Chapter 5

Some Useful Experiences

After discussing the implementation of the whole system, we hope that we can provide some useful experiences for those who would like to build up a similar system. In this chapter, we first introduce how to do system level evaluation before implementation. Then, since there are more than one hardware modules sharing the tasks of the whole system, we will provide some principles for the job of hardware partition. Later, in such a system equipped with FPGA and DSP modules, there may exist some implementation tips in the stage of development, and all of them will be provided. Finally, we will give a complete example about replacing the DFT (discrete time Fourier transform) component to FFT to show how they are carried out.

5.1 System Level Evaluation

In general, digital signal processing algorithms are becoming so complex that developers have little choice but to use a specialized modeling language like MATLAB to develop their intellectual properties. In particular, using such high level language offers the advantage of being able to use the analysis capabilities built inside, such as the “plot” function, or many of other built-in functions that facilitates us to roughly evaluate the performance before hardware implementation.

The system level evaluation mainly refers to the pre-evaluation of performance for some undecided algorithms in the same block, or the compatibility problem for the existence of each space-time mode working under the same system. Some competitive candidates of algorithms may not be considered due to their high computational complexity though they may have better performance than the other ones; and such

system performance evaluations can easily be done in high level language. The compatibility among three modes is also an important issue of system level evaluation in our implementation. Since three kinds of modes require different sizes of variables, which imply to different size of memory in hardware realization, we need to find out a solution that can support all three modes. For STBC mode, owing to the 3/4 code rate, it extends the original data symbols to a number of 4/3 times; the other two modes will not extend the number of symbols, but reduce to 1/2 and 1/4 times instead, since there are different ways, which depends on the number of data streams, for four transmit antennas to transmit symbols. In order to be compatible among the three modes, we find that 12 information symbols may be a good choice since it is the least common multiple of 3, 2, and 4. It means that the encoded symbols in each transmit antenna become 16, 3, and 6, for STBC, V-BLAST, and G-STBC respectively as shown in the upper part of Table 5.1, which implies that the processing time will be dominated by the 16 encoded symbols of STBC mode even though there are only 3 encoded symbols needed to process in the V-BLAST mode. This is because FPGA is designed for the mode that has the maximum number of encoded symbols to be processed. Therefore, to avoid dominating in STBC mode, we choose another factor 8 as our best choice though it induces slight difference in the generation of data source; the related result is shown in the lower part of Table 5.1.

Table 5.1: System level evaluation of data sizes among three modes

	STBC	VBLAST	GSTBC
Info. Sym.	12	12	12
ST-Coding	3/4	1/4	2/1
Ant. 1	16	3	6
Ant. 2	16	3	6
Ant. 3	16	3	6
Ant. 4	16	3	6
Info. Sym.	6	8	8
ST-Coding	3/4	1/4	2/1
Ant. 1	8	2	4
Ant. 2	8	2	4
Ant. 3	8	2	4
Ant. 4	8	2	4

5.2 Hardware Partition

If we have more than one programmable module to share the tasks of the whole system, we need to do the job of hardware partition. In our situation, there are some FPGA modules and a DSP module that can be programmed to realize our communication system. To find out the best way of partition, we provide some useful principles as follows; they are complexity, resource, and fluency, as shown in the Figure 5.1. In the figure, we first take a look at the complexity. Complexity refers to the intrinsic features of modules themselves, such as the support of floating-point expressions in DSP module and the only supported expression of fixed point in FPGA modules. The support of floating points implies that we can implement more complicated algorithms on DSP module with less effort than on FPGA modules, but it also implies that the efficiency is usually lower than fixed-point expression, since floating point expression requires a good compiler to translate C codes to assembly language so as to execute on DSP processor, while the fix-point expressions of FPGA are the real actions directly mapped to the circuit. Secondly, the resource is also an important issue to be considered. Since each of FPGA modules has limited number of programmable logic gates, we need to predict whether such a degree of complexity can

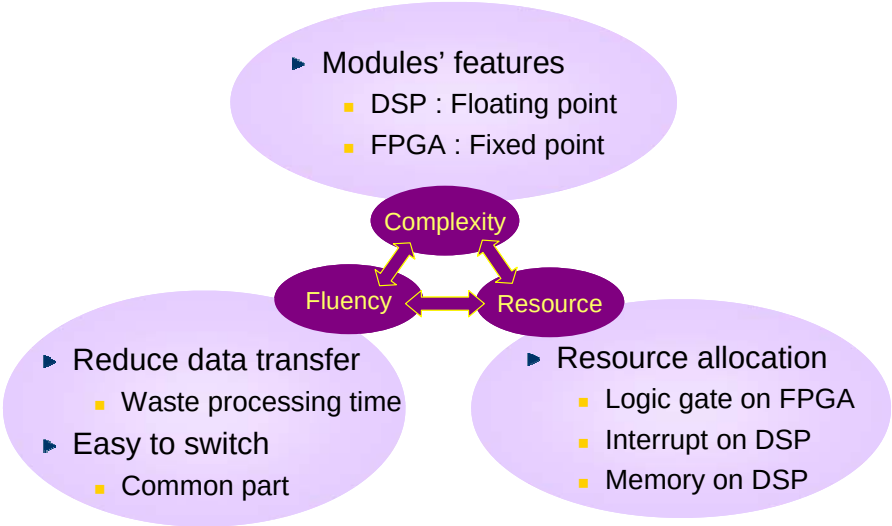


Figure 5.1: Some principles of hardware partition

be implemented or not. DSP module also has some restrictions on its numbers of interrupts and memories, implying that we cannot partition too many jobs on it though it is easier to implement. Finally, the fluency is also an important issue and the number of data transfer from one module to the other kind of module can reflect the degree. In the view of processing a packet, the more interfaces between two different modules, the more time will be consumed on data transfer, and it will results in less efficiency. The final version of our partition can be shown in Figure 5.2. The original number of interfaces between DSP and FPGA modules is only 2 in a packet processing cycle, which quite satisfies the principle of fluency. However, we finally decide to move Viterbi decoder from DSP to FPGA for its improvement on efficiency, and thus it induces another two interfaces before and after it. As far as complexity is concerned, we plan to realize the space-time coding and decoding which are the most complicated components of all in the field of DSP module.

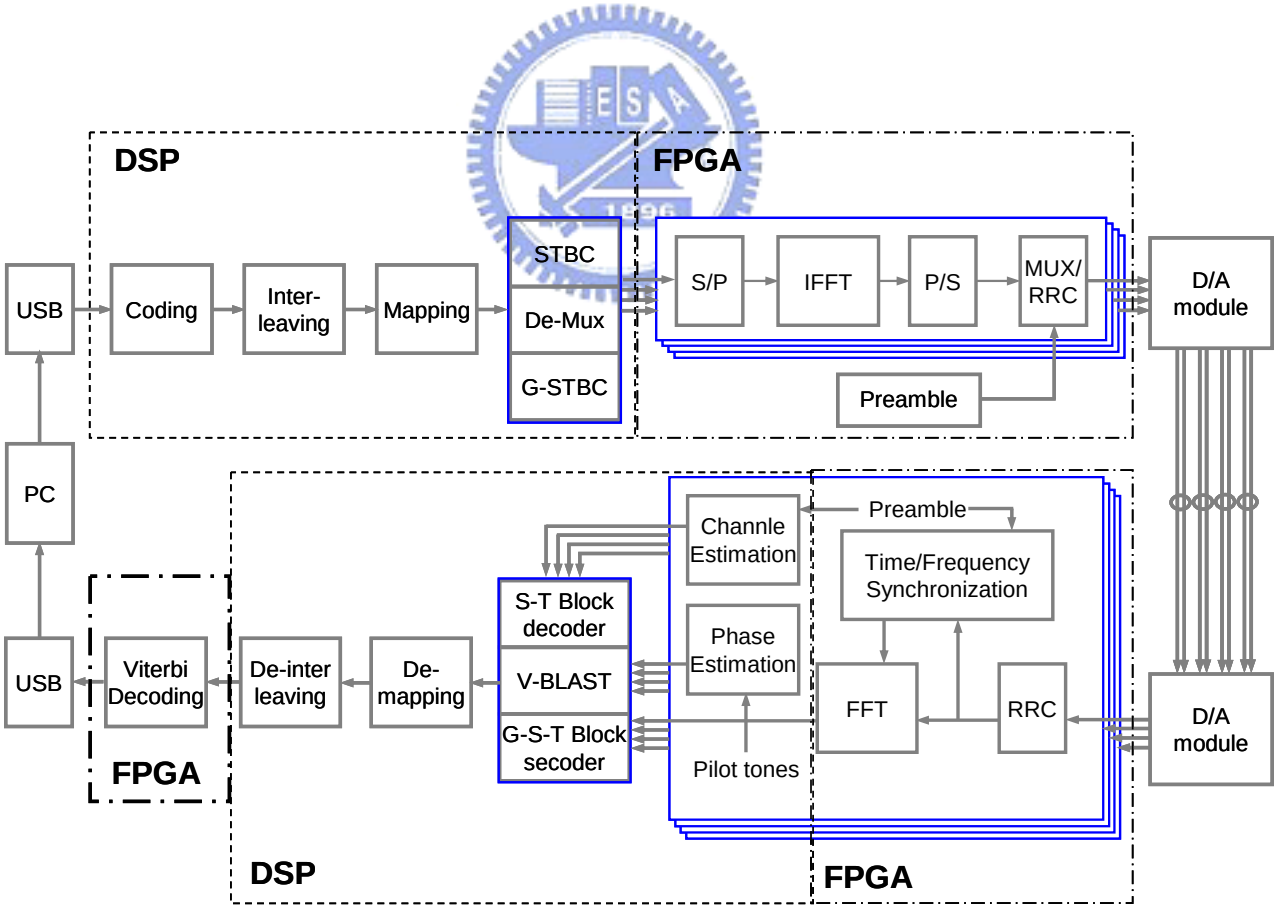


Figure 5.2: Partition of MIMO-OFDM system

5.3 Some Implementation Tips

The field of hardware implementation is highly empirical, meeting that the accumulation of useful knowledge and experiences is very important. Here we provide some useful implementation tips during the implementation of FPGA and DSP. At the end of this section, we will give an example about replacing FFT component to DFT component to share the experiences of implementation.

5.3.1 On FPGA Implementation

1. With the top-down design concept, FPGA is usually built up by several elementary components. Based on those components, some larger or more complicated components can be constructed. Therefore, we need to verify each elementary component carefully with the aid of functional simulation provided from EDA tools before directly verifying the whole blocks whenever we build up a new functional block.

2. Sometimes, to ensure that the elementary components in a huge block of circuit function correctly, like a whole FPGA module, we can link the major I/O of each elementary block to the top level design, so that we can check which component functions incorrectly once some unexpected results happen.

3. As shown in Table 5.2 [13], in FPGA design, the multiplier always consumes much more logic gates than the adder, which implies that we prefer using a series of adders instead of multiplier if not too many of adders are required to handle the same job. Once we have no choice but to use the multiplier, we have to lower the width of bus as much as possible to save logic gates.

Table 5.2: Comparison of the logic gate count between adder and multiplier

	Adder	Multiplier
4 bits	68	245
8 bits	157	927
16 bits	317	3563
32 bits	637	13875

4. RAM is also a gate-consuming component in FPGA design, which is usually used as a data buffer. To avoid the waste of logic gates, the size of RAM has to be designed carefully, but on the other hand, it implies that the size of RAM cannot be scaled and varied easily with input data that have different sizes.

5. During the design of FPGA, we realize that there is no short cut for debugging except for confirming each block step by step. There are too many times we try to skip over the routine and boring jobs for checking each of blocks step by step, but it always wastes us much more time than the one we check step by step. Therefore, we have to verify each of the related blocks step by step without skipping over whenever a unexpected result occur, or much more time will be wasted.

5.3.2 On DSP Implementation

1. Both C language and Assembly language can be used for the development of DSP. Although C is more friendly than Assembly, C has poorer efficiency than Assembly on execution time since C will be compiled to Assembly finally by a non-optimal compiler. Therefore, we can try to program with Assembly if possible, but if it is too difficult, we may develop the main algorithms in C language and then fine tune the programs using Assembly to improve the performance as much as possible [39][40].

2. The execution time of DSP highly depends on the data type declared in programs. The operation based on floating point variables which requires more than 64 bits to store performs poorest, since it involves with large amount of calculation. Therefore, we should try to use integer type variables as much as possible. In integer type, it can be further divided into short integer type and long integer type where they occupied the memory 16 bits and 32 bits, respectively. As the same concept mentioned before, once we can perform the same job with shorter variables, we should not declare longer variable so as to avoid the increase on execute time or the waste on memory.

3. Sometimes if the accuracy is our major concern, we may use a larger variable than normal to save the result. For example, when two 16-bit short integers are multiplied together, we may lose the accuracy if the output is too large. In this case, we should use a 32-bit long integer to store.

4. The declaration of global variable or local variable is also another important issue. There are mainly three suitable time to declare a variable as a global one. First, if the utility rate of a variable is high, we may consider using global type since it can reduce the I/O declaration among functions. Second, if the I/O for an function have the format of array, global variable could be a good choice to figure out the problem of data passing among functions with the format of array variables. Third, if some variables need to be watched to facilitate the task of debugging during execution, we should declare it as global type, since only global variable can be watched in memory watching window. Although global variables are useful, too much global variable will run out of memory and fail the building process.

5. Interrupt driven programming scheme requires us to program the interrupt subroutine in advance. To save more memory, we can abstract common functions into the global functions so that each interrupt subroutine can access. The major advantage of functional-oriented programming skill is to save the memory and to decrease the code size of interrupt subroutines for the inaccessible problem while the code size is too large.

6. We, like many other developers, use a specialized modeling language such as MATLAB to develop and verify our system in advance, and re-implement our algorithms using C language in DSP, where DSP can be taken as a simple version of MATLAB programs because many algorithms may be further simplified in DSP. Therefore, since we have the complete system in MATLAB where can provide friendly graphic user interface (GUI), inside mathematical functions, and large memories, it can be utilized to be a comparison system of DSP program during debugging. By using this skill, DSP can be programmed more easily and the debugging time can be much shortened.

5.4 Example: From DFT to FFT

Background:

Assume we are now in the middle stage of development. In the transmitter side of DSP, we have constructed the complete transmitter algorithms. In the transmitter side of FPGA, we have built up a complete algorithms mentioned before except that the IDFT block is implemented with IDFT algorithms. In receiver side of FPGA, we have also finished the development of timing synchronization algorithms and the DFT block with DFT algorithms. In receiver side of DSP, we can receive the signal from receiver FPGA only and the rest receiver processing have not been constructed.

Long-term Target:

Assume only one IP, FFT block, is available in FPGA. Our long-term target is to replace the IDFT block with the IFFT block, and replace the DFT block with the FFT block.

IP Description:

IDFT (DFT):

Type: Bit-reversed output (No need to do bit reversal)

Input data bus: 8 bits

Output data bus: 8 bits

Delay time between the first input and the first output: 69 clocks

FFT:

Type: Non Bit-reversed output (Need to do bit reversal)

Input data bus: 20 bits

Output data bus: 20 bits

Delay time between the first input and the first output: 90 clocks

---

- **Experiment 1:**

Since we have only an IP of FFT block, we need to realize the IFFT component by ourselves.

- **Process 1:**

Owing to the existence of a relationship between the FFT algorithm and the IFFT algorithm, we can create an IFFT component through some adequate modification on the FFT component. Therefore, we first try to derive the relationship between IFFT and FFT. The algorithms of FFT and IFFT can be expressed as follows:

$$\text{FFT:} \quad K(k) = \sum_{n=0}^{N-1} x(n) W^{nk}$$

$$\text{IFFT:} \quad x(n) = \frac{1}{N} \sum_{k=0}^{N-1} X(k) W^{-nk} \quad \text{where} \quad W = e^{-j\frac{2\pi}{N}}.$$

By proper derivation from IFFT algorithm we can find the structure of FFT algorithm imbedded as shown in the following expression:

$$X(k) = \sum_{n=0}^{N-1} x(n) W^{-nk} = N\left[\frac{1}{N} \sum_{n=0}^{N-1} x^*(n) W^{nk}\right]^*$$

It implies that the output of IFFT can simply be obtained by multiplying a scalar N with the complex-conjugated output of FFT that has a complex-conjugated input, and it can be summarized as follows:

$$\text{IFFT(in)} = N\left[\text{FFT(in}^*\text{)}\right]^*$$

After obtaining the information of their relationship, now we can modify the FFT component to an IFFT component, and the circuit is shown in Figure 5.3.

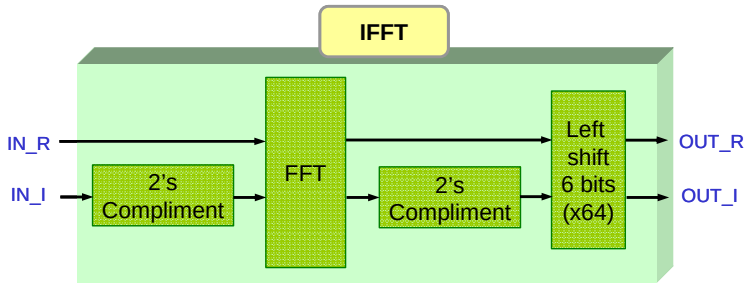


Figure 5.3: Circuit design of IFFT implemented by FFT

- **Result 1:**

After building the block of IFFT, we can start to deal with the problem of truncation from a 20-bit output databus to an 8-bit output databus.

- **Experiment 2:**

  Try to fit the new 20-bit I/O databus into the old 8-bit I/O databus.

- **Process 2:**

  In order to determine where the old 8-bit databus locates in the new 20-bit one, we first build up an adaptive mechanism for bit truncation of the output from 20 bits to 8 bits for two target components, whereas only the skill of zero padding to 20 bits is required at their input side. The control bits of adaptive mechanism are sent by pattern generator, where an 8-bit pattern is supported. Among them, one bit is used for reset signal; four are used for FFT, and three are used for IFFT. The mapping between control bits and truncation range are listed in Tables 5.3 and 5.4, respectively.

Table 5.3: Mapping table of control bits for output truncation of IFFT

Ctrl. Bits	Truncation	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
000	10 downto 3										■	■	■	■	■	■	■	■			
001	11 downto 4									■	■	■	■	■	■	■	■				
010	12 downto 5								■	■	■	■	■	■	■	■					
011	13 downto 6							■	■	■	■	■	■	■	■						
100	14 downto 7						■	■	■	■	■	■	■	■							
101	15 downto 8					■	■	■	■	■	■	■	■								
110	16 downto 9				■	■	■	■	■	■	■	■									
111	17 downto 10			■	■	■	■	■	■	■	■										

Table 5.4: Mapping table of control bits for output truncation of FFT

Ctrl. Bits	Truncation	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0000	7 downto 0													■	■	■	■	■	■	■	■
0001	8 downto 1												■	■	■	■	■	■	■	■	
0010	9 downto 2											■	■	■	■	■	■	■	■		
0011	10 downto 3										■	■	■	■	■	■	■	■			
0100	11 downto 4									■	■	■	■	■	■	■	■				
0101	12 downto 5								■	■	■	■	■	■	■	■					
0110	13 downto 6							■	■	■	■	■	■	■	■						
0111	14 downto 7						■	■	■	■	■	■	■	■							
1000	15 downto 8					■	■	■	■	■	■	■	■								
1001	16 downto 9				■	■	■	■	■	■	■	■									
1010	17 downto 10			■	■	■	■	■	■	■	■										
1011	18 downto 11		■	■	■	■	■	■	■	■											
1100	19 downto 12	■	■	■	■	■	■	■	■												

■ **Result 2:**

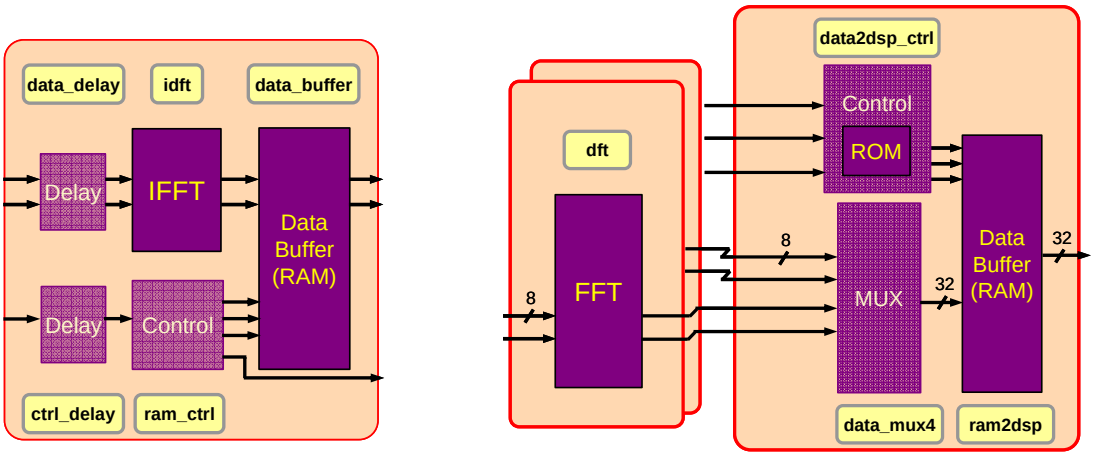
After trying all the combinations of adaptive truncation, we find that no combination can result in an expected output after passing through IFFT and FFT components. After a while, we finally realize that we need to do bit-reversal first for both the outputs of IFFT and FFT blocks. No wonder we cannot obtain the correct result.

■ **Experiment 3:**

Do bit-reversal for both the outputs of IFFT and FFT blocks.

■ **Process 3:**

In the output data buffer of IFFT, the bit-reversed writing address can be simply generated by the control block *ram_ctrl* as shown in Figure 5.4 (a), since there are no other tasks which are jointly considered during the address generation. On the contrary, address generation in the output data buffer of FFT is so complicated that we have to use a pre-written ROM to store the final address in the control block *data2dsp_ctrl* as shown in Figure 5.4 (b), since bit-reversal, remove of zero tones, and rearrangement of pilot tones are jointly considered. The tasks of zero tone removal and pilot tone arrangement is illustrated in Figures 5.5 (a) and 5.5 (b).



(a) (b)

Figure 5.4: (a) IFFT component and its output data buffer

(b) FFT components and its output data buffer

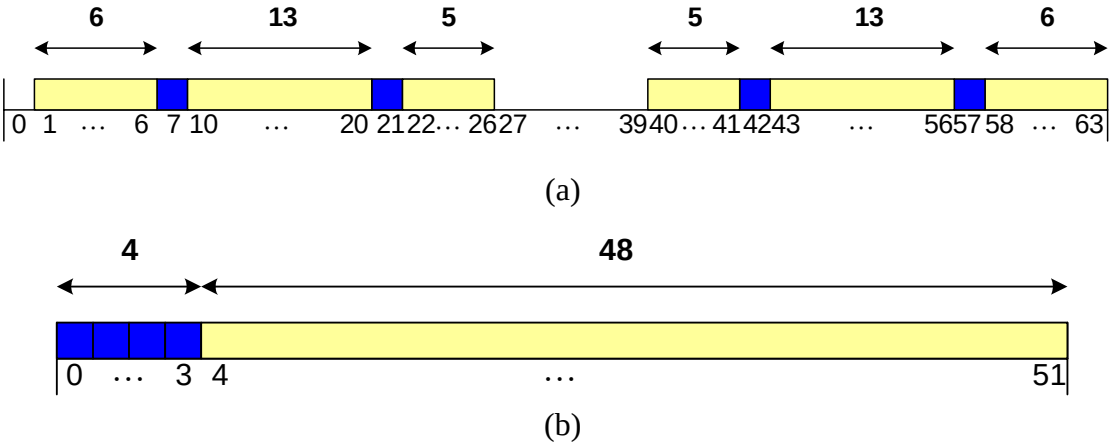


Figure 5.5: (a) Data before zero tone removal and pilot tone rearrangement

(b) Data after zero tone removal and pilot tone rearrangement

Table 5.5: Comparison between normal (NM) and bit-reversed (BR) order

NM Special bits	Z						P													P					Z	Z	Z	Z	Z
NM Order	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
NM Address	4	4	5	6	7	8	9	0	10	11	12	13	14	15	16	17	18	19	20	21	22	1	23	24	25	26	27	28	28	28	28	28
BR Address	18	18	18	37	10	30	25	45	7	22	22	41	14	33	48	48	5	20	20	39	12	32	27	46	9	28	23	43	16	35	50	50
BR Order	0	32	16	48	8	40	24	56	4	36	20	52	12	44	28	60	2	34	18	50	10	42	26	58	6	38	22	54	14	46	30	62
BR Special bits	Z	Z								Z					Z																Z	
NM Special bits	Z	Z	Z	Z	Z	Z					P															P						
NM Order	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
NM Address	28	28	28	28	28	28	28	29	30	31	32	2	33	34	35	36	37	38	39	40	41	42	43	44	45	3	46	47	48	49	50	51
BR Address	4	19	19	38	11	31	26	3	8	1	1	42	15	34	49	49	6	21	21	40	13	2	47	47	0	29	24	44	17	36	51	51
BR Order	1	33	17	49	9	41	25	57	5	37	21	53	13	45	29	61	3	35	19	51	11	43	27	59	7	39	23	55	15	47	31	63
BR Special bits		Z					P		Z	P				Z				Z				P	Z		P					Z	

As illustrated in Figure 5.5 (a), the indices zero, and from 27 to 39 denote zero tones, while the indices 7, 21, 42, and 57 represent pilot tones. After the removal and rearrangement, we discard the data on zero tone by using the same address for that on data tone, resulting in the coverage by latter data, and we also rearrange the pilot tones to the most front part of the data as depicted in Figure 5.5 (b). Moreover, the complicated address is listed in the row of bit-reversed (BR) address in Table 5.5. As illustrated in the above table, the bit-reversed special bits are dispersed so that the BR address is difficult to generate due to the disorder of index when compared with the generation of somehow ordered normal (NM) address.

- **Result 3:**

Although the generation of address is checked again and again, the expected results are still not found. We finally realized that the delay time between IDFT and IFFT, or DFT and FFT is different and it results in the incorrect receiving for the IFFT or FFT blocks. Therefore, we need to adjust the blocks that control the arrival time of data or control signals.

- **Experiment 4:**

  Adjust the delay controlling blocks to achieve the correct receiving of IFFT (FFT) blocks.

- **Process 4:**

  As illustrated in Figure 5.6, the first part shows the old relationship among original coming data (*DATA*), control signals (*CTRL*), and the active period of IDFT (or DFT) block. To align the data signal with the rising edge of active period and align the control signals with the falling edge of active period, we adopt two delay blocks as shown in Figure 5.4 (a) to achieve the goal, and the signal after delayed is shown in the second part in Figure 5.6, where the alignment of both DATA_Delay and CTRL_Delay perform well. After replacing DFT to FFT, since the delay times (active period) are different, an incorrect result due to no alignment between control signal and the falling edge of FFT active period will be obtained, as we can see in the third part. Through the analysis, we can correct the error receive timing by simply delaying another 21 clocks in *ctrl_delay* block, and the final result with correct timing can be seen in the fourth part.

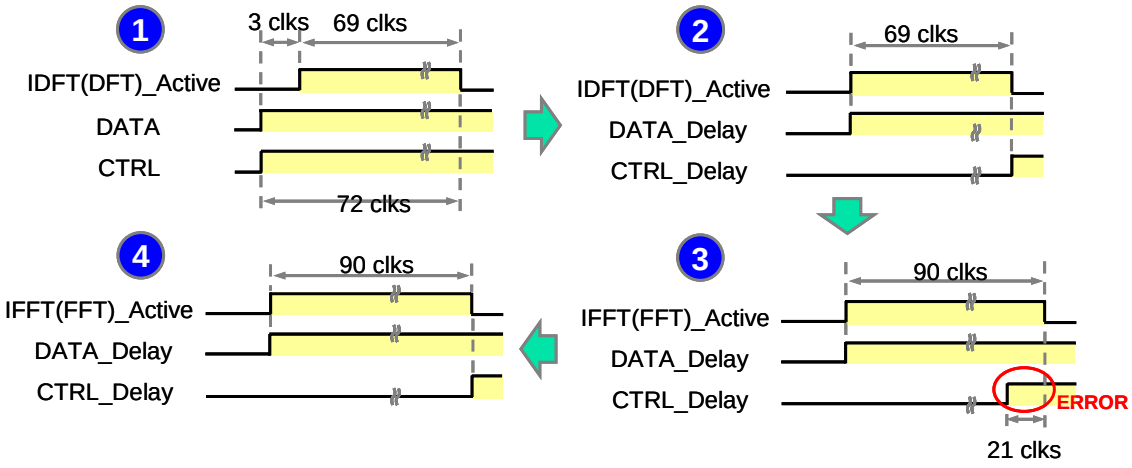


Figure 5.6: Process of adjusting the delay of control signal

we previously have an IDFT/DFT imbedded system that works properly, we can take the IDFT and DFT blocks as two black boxes and replace them with equivalent blocks. We believe that if we can create an equivalent block especially in the scalar factor, the replacement can be processed without any problem with regard to the reduction of the carried information. Therefore, our next experiment is trying to find out the same black boxes for both IDFT and DFT blocks.

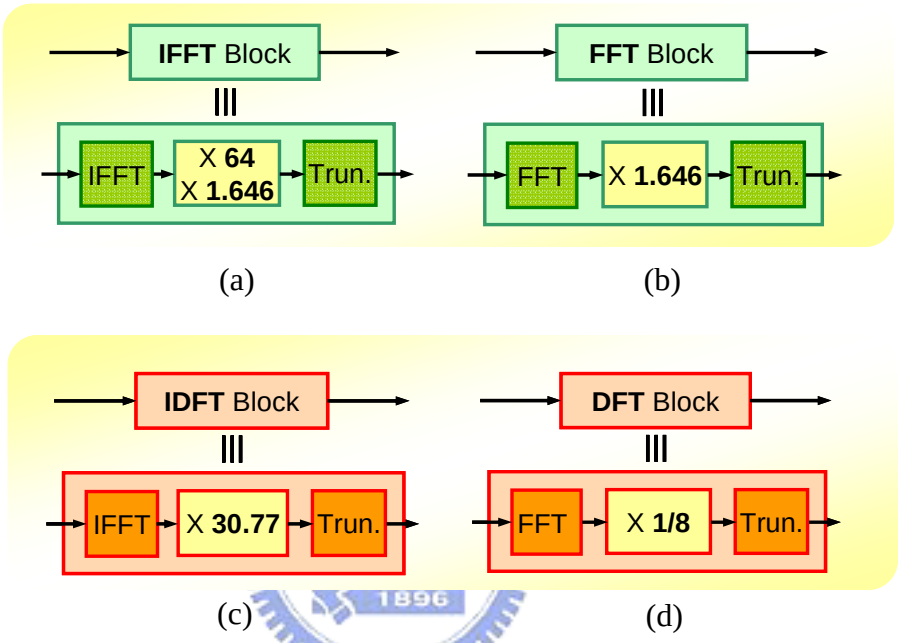


Figure 5.9: Relationship between MATLAB function IFFT (FFT) with (a) IDFT block (b) IFFT block (c) DFT block (d) FFT block

---

- **Experiment 7:**

  Find out equivalent black boxes for both DFT and IDFT blocks.

- **Process 7:**

  From the results of the previous experiment, the equivalent black boxes for both DFT and IDFT blocks can simply be obtained by a further analysis and adjusting the scalar of IFFT (FFT) block to be the same as that of IDFT (DFT) block. As far as IDFT block is concerned, since it has a scalar 30.77 between its input and output, we have to multiply an additional scalar 0.292 at the output of IFFT block, where the value 0.292 can be obtained by $30.77/(64*1.646)$ as shown in Figure 5.10 (a). On the other hand, when DFT block is concerned, the additional output scalar can originally be calculated by $(1/8)/1.646$, and we further

■ **Result 4:**

After the adjustment, the data values out of FFT at the receiver side still do not match the expected values. Checking again and again, we still cannot obtain the expected values. To stop meaningless checking for each of the places we doubt, we decide to build up another project in FPGA where there are only a pair of IFFT and FFT, some data buffers, and related control blocks, to verify whether we can obtain the same signals or not if we pass the data through IFFT first and FFT secondly. By the way, we can also observe the truncation behavior in a purer environment instead of in a complicated one.

■ **Experiment 5:**

Build up a new project to verify whether we can obtain the same signals if we pass the data through IFFT first and FFT secondly.

■ **Process 5:**

The circuit design of the testing project is depicted in Figure 5.7, including a central controller, a ROM-embedded data generation block, an IFFT/FFT pair, and two following data buffers. The central controller is the major part that controls the timing when each block should be turned on or off.

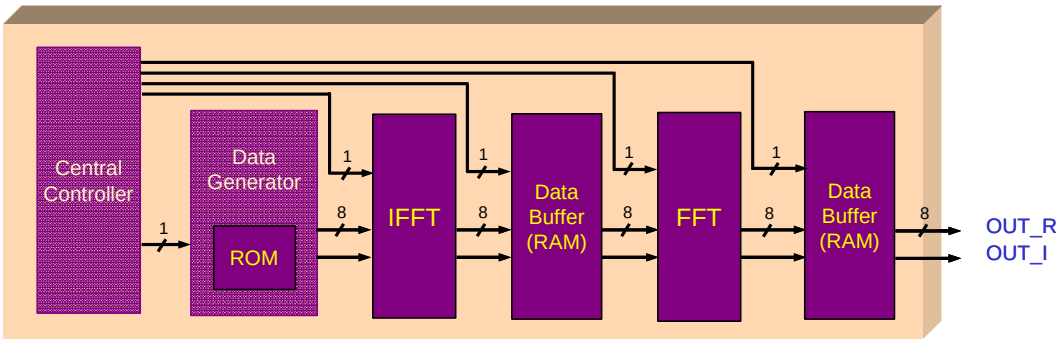


Figure 5.7: Circuit design of the testing project

■ **Result 5:**

By functional simulation, we find that the output data, *OUT_R* and *OUT_I* cannot match with the input data generated from the pre-written ROM in Figure 5.7. According the result, we may guest that maybe the FFT component is not exactly the same as the FFT function in MATLAB and there may exist a scalar difference

between them. Once there is a scalar k, the output of passing through IFFT and FFT may become k^2, resulting in the mismatch between the input and output and the difficulty to recognize the existence of the scalar k^2 for the data represented in a binary form is not so straightforward. Therefore, our next experiment is to find out the scalars of both DFT and FFT blocks respectively when compared with the FFT function built in MATLAB.

- **Experiment 6:**

  Find out the scalars exist in both DFT block and FFT block compared with FFT function built in MATLAB.

- **Process 6:**

  Simulating the truncation mechanism in MATLAB is necessary so that a closer environment can be created. There are two typical truncation situations, truncation that happens at tail and at head, as illustrated in Figure 5.8 and labeled "1" and "2" respectively. The left hand side shows two truncation cases in hardware, and the right hand side shows the corresponding expressions used to simulate in MATLAB. By comparing the output of DFT and IFT built in previous experiment with the truncated version of DFT and IFFT simulated in MATLAB, we can start to find out the relation ship between them.

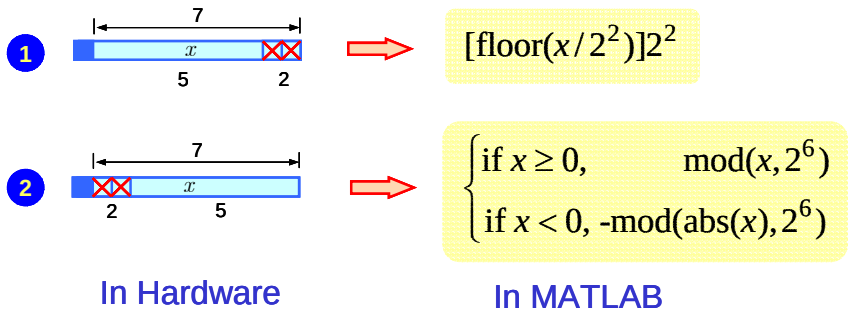
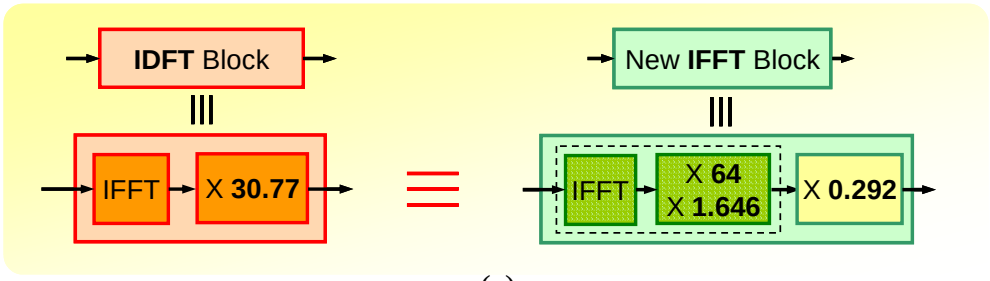


Figure 5.8: Truncation behavior of IFFT (FFT) block

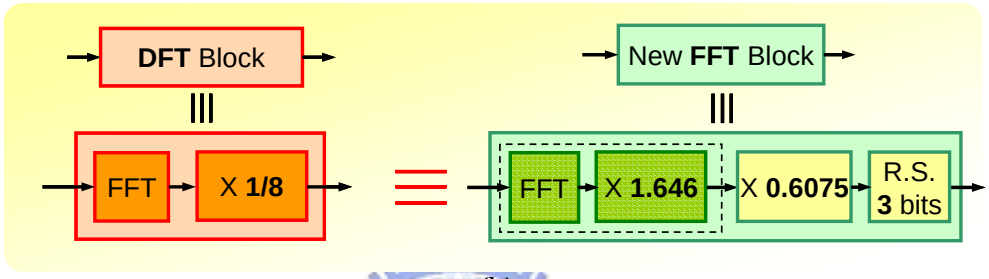
- **Result 6:**

  By fixing the input data of DFT and FFT blocks in hardware, and the input of truncated version of FFT function in MATLAB, we can finally obtain a relationship among them as shown in Figure 5.9. The results inspire us that since

separate it into two terms, 1/8 and 1/1.646, where 1/8 can be realized by right shifting (R.S.) 3 bits in hardware, as illustrated in Figure 5.10 (b).



(a)



(b)

Figure 5.10: (a) Adjusted new IFFT block (b) Adjusted new FFT block

- **Result 7:**

  After adjust the previous IFFT block by adding a scalar at its output, we can obtain the same value as we expected at the outer output of IFFT block. As done in FFT block, the previous FFT block is also modified by putting a scalar at its output, but unfortunately the results do not seem to match what we expected. Therefore, our next experiment is trying to figure out why we cannot obtain the expected value.

- **Experiment 8:**

  Figure out the reason that causes the output of FFT block to mismatch the results we expected.

- **Process 8:**

  So far, we can obtain the same value of FFT input in the receiver side of FPGA compared with in MATLAB, but we cannot obtain the expected value after the FFT block. Therefore, we try to analysis the range of input data to ensure that all

the input data fall into the safe range and the information within will not be truncated after FFT processing. Fortunately, we find that there are some input values like 44 or -28 which may be too large to pass the FFT block without saturation. To make sure if the values 44 and -28 are too large or not, a small trial in MATLAB is processed and the command is expressed as follows

$$\text{fft}([45*\text{ones}(1,32)\text{-}29*\text{ones}(1,32)])$$

The result of this trial shows that the output may range from -1506 to 1506 and we can not express the output by using only 8 bits, which implies the occurrence of saturation and truncation. A general saturation example is shown in Figure 5.11. If we have too large input values, the inner output of FFT may become so large that some values cannot be stored, which results in the saturation on outer output. To avoid this situation, we have to scale down the input. As illustrated in upper part of Figure 5.12, we move the right-shifting block at the output to the input side, since the action of right shifting 3 bits is equivalent to scaling down by a factor of 8. By this method, the whole FFT block can experience still the same scalar and achieve the goal of saturation avoidance almost, but the results show that saturation still happens on a few larger inputs. To further reduce the situation, we right shift one more bit at input side, and we add a 1-bit-left-shifting block to compensate. The final version is shown in the lower part of Figure 5.12.

- **Result 8:**

  After the modification for the avoidance of saturation, we eventually obtain the correct value as we expected in the output of FFT block, which also achieves the long-term target, replacing the IDFT (DFT) blocks with IFFT (FFT) blocks

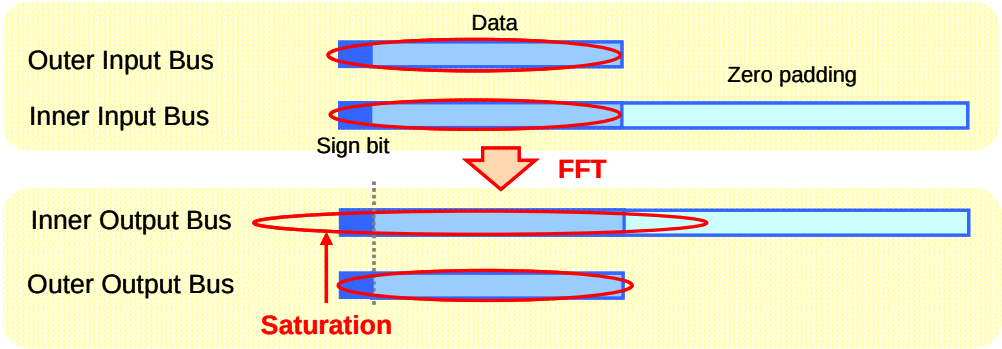


Figure 5.11: Saturation example of FFT block

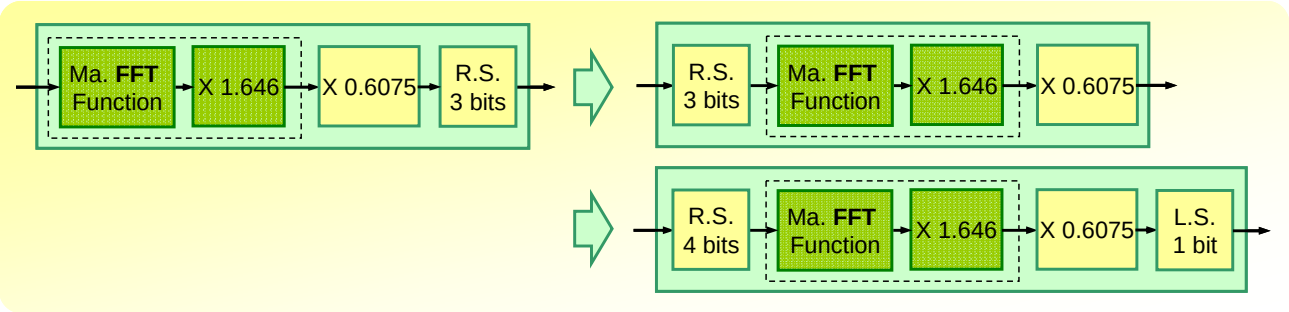


Figure 5.12: Modification for the avoidance of saturation

5.5 Summary

In this chapter, we provide some useful experiences for those who would like to build up such a similar prototype system. We first give an overview of how to do system level evaluation before implementation. Then, the job of hardware partition is also introduced since we have more than one hardware modules to share the whole tasks. Moreover, during the implementation, some precious experiences and tips are provided both in FPGA realization and DSP realization, respectively. In the end of this chapter, we give a detailed example on replacing DFT (IDFT) components to FFT (IFFT) components by means of showing a series of arising problems and the corresponding solutions.