Single LUT reprogram

1. Background

1.1 basic knowledge

FPGA is an important tool for high-performance computing and networking. Thanks to its high degree of parallelism and user-programmable features, FPGAs are becoming more widely used. The FPGA is composed of CLB (Configurable Logic Block), BRAM (Block RAM), DSP48E1 (Dedicated Digital Processing Unit), programmable routing resources, programmable IO resources, etc. , where CLB is the implementation logic. The basis of the function, the internal structure of the Xilinx 7 series FPGA CLB is shown in Figure 1.1 [1].

As can be seen from Figure 1.1, a CLB consists of two SLICEs, and SLICE can be divided into SLICEL and SLICEM. SLICEL can only implement logic functions. In addition to logic functions, SLICEM can also achieve a bit width of 1 bit. The structure of 64-bit RAM, SLICEL and SLICEM is shown in Figure 1.2 and Figure 1.3 [1]. SLICEL and SLICEM are composed of four 6-input LUTs (Look up table) and three MUXs, a Carry Chain and 8 Flip-Flop , where the LUT can be seen as a ROM with 6bit address and 1bit output (it can also be configured as a 5-inputs LUT, with 2 outputs). By changing the content of LUT, you can get a variety of truth tables. It means a variety of logic functions can be achieved, which is one of the foundations of FPGA programmability.

Comparing Figure 1.2 and Figure 1.3, it can be found that the four LUTs of SLICEL only have 6-bit address input and 1-bit data output. When the bitstream file is configured, the user cannot change the LUT content. It can be regarded as a ROM; In addition to the 6-bit read address input and 1-bit data output, SLICEM has 6-bit write address input (W6: W1 in Figure 1.2) and 1-bit write data enable (WEN in Figure 1.2). 2bit write data input (DI1 and DI2), which is why SLICEM's LUT can be configured as a 1-bit wide and 64-bit deep RAM.

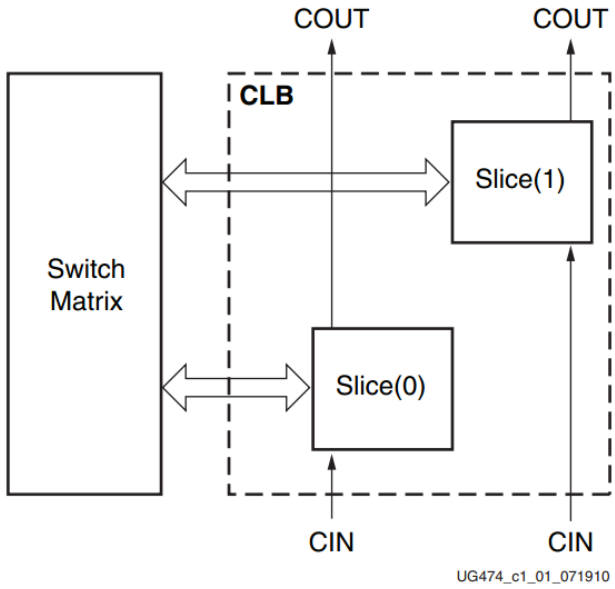

Fig 1.1 CLB architecture

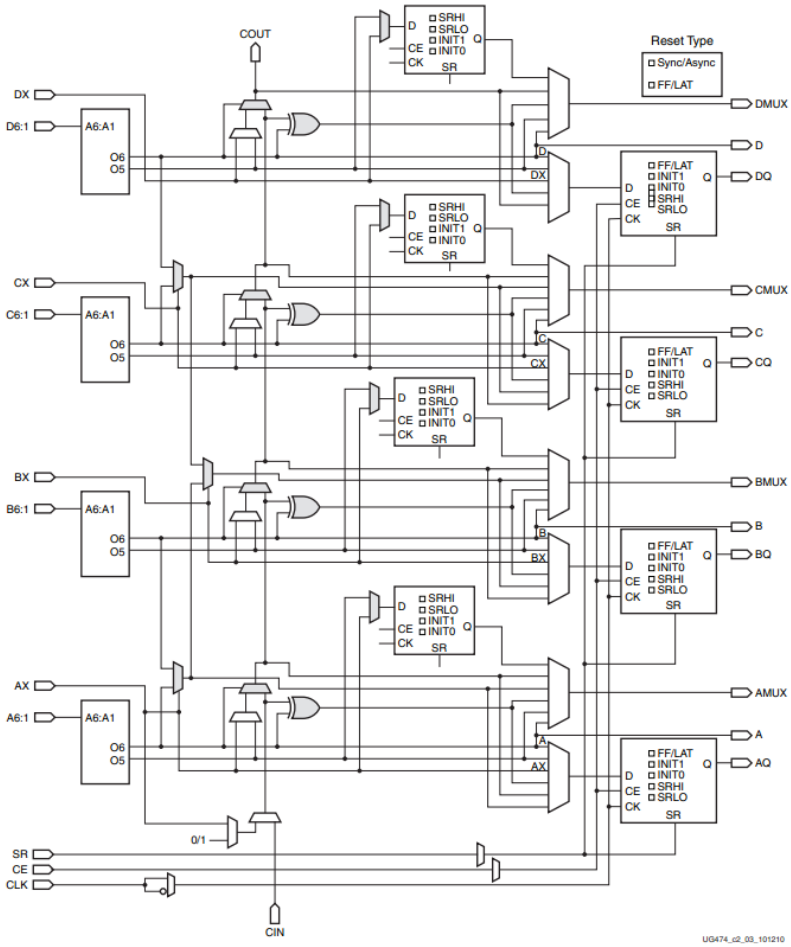

Fig 1.2 SLICEL architecture

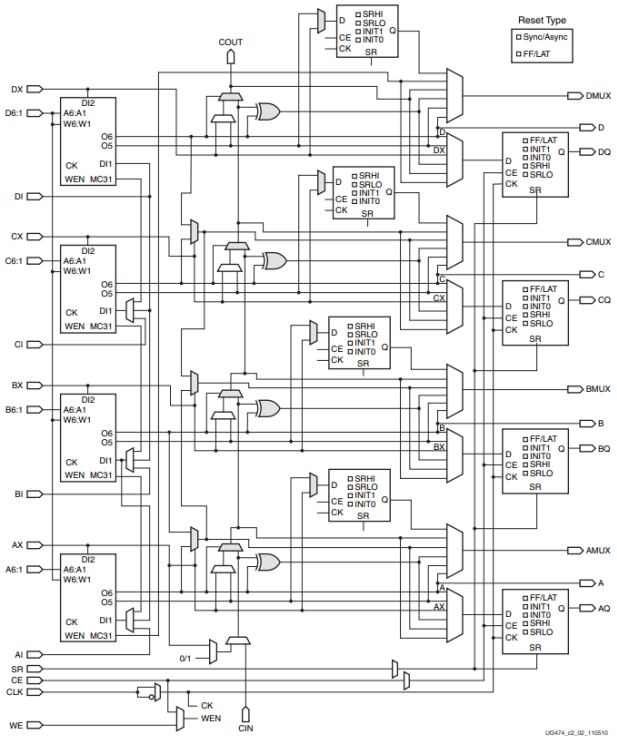

Fig 1.3 SLICEM architecture

1.2 Research object

For the 7 series FPGA, nearly 2/3 of SLICE is SLICEL, and the rest is SLICEM[1]. That is to say, after the 2/3 resources in the FPGA are downloaded, the logic function cannot be changed unless modify the code and generate a new bitstream file. This brings a lot of trouble to the research of reconfigurable computing, LUT-based high-bandwidth lookup algorithms. These studies hope to dynamically modify the contents of the LUT while the device is running, and only the SLUTM LUT has a write data interface. It is the algorithms in these studies that can only use 1/3 of the LUT resources.

We encountered the above problems when studying FPGA-based high-performance reconfigurable CRC algorithm. The key computational module of programmable CRC is implemented by LUT. In order to realize CRC-generated programming, SLLICEM LUT must be used, which results in a bad timing result complex configuration circuit.

Inspired by [2], we found that using the HWICAP IP core provided by Xilinx can realize the dynamic programming of the SLICEL LUT, which can realize the reconfiguration latency of ms and even us [2][3], which is based on FPGA implementation. Ideal for high performance reconfigurable CRC.

Most of the papers focus on how to design high-performance ICAP controllers to replace low-performance HWICAP controllers. So far, there is no way to find out how to use HWICAP to implement content reconfiguration of a single LUT. In this guide, the whole procedure of reprogramming a single LUT is given. Based on an Artix-7 FPGA board, we implements reading and writing the contentof a single LUT. The remainder of this guide is organized as follows.

The second section introduces the addressing and configuration of the LUT. The third section introduces the LUT information extraction, including the LUT location extraction and LUT pin mapping relationship extraction; The fourth section analyzes the rbt file (only for the LUT content part). Only by understanding how the initial value of the LUT is mapped to the rbt file can the HUTCAP be used to implement the correct configuration of the LUT content; The fifth section describes the specific operations of the HWICAP, including HWICAP IP core operation and ICAP operation commands; The sixth section builds a verification system based on the Artix-7 FPGA development board, using the TCL command in the Vivado environment to implement reconfiguration of a single LUT; The seventh section summarizes the above work, and point out the direction of the work later.

2. LUT addressing and configuration

2.1 FPGA architecture

In order to address the LUT, you must know how the LUT is organized in the FPGA, that is, know the structure of the FPGA. It should be noted that the composition of the Xilinx 7 series FPGA has changed greatly compared with its predecessor. The specific structure of the Virtex-5 FPGA and Artix-7 FPGA is introduced here.

Virtex-5 series FPGAs and Artix-7 series FPGAs are based on the ASMBL (Advanced Silicon Modular Block) architecture [1][4] (but Virtex-5 is based on second-generation ASMBL technology and Artix-7 is based on four generations of ASMBL technology). Here is a 7-series FPGA as an example to briefly introduce the ASMBL architecture.

7 series FPGA ASMBL architecture shown in Figure 2.1 [1]. The resources of this architecture are arranged in columns, and the resources in the same column are the same.

Let's look at the internal structure of a real FPGA (the device is XC7A100T, this figure is obtained by Vivado software). As shown in Figure 2.2, the FPGA is divided into 8 regions (8 Clock Regions). The blue irregular part is the module that has been occupied. The enlarged picture of the X0Y3 area is shown in Figure 2.3. The leftmost orange part of the figure is the IO interface. The blue part is CLB. The red part is BRAM resource (RAM36E1), and the green part is DSP resource (DSP48E1). The resource organization manner is completely consistent with the ASMBL architecture mentioned above. We will enlarge the blue square in Figure 2.2 again, as shown in Figure 2.3. It can be seen that each blue square is composed of two SLICEs, which is in line with Figure 1.1 above. In a column of blue squares, the number of blue squares is 50, that is, a CLB contains 50 CLBs (this knowledge will be used later); in a column of red squares, the number of red squares is 10. A column of BRAM contains 10 RAM36E1; in a column of green squares, the number of red squares is 20, that is, a column of DSP contains 20 DSP48E1; it is necessary to pay special attention to that, in the middle part of the FPGA, there are a group of BUFG. As shown in Figure 2.5 (enlarged in Figure 2.2, observing the center of the FPGA), above the BUFG, is the top half of the FPGA (top), below the FPGA is the bottom part (bottom) [6] [7]. This knowledge will be used later. Someone may think that the structure of Figure 2.2 is so symmetrical. Is the upper part and the lower part not clear at a glance? But we need to know that not all FPGAs are four rows and two columns of eight Clock Regions, as shown in Figure 2.6 (chip model XC7Z020), three rows of two columns of six Clock Region. Then only by observing BUFG Position (the location of the white box in Figure 2.6), it can be determined which lines are the top part, and which lines are the bottom part.

Let's take a look at the internal structure of the Virtex-5 FPGA, as shown in Figure 2.7 [3]. It can be seen that the structure of the Virtex-5 FPGA is similar to that of the 7 series FPGAs, which are modular and arranged in columns; Focus on the difference between CLB. Comparing Figure 2.3 and Figure 2.7, it can be found that the number of CLBs in a column of a 7-series FPGA is 50, and the number of CLBs in a column of a Virtex-5 FPGA is 20, which is the difference. It will be reflected in the later introduction of LUT addressing.

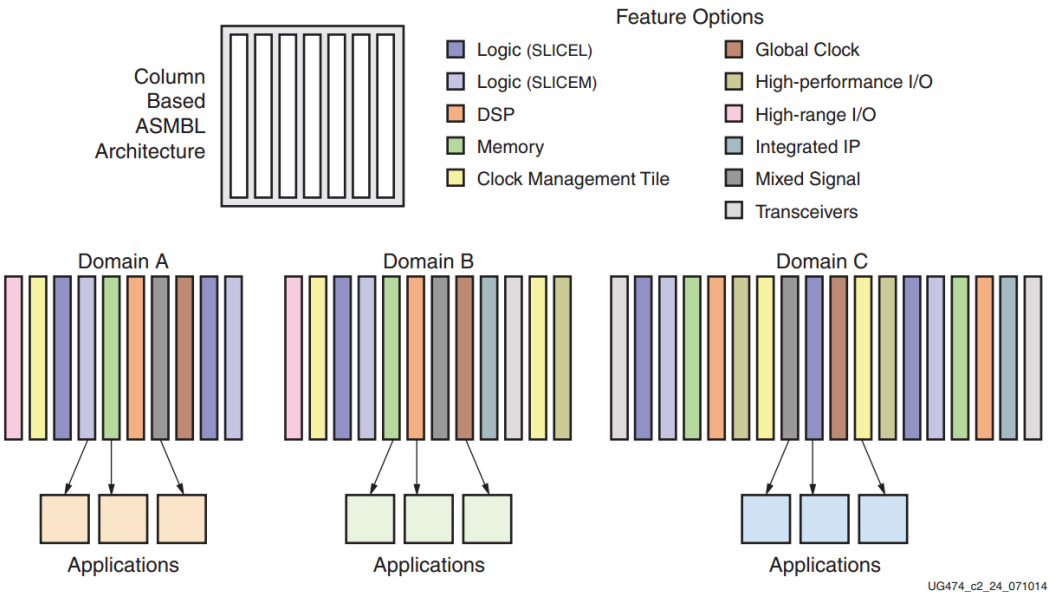
Column Based ASMBL Architecture
Feature Options
Logic (SLICEL)
Logic (SLICEM)
DSP
Memory
Clock Management Tile
Global Clock
High-performance I/O
High-range I/O
Integrated IP
Mixed Signal
Transceivers
Domain A
Domain B
Domain C
Applications
Applications
Applications
UG474_c2_24_071014

Fig. 2.1 ASMBL

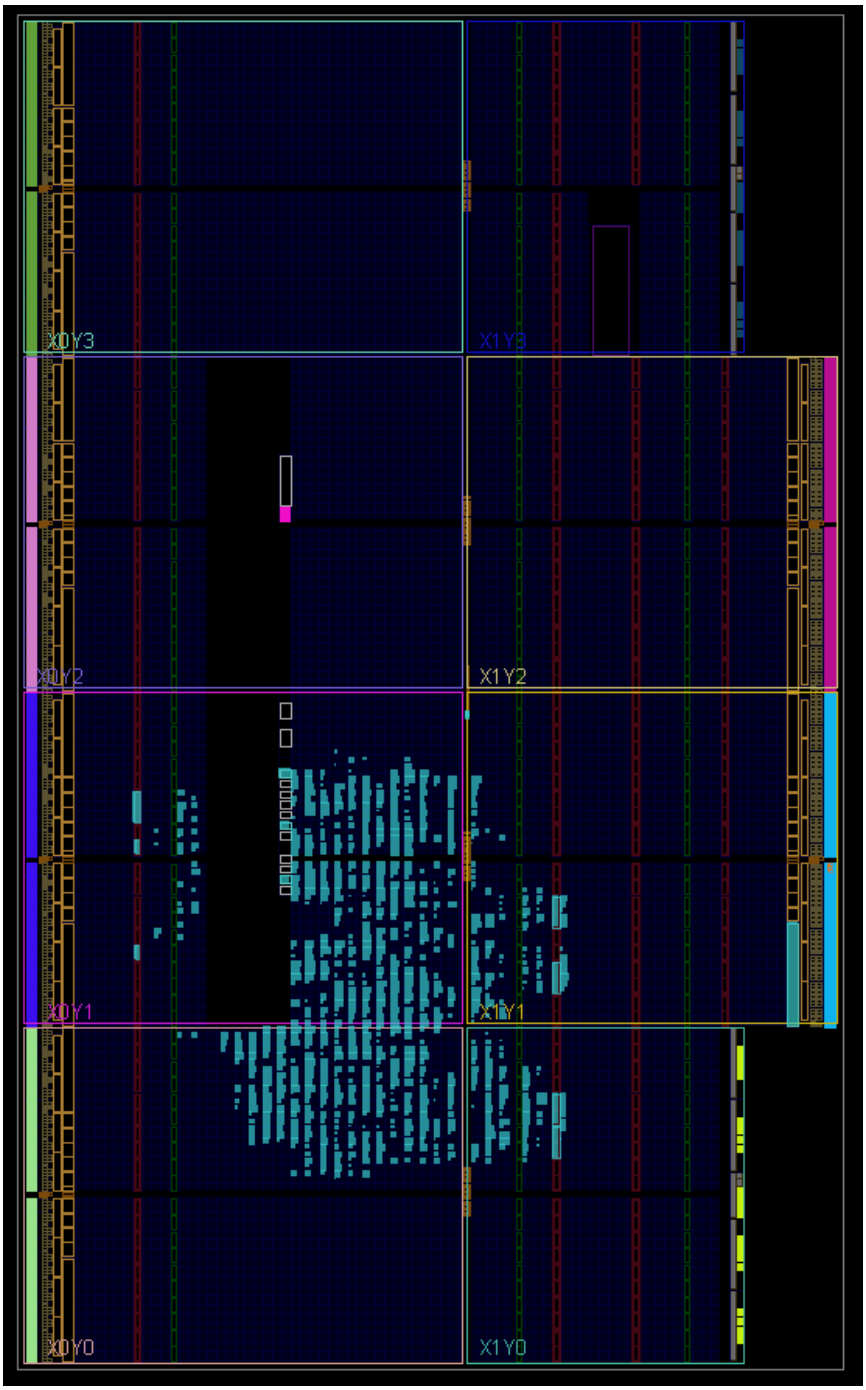

Fig. 2.2 XC7A100T architecture

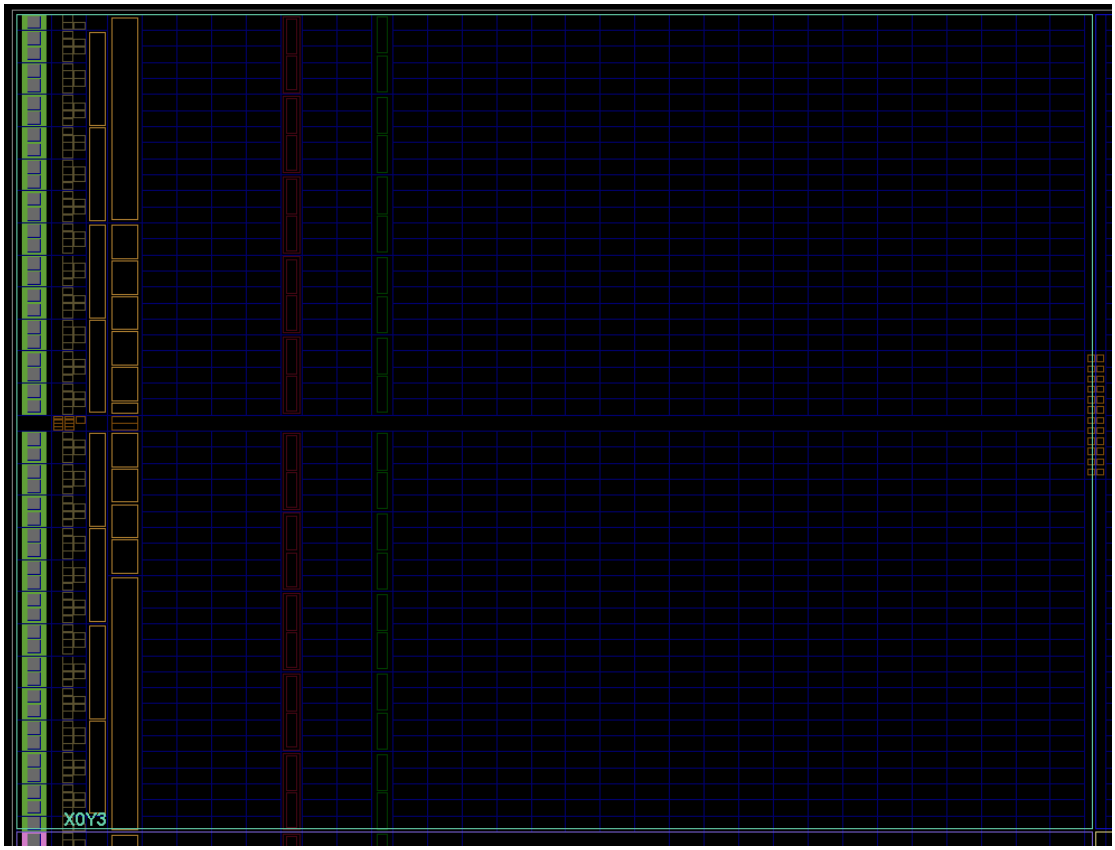

Fig 2.3 X0Y3 architecture

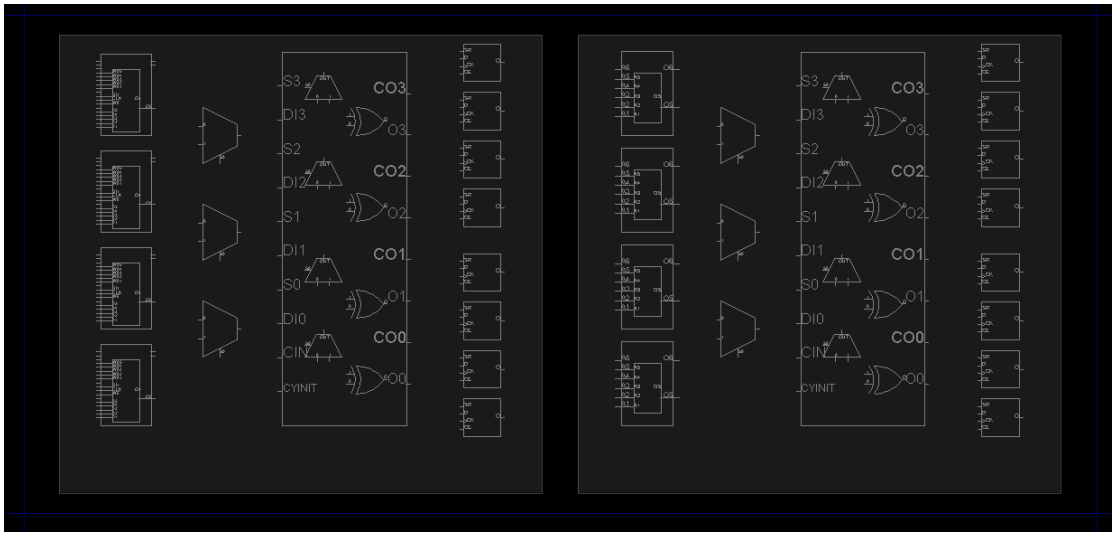

Fig 2.4 CLB architecture

Fig 2.5 BUFG

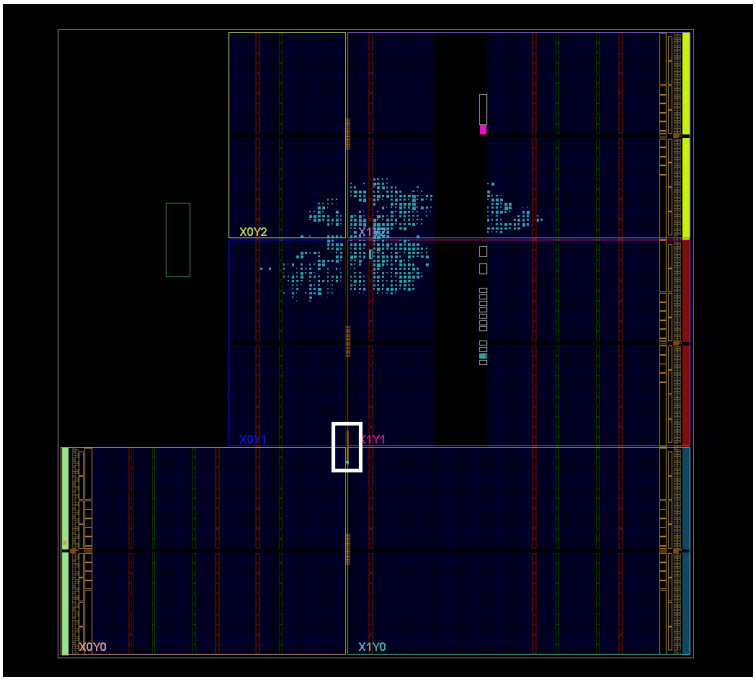

Fig 2.6 XC7Z020 architecture

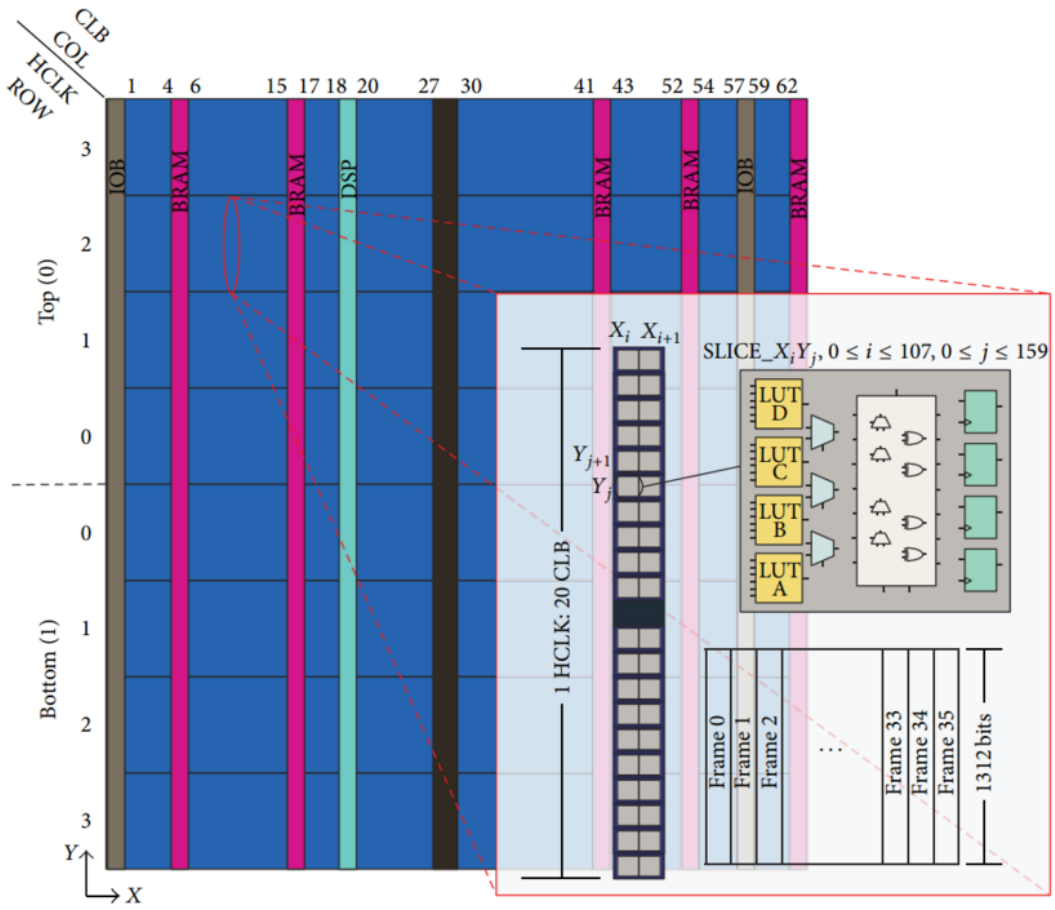

Fig 2.7 XC5VLX110T

2.2 FPGA configuration frame format and addressing format

The minimum unit of FPGA configuration data is frame. Let's take Virtex-5 as an example to introduce the format and addressing mode of configuration frame. Then introduce the difference between 7 series FPGA and Virtex-5 in configuration frame format and addressing mode.

2.2.1 Virtex-5 FPGA configuration frame format and addressing format

The Virtex-5 configuration frame format is shown in Figure 2.8. There are 36 frames in the picture. Each frame contains 41 words. Each word consists of 32 bits. The “X odd” part in the figure represents a SLICE with an odd number in the CLB. "X even" represents the SLICE numbered evenly. "X odd" represents the SLICE on the right side of Figure 2.4, and "X odd" represents the SLICE on the left side of Figure 2.4;

Frame26-Frame29 and Frame32-Frame35 are enough to configure 8 LUTs in a CLB. Why 36 frames are needed to configure a CLB? Because in addition to the need to configure the LUT, you need to configure other parts such as DFF, switchbox, etc. We only focus on how the LUT content is reconfigured. In addition, 1 frame is not enough to confiure a single LUT. Because 1 frame can only be configured with 2 bytes of 1 LUT (the initial value of 6 input LUT is 64 bits, which is 8 bytes). It takes 4 frames to configure a LUT. However, A frame also involves the configuration information of 20 LUTs, that is, one frame will configure the LUT in a column of SLICE (as mentioned earlier, the number of CLBs in the Virtex-5 column of CLB is 20).

In Figure 2.8, one configuration frame contains 41 words, and one column of SLICE contains 20 SLICEs. Two words can be configured with one SLICE 4 LUTs. What is the use of the one more word? The answer can be found in Figure 2.9 [5]. The first 20 words and the last 20 words of a configuration frame are used to configure the LUT, while the middle LUT is used for other functions. For more information, refer to the Xilinx documentation: UG191 [5].

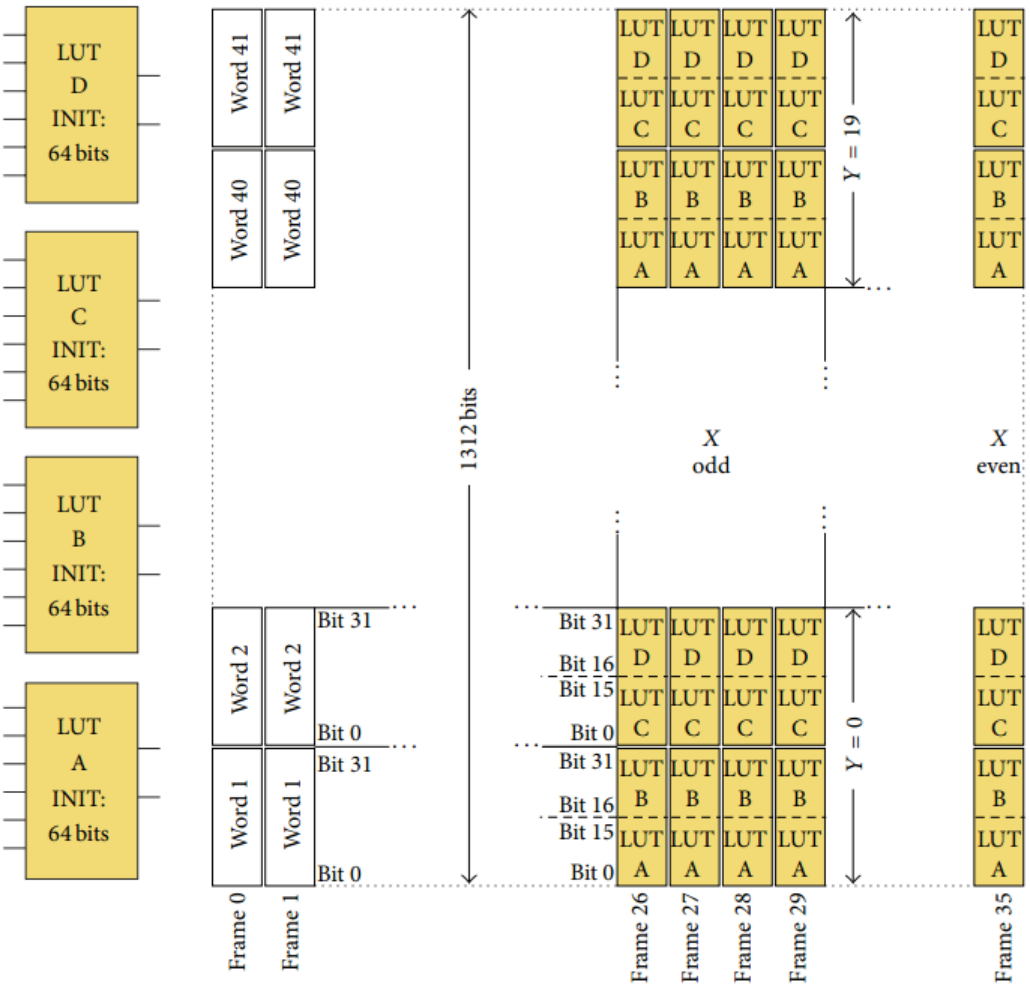

FIGURE 4: Frame bits for LUT configuration.

Fig 2.8 Virtex-5 configuration frame format

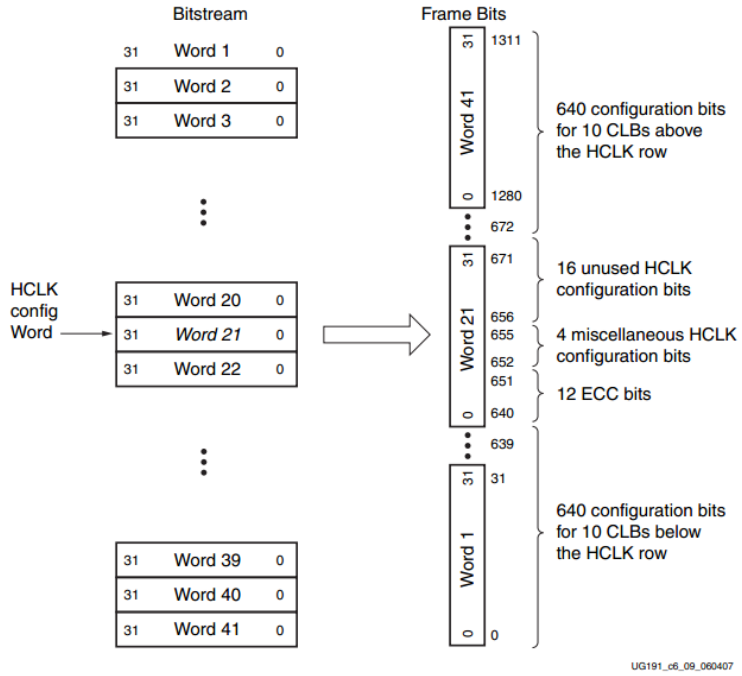

Figure 6-9: **Configuration Words in the Bitstream and Configuration Bits in a Frame**

Fig 2.9 detail format of a configuration format

The addressing format of the Virtex-5 FPGA configuration frame is shown in Figure 2.10 [5]. The address of the Virtex-5 series FPGA configuration frame is composed of 24 bits. The bit 23-bit21 is used to indicate the type of the configuration object, for example, the value 000 means to configures the CLB; the bit20 function indicates whether the configured object is in the upper part or the lower part of the FPGA (refer to Figure 2.5 and Figure 2.6); bit19-bit15 is used to select the row, as shown in Figure 2.2, the FPGA has 4 rows. Is the addressing method 0, 1, 2, 3 from top to bottom? Of course not. The addressing method is shown in Figure 2.11. The upper part and the bottom part are separately addressed. The address of the middle row is 1. For example, in Figure 2.2, X0Y2, X1Y2, X0Y3, and X1Y3 are in the upper half, X0Y2 and X1Y2 are in the row of address 0, and X0Y3 and X1Y3 are in the row of address 1; in Figure 2.2, X0Y0, X1Y0, X0Y1, and X1Y1 are in the lower half. The row address of X0Y1 and X1Y1 is 0, the row address of X0Y0 and X1Y0 is 1; The bit 14-bit7 is the column address. It should be noted that the column address don’t care about clock regions. For an example, the first column of X0Y2 and X0Y3 in Figure 2.2 has the same address. In addition, the CLB, BRAM, DSP and other columns are uniformly addressed. The column address can be extracted by the TCL command using the Viavdo software. The bit 6-bit0 is the address of a specific frame. Configuring one CLB (or configuring a column of CLBs) requires 36 frames, but when reconfiguring the LUT, not all frames

need to be re-executed for RMW, only four frames associated with the LUT need to be performed. The RMW operation can be performed on a specific frame of 36 frames. And then, we need to use a minor address.

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.

Fig 2.10 Virtex-5 FPGA configuration frame address format

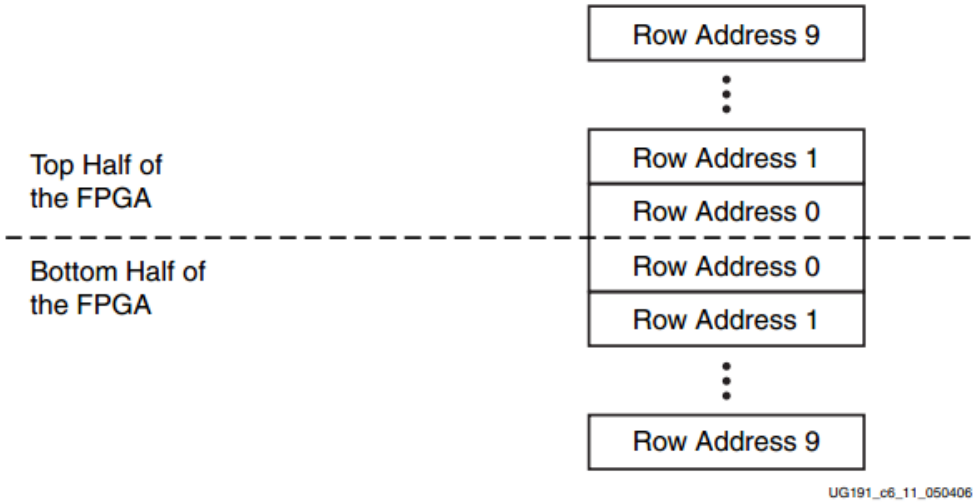

Figure 6-11: **Row Addresses in the FPGA**

Fig 2.11 row address

2.2.2 7 Series FPGA configuration frame format and addressing format

The 7 series FPGA configuration frame format is similar to that of Figure 2.8. The difference lies that : since the 7 series FPGA CLB consists of 50 CLBs (the Virtex-5 is 20), a 7 series FPGA configuration frame contains 101 words;

The addressing format of the configuration frame of the 7 series FPGA is shown in Figure 2.12 [8]. The configuration frame address of the 7 series FPGA consists of 26 bits. After comparing with Figure 2.10, it can be found that the 7 series FPGA column address is 10 bits. The Virtex-5 FPGA column address is 8bit, which is due to the larger FPGA size; the rest of the address is nearly unchanged.

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.

Fig 2.12 7Series FPGA configuration frame format

3. .rbt format

3.1 Why we should know the format of .rbt file

The rbt file is is the configuration file of the FPGA, just as the bitstream file. Bitstream file is in binary format. It is very inconvenient to observe. The rbt file can be regarded as the ASCII version of the bit file. The specific difference is shown in Figure 3.1 [8].

Because of, there is still an unclear point: in Figure 2.8, half of the word can be configured with 1/4 of a LUT, There is an bit order of the half word, and the LUT initial value in the Verilog code also has an order. Is the two bit order are the same? Is there a reflection? In order to explore this, my first thought is to parse the rbt file. Specifically,

create a project, initialize a LUT, and generate a rbt file, then observe the corresponding initialization value in the rbt file. The results show that the initialization value of the LUT is not equal to the corresponding content in the rbt file. There is a specific reflection between the two values, and the LUT in SLICEL and SLICEM has different reflection. The research process is described below.

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.

Fig 3.1 FPGA configuration file format

3.2 preparatory work

The configuration data in the rbt file is in binary format. Observing is very inconvenient, so the first preparation is to convert the binary rbt file to hexadecimal. This work is implemented by a python script(rbt_translation.py. The following is a description of what to note with the code.

（1）The first 7 lines of the rbt file are the introduction information of the rbt file, as shown in Figure 3.3. Before using the script to convert the radix, these 7 lines need to be manually removed.

（2）The NUM of the first line in Figure 3.2 is 956447. This is the number of lines after the first 7 lines of the rbt file are removed. It should be noted that the FPGA for this project is XC7A100T. The scale of each FPGA is different, and the value of NUM also change accordingly;

（3）In figure 3.2, the file path is an absolute path and needs to be modified according to the actual situation.

```python
NUM = 956447

f_read = open('X:/project/TEST_2019/read_frames/jtag_axi_icap_lut_AX7103/py_code/txt/xor_bin.txt', 'r')
f_write= open('X:/project/TEST_2019/read_frames/jtag_axi_icap_lut_AX7103/py_code/txt/xor_hex.txt', 'w')
a=f_read.readlines()
b=list(range(0,NUM))

for i in range(0,NUM):
    b[i]='{:08x}'.format(int(a[i],2))
    f_write.writelines(b[i])
    f_write.writelines('\n')



f_read.close()
f_write.close()
```

Fig 3.2 python code of bin to hex

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

Fig 3.3 rbt file information

In UG470, there is a part that introduces the composition of bitstream. This part needs to be focused on. For details, see table 5-19 of UG470.

3.3 detailed process

First, establish a project to instantiate a LUT primitive in the project. The position of the LUT must be fixed, which is convenient for control experiments. As shown in Figure 3.4, the LUT position of the LUT is SLICE_X57Y53 As shown, for the sake of simplicity, the initial value of the LUT is set to "0x6996966996696996". Why is this value set? Because the initial value of the LUT is this value, it acts as a six-input-output XOR gate, and the XOR gate is the basic unit in the CRC algorithm.

Generate the rbt file, convert it to hex, the result is shown in table 3.1.

```
(*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
   );
```

Fig 3.4 LUT instantiated-1

Tbale 3.1 rbt line number and value-1

rbt file line number	Value in rbt file
510521	0x69960000
510622	0x 96690000
510723	0x 69960000
510824	0x 96690000

Observing Table 3.1, it can be found that the number of rows corresponding to the value of the rbt file is always 101 words, which is corresponding to the size of the configuration frame described in Section 2.2. Is it possible to say that the bit order of the initial value of the LUT in the Verilog code (Figure 3.4) and the bit order in rbt are consistent? It cannot be because the binary representation of "6" is "0110" and the binary representation of 9 is "1001", it can be found that both values are very symmetrical.

In order to further verify whether there is a certain bit order mapping relationship, a new project is established. The LUT position is unchanged, but the initialization value is set to "0x0123456789ABCDEF", as shown in Figure 3.5, and the results are shown in Table 3.2.

```
(*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
   );
```

Fig 3.5 LUT instantiated -2

Table 3.2 rbt line number and value -2

rbt file line number	Value in rbt file
510521	0x fe760000
510622	0x ba320000
510723	0x 98100000
510824	0x dc540000

It can be found that the rbt file value is completely different from the LUT initialization value, and it can be proved that there is a mapping relationship. How to determine this mapping relationship? The simplest is to create multiple projects, each project initialization value is only 1bit (such as 0x0000000000000001, 0x0000000000000002, 0x0000000000000004, 0x0000000000000008, etc.), observe the location of the corresponding value in the generated rbt file, theoretically need to establish 64 projects to fully determine this Relationships. The test raw data is shown in Table 3.3 ("X" stands for 0x0000)

Table 3.3 test results-1

LUTinitial value	Value in rbt
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

The highest bit (MSB) of the 64-bit initial value is defined as bit-64, and the least significant bit (LSB) is defined as bit-1. The positional mapping relationship between the initial value of the LUT and the rbt file is as shown in Table 3.4.

Table 3.4 mapping relationship -1

LUT	rbt
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

After observing the law, the mapping rule can be expressed in python code (map_SliceL.py), as is shown in Fig 3.6 .

```
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)
```

Fig 3.6 map_SliceL.py

The LUT in SLICEL and SLICEM is different in the relationshio between the initial value of verilog code and the value of rbt file. The above is the mapping relationship of SLICEL. The test data, mapping relationship and python code(map_SliceM.py) of SLICEM (location is SLICE_X56Y53) are shown in Table 3.5. Table 3.6 and Figure 3.7

Table 3.5 Test data-2

LUT initial value	Value in rbt
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

Table 3.6 mapping relationship-2

LUT Initial value	Value in rbt
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

```
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='0000000100100011010001010110011111000100110101011110011010111101111' # 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)
```

Fig 3.7 map_SliceM.py

After the above relationship is derived, the results of Table 3.2 are still not obtained. In theory, after the initial value mapping, the result should be as shown in Table 3.7. After several exploration attempts, the reason is found: The mapping between the LUT instantiated in the verilog code and the pin of the LUT on the FPGA is different.For an example, as is shown in Figure 3.8, the LUT instantiated by the Verilog code, the initialization value "0x0123456789ABCDEF", the corresponding 6-bit address is I5-I0, but the actual FPGA internal The LUT is shown in Figure 3.9, and the corresponding address is A6-A1.

Table 3.7 rbt file -3

Rbt file lines	Value in rbt
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
   );
```

Fig 3.8 LUT instantiated

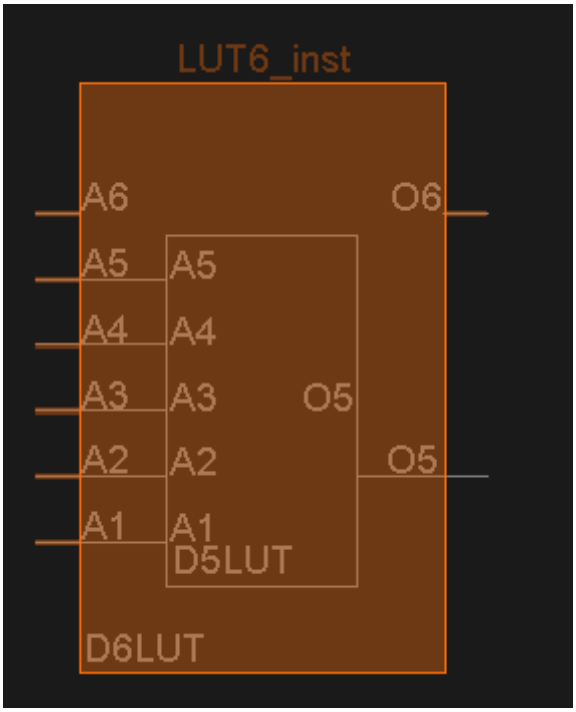

Fig 3.9 actual LUT in FPGA

The corresponding relationship between the LUT in verilog and the actual LUT is shown in Figure 3.10. There is a mapping relationship between them. The reason is that the integrated tool Vivado will consider various situations. Choosing the optimal wiring, in order to achieve this optimality, allows some LUT address lines to be flipped.

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		✓

Fig 3.10 LUT pin mapping

To support the above flip, we wrote a simple python script (lut_pin_map.py), as is shown in Figure 3.11, if the data m is the LUT initialization value in the Verilog code, then n is the LUT initialization value mapped to the actual FPGA.

```
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='0000000100100011010001010110011111000100110101011110011010111101111'
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)
```

Fig 3.11 lut_pin_map.py

3.4 conclusion

From the initial value of the LUT in the Verilog code to the initial value of the LUT in the rbt file, the flow following is shown in Figure 3.12. After two conversions,

the initial value of the LUT in the Verilog code can be converted to the initial value of the LUT in the rbt file.

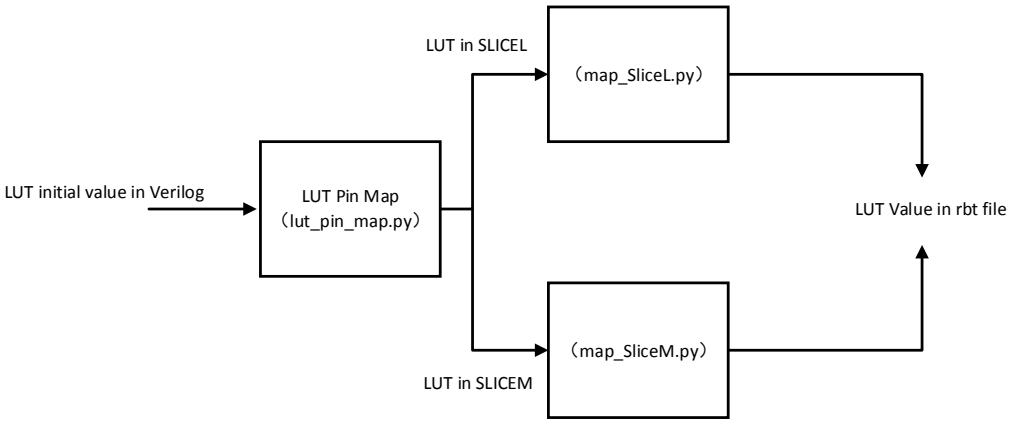

Fig 3.12 LUT Initial value conversion process

4、 LUT Information extraction

How can we know LUT location? What is the mapping relationship between the LUT in the Verilog and the actual LUT? The information of the LUT can be extracted by Vivado software and TCL commands.

```
(* 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 LUT instantiated

Taking the LUT instantiated in Figure 4.1 as an example, after generating the rbt file, the LUT location information extraction command is

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

The result is：CLBLM_L_X34Y53

The above result gives the location of the CLB where the LUT is located, where the value of X (34) is the column address in Figure 2.12, and the top/bottom address

and row address in Figure 2.12 can be converted by the address of Y.

The LUT pin mapping relationship extraction command is

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

The result is：

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

The above results are consistent with the mapping relationship in Figure 3.10.

5 HWICAP operation method

The previous work solved the problem of configuration frame addressing and configuring the contents of the configuration frame. The following begins to introduce the ICAP primitive and the HWICAP IP core. The ICAP primitive is the FPGA internal configuration interface that Xilinx opens to the user. As shown in Figure 5.1 [9], using the ICAP primitive, you need to write a controller to handle the interface timing problem. Xilinx provides this controller, which is HWICAP IP core [10], as is shown in Figure 5.2, this IP core presents the AXI4-Lite Slave interface, through which the user can manipulate the ICAP primitives indirectly.

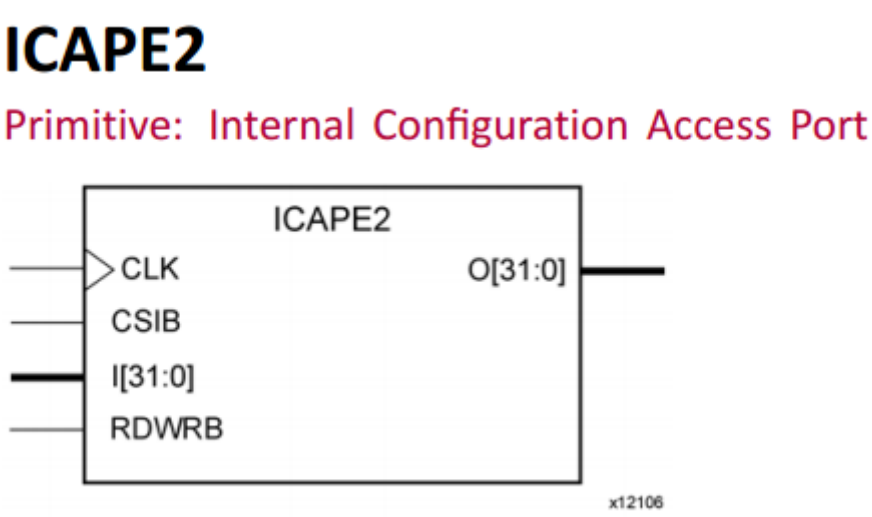

图 5.1 ICAP primitive

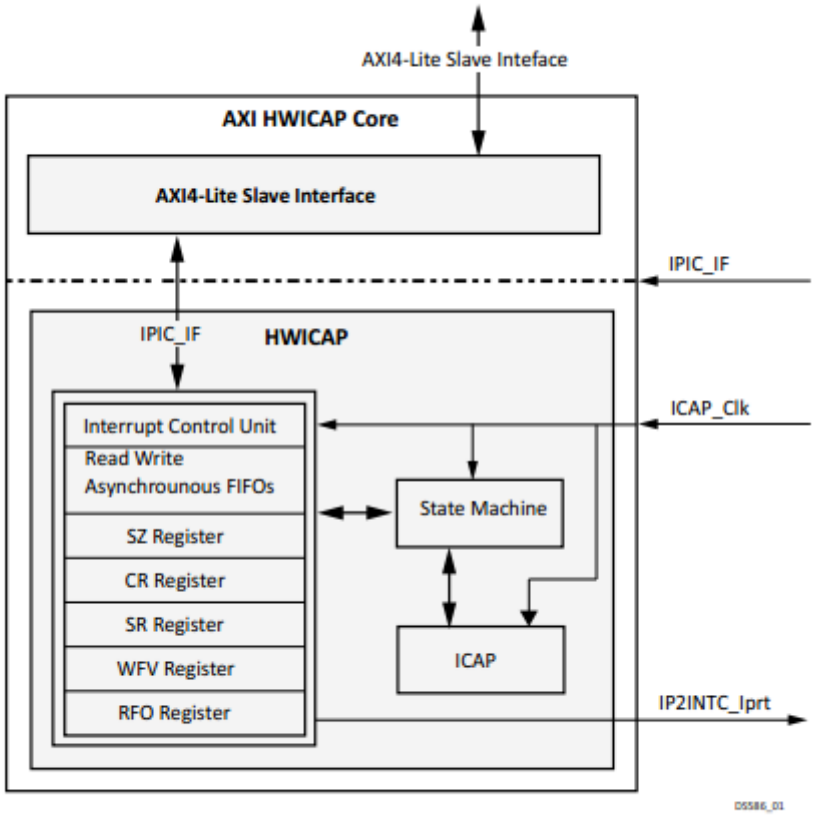

Figure 1-1: **Top Level Block Diagram for the AXI HWICAP Core**

Fig 5.2 HWICAP IP Core

5.1 ICAP operation method

ICAP operation method mainly refers to UG953 [9] and UG470 [8], especially UG470. Section 5 of UG470(Configuration Details) introduces a lot of useful information. We needs to be focused on:

（1）Bitstream Composition Section. This section describes the composition of the bit file/rbt file. You can get information about the configuration process.；

（2）Configuration Memory Frames Section. It correspond to the frame format introduced in Section 2.2；

（3）Configuration Packets Section. The details of Configuration Packets are introduced；

（4）Configuration Registers Section， As is shown in Fig 5.3.

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

Fig 5.3 Configuration Registers

5.2 HWICAP operatiopn method

As can be seen from Figure 5.2, the HWICAP IP core only supports the AXI4-Lite

interface. That means, when we operate the HWICAP IP core, actually the register is operated. The corresopnding registers are shown in Figure 5.4 [10].

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 registers in HWICAP IP Core

The following highlights several of the registers used. Let's first analyze what we need to transfer the configuration frame into.

Write configuration frame：（1）Write frame into FIFO（2）Start configuration after writing。

The first step corresponds to the Write FIFO Keyhole Register,as is shown in Fig 5.5; The second step corresponds to the Control Register,as is shown in Fig 5.6.

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.

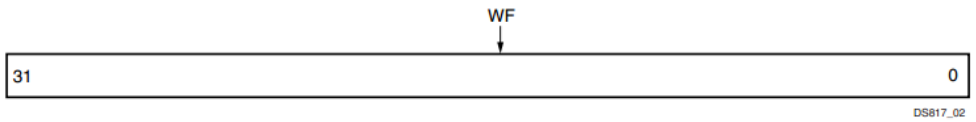

Figure 2-4: **Write FIFO (WF)**

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

Fig 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.

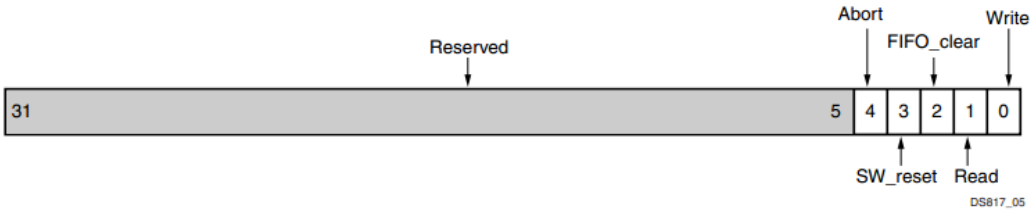

Figure 2-7: **Control Register (CR)**

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

Fig 5.6 Control Register

Read configuration frame：（1）Tell the IP core how many frames to read（2）Start read。

The first step corresponds to the Size Register,as is shown in Fig 5.7; The second step corresponds to the Control Register,as is shown in Fig 5.6.

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.

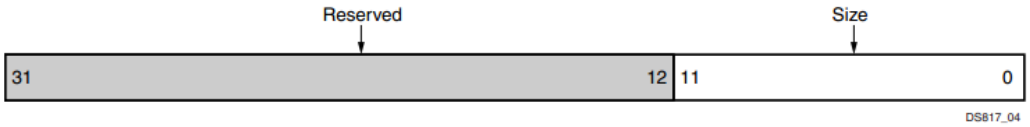

Figure 2-6: **Size Register (SZ)**

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

Fig 5.7 Size Register

The reader is advised to refer to PG134 for the function of other registers.

6 Verification system

A lot of research are done in the previous to enable online reading and writing of a single LUT. How can we prove that the work we have done before is correct? The answer is to establish a verification system. In the actual FPGA system, a specific LUT content inside the FPGA is read out online, and then some content is written online. If this can be done successfully, it can be said that the research work is basically successful.

6.1 Development board and connection topology

The development board connection topology is shown in Figure 6.1, and the FPGA development board model is Black Gold AX7103 [11], as shown in Figure 6.2. The reader can also use other Xilinx FPGA boards to build the verification system (the previous knowledge applies to all 7 series FPGAs). The PC and the FPGA are connected by JTAG, the Vivado software runs on the PC, and the TCL command is used in the Vivado TCL Console.

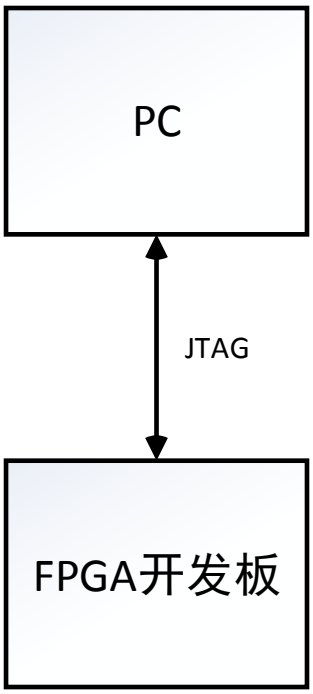

Fig 6.1 connection topology

Fig 6.2 FPGA Board

6.2 FPGA Internal module design

The JTAG2AXI IP core and the HWICAP IP core are responsible for receiving the control commands from the PC to read and write the contents of the LUT; The target LUT is the object to be read and written, and its Verilog instantiation is shown in Figure 6.4.

How to verify the contents of the LUT? Answer: In the PC's Vivado software TCL Console, write the TCL command, read the target LUT content, and read the content and the corresponding position of the rbt file.

How to verify the contents of the LUT? Answer: In the Vivado software TCL Console of the PC, write the TCL command to configure the target LUT content. After the configuration is complete, observe the ILA. Before the configuration, the initial value of the target LUT is 0x0123456789ABCDEF, and the counter generates a decremented address. Therefore, the changed data can be observed by the ILA; the configuration content is all 0s, and after the configuration is completed, the content observed by the ILA is all 0s, which proves that the configuration is successful.

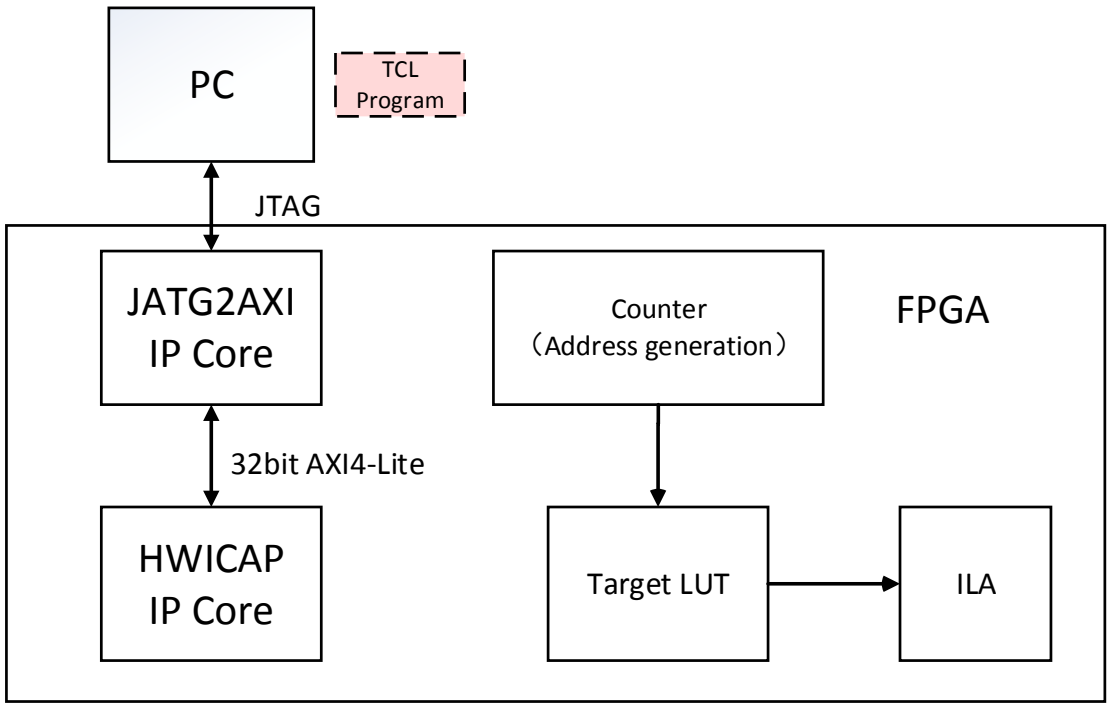

Fig 6.3 FPGA internal module

```
(* 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
);
```

Fig 6.4 LUT instantiated

6.3 How to write TCL code

To learn the syntax of TCL, you need to refer to the Xilinx official document, UG835 [12], UG894 [13]. In addition, Xilinx senior SAE Gao Yajun's TCL series tutorial [14] is also very good.

To learn the specific operation methods of HWICAP and ICAP in Section 5, refer to the HWICAP official driver given by Xilinx [15] , and the author also gives the TCL code needed to read and write LUT. The code written by the author refers a lot to the driver given by Xilinx.

6.4 Experimental result

Read LUT content:

Download the bitstream of the project "jtag_axi_icap_lut_AX7103_0123" (remember to reset by VIO), run the "read_frame_510521.tcl" file, the file reads 1/4 of the contents of the LUT, corresponding to the 510521 line of the rbt file (see Table 3.7), the results of reading are shown in Figure 6.5. It can be found that the two are corresponding.

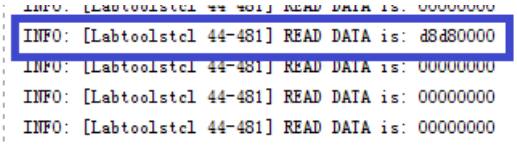

Fig 6.5 read result

Write LUT content:

Using the same bit file, before the LUT content is configured, the waveform can be seen in Figure 6.6. The three signals from top to bottom are the input (address) of

the LUT, the output of the LUT, and the output of the LUT. Parallelization (LUT output is not convenient to observe, after parallelization, it is convenient to observe the initial value of LUT); after the "source_write_all.tcl" file in the Vivado TCL Console line , the content of the LUT is reset to full 0, as shown in Figure 6.8, the configuration is successful.

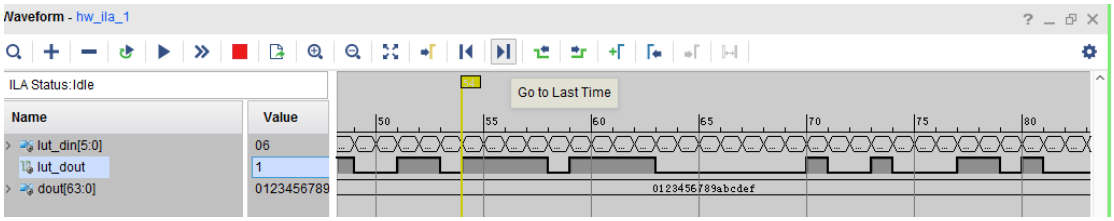

Fig 6.6 The Output of the LUT before reconfiguration

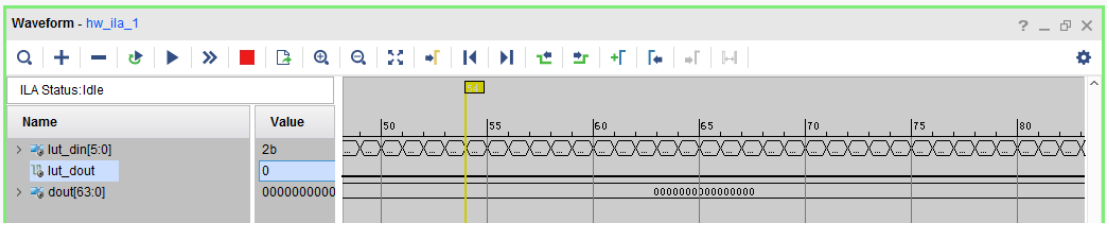

Fig 6.8 The Output of the LUT after reconfiguration

7 summary and prospect

Achievements：

The process of dynamic reading and writing of a single LUT is achieved, including LUT addressing, LUT configuration frame format, rbt file parsing (for comparison), LUT information extraction, LUT configuration method (HWICAP and ICAP operation), and demonstration verification system construction.

To be done later：

（1）In the later stage, the embedded CPU is required to implement the operation of the HWICAP IP core instead of the PC, because the PC+Vivado+TCL method is too inefficient;

（2）The above operation method is not applicable to the Zynq series. The reason is unknown. It is suspected that some operations are not in place. The single LUT of

the Zynq series is dynamically read and written. Further research is needed.

8 references

[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