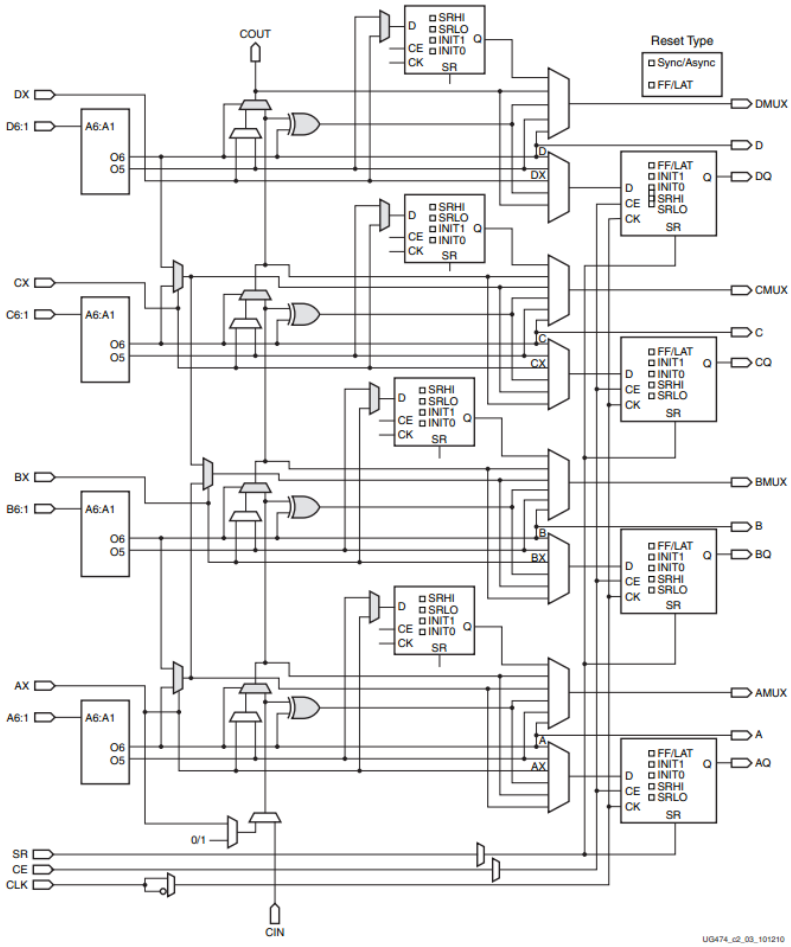

图 1.2 SLICEL 内部结构

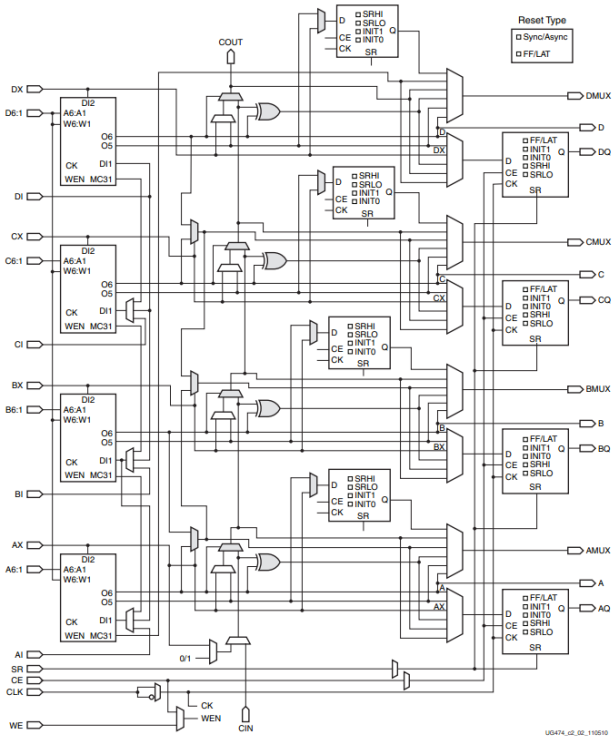

图 1.3 SLICEM 内部结构

1.2 研究目的

在 7 系列 FPGA 中，将近 2/3 的 SLICE 是 SLICEL，其余的是 SLICEM[1]，也就是说，FPGA 内 2/3 的资源在 bitstream 文件下载后，其逻辑功能就无法更改了，除非修改代码并生成新的 bitstream 文件。这给可重构计算、基于 LUT 的高带宽查找算法等研究带来了很大的困扰：这些研究希望在设备运行时对 LUT 的内容进行动态修改，而只有 SLICEM 的 LUT 存在写数据接口，也就是这些研究中的算法只能利用 1/3 的 LUT 资源。

作者在研究基于 FPGA 的高性能可重构 CRC 算法时，就遇到了上述问题，可编程 CRC 的关键计算模块由 LUT 实现，而为了实现 CRC 生成式的可编程，必须要使用 SLLICEM 的 LUT，导致时序不理想，且需要额外使用复杂的配置电路。

作者受到[2]的启发，发现使用 Xilinx 提供的 HWICAP IP 核即可实现对 SLICEL 的 LUT 的动态编程，无需 Vivado 软件参与 ，即可实现 ms 乃至 us 级别的重配置速度[2][3]，该方案是基于 FPGA 实现高性能可重构 CRC 的理想选择。

然而，网上关于该研究的资料是十分稀缺的；Xilinx 官方提供的内容也十分分散且有限，部分内容还被有意地忽略了；学术界的论文大多集中在如何设计高性能的 ICAP 控制器，以取代性能较低的 HWICAP 控制器。到目前为止，还无法找到关于如何使用 HWICAP，实现单个 LUT 的内容重配置的内容。作者经过

1 个多月的摸索，基本打通了单个 LUT 重配置的各个环节，最终基于 Artix-7 FPGA 开发板，搭建了一套演示环境，实现了单个 LUT 内容的读取与重配置，该项研究的各项内容如下所述：

第二节介绍 LUT 的寻址与配置方式，只有知道 LUT 如何寻址，才能对特定位置的 LUT 进行重配置；第三节介绍 LUT 的信息提取，包括 LUT 位置提取与 LUT 引脚映射关系提取；第四节对 rbt 文件进行解析（仅限于 LUT 内容部分），只有了解 LUT 初始值是如何映射到 rbt 文件中，才能利用 HWICAP 实现 LUT 内容的正确配置；第五节介绍 HWICAP 的具体操作，包括 HWICAP IP 核的操作与 ICAP 的操作命令；第六节搭建了验证系统，基于 Artix-7 FPGA 开发板，在 Vivado 环境下，使用 TCL 命令，实现单个 LUT 的重配置;第七节对上述工作进行总结，指出后面的工作方向。

2、LUT 寻址与配置方式

1、FPGA 基本结构

想要对 LUT 进行寻址，必须要知道 LUT 在 FPGA 中的组织方式，也就是知道 FPGA 的结构，需要说明的是，Xilinx 7 系列 FPGA 与其前代产品相比，组成出现了较大的变化，在这里介绍 Virtex-5 FPGA 与 Artix-7 FPGA 的具体结构。

Virtex-5 系列 FPGA 和 Artix-7 系列 FPGA 都是基于 ASMBL（Advanced Silicon Modular Block）架构的[1][4] (但是 Virtex-5 基于二代 ASMBL 技术，Artix-7 基于四代 ASMBL 技术) ，这里以 7 系列 FPGA 为例，对 ASMBL 架构进行简单介绍。

7 系列 FPGA　ASMBL 架构如图 2.1 所示[1]，该架构的关键在于，资源按列排布，同一列的资源是相同的，通过组合不同的列，可以得到面向各种应用、满足各种功能的 FPGA，该架构的模块化思想，大大简化了 FPGA 的设计。我们来看一张真实的 FPGA 的内部结构图（器件型号为 XC7A100T 该图通过 Vivado 软件得到），如图 2.2 所示，该 FPGA 分为 8 个区域（8 个 Clock Region），图中大量的淡蓝色非规则部分，是已经被占用的模块，我们选择比较“干净”的 X0Y3 区域进行介绍；X0Y3 区域放大后的图片如图 2.3 所示，图中最左侧橘色部分为 IO 接口，蓝色部分为 CLB，红色部分为 BRAM 资源（RAM36E1），绿色部分为 DSP 资源（DSP48E1），这样的资源组织方式与前面说的 ASMBL 架构是完全吻合的。我们再次将图 2.2 中的蓝色方块放大，如图 2.3 所示，可以看到，每个蓝色方块

内部，都由 2 个 SLICE 组成，这又与上面列的图 1.1 是吻合的。作者肉眼数了一下，一列蓝色方块中，蓝色方块的数量是 50 个，也就是一列 CLB 中包含 50 个 CLB（这个知识后面要用到）；一列红色方块中，红色方块的数量是 10 个，也就是一列 BRAM 中包含 10 个 RAM36E1；一列绿色方块中，红色方块的数量是 20 个，也就是一列 DSP 中包含 20 个 DSP48E1；需要特别注意的是，在 FPGA 的中间部分，是一组 BUFG，如图 2.5 所示（图 2.2 放大，观察 FPGA 正中央），在 BUFG 的上方，是 FPGA 的上半部分（top），FPGA 的下方是下办部分（bottom）[6][7]，这个知识后面会用到，可能有人会有疑问，图 2.2 结构如此对称，上半部分和下办部分不是一目了然的么？但是我们需要知道，不是所有的 FPGA 都是四行两列八个 Clock Region 的，如图 2.6（芯片型号为 XC7Z020），为三行了两列六个 Clock Region，这时只能通过观察 BUFG 的位置（图 2.6 中白框所在位置），判断哪几行是 top 部分，哪几行是 bottom 部分。

再来看一下 Virtex-5 FPGA 的内部结构，如图 2.7 所示[3]，可以看到，Virtex-5 FPGA 的结构与 7 系列 FPGA 很相似，都是模块化的、按列排布的；我们重点关注一下 CLB 方面的区别，对比图 2.3 和图 2.7 可以发现，7 系列 FPGA 一列 CLB 包含 CLB 的个数为 50 个，而 Virtex-5 FPGA 一列 CLB 包含 CLB 的个数为 20 个，这种差别在后期介绍 LUT 寻址时会体现出来。

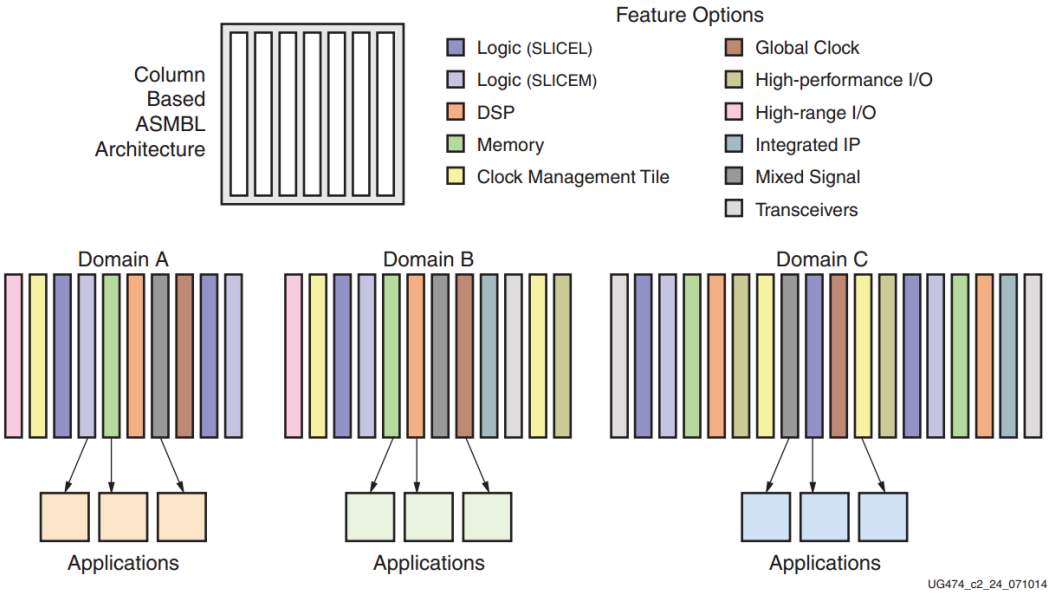

图 2.1 ASMBL 架构

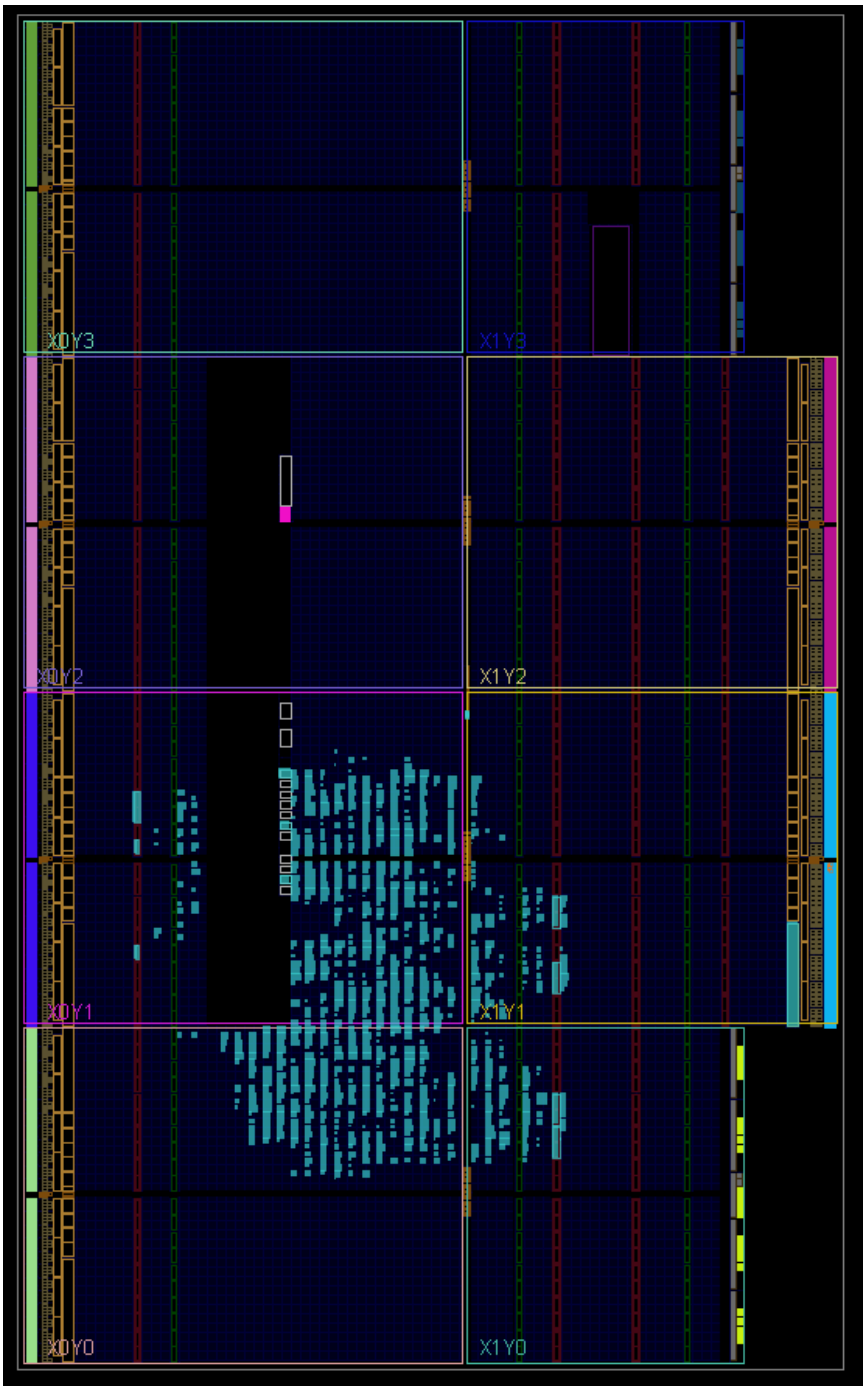

图 2.2 XC7A100T 内部结构

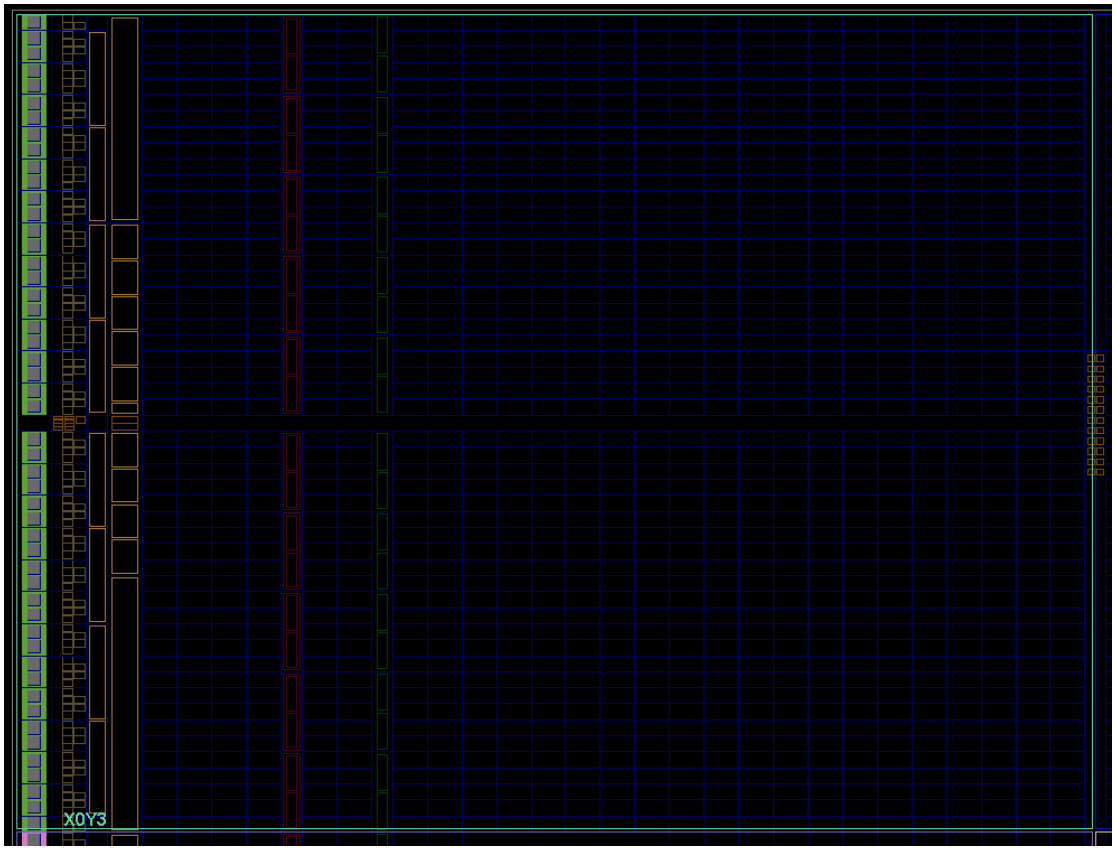

图 2.3 X0Y3 区域结构

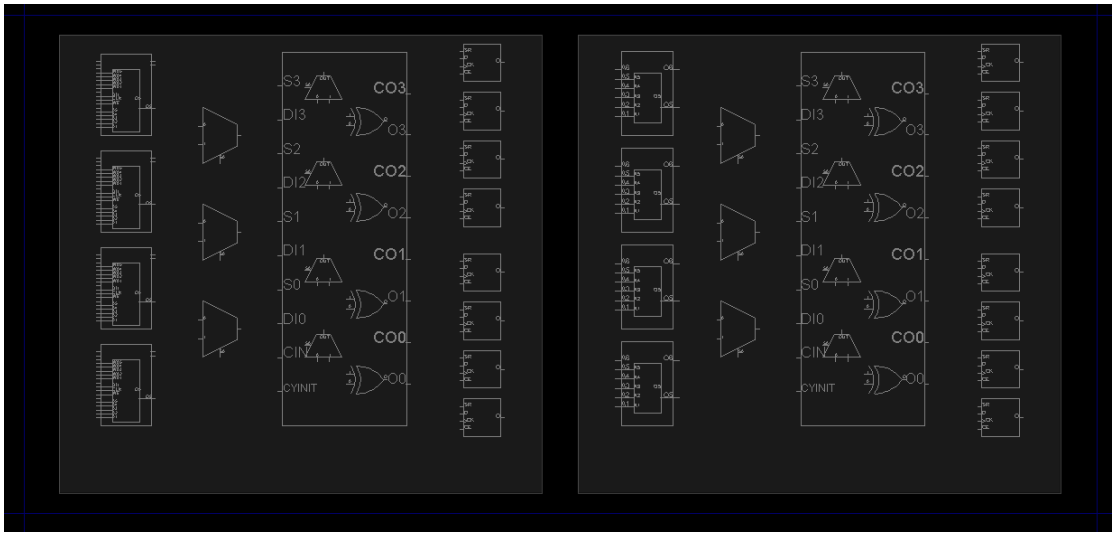

图 2.4 CLB 内部结构

图 2.5 FPGA 中间部位的 BUFG

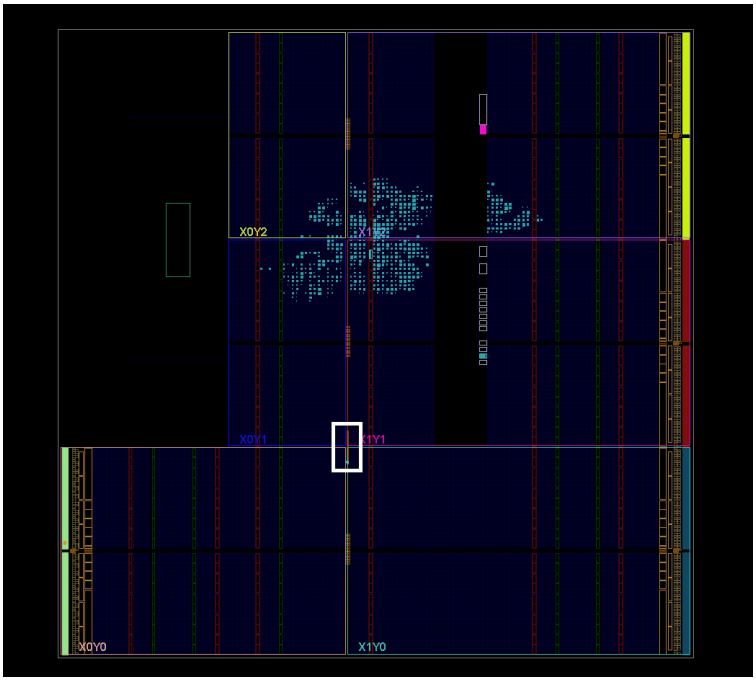

图 2.6 XC7Z020 内部结构

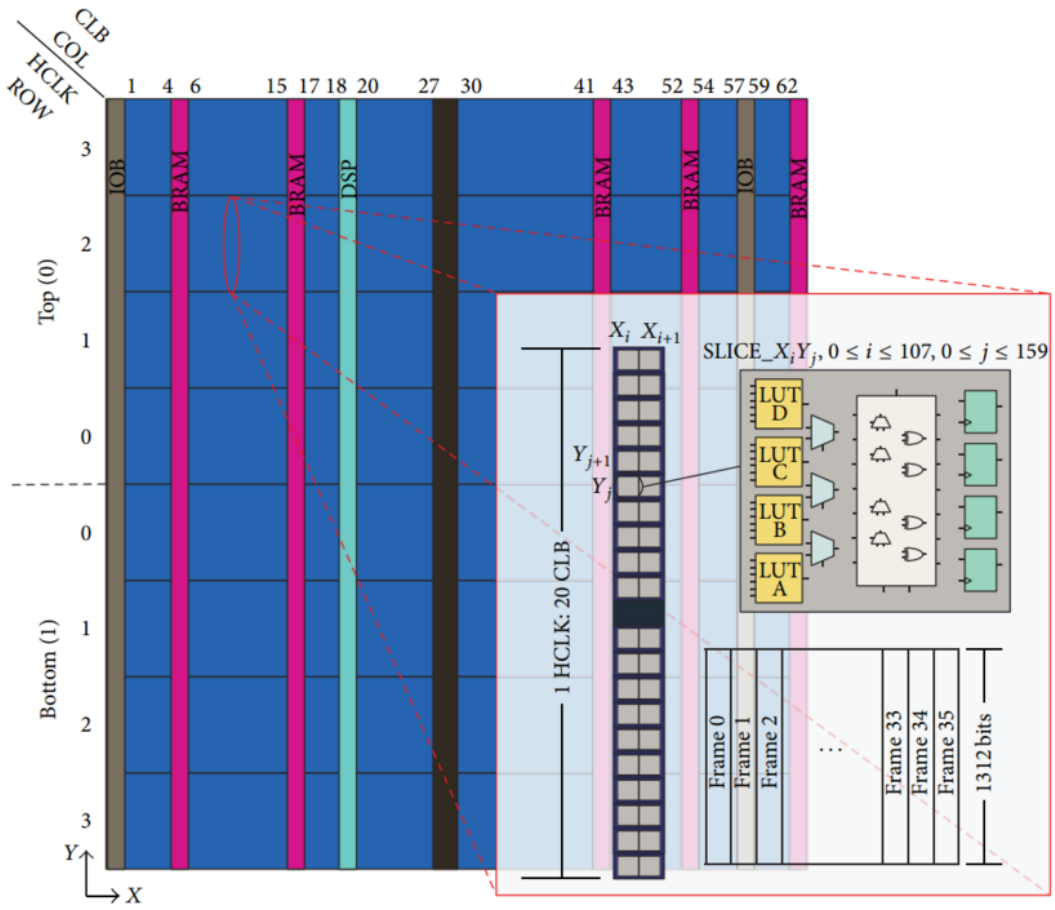

图 2.7 XC5VLX110T 所示

2.2 FPGA 配置帧格式与寻址格式

FPGA 配置数据的最小单位是帧，下面先以 Virtex-5 为例，介绍配置帧的格式与寻址方式；然后介绍 7 系列 FPGA 与 Virtex-5 在配置帧格式与寻址方式方面的区别。

2.2.1 Virtex-5 FPGA 配置帧格式与寻址格式

Virtex-5 配置帧格式如图 2.8 所示，图中共有 36 个帧，每个帧包含 41 个 word，每个 word 由 32bit 组成，图中“X odd”部分代表一个 CLB 中编号为奇数的 SLICE，而“X even”代表编号为偶数的 SLICE，也就是“X odd”代表图 2.4 中右侧的 SLICE，“X odd”代表图 2.4 中左侧的 SLICE；Frame26-Frame29 和 Frame32-Frame35 已经可以用来配置一个 CLB 内的 8 个 LUT 了，为什么配置一个 CLB 需要 36 个帧呢？因为除了需要配置 LUT 外，还需要配置触发器、连接线等其他的部分，因为我们仅需要研究 LUT 内容是如何重配置的，因此我们不对这些内容进行研究；另外，想要配置一个 LUT，使用 1 个帧是搞不定的，因为 1 个帧只

能配置 1 个 LUT 的 2 个字节（6 输入 LUT 初始值为 64bit，也就是 8 字节），需要 4 个帧才能配置一个 LUT，但是，一个帧又同时涉及到了 20 个 LUT 的配置信息，也就是一个帧会对一列 SLICE 中的 LUT 进行配置（前面提到过，Virtex-5 一列 CLB 中，CLB 数量是 20），这也是为什么要使用 RMW（read-modify-write）的思想，也就是想配置 1 个 LUT，需要先把这个 LUT 对应的 4 个帧读出来，修改该 LUT 对应的内容后，再将这 4 个帧写回去，这样才能保证其他的 LUT 不受影响。

细心的读者可以发现，图 2.8 中一个配置帧包含 41 个 word，而一列 SLICE 包含 20 个 SLICE，2 个 word 可以配置一个 SLICE 的 4 个 LUT，怎么多出了一个 word？答案在图 2.9[5]中可以找到，一个配置帧的前 20 个 word 和后 20 个 word 都是用于配置 LUT 的，而中间的 LUT 用于其它功能，更多信息可以参考 Xilinx 官方文档：UG191[5]。

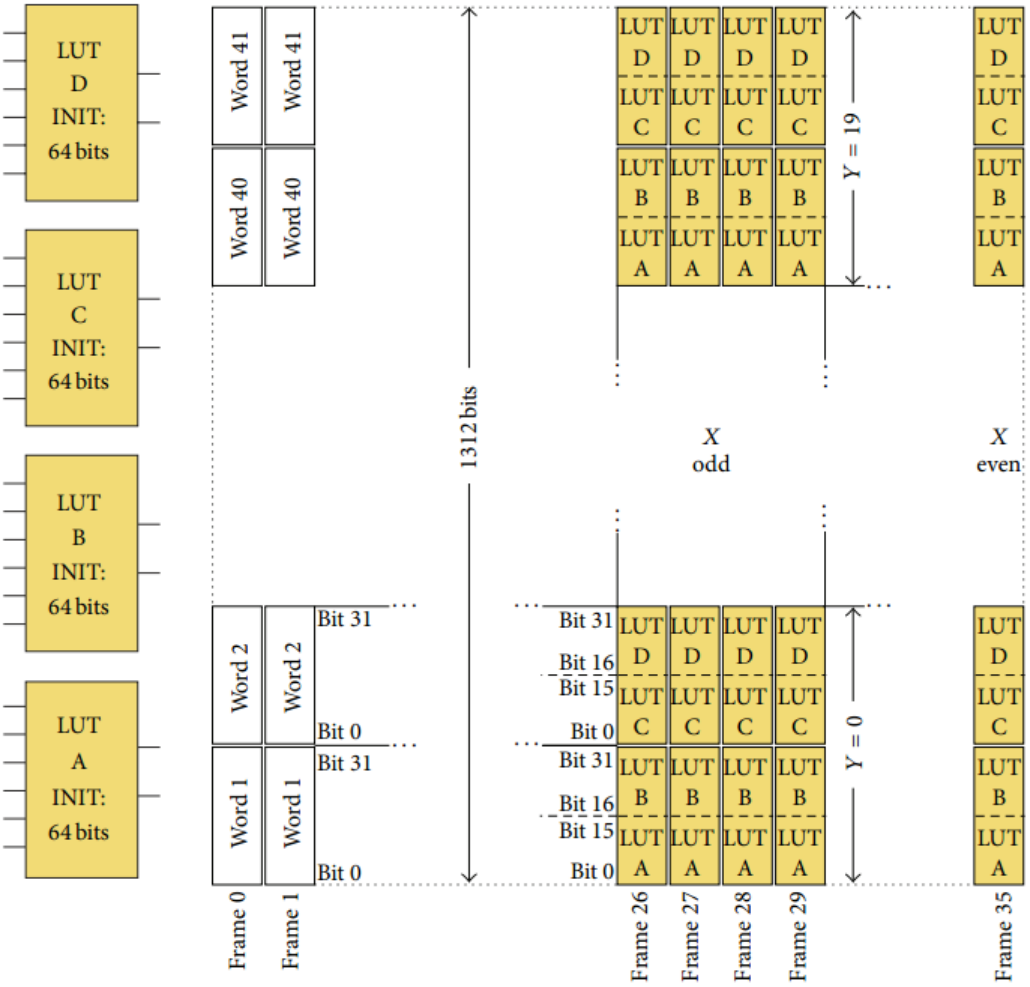

FIGURE 4: Frame bits for LUT configuration.

图 2.8 Virtex-5 配置帧格式

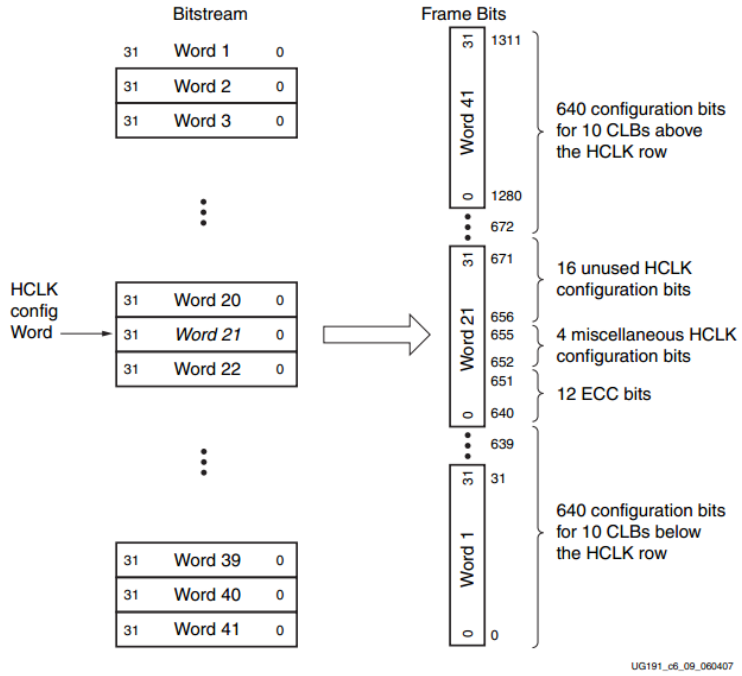

Figure 6-9: **Configuration Words in the Bitstream and Configuration Bits in a Frame**

图 2.9 一个配置帧的具体格式

Virtex-5 FPGA 配置帧的寻址格式如图 2.10[5]所示，Virtex-5 系列 FPGA 配置帧的地址由 24bit 组成，其中，bit23-bit21 作用是指示配置对象的类型，如取值为 000 代表对 CLB 进行配置；bit20 作用是指示配置的对象在 FPGA 的上半部分还是下半部分（相关内容参考图 2.5 及图 2.6）；bit19-bit15 是选择行的，如图 2.2 所示，该 FPGA 有 4 行，但是其编址方式是不是从上到下为 0、1、2、3 呢？当然不是的，其编址方式如图 2.11 所示，上半部分（top）和下半部分（bottom）是分开编址的，接近中间的行，地址为 1，远离中间的行，地址依次加 1，举个例子，图 2.2 中 X0Y2、X1Y2、X0Y3、X1Y3 位于上半部分，X0Y2、X1Y2 行地址为 0，X0Y3、X1Y3 行地址为 1；图 2.2 中 X0Y0、X1Y0、X0Y1、X1Y1 位于下半部分，X0Y1、X1Y1 行地址为 0，X0Y0、X1Y0 行地址为 1；bit14-bit7 是列地址，需要注意的是，列地址是不分 clock region 的，比如图 2.2 中 X0Y2、X0Y3 的第一列，地址是相同的，另外，CLB、BRAM、DSP 等列是统一编址的，列地址可以利用 Viavdo 软件，通过 TCL 命令提取出来；bit6-bit0 是选择具体某一帧的地址，由图 2.8 可知，配置 1 个 CLB（或者说配置一列 CLB），需要 36 个帧，但是在对 LUT 重配置的时候，并不是所有帧都要重新进行 RMW 操作，只需要对与该 LUT 相关的四个帧进行 RMW 操作即可，对 36 个帧中某一具体的帧进行寻址，就需要用到 minor address 了。

Table 6-8: **Frame Address Register Description**

Address Type	Bit Index	Description
Block Type	[23:21]	Block types are: Interconnect and Block Configuration (000), Block RAM Content (001), Interconnect and Block Special Frames (010 - typically not used by users), and Block RAM Non-Configuration Frames (011 - not used by users).
Top_B Bit	20	Select between top-half rows (0) and bottom-half rows (1).
Row Address	[19:15]	Selects the current row. The row addresses increase from bottom to top.
Column Address	[14:7]	Selects a major column, such as a column of CLBs. Column addresses start at 0 on the left and increase to the right.
Minor Address	[6:0]	Selects a frame within a major column.

图 2.10 Virtex-5 FPGA 配置帧寻址格式

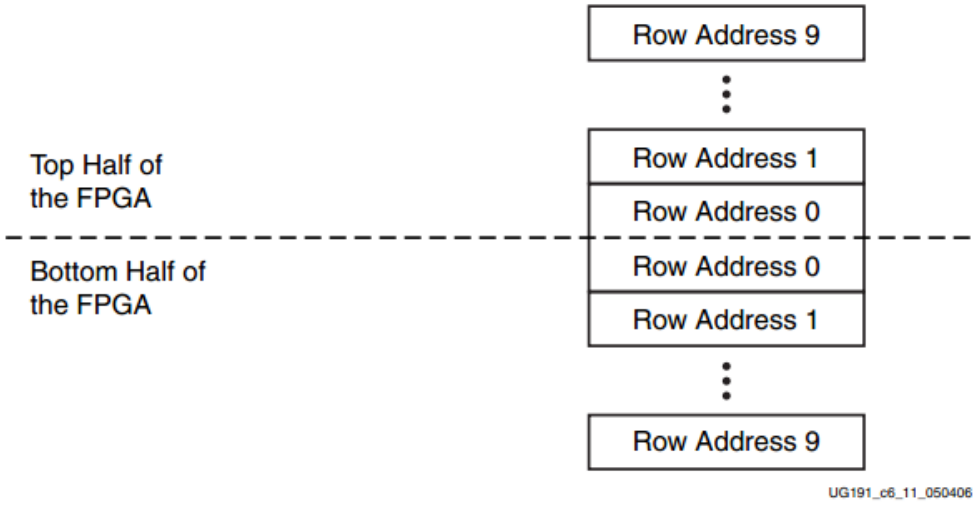

Figure 6-11: **Row Addresses in the FPGA**

图 2.11 行编址

2.2.2 7 系列 FPGA 配置帧格式与寻址格式

7 系列 FPGA 配置帧格式与图 2.8 基本类似，但是由于 7 系列 FPGA 一列 CLB 由 50 个 CLB 组成（而 Virtex-5 为 20 个），因此一个 7 系列 FPGA 配置帧包含 101 个 word；

7 系列 FPGA 的配置帧的寻址格式如图 2.12[8]所示，7 系列 FPGA 的配置帧

地址由 26bit 组成，与图 2.10 比较之后，可以发现，7 系列 FPGA 列地址为 10 个 bit，而 Virtex-5 FPGA 列地址为 8bit，这是由于 FPGA 规模变大所致；地址其他部分基本没有发生变化。

Table 5-24: **Frame Address Register Description**

Address Type	Bit Index	Description
Block Type	[25:23]	Valid block types are CLB, I/O, CLK (000), block RAM content (001), and CFG_CLB (010). A normal bitstream does not include type 011.
Top/Bottom Bit	22	Select between top-half rows (0) and bottom-half rows (1).
Row Address	[21:17]	Selects the current row. The row addresses increment from center to top and then reset and increment from center to bottom.
Column Address	[16:7]	Selects a major column, such as a column of CLBs. Column addresses start at 0 on the left and increase to the right.
Minor Address	[6:0]	Selects a frame within a major column.

图 2.12 7 系列 FPGA 的配置帧格式

3、rbt 文件解析

3.1 为什么要做 rbt 文件解析

rbt 文件与 bit 文件一样，都是 FPGA 的配置文件，bit 文件是二进制的，观察十分不方便，而 rbt 文件可以看作是 bit 文件的 ASCII 版本，其具体区别如图 3.1[8]所示。

为什么要做 rbt 文件解析？因为 2.2.1 节配置帧格式的研究中，还存在一个不清楚的地方：图 2.8 中半个 word 可以配置 1 个 LUT 的 1/4，那这半个 word 的 bit 顺序，与 Verilog 代码中 LUT 初始值的 bit 顺序，是否是一致的？是否存在某种映射关系？为了探索这一点，我首先想到的是对 rbt 文件进行解析，具体来说，建立一个工程，对一个 LUT 进行初始化，生成 rbt 文件后，观察 rbt 文件中相应的初始化值是怎样的。后来发现，进行这项研究是十分必要的，研究结果表明，LUT 的初始化值与 rbt 文件中对应的内容，并不是相等的，二者存在特定的映射关系， 且 SLICEL 和 SLICEM 中的 LUT，映射关系是不一样的。下面对研究过程进行介绍。

Table 5-1: **Xilinx Configuration File Formats**

File Extension	Bit Swapping(1)	Xilinx Tool(2)	Description
BIT	Not Bit Swapped	ISE BitGen or Vivado write_bitstream (generated by default)	Binary configuration data file containing header information that does not need to be downloaded to the FPGA. Used to program devices from the iMPACT tool or Vivado device programmer with a programming cable.
RBT	Not Bit Swapped	ISE BitGen generated if **-b** option is set) or Vivado write_bitstream (generated with -raw_bitfile argument)	ASCII equivalent of the BIT file containing a text header and ASCII 1s and 0s. (Eight bits per configuration bit.)
BIN	Not Bit Swapped	ISE BitGen (generated if **-g binary:yes** option is set) or PROMGen, or Vivado write_bitstream (generated with -bin_file argument)	Binary configuration data file with no header information. Can be used for custom configuration solutions (for example, microprocessors), or in some cases to program third-party PROMs.
MCS	Bit Swapped(3)	ISE PROMGen or iMPACT, or Vivado write_cfgmem -format MCS	ASCII PROM file format containing address and checksum information in addition to configuration data. Used mainly for device programmers and the iMPACT tool.

图 3.1 FPGA 配置文件格式

3.2 准备工作

rbt 文件中的配置数据是二进制格式的，观察十分不方便，因此第一项准备工作就是将二进制的 rbt 文件转换为十六进制，这项工作是通过一个 python 脚本实现的（位置：代码\rbt_translation.py），如图 3.2 所示，因为作者此前没有写过 python 脚本，因此代码很不完善，下面结合代码说明需要注意的地方：

（1）rbt 文件的前 7 行是 rbt 文件的介绍信息，如图 3.3 所示，在用脚本转换进制之前，这 7 行需要手动去掉（作者 python 水平受限）；

（2）图 3.2 第一行的 NUM 值为 956447，这是 rbt 文件去除前 7 行后的行数；需要注意的是，本工程针对的 FPGA 型号是 XC7A100T，每个型号的 FPGA 规模不同，NUM 的值也要相应变化；

（3）图 3.2 第三、四行为原始 rbt 文件的路径（后缀改为 txt）与转换后文件的路径，这个路径是绝对路径，需要根据实际情况进行修改。

```
1   NUM = 956447
2
3   f_read = open('X:/project/TEST_2019/read_frames/jtag_axi_icap_lut_AX7103/py_code/txt/xor_bin.txt', 'r')
4   f_write= open('X:/project/TEST_2019/read_frames/jtag_axi_icap_lut_AX7103/py_code/txt/xor_hex.txt', 'w')
5   a=f_read.readlines()
6   b=list(range(0,NUM))
7
8   for i in range(0,NUM):
9       b[i]='{:08x}'.format(int(a[i],2))
10      f_write.writelines(b[i])
11      f_write.writelines('\n')
12
13
14
15  f_read.close()
16  f_write.close()
```

图 3.2 rbt 进制转换代码

```
1   Xilinx ASCII Bitstream
2   Created by Bitstream 2017.2 SW Build 1909853 on Thu Jun 15 18:39:09 MDT 2017
3   Design name:    jtag_axi_icap_top;UserID=0XFFFFFFFF;Version=2017.2
4   Architecture:   artix7
5   Part:           7a100tfgg484
6   Date:           Wed May 01 22:41:59 2019
7   Bits:           30606304
8   11111111111111111111111111111111
9   11111111111111111111111111111111
10  11111111111111111111111111111111
11  11111111111111111111111111111111
```

图 3.3 rbt 文件介绍信息

在 UG470 中，专门有一部分介绍 bitstream 的组成，这部分需要重点了解，详细内容见 UG470 的 table 5-19。

3.3 具体流程

首先建立一个工程，在工程中例化一个 LUT，这个 LUT 的位置必须是固定的，方便进行多次对照试验，如图 3.4 所示，例化 LUT 的位置为 SLICE_X57Y53 的 LUT-D，如图 3.4 所示，为了简单起见，该 LUT 的初始值设置为“0x6996966996696996”，为什么设置成这个值呢？因为 LUT 初始值为该值时，作用为一个六输入一输出异或门，而异或门是 CRC 算法中的基本单元。

综合工程（工程在：工程\jtag_axi_icap_lut_AX7103_simple_lut_6996 文件夹下），生成 rbt 文件后，将 rbt 文件转成十六进制（见 3.2 节），结果如表 3.1 所示：

```
(*BEL="D6LUT",LOC="SLICE_X57Y53"*)    LUT6 #(
    .INIT(64'h6996966996696996)  // Specify LUT Contents
) LUT6_inst (
    .O(dout),   // LUT general output
    .I0(din[0]), // LUT input
    .I1(din[1]), // LUT input
    .I2(din[2]), // LUT input
    .I3(din[3]), // LUT input
    .I4(din[4]), // LUT input
    .I5(din[5])  // LUT input
);
```

图 3.4 LUT 初始化-1

表 3.1 rbt 文件对应行数和值-1

rbt 文件行数	rbt 文件值
510521	0x69960000
510622	0x 96690000
510723	0x 69960000
510824	0x 96690000

观察表 3.1 可以发现，rbt 文件对应值的行数，总是差了 101 个 word，与 2.2 节描述的配置帧格式是可以对应上的。那是不是可以说 Verilog 代码中 LUT 初始值的 bit 顺序（图 3.4）和 rbt 中的 bit 顺序，是一致的呢？还不能这么说，因为“6”的二进制表示是“0110”，9 的二进制表示为“1001”，可以发现这两个值都是十分对称的值。

为了进一步验证是否存在某种 bit 顺序映射关系，建立新的工程。LUT 位置不变，但初始化值设置为“0x0123456789ABCDEF”，如图 3.5 所示，综合工程（工程在：工程\jtag_axi_icap_lut_AX7103_simple_lut_0123 文件夹下），生成 rbt 文件后，将 rbt 文件转成十六进制（见 3.2 节），结果如表 3.2 所示：

```
(*BEL="D6LUT",LOC="SLICE_X57Y53"*)     LUT6 #(
      .INIT(64'h0123456789ABCDEF)  // Specify LUT Contents
   ) LUT6_inst (
      .O(dout),    // LUT general output
      .I0(din[0]), // LUT input
      .I1(din[1]), // LUT input
      .I2(din[2]), // LUT input
      .I3(din[3]), // LUT input
      .I4(din[4]), // LUT input
      .I5(din[5])  // LUT input
   );
```

图 3.5 LUT 初始化-2

表 3.2 rbt 文件对应行数和值-2

rbt 文件行数	rbt 文件值
510521	0x fe760000
510622	0x ba320000
510723	0x 98100000
510824	0x dc540000

可以发现，rbt 文件值与 LUT 初始化值完全不一样，可以证明，确实存在某种特殊的映射关系。如何确定这种映射关系呢？最简单的是，建立多个工程，每个工程初始化值只有 1bit（如 0x0000000000000001、0x0000000000000002、

0x0000000000000004、0x0000000000000008 等），观察生成 rbt 文件中对应值的位置，理论上需要建立 64 个工程才能完全确定这个关系（但其实存在规律，不用这么多），成功破译了其对应关系，测试原始数据如表 3.3 所示（为了简单起见，将 1 个 LUT 的 4 部分数据写到一起，作为一个完整的 word,“X”代表 0x0000）：

表 3.3 测试原始数据-1

LUT 初始值	rbt 数据
0x X_X_X_0001	0x 8000_X_X_X
0x X_X_X_0002	0x X_8000_X_X
0x X_X_X_0004	0x 4000_X_X_X
0x X_X_X_0008	0x X_4000_X_X
0x X_X_X_0010	0x 2000_X_X_X
0x X_X_X_0020	0x X_2000_X_X
0x X_X_X_0040	0x 1000_X_X_X
0x X_X_X_0080	0x X_1000_X_X
0x X_X_X_0100	0x X_X_X_8000
0x X_X_X_0200	0x X_X_8000_X
0x X_X_X_0400	0x X_X_X_4000
0x X_X_X_0800	0x X_X_4000_X
0x X_X_X_1000	0x X_X_X_2000
0x X_X_X_2000	0x X_X_2000_X
0x X_X_X_4000	0x X_X_X_1000
0x X_X_X_8000	0x X_X_1000_X
0x 0100_X_X_X	0x X_X_X_0008
0x 0200_X_X_X	0x X_X_0008_X
0x X_0100_X_X	0x X_X_X_0080
0x X_0200_X_X	0x X_X_0080_X
0x X_X_0100_X	0x X_X_X_0800
0x X_X_0200_X	0x X_X_0800_X
0x 0001_X_X_X	0x 0008_X_X_X
0x 0002_X_X_X	0x X_0008_X_X
0x X_0001_X_X	0x 0080_X_X_X
0x X_0002_X_X	0x X_0080_X_X
0x X_X_0001_X	0x 0800_X_X_X
0x X_X_0002_X	0x X_0800_X_X

将 64bit 初始值的最高位（MSB）定义为 bit-64，最低位（LSB）定义为 bit-

1，则 LUT 初始值与 rbt 文件的位置映射关系如表 3.4 所示：

表 3.4 映射关系-1

LUT 初始值	rbt 数据
1	64
2	48
3	63
4	47
5	62
6	46
7	61
8	45
9	16
10	32
11	15
12	31
13	14
14	30
15	13
16	29
57	4
58	20
41	8
42	24
25	12
26	28
49	52
50	36
33	56
34	40
17	60
18	44

观察规律后，将该映射规律用 python 代码（文件位置：代码\map_SliceL.py）表示出来，如图 3.6 所示，

```python
a=list(range(0,64))
for n in range(0,4):
    for m in range(0,4):
        a[16*n+2*m]    =  63-4*n-m
        a[16*n+1+2*m]  =  47-4*n-m
        a[16*n+8+2*m]  =  15-4*n-m
        a[16*n+9+2*m]  =  31-4*n-m

# x='0000000100100011010001010110011110001001101010111100110111101111' # 0123_4567_89AB_CDEF 通过测试
#x='0110100110010110100101100110100110010110011010010110100110010110' # 6996966996696996 通过测试
# ------------------------------------------------------------------------------------------
# X57 初始值0123 不锁定引脚 通过测试，结果为 AFA0CFCFC0C0AFA0 与rbt文件获得的结果一致
# 需要先将0123经过X57_lock_to_nolock 转换一下，得到下面的初始值（下面的x），再经过本程序转换，可以得到AFA0CFCFC0C0AFA0
# Vivado 工程名：用everything搜索 X57_nolock
x='0000000010101010000110110001101101010101111111110001101100011011'

y=list(range(0,64))


for i in range(0,64):
    if  x[63-i] == '1' :
        y[63-a[i]] = 1
    else :
        y[63-a[i]] = 0

print (y)
```

图 3.6 对应 python 代码

后来又发现，SLICEL 和 SLICEM 的 LUT，从 verilog 代码初始值到 rbt 文件值的映射关系还是不一样的，上面是 SLICEL 的映射关系，SLICEM（位置是 SLICE_X56Y53）的测试数据、映射关系、python 代码分别如表 3.5、表 3.6、图 3.7 所示,表 3.6 中红色字体，代表这几组数据是推测得来的（后来证实推测正确）。

表 3.5 测试原始数据-1

LUT 初始值	rbt 数据
0x X_X_X_0001	0x X_X_8000_X
0x X_X_X_0002	0x X_X_X_8000
0x X_X_X_0004	0x X_X_4000_X
0x X_X_X_0008	0x X_X_X_4000
0x X_X_X_0010	0x X_X_2000_X
0x X_X_X_0020	0x X_X_X_2000
0x X_X_X_0040	0x X_X_1000_X
0x X_X_X_0080	0x X_X_X_1000
0x X_X_X_0100	0x 8000_X_X_X
0x X_X_X_0200	0x X_8000_X_X
0x 0100_X_X_X	0x 0008_X_X_X
0x 0200_X_X_X	0x X_0008_X_X
0x X_0100_X_X	0x 0080_X_X_X
0x X_0200_X_X	0x X_0080_X_X

0x X_X_0100_X	0x 0800_X_X_X
0x X_X_0200_X	0x X_0800_X_X
0x 0001_X_X_X	0x X_X_0008_X
0x 0002_X_X_X	0x X_X_X_0008
0x X_0001_X_X	0x X_X_0080_X
0x X_0002_X_X	0x X_X_X_0080
0x X_X_0001_X	0x X_X_0800_X
0x X_X_0002_X	0x X_X_X_0800

表 3.6 映射关系-1

LUT 初始值	rbt 数据
1	32
2	16
3	31
4	15
5	30
6	14
7	29
8	13
9	64
10	48
11	63
12	47
13	62
14	46
15	61
16	45
57	52
58	36
41	56
42	40
25	60
26	44
49	20
50	4
33	24

34	8
17	28
18	12

```python
a=list(range(0,64))
for n in range(0,4):
    for m in range(0,4):
        a[16*n+2*m]    =  31-4*n-m
        a[16*n+1+2*m]  =  15-4*n-m
        a[16*n+8+2*m]  =  63-4*n-m
        a[16*n+9+2*m]  =  47-4*n-m

# x='0000000100100011010001010110011111000100110101011110011011101111' # 0123_4567_89AB_CDEF 通过测试
x='0110100110010110100101100110100110010110011010010110100110010110' # 6996966996696996 通过测试
y=list(range(0,64))


for i in range(0,64):
    if  x[63-i] == '1' :
        y[63-a[i]] = 1
    else :
        y[63-a[i]] = 0

print (y)
```

图 3.7 对应 python 代码

上述关系推导出来之后，还是得不到表 3.2 的结果，理论上，初始化值映射后，结果应如表 3.7 所示，经过多次探索尝试后，发现原因：Verilog 代码中例化的 LUT 与 FPGA 上 LUT 的管脚的映射关系是不同的，通过一个例子说明，如图 3.8 所示，Verilog 代码例化的 LUT，初始化值“0x0123456789ABCDEF”，对应的 6 位地址是 I5-I0，但实际 FPGA 内部的 LUT 如图 3.9 所示，对应的地址是 A6-A1。

表 3.7 rbt 文件对应行数和值-3

rbt 文件行数	rbt 文件值
510521	0x d8d80000
510622	0x ffaa0000
510723	0x 55000000
510824	0x d8d80000

```
(*BEL="D6LUT",LOC="SLICE_X57Y53"*)     LUT6 #(
      .INIT(64'h0123456789ABCDEF)  // Specify LUT Contents
  ) LUT6_inst (
      .O(dout),   // LUT general output
      .I0(din[0]), // LUT input
      .I1(din[1]), // LUT input
      .I2(din[2]), // LUT input
      .I3(din[3]), // LUT input
      .I4(din[4]), // LUT input
      .I5(din[5])  // LUT input
  );
```

图 3.8 Verilog 代码例化的 LUT

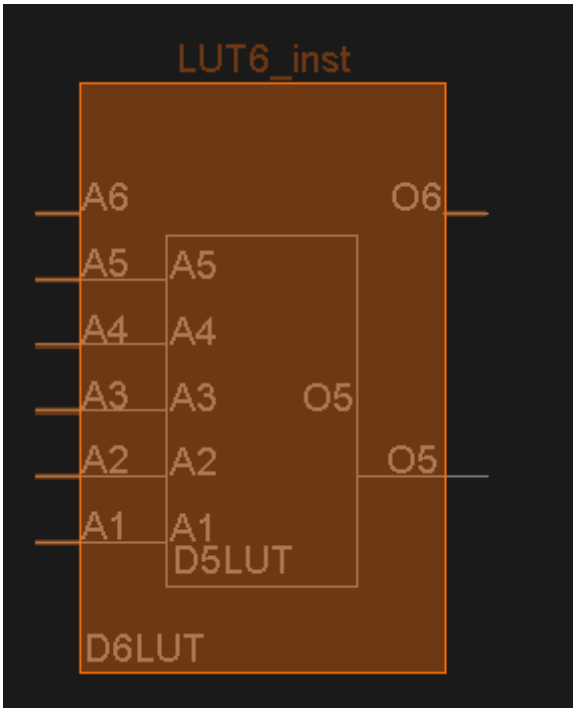

图 3.9 FPGA 内实际的 LUT

而 Verilog 代码例化的 LUT 与实际的 LUT，地址引脚的对应关系如图 3.10 所示，二者又存在一种映射关系，发生这种情况的原因是，综合工具 Vivado 会综合考虑各种情况，选择最优的布线，为了达到这种最优，允许一个 LUT 的地址线进行某种翻转。

Cell Properties

LUT6_inst

Name	Dir	BEL Pin	Net	Net Delay (ps)	Routed
I0	Input	A2	din_IBUF[0]		✓
I1	Input	A3	din_IBUF[1]		✓
I2	Input	A6	din_IBUF[2]		✓
I3	Input	A1	din_IBUF[3]		✓
I4	Input	A4	din_IBUF[4]		✓
I5	Input	A5	din_IBUF[5]		✓
O	Output	O6	dout_OBUF		✓

图 3.10 引脚对应关系

为了支持上述的翻转，作者写了一个简单的 python 脚本（目录：代码\lut_pin_map.py），如图 3.11 所示，假如数据 m 是 Verilog 代码中的 LUT 初始化值，则 n 是映射到实际 FPGA 中的 LUT 初始化值。

```
b=list(range(0,64))
for a in range(0,64):
    a0 = (a   )&1
    a1 = (a>>1)&1
    a2 = (a>>2)&1
    a3 = (a>>3)&1
    a4 = (a>>4)&1
    a5 = (a>>5)&1
    b[a] = a0*2+a1*4+a2*32+a3+a4*8+a5*16

print (b)

m='0000000100100011010001010110011111000100110101011110011011110111111'
n=list(range(0,64))
for i in range(0,64):
    if m[63-i] == '1':
        n[63-b[i]] = 1
    else:
        n[63-b[i]] = 0

print (n)
```

图 3.11 LUT 引脚映射代码

3.4 总结

从 Verilog 代码中的 LUT 初始值到 rbt 文件中的 LUT 初始值，遵循的流程如

图 3.12 所示，经过两次转换后，Verilog 代码中的 LUT 初始值可以转换为 rbt 文件中的 LUT 初始值。

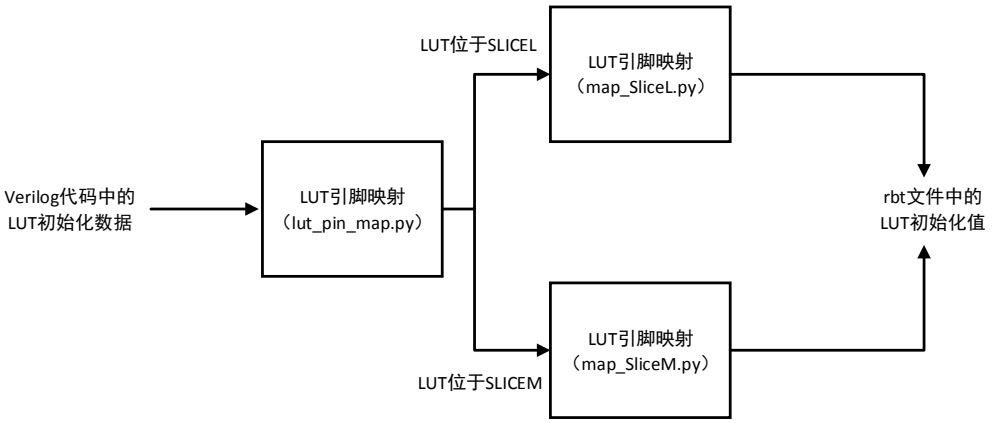

图 3.12 LUT 初始化值转换流程

4、 LUT 信息提取

我们如何知道待配置的 LUT，其位置是多少？代码中的 LUT 与实际的 LUT 引脚映射关系是怎样的？当然可以通过 Verilog 代码中的位置约束（如 BEL="D6LUT",LOC="SLICE_X57Y53"）和 xdc 文件中的引脚锁定约束（如 set_property LOCK_PINS {I0:A1 I1:A2 I2:A3 I3:A4 I4:A5 I5:A6} [get_cells u_xor_lut/LUT6_inst_D_right]），在生成 rbt 文件之前，人为地分配好一切，但是，这样做可能会导致设计性能下降，因为 Vivado 的综合实现算法，是 Xilinx 公司多年研究的成果，可以认为其是最优的。因此，作者不建议通过添加位置约束和 LUT 引脚锁定约束的方式，确定待配置 LUT 的信息，而是生成 rbt 文件之后，通过 Vivado 软件和 TCL 命令，将 LUT 的位置信息和引脚约束信息提取出来。

```
(* DONT_TOUCH= "TRUE" *) (*BEL="D6LUT",LOC="SLICE_X57Y53"*)     LUT6 #(
    .INIT(64'h0123_4567_89AB_CDEF)  // Specify LUT Contents // 0212_2232_4252_6272
  ) LUT6_inst_D_right (
    .O(dout_pre),   // LUT general output
    .I0(din_ff[0]), // LUT input
    .I1(din_ff[1]), // LUT input
    .I2(din_ff[2]), // LUT input
    .I3(din_ff[3]), // LUT input
    .I4(din_ff[4]), // LUT input
    .I5(din_ff[5])  // LUT input
  );
```

图 4.1 Verilog 代码中的 LUT 例化

以图 4.1 中例化的 LUT 为例，生成 rbt 文件后，

LUT 位置信息提取命令是：

get_tiles -of_objects [get_property SITE [get_cells -hierarchical "*LUT6_inst_D_right*"]]

得到的结果是：CLBLM_L_X34Y53

上述结果给出的是 LUT 所在 CLB 的位置，其中 X 的值（34）就是图 2.12 中的 column address，可以通过 Y 的地址，换算图 2.12 中的 top/bottom address 和 row address。

LUT 引脚映射关系提取命令是：

get_bel_pins -of_objects [get_pins -hierarchical "*LUT6_inst_D_right*"]

得到的结果是：

SLICE_X57Y53/D6LUT/O6　　SLICE_X57Y53/D6LUT/A1
SLICE_X57Y53/D6LUT/A2　　SLICE_X57Y53/D6LUT/A3
SLICE_X57Y53/D6LUT/A4　　SLICE_X57Y53/D6LUT/A5
SLICE_X57Y53/D6LUT/A6

上述结果与图 3.10 中的映射关系是一致的。

5、HWICAP 操作方法

前面的工作解决了配置帧寻址与配置帧内容怎么写的问题，下面要开始介绍 ICAP 原语和 HWICAP IP 核。ICAP 原语是 Xilinx 公司开放给用户的 FPGA 内部配置接口，如图 5.1[9]所示，使用 ICAP 原语，需要写一个控制器，处理接口时序的问题，Xilinx 提供了这个控制器，就是 HWICAP IP 核[10]，如图 5.2 所示，这个 IP 核对外呈现 AXI4-Lite Slave 接口，使用者可以通过该接口，间接地操纵 ICAP 原语。

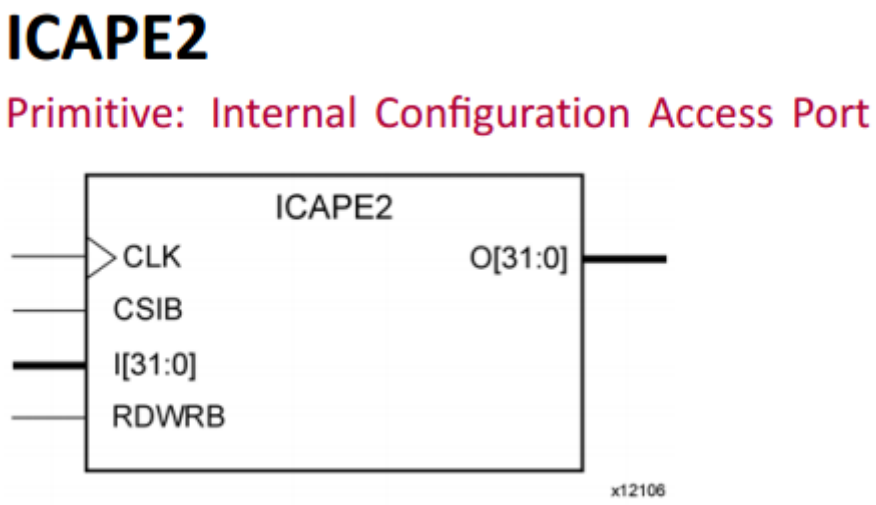

图 5.1 ICAP 原语

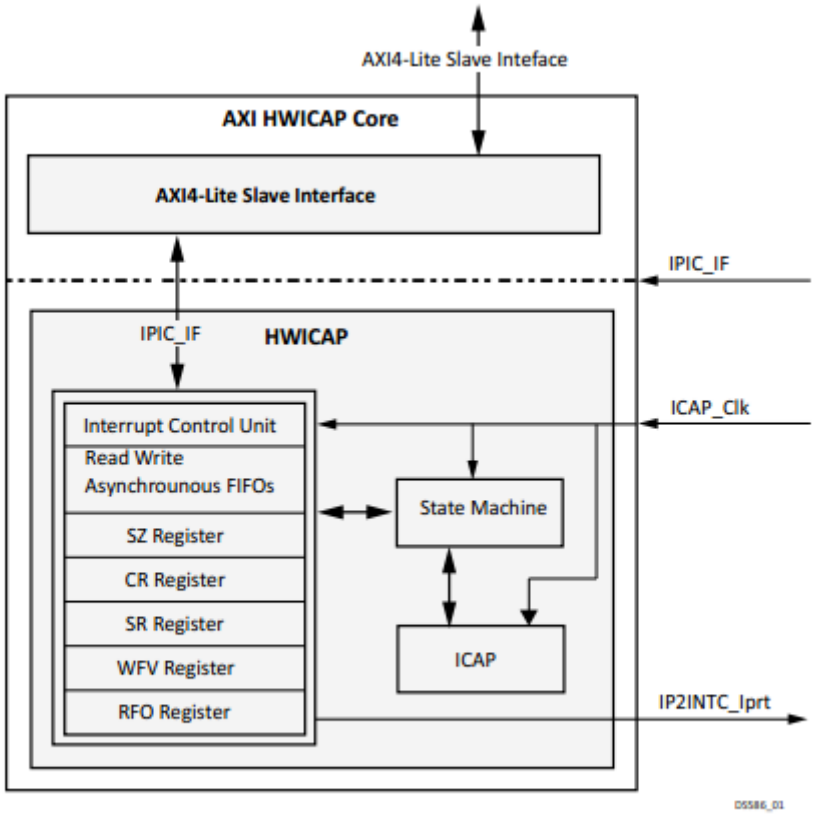

Figure 1-1: **Top Level Block Diagram for the AXI HWICAP Core**

图 5.2 HWICAP IP 核

5.1 ICAP 操作方法

ICAP 操作方法主要参考 UG953[9]与 UG470[8]，尤其是 UG470，第五节 Configuration Details 介绍了大量有用信息，需要重点关注的有：

（1）Bitstream Composition 小节，介绍了 bit 文件/rbt 文件的组成形式，可以从中看出一些配置流程相关的信息；

（2）Configuration Memory Frames 小节，与 2.2 节介绍的帧格式是对应的；

（3）Configuration Packets 小节，介绍了 Configuration Packets 的细节，上面说的 Configuration Memory Frames 主要是用于传输配置内容（如 LUT 的初始化值），而 Configuration Packets 主要用于传输一些配置相关的指令，如读写配置寄存器。Configuration Packets 分为两种，即 Type 1 packet 和 Type 2 packet，Type 2 报文必须跟在 Type 1 报文后面，个人认为 Type 2 报文的作用是对 Type 1 报文进行补充，因为 Type 1 报文 word count 字段仅有 11bit，只支持 2048 个 word 传

输，而 Type 2 报文 word count 字段有 27bit，支持 134217728 个 word，具体内容请参考 UG470 Table 5-20、Table 5-21、Table 5-22；

（4）Configuration Registers 小节，配置相关寄存器，如图 5.3 所示，详细内容请参考 UG470 Configuration Registers 小节。

Table 5-23: **Type 1 Packet Registers**

Name	Read/Write	Address	Description
CRC	Read/Write	00000	CRC Register
FAR	Read/Write	00001	Frame Address Register
FDRI	Write	00010	Frame Data Register, Input Register (write configuration data)
FDRO	Read	00011	Frame Data Register, Output Register (read configuration data)
CMD	Read/Write	00100	Command Register
CTL0	Read/Write	00101	Control Register 0
MASK	Read/Write	00110	Masking Register for CTL0 and CTL1
STAT	Read	00111	Status Register
LOUT	Write	01000	Legacy Output Register for daisy chain
COR0	Read/Write	01001	Configuration Option Register 0
MFWR	Write	01010	Multiple Frame Write Register
CBC	Write	01011	Initial CBC Value Register
IDCODE	Read/Write	01100	Device ID Register
AXSS	Read/Write	01101	User Access Register
COR1	Read/Write	01110	Configuration Option Register 1
WBSTAR	Read/Write	10000	Warm Boot Start Address Register
TIMER	Read/Write	10001	Watchdog Timer Register
BOOTSTS	Read	10110	Boot History Status Register
CTL1	Read/Write	11000	Control Register 1
BSPI	Read/Write	11111	BPI/SPI Configuration Options Register

图 5.3 配置寄存器

5.2 HWICAP 操作方法

从图 5.2 可知，HWICAP IP 核仅支持 AXI4-Lite 接口，也就是我们操作 HWICAP IP 核的时候，其实是在配置寄存器，那么该 IP 核有哪些寄存器呢？如图 5.4[10]所示。

Table 2-4: **Register Address Map**

Address Offset	Register Name	Access	Default Value	Description
1Ch	GIER	Read/Write	0x0	See Global Interrupt Enable Register.
20h	ISR	Read/Write	0x0	See Abort Status Register.
028h	IER	Read/Write	0x0	See IP Interrupt Enable Register.
100h	WF	Write Only	0x0	See Write FIFO Keyhole Register.
104h	RF	Read Only	0x0	See Read FIFO Keyhole Register.
108h	SZ	Write Only	0x0	See Size Register.
10Ch	CR	Read/Write	0x0	See Control Register.
110h	SR	Read Only	0x0	See Status Register.
114h	WFV	Read Only	(1)	See Write FIFO Vacancy Register.
118h	RFO	Read Only	0x0	See Read FIFO Occupancy Register.
11Ch	ASR	Read Only	0x0	See Abort Status Register.

Notes:

1. This value is based on the actual Write FIFO size. For example, if the **Write FIFO depth** is set to 1024 during customization, the actual FIFO depth is 1023 (or 0x3FF). This register reports the *actual* Write FIFO vacancy.

图 5.4 HWICAP IP 核的寄存器

下面重点介绍几个用到的寄存器。我们先分析一下我们将配置帧传输进去，需要哪些操作。

写配置帧：（1）将帧写入 FIFO（2）写完后开始配置。

上面第一步，将帧写入 FIFO，对应的是 Write FIFO Keyhole Register（keyhole 就是钥匙孔，可以说十分形象了），寄存器格式如图 5.5 所示；第二步，开始配置，对应的寄存器是 Control Register，该寄存器可以控制读，也可以控制写，寄存器格式如图 5.6 所示。

Write FIFO Keyhole Register

The Write FIFO (WF) register shown in Figure 2-4 is a 32-bit keyhole register into a Write FIFO. The bit definitions for the Write FIFO are shown in Table 2-8.

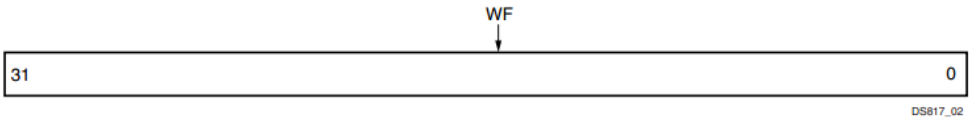

Figure 2-4: **Write FIFO (WF)**

Table 2-8: **Write FIFO Bit Definitions (100h)**

Bits	Name	Access	Reset Value	Description
31–0	WF	Write Only	0	Data written into the FIFO

图 5.5 Write FIFO Keyhole Register

Control Register

The Control Register (CR) shown in Figure 2-7 is a 32-bit read/write register that determines the direction of the data transfer. It controls whether a configuration or readback takes place. Writing to this register initiates the transfer. The bit definitions for the register are shown in Table 2-11.

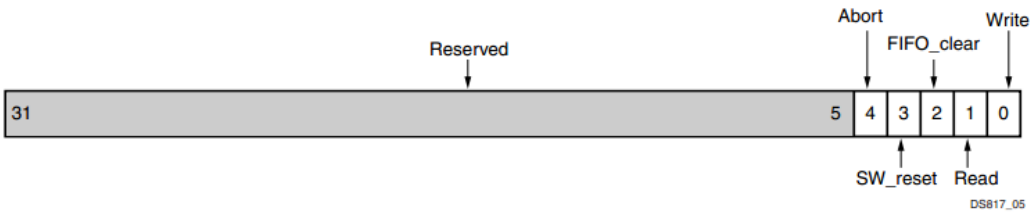

Figure 2-7: **Control Register (CR)**

Table 2-11: **Control Register Bit Definitions (10Ch)**

Bits	Name	Access	Reset Value	Description
31–5	Reserved	N/A	0	Reserved bits
4	Abort	Read/ Write	0	1 = Aborts the read or write of the ICAPEn and clears the FIFOs

Table 2-11: **Control Register Bit Definitions (10Ch) *(Cont'd)***

Bits	Name	Access	Reset Value	Description
3	SW_reset	Read/ Write	0	1 = Resets all the registers
2	FIFO_clear	Read/ Write	0	1 = Clears the FIFOs
1	Read	Read/ Write	0	1 = Initiates readback of bitstream in to the Read FIFO
0	Write	Read/ Write	0	1 = Initiates writing of bitstream in to the ICAPEn

图 5.6 Control Register

读配置帧：（1）告诉 IP 核读几帧（2）开始读。

第一步对应的寄存器是 Size Register，定义如图 5.7 所示，第二步对应的还是 Control Register，如图 5.6 所示。

Size Register

The Size (SZ) register shown in Figure 2-6 is a 12-bit write only register that determines the number of 32-bit words to be transferred from the ICAPEn to the read FIFO. (This means how many 32-bit data beats are expected.) The bit definitions for the register are shown in Table 2-10.

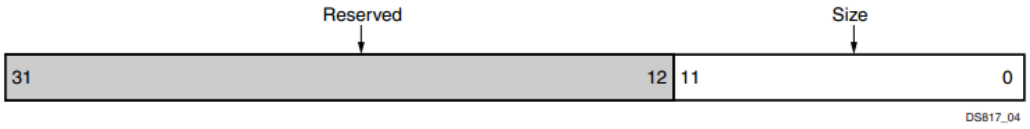

Figure 2-6: **Size Register (SZ)**

Table 2-10: **Size Register Bit Definitions (108h)**

Bits	Name	Access	Reset Value	Description
31–12	Reserved	N/A	0	Reserved bits
11–0	Size	Write Only	0	Number of words to be transferred from the ICAPEn to the FIFO

图 5.7 Size Register

其他的寄存器也有各自的用处，建议读者参考 PG134，进一步了解。

6、验证系统

前面做的大量调研与学习，目的是为了能够做到对单个 LUT 的在线读和写，如何证明我们前面做的工作是正确的呢？答案是建立一个验证系统，在实际的 FPGA 系统中，将 FPGA 内部某个特定的 LUT 内容在线读出来，然后在线将某些内容写进去，如果可以成功做到这一点，可以说研究工作基本成功了。下面我们来介绍验证系统搭建的详细流程。

6.1 开发板及连接拓扑

开发板连接拓扑如图 6.1 所示，FPGA 开发板型号为黑金 AX7103[11]，如图 6.2 所示，读者也可以用其他 Xilinx FPGA 板卡搭建验证系统（前文的知识适用于所有的 7 系列 FPGA），但是 Zynq 系列暂时还无法调通，应该是作者忽略了些什么；PC 与 FPGA 通过 JTAG 连接，PC 上运行 Vivado 软件，在 Vivado TCL Console 中输入 TCL 命令，可以做到 PC 与 FPGA 的交互。

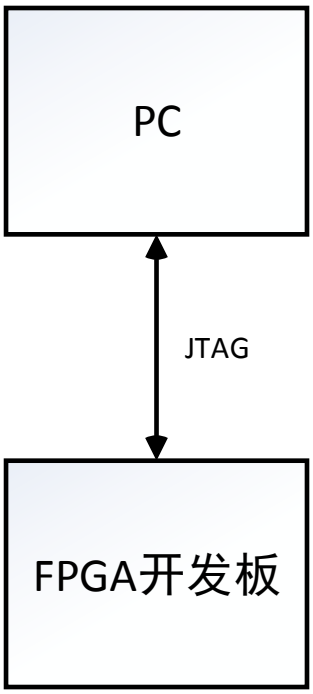

图 6.1 验证环境拓扑

图 6.2 FPGA 开发板

6.2 FPGA 内部模块设计

FPGA 内部模块设计如图 6.3 所示（工程位置：工程\jtag_axi_icap_lut_AX7103_0123），JTAG2AXI IP 核和 HWICAP IP 核负责实现接收 PC 的控制指令，实现对 LUT 内容的读写；目标 LUT 是被读写的对象，其 Verilog 例化如图 6.4 所示。

读 LUT 内容怎么验证？答案：在 PC 的 Vivado 软件 TCL Console 内，写 TCL 命令，读取目标 LUT 内容，读取内容与 rbt 文件的相应位置比对（参见表 3.7，为了简单起见，我们采用 set_property LOCK_PINS {I0:A1 I1:A2 I2:A3 I3:A4 I4:A5 I5:A6} [get_cells u_xor_lut/LUT6_inst_D_right]命令，保证代码例化 LUT 与真实 LUT 管脚一一对应，避免出现图 3.10 的情况）。

写 LUT 内容怎么验证？答案：在 PC 的 Vivado 软件 TCL Console 内，写 TCL 命令，配置目标 LUT 内容，配置完成后，观察 ILA。在配置之前，目标 LUT 初始值是 0x0123456789ABCDEF，计数器产生递减的地址，因此可以用 ILA 观察到变化的数据；配置内容为全 0，配置完成后，ILA 观察到的内容为全 0，证明配置成功。

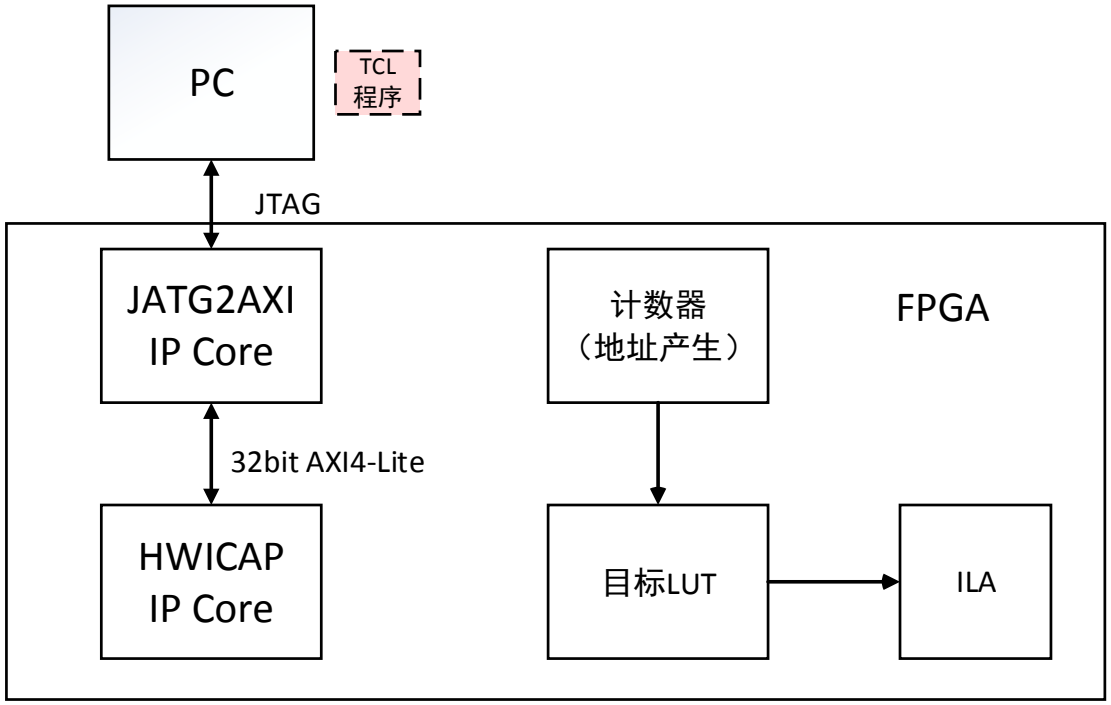

图 6.3 FPGA 内部模块设计

```
(* DONT_TOUCH= "TRUE" *) (*BEL="D6LUT",LOC="SLICE_X57Y53"*)     LUT6 #(
    .INIT(64'h0123_4567_89AB_CDEF)  // Specify LUT Contents // 0212_2232_4252_6272
) LUT6_inst_D_right (
    .O(dout_pre),   // LUT general output
    .I0(din_ff[0]), // LUT input
    .I1(din_ff[1]), // LUT input
    .I2(din_ff[2]), // LUT input
    .I3(din_ff[3]), // LUT input
    .I4(din_ff[4]), // LUT input
    .I5(din_ff[5])  // LUT input
);
```

图 6.4 LUT 例化

6.3 如何编写 TCL 代码

学习 TCL 本身的语法，需要参考 Xilinx 官方文档，UG835[12]、UG894[13]，熟悉 TCL 基本操作，另外，Xilinx 高级 SAE 高亚军老师的 TCL 系列教程[14]写的也很好；

学习第 5 节 HWICAP 和 ICAP 的具体操作方法，可以参考 Xilinx 给的 HWICAP 官方驱动[15]（其中的 example 和 src 部分让作者受益匪浅），作者也给出了读写 LUT 需要的 TCL 代码（位置：代码\读 LUT-TCL 指令 和代码\写 LUT-TCL 指令），作者写的这些代码大量参考了 Xilinx 给的驱动（可以从作者代码的注释中看出来），另外一定要注意，TCL 文件里面的路径是绝对路径，要修改成自己电脑上的路径。

6.4 验证结果

读 LUT 内容：

下载工程“jtag_axi_icap_lut_AX7103_0123”的 bit 程序（记得通过 VIO 复位，复位是低电平复位），运行“read_frame_510521.tcl”文件，该文件读取 LUT 的 1/4 的内容，对应 rbt 文件的 510521 行（见表 3.7），读出的结果如图 6.5 所示（如果读出的结果不能完全显示出来，使用“set_param messaging.defaultLimit 5000”命令，修改 TCL Console 显示的消息数量限制），可以发现是对应的（其他三行的读操作可以运行同目录下的其他 TCL 文件，结果都是正确的）。

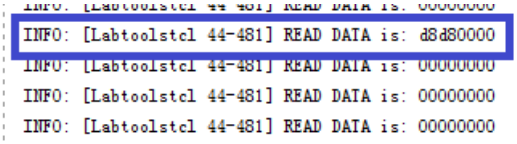

图 6.5 读结果

写 LUT 内容

仍然是同样的 bit 文件，在配置 LUT 内容之前，ILA 可以看到的数据如图 6.6 所示，从上到下的三个信号，分别是 LUT 的输入（地址），LUT 的输出，与 LUT 输出的并行化（LUT 输出不方便观察，并行化之后，方便观察 LUT 初始值）；在 Vivado TCL Console 行“source_write_all.tcl”文件后（记得改地址!!!），LUT 的内容被重置为全 0，如图 6.8 所示，配置成功。

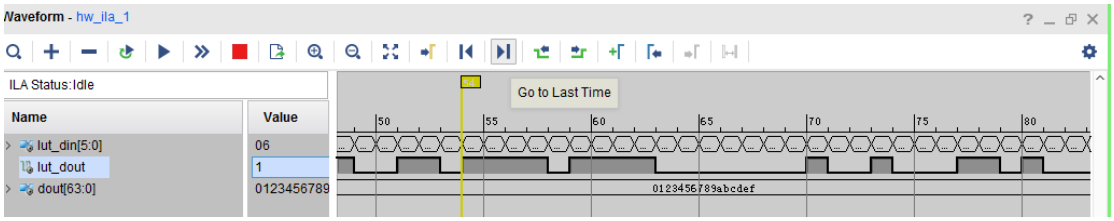

图 6.6 配置前的 LUT 输出

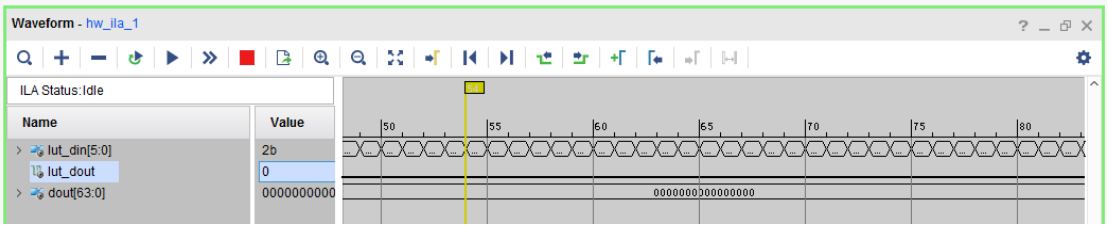

图 6.8 配置全 0 后 LUT 的输出

7、总结与展望

获得的成果：

基本打通了单个 LUT 动态读写的流程，包括 LUT 寻址、LUT 配置帧格式、rbt 文件解析（对照用）、LUT 信息提取、LUT 配置方法（HWICAP 和 ICAP 操作）、演示验证系统搭建。

后期还需要做的工作：

（1）后期需要用嵌入式 CPU 实现对 HWICAP IP 核的操作，而不是 PC，因为 PC+Vivado+TCL 的方式效率太低了；作者对软件开发了解有限，需要一个精通软件的人员，协助作者开发上层软件及应用，比如开发文献[2]中的配表系统；

（2）上述操作方法不适用于 Zynq 系列，原因未知，目前怀疑是某些操作不到位，Zynq 系列的单个 LUT 动态读写，还需要进一步研究。

8 参考文献

[1]. UG474 Xilinx.7 Series FPGAs Configurable Logic Block,

[2]. Reviriego P, Ullah A, Pontarelli S. PR-TCAM: Efficient TCAM Emulation on Xilinx FPGAs Using Partial Reconfiguration[J]. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2019.

[3]. Cardona L A, Ferrer C. AC_ICAP: A flexible high speed ICAP controller[J]. International Journal of Reconfigurable Computing, 2015, 2015: 9.

[4]. DS100 Xilinx. Virtex-5 Family Overview

[5]. UG191 Xilinx. Virtex-5 FPGA Configuration User Guide

[6]. Project X-Ray. https://github.com/SymbiFlow/prjxray

[7]. UG472 Xilinx. 7 Series FPGAs Clocking Resources

[8]. UG470 Xilinx. 7 Series FPGAs Configuration

[9]. UG953 Xilinx. Vivado Design Suite 7 Series FPGA and Zynq-7000 SoC Libraries Guide

[10]. PG134 Xilinx. AXI HWICAP v3.0 LogiCORE IP Product Guide

[11]. AX7103 User Guide. ALINX http://www.alinx.vip:81/ug/AX7103_UG.pdf

[12]. UG835 Xilinx.Vivado Design Suite Tcl Command Reference Guide

[13]. UG894 Xilinx. Vivado Design Suite User Guide: Using Tcl Scripting

[14]. TCL 知识库.https://mp.weixin.qq.com/s/5xHYli4NqCVLkrJXUPXT2w

[15]. HWICAP 驱动 https://github.com/Xilinx/embeddedsw/tree/master/XilinxProcessorIPLib/drivers/hwicap