Chapter 4

System Realization on FPGA and DSP

Chapter 4 is the major part of this thesis, which is organized as follows. In the first subsection, we provide some pre-implementation works, such as preamble design for MIMO structure in MATLAB, interrupt scheduling in DSP, arrangement of data buffer insertion in FPGA, and hardware partitioning for both DSP and FPGA. Then, we give our detailed implementation results on DSP and FPGA respectively and chronologically in the sense of processing a packet. In addition, to achieve the goal of being an adaptive system, we first show the experimental results under different channel conditions. By analyzing the data, we try to build up the tables for adaptive mechanism, which makes the mode selection strategies more complete. Finally, we will provide some simulation results and prove that under a varying channel condition the system throughput can be improved significantly by using the strategies we proposed.

4.1 Pre-Implementation Works

Before the implementation of the MIMO-OFDM system, some works are required to be done in advance. In the algorithm development stage, we have to decide which algorithm we should adopt, such as preamble design for the MIMO system. After that, some pre-works for hardware realization are necessary. In DSP, the most important pre-work is the interrupt scheduling, since the interrupt is a limited resource that we have to schedule carefully before implementation. Moreover, the arrangement of data buffer insertion is also required before writing the HDL codes in FPGA. Finally, we will show the results of hardware partition to give a picture of the whole system.

4.1.1 MATLAB: Preamble Design

Unlike the preambles defined in the IEEE 802.11a standard [10], the system we implement is a MIMO system, implying that we have to estimate the MIMO channels which may require more long preambles to perform the task and need an advanced design to separate the mixed received preambles at the receiver. Therefore, the transmitted preambles among different transmit antennas have to be designed carefully.

To find out the proper design, we first define received preambles $\mathbf{Z}$ as follows.

$$\mathbf{Z} = \mathbf{H}(\mathbf{T} \otimes \mathbf{p})$$

where $\mathbf{p}$ is an $1 \times k$ long preamble row vector with k being the length of long preamble, $\mathbf{T}$ is a 4×4 structure matrix, $\mathbf{H}$ is a 4×4 channel matrix, $\mathbf{Z}$ is the cascaded received vector with size of $4 \times k$, and $\otimes$ denotes the Kronecker product. At the receiver, we may multiply a vector $\mathbf{T}^T \otimes \mathbf{p}^T$ to match the mixed preambles

$$\begin{aligned}\mathbf{Z}(\mathbf{T}^T \otimes \mathbf{p}^T) &= \mathbf{H}(\mathbf{T} \otimes \mathbf{p})(\mathbf{T}^T \otimes \mathbf{p}^T) \\ &= \mathbf{H}(\mathbf{T}\mathbf{T}^T \otimes \mathbf{p}\mathbf{p}^T) = k\mathbf{H}\end{aligned}$$

According to some properties of Kronecker product and the fact that $\mathbf{p}\mathbf{p}^T$ is a scalar, we can finally obtain a scaled channel matrix only if $\mathbf{T}\mathbf{T}^T$ is a scalar. Therefore, the structure matrix $\mathbf{T}$ should be designed as an orthogonal matrix. In fact, there are some common 4×4 orthogonal matrices, such as identity matrix $\mathbf{I}$, or the matrix used for STBC of four transmit antennas mentioned in Section 2.4.1. To reduce complexity, we choose identity matrix $\mathbf{I}$ as our structure matrix as illustrated in Figure. 4.1.

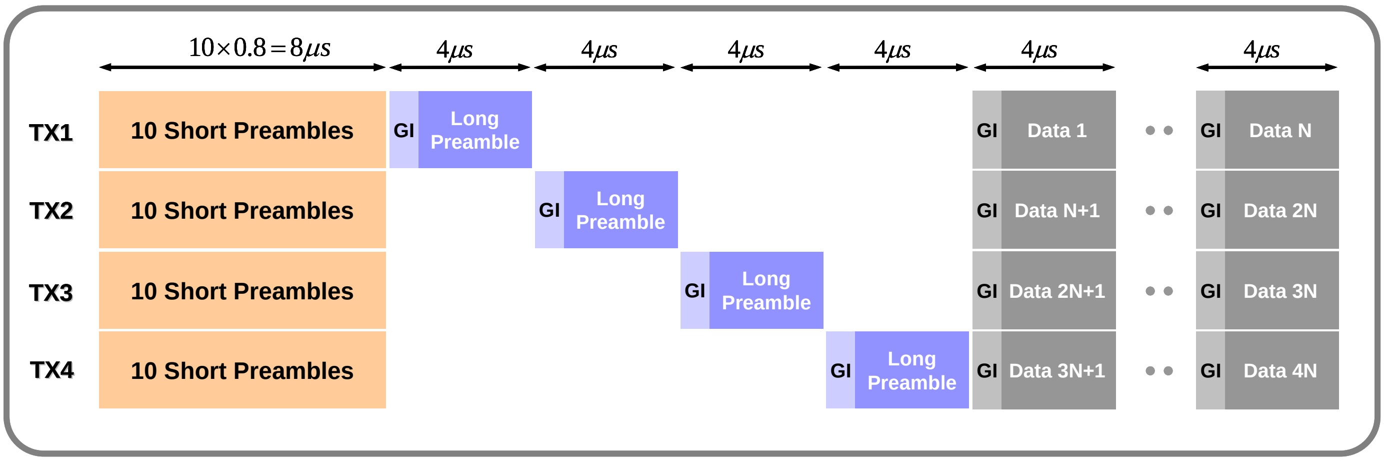


Figure 4.1: Frame structure using identity matrix as structure matrix

4.1.2 DSP: Interrupt Scheduling

The type of DSP program can be divided into interrupt-driven program and polling-based program. With an interrupt-driven program, an interrupt is selected (usually refers to INT11) once the nearby device triggers. A polling-based program (non-interrupt driven) continuously polls or tests whether or not data are ready to be received or transmitted; this is less efficient than the interrupt scheme [37]. Therefore, we adopt the interrupt scheme to realize our system partitioned in DSP. To select interrupt INT11, a branch instruction to the interrupt service routine (ISR) located in C program is placed at the address INT11 in the vector file, *vetors_11.asm*, so that we can know which subroutine will be executed once an external interrupt is triggered.

Interrupt scheduling refers to the organization of those limited interrupts used either for major subroutines or for handshakes of data transfer. Actually, there are four external interrupts for major subroutines (trigger signals *EXT0~3* corresponding to *EXTINT1~4{}* subroutines) and four internal interrupts for handshakes of data transfer. Besides, there are three modules, FPGA, DSP, and USB modules, which communicate with one another. The finalized schedule result of interrupts is illustrated in Figure 4.2. As shown in the figure, we use two major subroutines *EXTINT2{}* and *EXTINT1{}* as the transmitter and the receiver respectively in the portion of DSP, and their corresponding trigger signals from FPGA are *EXT1* and *EXT0*. In the beginning, when the *RST* signal which comes from the pattern generator is received, FPGA sends the signal *EXT1* to trigger the subroutine *EXITINT2{}*, and the system does not start to process until now. The handshake interrupt **inter3_ptr* is used to ask USB module for image data generated from the web camera on PC as the source data. Then, after the portion of transmitter in DSP is finished and ready to send the traffic data, **inter2_ptr* is used to handshake with FPGA. After a while, data are processed from the transmitter in FPGA via DAC and ADC to the receiver in FPGA, and again, *EXT0* is sent to trigger the receiver subroutine *EXTINT1{}* so as to start the portion of receiver in DSP. Once the DSP subroutine is triggered, DSP plays an active role to handle all the tasks of handshaking. Hence, data in FPGA are returned to DSP with the aid of **inter0_ptr*. Since the viterbi decoder is located in FPGA, it is necessary to transfer the data to

FPGA again, and we use the last unused handshake interrupt **inter1_ptr* to perform the task. Finally, we return the decoded data to USB modules by **inter3_ptr*, so that we can compare the unprocessed image source with the processed data received on PC through the interface of USB 2.0. Note that DSP does not immediately receive the data right after the process of the Viterbi decoder in FPGA. The decoded data are stored until the receiver FPGA is returning its data to receiver DSP in the next packet. Technically, they are imbedded in the unused bits of the databus. By using this skill, a significant period of time for data transfer can be saved, but it also induces a one-packet delay of data arrival at the same time.

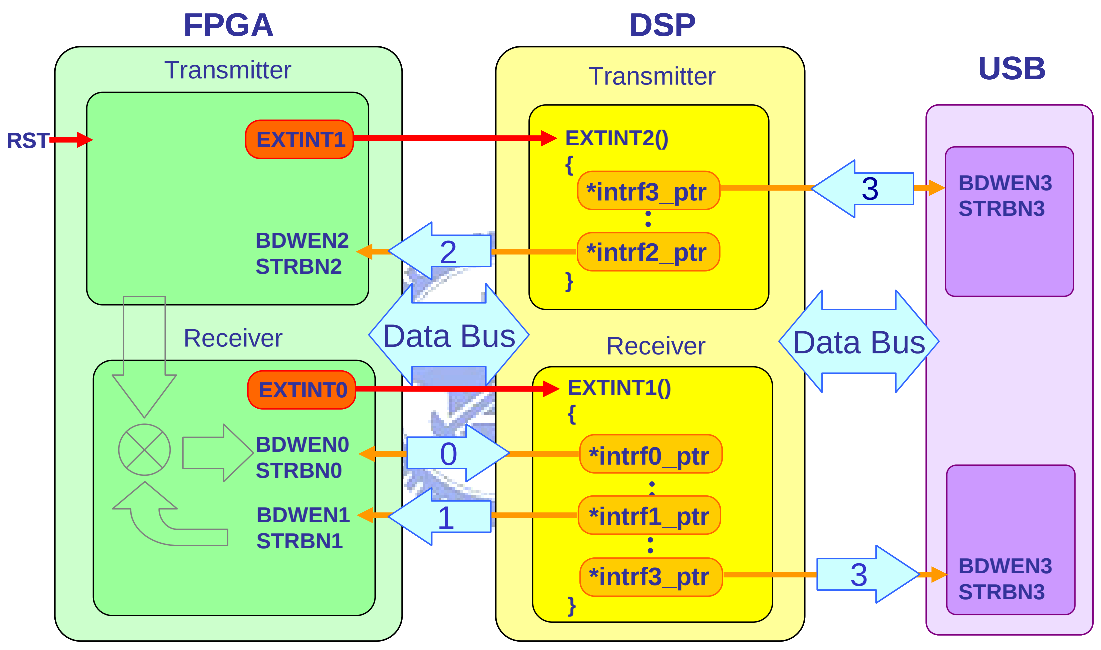


Figure 4.2: Results of interrupt scheduling

4.1.3 FPGA: Data Buffer Insertion

The programming concepts in high level language like MATLAB and in hardware description language like VHDL are quite different. In general, high level language keeps its temporary data in a form of variables, and simply assigns the stored variable to another one which is used to be the input of next stage or functions if necessary, whereas hardware description language may need extra data buffer and related components to perform the same task.

As shown in Figure 4.3, we first give an example around IFFT stage in high level language such as MATLAB. The previous stage generates some output data which are stored in the variable with a format of vector or array. Then all we need to do is just to copy the variable to another one and feed it into the following function, IFFT, and in a while the output is stored again in the variables waiting for the process of next stage. In other words, the data can be temporarily stored whenever needed in the memory pre-allocated by the application software, which facilitates us to develop our system.

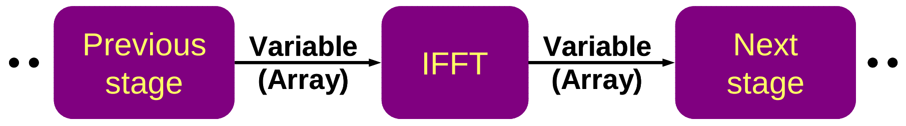


Figure 4.3: Programming concept in high level language

Unfortunately, the hardware description language does not have such merits. Instead, some extra memory blocks are needed to keep those data from previous stage in “mind” until the next stage handles. As illustrated in Figure 4.4, we not only have to prepare the memory blocks but also the control elements such as address and control signal generator to perform the same task as in high level language. These memories are also called buffers, which can further be classified into input data buffers and output data buffers depending on the place it locates. Since we have no choice but to add those extra elements, some index-related jobs can be performed in the same time, such as zero padding, bit reversing, or adding cyclic prefix and so forth. Therefore, it has to be planned carefully to insert data buffers between any two major adjacent stages in FPGA. Particularly, when the insertion is combined with the index-related jobs, the generation of address will become very complicated, which takes us a lot of efforts to implement.

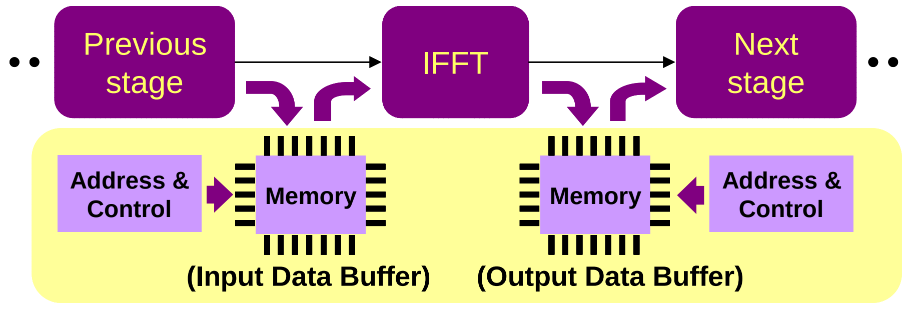


Figure 4.4: Programming concept in hardware description language

4.1.4 Overview of Hardware Partition

Hardware partition is also an important pre-work before implementation. Actually, some principles can be referred to perform the task of hardware partition, such as the consideration of implementation complexity, overall fluency, and limited resource, which we will have a more detailed discussion in Chapter 5. Here, we just show the result of our partition in Figure 4.5 to give a picture on the whole system.

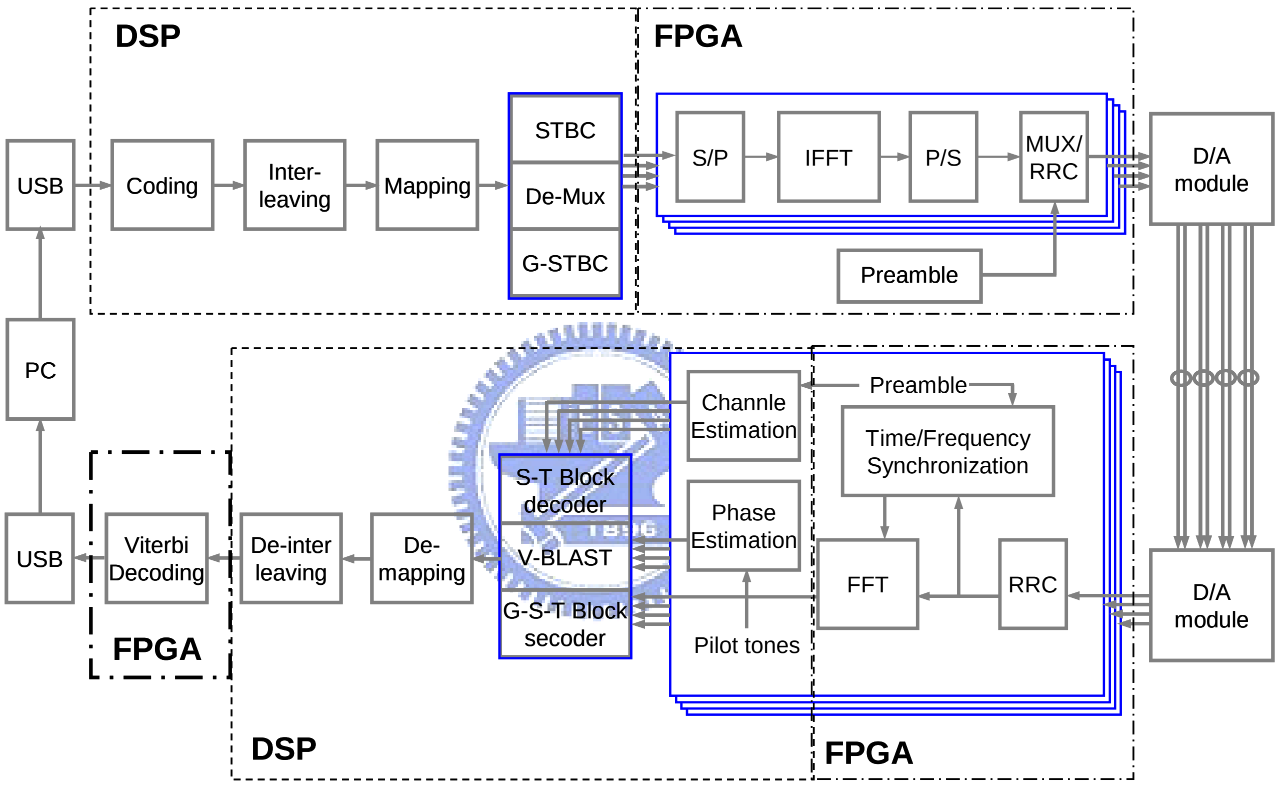


Figure 4.5: Partition result of MIMO-OFDM system

4.2 Transmitter on DSP

As mentioned in Section 4.1.2, a packet is first processed in DSP, and the major tasks are illustrated in Figure 4.6. Channel data is declared in the form of matrix with a size of 32×8 where the number of columns refers to the number of symbols transmitted in a packet. After the 1/3 rate convolutional encoder as introduced in Section 2.2.1, each column is extended to 96 bits, and then the inter-leaver block in

Section 2.2.2 mixes the data on each column individually keeping the size still 96×8. Next, the mapper tries to combine two bits into a complex symbol. In DSP the programming of C language does not support the complex data type. Therefore, we define a simple class containing only two member variables, i and q, referring to real part and imaginary part respectively so as to simplify our development later. After that, two adjacent bits in the same column are combined and mapped into a complex number and stored in the self-defined complex data type, which forms a 48×8 complex data.

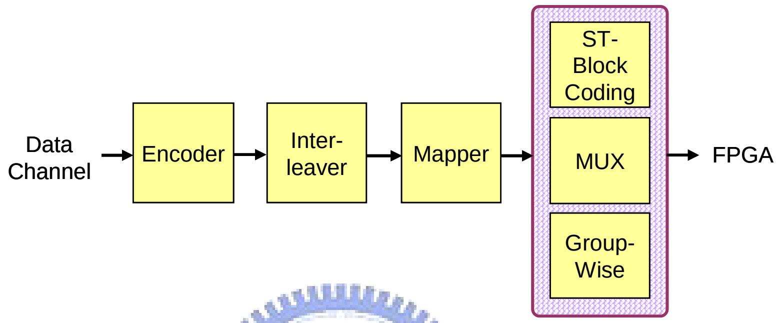


Figure 4.6: Diagram of transmitter in DSP

The previous part in DSP can be considered the common part of transmitter. The following space-time encoder is quite different among three methods. V-BLAST do nothing but branches the 48×8 complex data source into four 48×2 branches for four transmit antennas, implying only two symbols will be transmitted in V-BLAST mode. In G-STBC mode, owing to performing Alamouti space-time block coding in each group, each four incoming symbols will provide the data for four transmit antennas and double the symbols needed to be transmitted. Therefore, each transmit antenna needs to send four symbols for eight data source symbols defined in a packet. All the above-mentioned data size is gathered in Table 4.1. Since the process right after transmitter in DSP is to transfer data to FPGA, which means that the space-time encoded data has to be replaced by limited bits so as to pass through the databus, STBC becomes the most complicated mode, not because it has complex computation, but because it produces floating-valued output such as $\pm 1/\sqrt{2}$, which takes lots of bits to represent. Unlike STBC, the other two modes generate only fixed-valued output such as ± 1 in either real part or imaginary part. In order to have consistency among three modes, we figure out this problem by performing a mapping before data transfer. As shown in Table. 4.2, the mapping skill makes it possible to use only two bits to represent the data in each I/Q channel on each transmit antenna. Since we have four transmit antennas and each antenna as two channels, therefore, an amount of 16 bits are needed to transfer data to FPGA at the same time. Actually, the databus has 32 bits and is sufficient enough to perform the task.

The constellation in frequency domain can be shown in Figure 4.7, where (a), (b), and (c) are the constellation that we impose an AWGN of SNR = 0 dB, 10 dB, 20 dB respectively on the first transmit antenna, and (d) is the constellation of the first receive antenna under an flat fading channel with a sample of condition number 10 at SNR = 20 dB. As in the Figures 4.7 (a), (b), and (c), we adopt QPSK as our modulation scheme, and the constellation disperses more seriously as the SNR increases. In Figure 4.7 (d), since different paths have different scalar, the received constellation looks very dispersive.

Table 4.1: Collection of data sizes in transmitter DSP

	STBC	VBLAST	GSTBC
Info. Bits	32*6	32*8	32*8
(1/3) Conv.	96*6	96*8	96*8
Interleaving	96*6	96*8	96*8
Mapping	48*6	48*8	48*8
ST-Coding	3/4	4	2
Ant. 1	48*8	48*2	48*4
Ant. 2	48*8	48*2	48*4
Ant. 3	48*8	48*2	48*4
Ant. 4	48*8	48*2	48*4

Table 4.2: Mapping table for data transfer

	STBC	VBLAST	GSTBC
+1	01	01	01
-1	11	11	11
$+1/\sqrt{2}$	10	X	X
$-1/\sqrt{2}$	00	X	X

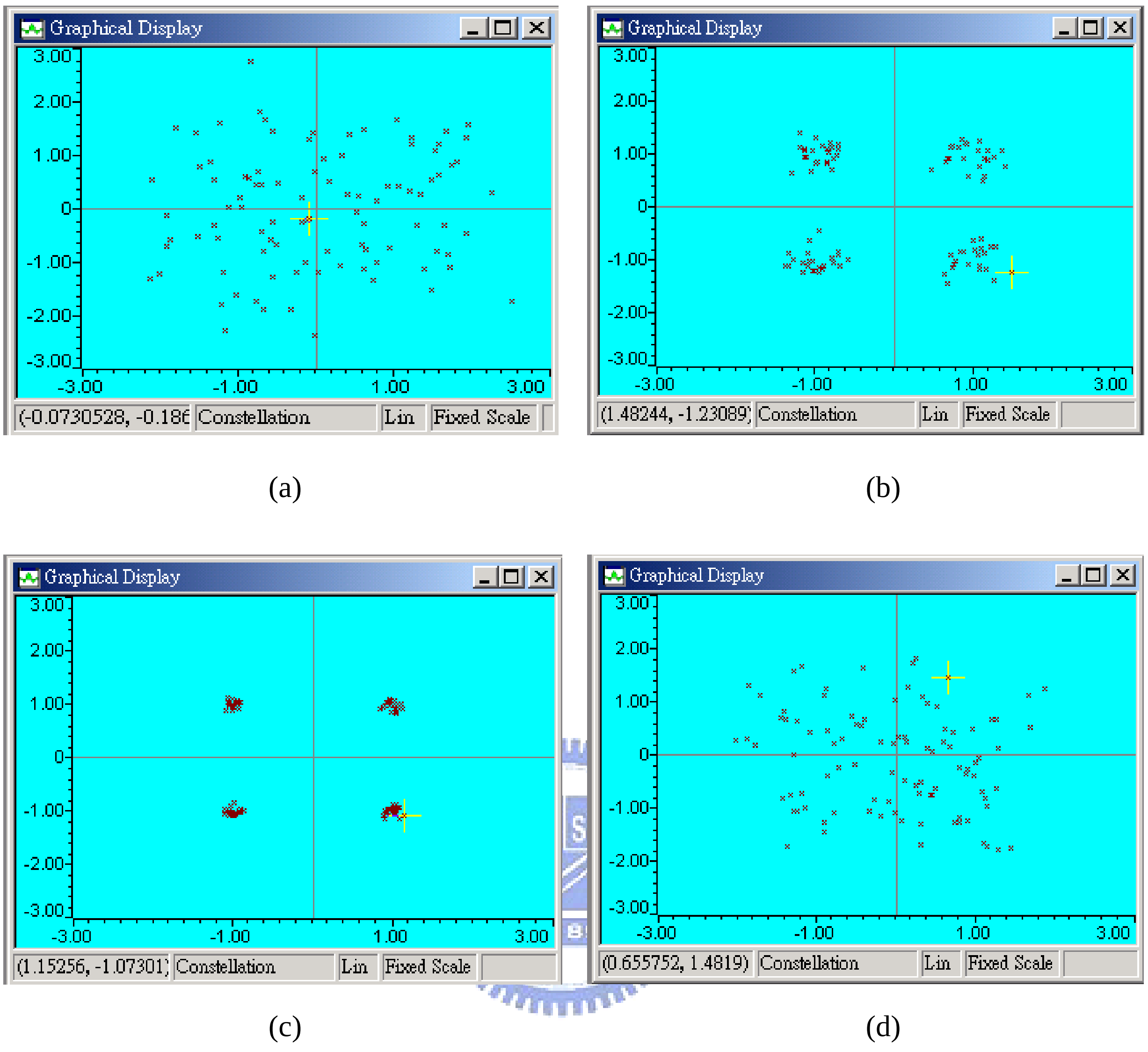


Figure 4.7: Constellation of the first transmit antenna at (a) SNR = 0 dB (b) SNR = 10 dB (c) SNR = 20 dB, and the constellation of first receive antenna at (d) SNR = 20 under a flat fading channel with a sample of condition number 10

4.3 Transmitter on FPGA

In this section, we first introduce the circuit design of the whole transmitter. Since the system is a MIMO system, such that there exists some parallel processing for individual transmit branch, we may duplicate some common part for each antenna to save the time of development. At last, the design of the common part in each transmit branch will be provided in the end of this section.

4.3.1 Circuit Design of Transmitter

As illustrated in Figure 4.8, the transmitter in DSP transfers the space-time processed data to the transmitter in FPGA through a 32-bit databus with **inter2_ptr* handshaking signals including *BDWEN2* and *STRBN2*. The first encountered component in FPGA is the multiplexer named *TX_DATA2FPGA_MUX* used to multiplex the information-carried 16 bits from the 32-bit databus whenever *BDWEN2* and *STRBN2* become low simultaneously. At the same time, each multiplexed signals are sent to the corresponding component to be processed for each transmitter and immediately stored in the input data buffers inside with the address generated by counting the times that *STRBN2* falls. This is because every time data transfer that happens in the direction from DSP to FPGA results in the fall of both handshaking signals, *BDWEN2* and *STRBN2*, while nothing happens except the fall of *BDWEN2* in the opposite direction. After a series of processing in each transmitter block, from *TX1* to *TX4*, data ready to be transmitted are passed to an eight-channeled DAC which is prepared for up-conversion to the appropriate RF band if needed. Later, we will give a more detailed circuit design for each transmit antenna.

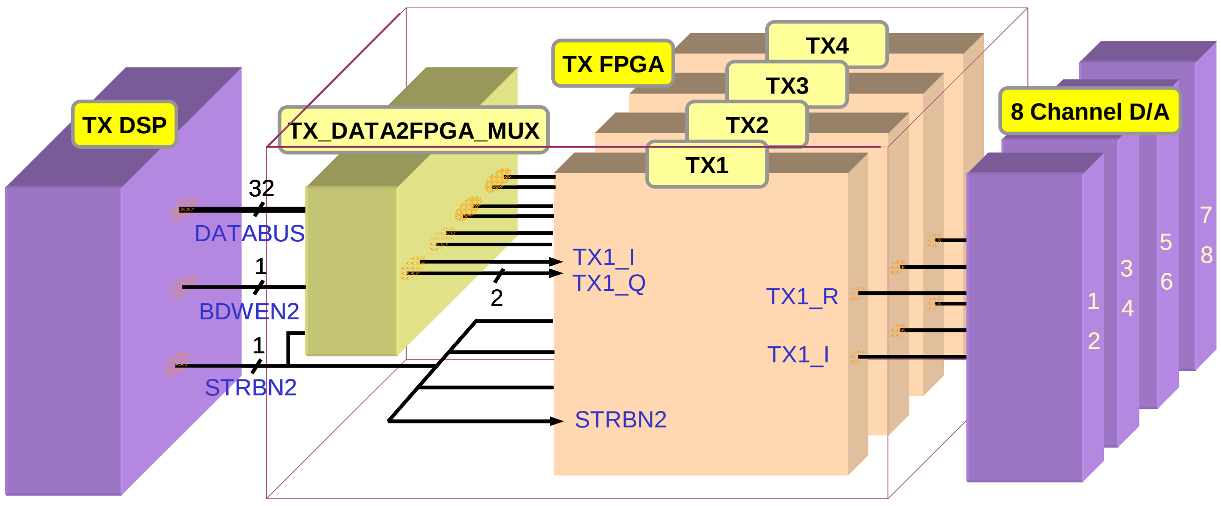


Figure 4.8: Circuit design of transmitter in FPGA

4.3.2 Circuit Design of Each Transmit Branch

Figure 4.9 shows the circuit design of each transmit antenna, which is mainly composed of an input data buffer, an IFFT block, an output data buffer, a ROM, a multiplexer, and two RRC blocks. The other blocks also play an important role in completing the task. First, the strobe signal *STRBN2* with its fall state lasting for several clocks is trimmed to one clock by block *tx_strnc_gen*, since the following *tx_data_ctrl* block counts the clocks that trimmed *STRBN2* lasts as the writing address of input data buffer *D_CH*. After collecting all the data, *tx_data_ctrl* sends a reset signal to *data_ctrl* block so as to start the rest circuit up. The *data_ctrl* mainly performs the job of assembling OFDM frames including the insertion of pilot tones and zero tones, and by means of modifying the read address appropriately, this job can be performed while data are read to next stage. Undoubtedly, IFFT is the most time-consuming block in the development stage owing to its high complexity. Since IFFT induces a delay of 26 clocks from the incoming of the last bit to the outgoing of the first bit in a symbol of 64 bits, we use *data_delay* and *ctrl_delay* blocks to adjust the arrival timing of IFFT output and control signals for writing data to *data_buffer* block. Actually, the IFFT we designed is a bit-reversed version of its output, implying that we have to rearrange the order at the time of writing to data buffer, which

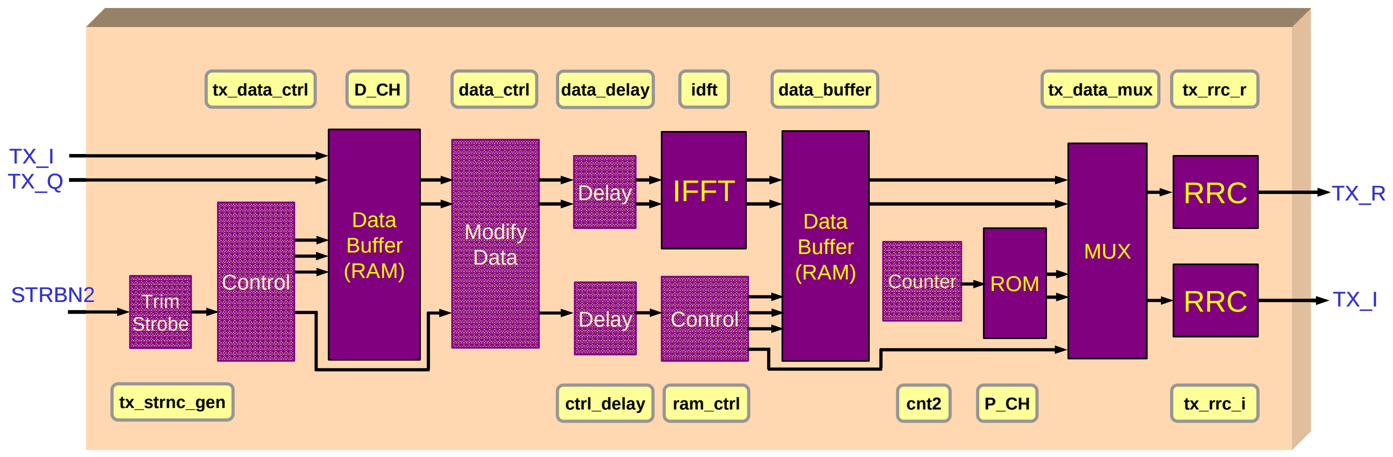


Figure 4.9: Circuit design of each transmit branch

somewhat makes *ram_ctrl* more complicated. While reading data from *data_buffer*, instead of simply increasing the address, we modify the address as illustrated in Figure 4.10 to read out the data with a cyclic prefix attached in the front of each symbol. Then the preamble data are read from a pre-defined ROM *P_CH* by a modified counter *cnt2* that can read the short preamble ten times and the cyclic-prefixed long preamble four times. Through the multiplexer *tx_data_mux*, preamble channels and data channels are combined together to form a complete OFDM packet. As mentioned in Section 2.2 and Section 4.1.1, we use the rule of STBC on short preambles and use the identity matrix as structure matrix on long preambles, which are both performed in the same multiplexer, *tx_data_mux*. At last, two RRC blocks are processed for individual I/Q channels, and then data are transmitted to the corresponding DAC for each transmit antenna.

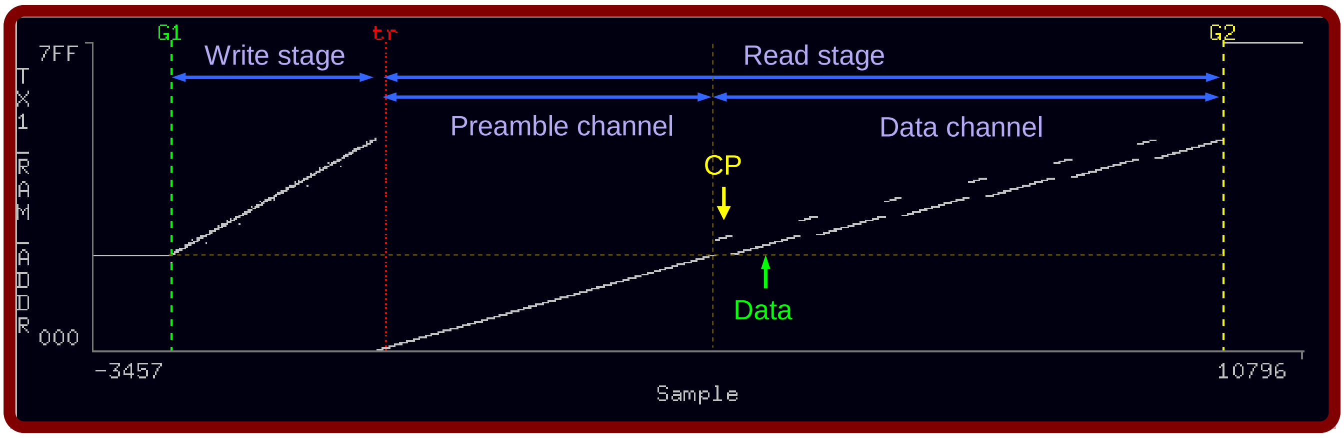


Figure 4.10: Address values of *ram_ctrl* block

4.4 Receiver on FPGA

At the receiver side, we first give a whole picture of the receiver on FPGA, which is partitioned into several parts. Among them, the RRC blocks and timing synchronization block which gives the information of when the packet starts are then provided. Later, we will give the circuit design of each processing block for two receive antennas. Finally, the rest components like the multiplexer, output buffers, and Viterbi decoder will be presented.

4.4.1 Circuit Design of Receiver

As we can see in Figure 4.11, four pairs of received channels are passed to the receiver FPGA from an eight-channeled ADC. These data are first fed to eight RRC blocks to perform waveform shaping, and then the output are matched with short preambles, delayed and summed, and filtered. By a series of processing, the maximum output is selected to find out when the packet starts by delaying a fixed time. After that the signal *FRM_STR* containing timing information is sent to the major processing blocks, *RX12* and *RX34*, in a form of enable signal so as to start the following components up. Each major processing block uses two pairs of RRC-processed signals as their inputs *RX1_I/Q* and *RX2_I/Q*, taking *RX12* for example, and combines them into one pairs *RX12_I* and *RX12_Q* as its output. The reason for combining is to reduce the number of FFT blocks which is the most gates-consuming part of all. By combination, a total of four parallel FFT processing can be shrunk to two parallel ones but twice longer data are processed than before, which implies we trade complexity for efficiency. Then *RAM_ENB* are used to inform the *RX_REST* block to enable the inside output buffer to store the incoming data. Finishing collecting and merging the data comes from previous stage and the output of *VITERBI_DECODER*, *RX_REST* uses *BDWEN0* and *STRBN0* to transfer data to receiver DSP whereas the input of *VITERBI_DECODER* is fed in previous packet as mentioned in section 4.1.2.

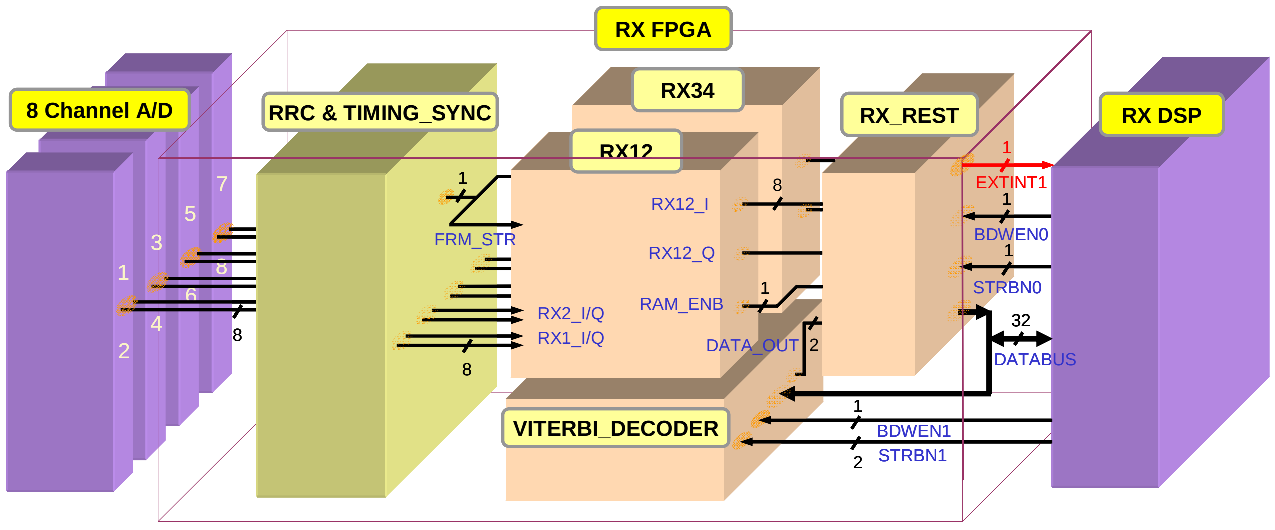


Figure 4.11: Circuit design of receiver in FPGA

4.4.2 RRC and Time Synchronization

Figure 4.12 shows the circuit design of RRC waveform shaping filter and time synchronization block. Again we use the concept of instantiation in hierarchical design to save the time of development, as we can see in the left hand side where two major pieces of identical processing blocks for each two receive antennas are instantiated from the same design. First, the incoming data which is 10-bits wide from ADC are truncated to 8 bits and passed to individual RRC blocks. After that, in order to do two-time downsampling, we use *switch* blocks to generate clock sources two time faster than the FPGA clock and perform the job in the followed matched filter *MF* which uses the generated clock as its working clock. Then a series of comparison are processed to find out the maximum absolute value among 16 paths at each time index. After collecting a data sequence that has the maximum absolute values, it is delayed by 16 clocks and summed up to enhance the peak values by the *rx_delay_sum* block where sixteen is the length of a short preamble and is also the expected distance between two adjacent peaks since they are matched with short preamble in the previous stage. Later, an FIR filter with response of some repeated {0,0,…,0,1} is applied to rake the values on each peaks, so that we can obtain a series of ascending values which become descending after a while as illustrated in Figure 4.13. Thus, we use the block *rx_select_max* to find out where the maximum value locates and

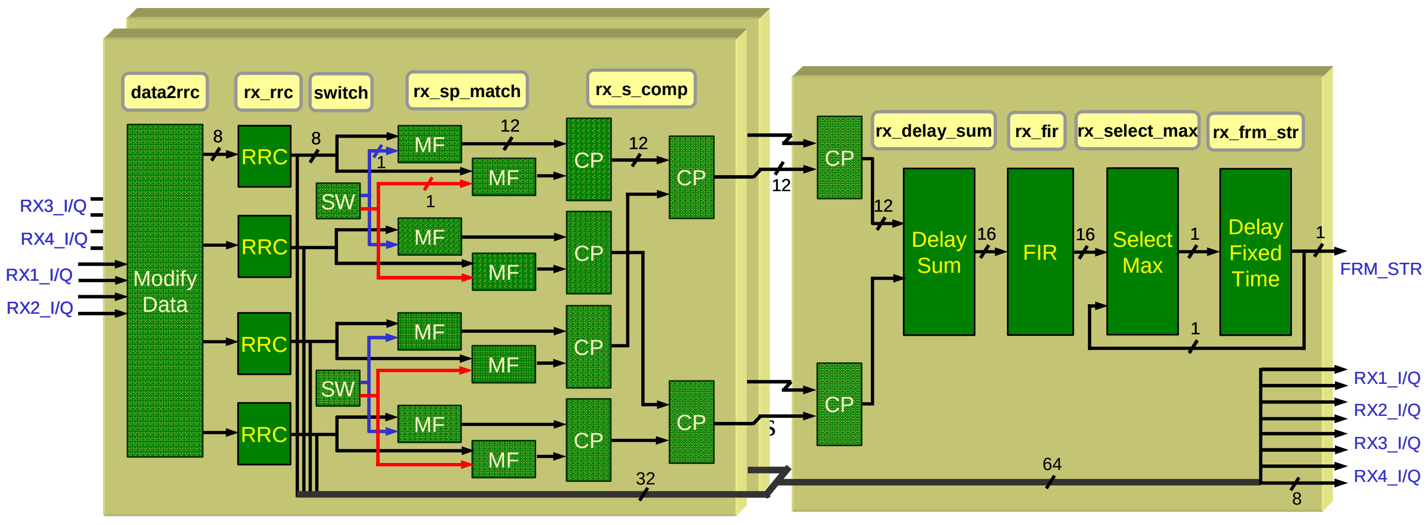


Figure 4.12: Circuit design of RRC and time synchronization

the following block *rx_frm_str* is used to delay a fixed number of clocks. The fixed numbers of clocks can be calculated, since we have the information of the processing period (clocks) of a packet (frame) and the offset between the frame start and the place that maximum value would appear. Therefore, the start of the next frame can be determined simply by delaying a fixed time, "frame time - offset", after we know the place that peak value appears. Finally, the timing information is carried by *FRM_STR* in a form of enable signal to start up the later circuit, and it also feeds back as an enable signal of the previous block *rx_select_max* to avoid selecting maximum values at the wrong time.

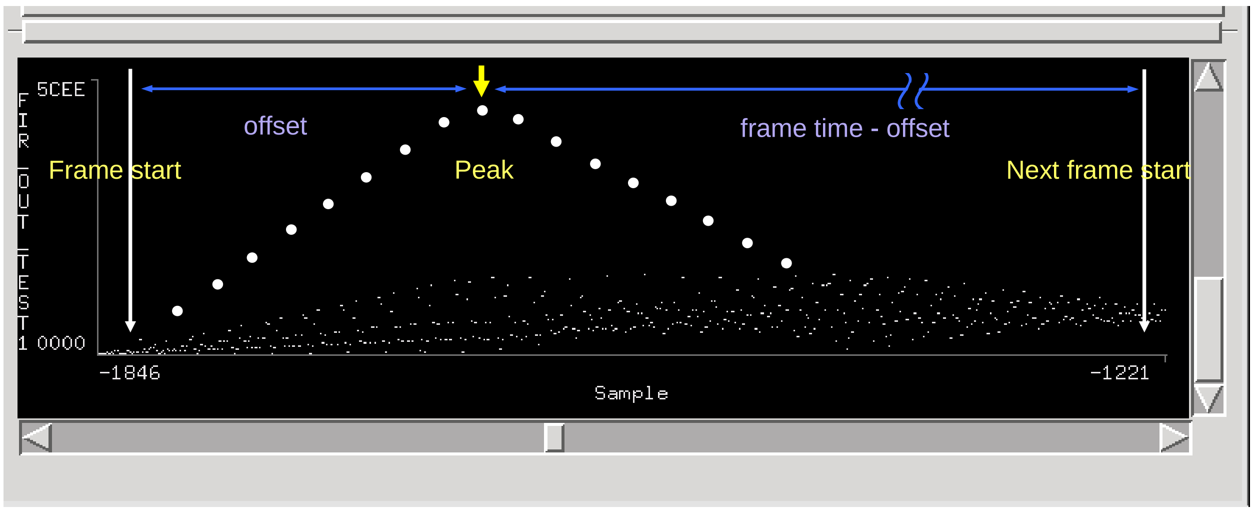


Figure 4.13: Concept of time synchronization method

4.4.3 Group of Two Receive Antennas and Other Components

After having the information about the start of frame, as illustrated in Figure 4.14, the *rx_databuf_ctrl* can generate the control signals and writing address to *rx_data_buffer* in a right time where the writing address is also used to perform the removal of cyclic prefix and the job of downsampling. Then, before FFT processing, two branches of data from different receive antennas are merged into one branch by *data2fft_mux* in the way of one attached after the other. Therefore, passed in the same data bus, data from the second antenna will be sent after that from the first antenna, which can save us a large amount of logic-gates for only one FFT block is required in each group of two receive antennas. Since the control block *rx_databuf_ctrl* can expect when the first output data from FFT will pass through the multiplexer and arrive at output data buffer *ram2dsp*, the notifying signal *RAM_ENB* will be sent at a proper time. After the block *data_mux4* multiplexes the data from two major processing blocks, *RX_12* and *RX_34*, the merged data are written into output data buffer by the complicated address that combines bit-reversal, discard of zero tones, and rearrangement of pilot tones, which is generated by *data2dsp_ctrl*. In order to let FFT-processed data and the output of Viterbi decoder travel back by the same databus simultaneously, we have to store the Viterbi output *DATA_OUT* in the data buffer *dual_ram* whenever the *READY_OUT* signal triggers the control block *dual_ram_ctrl*. Once the FFT-processed data ends its collection, *EXTINT1* will be sent to wake up the receiver interrupt function in DSP so as to start the data transfer from FPGA to DSP with handshake signals, *BDWEN0* and *STRBN0*. Therefore, the fall of voltage level on handshake signals will trigger the control block *data2dsp_ctrl* to read out the data from both of the RAMs, *ram2dsp* and *dual_ram*, and also trigger the multiplexer *data2dsp_mux* to combine the data that FFT processes and the output bit stream of Viterbi decoder. Notice that the job we perform in multiplexer *data2dsp_mux* is to merge a 32-bit databus and a 1-bit signal into a 32-bit databus, which implies we have

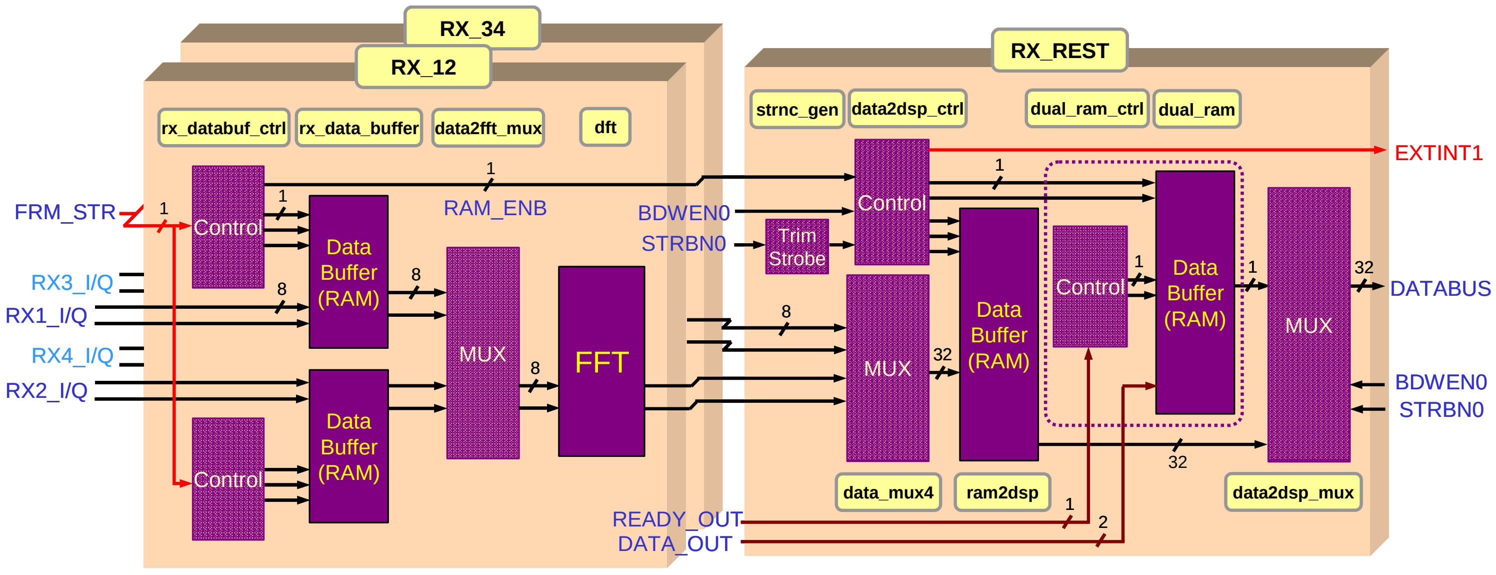


Figure 4.14: Circuit design of RRC and time synchronization

to ignore the source databus one bit out of 32 so that we can further imbed the 1-bit signal source in it. Undoubtedly, although the step will induce a little drop of performance, we can save the extra time required to transfer the output of Viterbi decoder.

Figure 4.15 is the Viterbi decoder and its related circuits. As mentioned in Section 4.1.2, we adopt handshake signal *STRBN1* for DSP to inform FPGA to receive the data after the process of de-interleaver and waiting for the process of channel decoder, Viterbi decoder. We can notice that dual-port-RAMs are used here and in the right hand side of Figure 4.14, which allow multiple reads or writes to occur at the same time, unlike single-ported RAM which only allows one access at a time. The control block *dual_ram_ctrl1* mainly handles the job of writing data to *dual_ram_1024x3* while *dual_ram_ctrl2* mainly performs the task of reading data to Viterbi decoder and preparing the required control signals for it. After a while, the decoded data *DATA_OUT* and notifying signal *READY_OUT* will be sent to the block *dual_ram* and *dual_ram_ctrl* in the previous figure to be stored and wait for the data transferred from FPGA to DSP in the next packet. Since the design of Viterbi decoder is an intellectual property (IP) developed from our laboratory, only the related circuits are shown as in Figure 4.15.

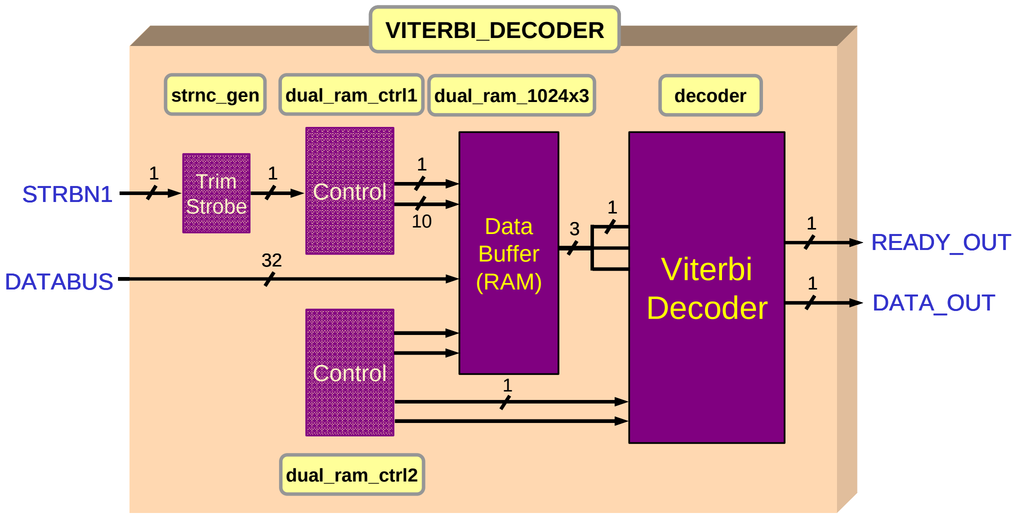


Figure 4.15: Viterbi decoder and its related circuits

4.5 Receiver on DSP

The major tasks of receiver in DSP are shown in Figure 4.16. The data received from FPGA forms a 52×12 matrix, where 52 is composed of four pilot tones in the top and 48 data tones in the rest part, and 12 consists of four long preambles and eight data symbols as illustrated in Figure 4.17. Then a series of estimation which has been introduced in Chapter 2 are processed including channel estimation, phase estimation, and frequency estimation with the aids of long preambles and pilot tones provided from the above-mentioned received matrix. After that, the data are space-time decoded to a fix-numbered complex symbol according to the mode it adopts at the transmitter, which is also the most time-consuming stage for its high complexity, and we will give a more detailed introduction of their decoding algorithms in the following sections. Later, the de-mapper transforms the data from a complex symbol into two simple bits, and after

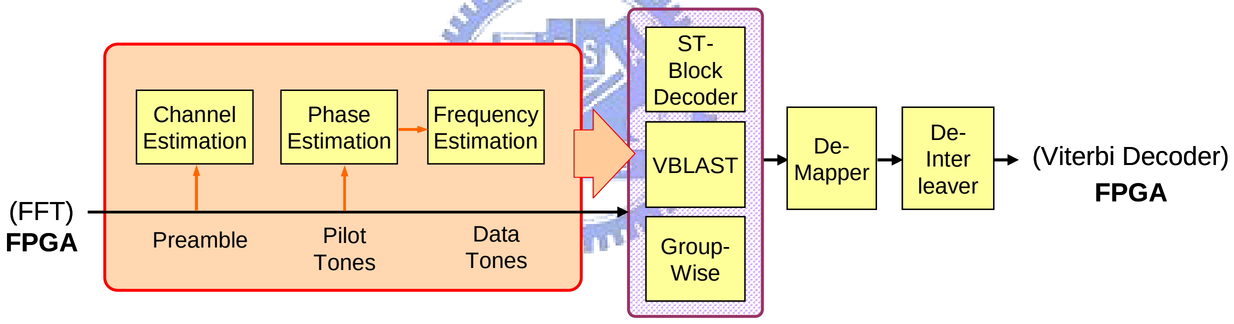


Figure 4.16: Diagram of receiver in DSP

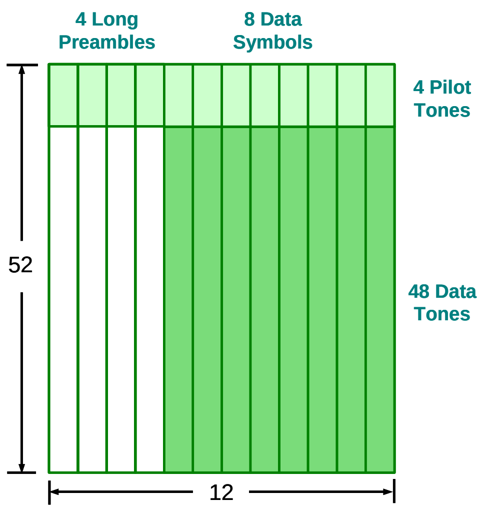


Figure 4.17: Received data from receiver FPGA

that the de-interleaver is performed to rearrange the data in a specific way. Finally, because the Viterbi decoder is located in FPGA, we need to transfer data back to FPGA again to be decoded.

4.5.1 Space-Time Block Decoding

The decoding procedure of space-time block code for four transmit antennas [17] is mainly to rearrange the estimated channel matrix in a specific method, and then to sum up the multiplication results with both the original received signals and the complex conjugated version. The signal model can be expressed as follows:

$$\widehat{\mathbf{A}}\mathbf{Z} + \widehat{\mathbf{B}}\mathbf{Z}^* = \widehat{\mathbf{S}}$$

where $\widehat{\mathbf{A}}$ and $\widehat{\mathbf{B}}$ is the rearranged channel matrix estimated at the receiver, $\mathbf{Z}$ is the received data matrix, and $\widehat{\mathbf{S}}$ is the combined result for decision making. The rearranged channel matrix can be further rewritten as follows:

$$\widehat{\mathbf{A}} = \begin{bmatrix} \mathbf{A}_1 & \mathbf{A}_2 & \mathbf{A}_3 & \mathbf{A}_4 \end{bmatrix}$$

$$\widehat{\mathbf{B}} = \begin{bmatrix} \mathbf{B}_1 & \mathbf{B}_2 & \mathbf{B}_3 & \mathbf{B}_4 \end{bmatrix}$$

where $\mathbf{A}_i$ and $\mathbf{B}_i$ are

$$\mathbf{A}_i = \begin{bmatrix} H_{1i} & 0 & -(H_{3i} - H_{4i})/2 & (H_{3i} - H_{4i})/2 \\ H_{2i} & 0 & (H_{3i} - H_{4i})/2 & (H_{3i} - H_{4i})/2 \\ (H_{3i} + H_{4i})/\sqrt{2} & (H_{3i} - H_{4i})/\sqrt{2} & 0 & 0 \end{bmatrix}^*$$

$$\mathbf{B}_i = \begin{bmatrix} 0 & H_{2i} & -(H_{3i} + H_{4i})/2 & -(H_{3i} + H_{4i})/2 \\ 0 & -H_{1i} & -(H_{3i} + H_{4i})/2 & (H_{3i} + H_{4i})/2 \\ 0 & 0 & (H_{1i} + H_{2i})/\sqrt{2} & (H_{1i} - H_{2i})/\sqrt{2} \end{bmatrix}$$

where H_{ij} is the channel response between *i*th transmitter and *j*th receiver obtained from the channel estimator in the same subcarrier. The received signal $\mathbf{Z}$ can usually be written in the following form:

$$\mathbf{Z} = \begin{bmatrix} \mathbf{z}_1 & \mathbf{z}_2 & \mathbf{z}_3 & \mathbf{z}_4 \end{bmatrix}^T$$

where $\mathbf{z}_i = \begin{bmatrix} z_{i1} & z_{i2} & z_{i3} & z_{i4} \end{bmatrix}^T$, and z_{ij} is the *i*th data received from the *j*th receiver. By the above method, we can obtain a 3×1 combined vector $\hat{\mathbf{S}}$ so that we can decide whether the decoded data of either real part of imaginary part should be 1 or -1 simply by setting the threshold of zero.

4.5.2 V-BLAST Decoding

As mentioned in Chapter 2, for the computational complexity is always our major concern rather than performance, we only use pure nulling method to perform the decoding at the receiver without incorporating with symbol cancellation which requires higher complexity and more hardware costs. The pure nulling method can be summarized as follows.

Assume we can obtain the received signals from four receive antennas and each of them can be expressed as

$$\mathbf{z}_i(n) = \{H_{1i}d_1(n) + H_{2i}d_2(n) + H_{3i}d_3(n) + H_{4i}d_4(n)\}e^{j\phi_{n,i}}$$

We can further combine all of them in a form of matrix as follows

$$\begin{bmatrix} \mathbf{z}_1(n)e^{-j\phi_{n,1}} \\ \mathbf{z}_2(n)e^{-j\phi_{n,2}} \\ \mathbf{z}_3(n)e^{-j\phi_{n,3}} \\ \mathbf{z}_4(n)e^{-j\phi_{n,4}} \end{bmatrix} = \begin{bmatrix} H_{11} & H_{21} & H_{31} & H_{41} \\ H_{12} & H_{22} & H_{32} & H_{42} \\ H_{13} & H_{23} & H_{33} & H_{43} \\ H_{14} & H_{24} & H_{34} & H_{44} \end{bmatrix} \begin{bmatrix} d_1(n) \\ d_2(n) \\ d_3(n) \\ d_4(n) \end{bmatrix}$$

where $d_i(n)$ is the data transmitted from the *i*th transmitter in *n*th time slot of a block, $e^{-j\phi_{n,i}}$ is the phase shift affected on *i*th receive antenna in *n*th time slot of a block, and $\mathbf{z}_i(n)e^{-j\phi_{n,i}}$ means the phase-compensated received data.

Then in order to detect the data, we have to find out the linear combining weight as shown in the following equation:

$$\begin{bmatrix} d_1(n) \\ d_2(n) \\ d_3(n) \\ d_4(n) \end{bmatrix} = \mathbf{W}^{-1} \begin{bmatrix} \mathbf{z}_1(n)e^{-j\phi_{n,1}} \\ \mathbf{z}_2(n)e^{-j\phi_{n,2}} \\ \mathbf{z}_3(n)e^{-j\phi_{n,3}} \\ \mathbf{z}_4(n)e^{-j\phi_{n,4}} \end{bmatrix}$$

where $\mathbf{W}$ is the linear combining weight. In general, the weight can be obtained by two kinds of criterion, which are ZF and MMSE. The corresponding solutions are listed as follows:

$$\mathbf{W}_{ZF} = \begin{bmatrix} \widehat{H}_{11} & \widehat{H}_{21} & \widehat{H}_{31} & \widehat{H}_{41} \\ \widehat{H}_{12} & \widehat{H}_{22} & \widehat{H}_{32} & \widehat{H}_{42} \\ \widehat{H}_{13} & \widehat{H}_{23} & \widehat{H}_{33} & \widehat{H}_{43} \\ \widehat{H}_{14} & \widehat{H}_{24} & \widehat{H}_{34} & \widehat{H}_{44} \end{bmatrix}^{\dagger} = \widehat{\mathbf{H}}^{\dagger}$$

$$\mathbf{W}_{MMSE} = \left\{ \widehat{\mathbf{H}}\widehat{\mathbf{H}}^{H} + \begin{bmatrix} \sigma_1^2 & 0 & 0 & 0 \\ 0 & \sigma_2^2 & 0 & 0 \\ 0 & 0 & \sigma_3^2 & 0 \\ 0 & 0 & 0 & \sigma_4^2 \end{bmatrix} \right\}^{\dagger} \widehat{\mathbf{H}}^{H}$$

where $\widehat{\mathbf{H}}^{\dagger}$ is the pseudo-inverse of estimated channel matrix, and σ_i^2 is the noise power of *i*th receive antenna. For MMSE has higher complexity than ZF, we only adopt the solution according to ZF criterion. In addition, since there is no closed-form for the matrix inversion of a 4×4 matrix, we use Gaussian-Jordan Elimination method to solve the matrix inversion problem in our implementation.

4.5.3 G-STBC Decoding

The four transmit antennas in our system are partitioned into two groups, with each group containing two antennas and transmitting independent data streams encoded by the Alamouti STBC individually. At the receiver, the decoding scheme of G-STBC is somewhat like a cascaded version of Alamouti STBC decoding scheme.

The decoding procedure [38] is as follows. We first write the received data in the 4×4 Alamouti G-STBC system as

$$\begin{bmatrix} r_{1,1} & r_{1,2} \\ r_{2,1} & r_{2,2} \\ r_{3,1} & r_{3,2} \\ r_{4,1} & r_{4,2} \end{bmatrix} = \begin{bmatrix} H_{11} & H_{21} & H_{31} & H_{41} \\ H_{12} & H_{22} & H_{32} & H_{42} \\ H_{13} & H_{23} & H_{33} & H_{43} \\ H_{14} & H_{24} & H_{34} & H_{44} \end{bmatrix} \begin{bmatrix} x_1 & x_2^* \\ x_2 & -x_1^* \\ x_3 & x_4^* \\ x_4 & -x_3^* \end{bmatrix}$$

where $r_{i,j}$ is the received data from the *i*th receive antenna in *j*th time slot of a block, and x_i is the data symbol to be encoded. Then rearranging the signal elements, we can have the following representation for the received data $\mathbf{r}_{12}$ on the first two receive antennas.

$$\mathbf{r}_{12} \overset{\text{def}}{=} \begin{bmatrix} r_{1,1} \\ -r_{1,2}^* \\ r_{2,1} \\ -r_{2,2}^* \end{bmatrix} = \underbrace{\begin{bmatrix} H_{11} & H_{12} & H_{13} & H_{14} \\ -H_{12}^* & H_{11}^* & -H_{14}^* & H_{13}^* \\ H_{21} & H_{22} & H_{23} & H_{24} \\ -H_{22}^* & H_{21}^* & -H_{24}^* & H_{23}^* \end{bmatrix}}_{\mathbf{H}} \underbrace{\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}}_{\mathbf{x}} = \begin{bmatrix} \mathbf{T}_{1,1} & \mathbf{T}_{1,2} \\ \mathbf{T}_{2,1} & \mathbf{T}_{2,2} \end{bmatrix} \mathbf{x}$$

where

$$\mathbf{T}_{1,1} = \begin{bmatrix} H_{11} & H_{12} \\ -H_{12}^* & H_{11}^* \end{bmatrix}, \quad \mathbf{T}_{1,2} = \begin{bmatrix} H_{13} & H_{14} \\ -H_{14}^* & H_{13}^* \end{bmatrix},$$

$$\mathbf{T}_{2,1} = \begin{bmatrix} H_{21} & H_{22} \\ -H_{22}^* & H_{21}^* \end{bmatrix}, \quad \mathbf{T}_{2,2} = \begin{bmatrix} H_{23} & H_{24} \\ -H_{24}^* & H_{23}^* \end{bmatrix}$$

It can be shown that

$$\mathbf{T}_{i,j}^H \mathbf{T}_{i,j} = \mathbf{T}_{i,j} \mathbf{T}_{i,j}^H = t_{i,j} \mathbf{I}_2, \qquad i, j = 1, 2,$$

where $t_{i,j} = \det(\mathbf{T}_{i,j})$ and $\mathbf{I}_2$ denotes the identity matrix of size 2. On the other hand, we can also obtain the same representation for the received data $\mathbf{r}_{34}$ on the last two receive antennas.

At the receiver, we adopt the zero-forcing interference cancellation (ZFIS) method to perform the job of decoding. Applying the following linear matrix filtering operation on $\mathbf{r}_{12}$ and $\mathbf{r}_{34}$, we have

$$\tilde{\mathbf{r}}_{12} = \begin{bmatrix} \mathbf{I}_2 & -\mathbf{T}_{1,2}\mathbf{T}_{2,2}^{-1} \\ -\mathbf{T}_{2,1}\mathbf{T}_{1,1}^{-1} & \mathbf{I}_2 \end{bmatrix} \mathbf{r}_{12} = \begin{bmatrix} \tilde{\mathbf{G}}_1\mathbf{x}_1 \\ \tilde{\mathbf{G}}_2\mathbf{x}_2 \end{bmatrix},$$

$$\tilde{\mathbf{r}}_{34} = \begin{bmatrix} \mathbf{I}_2 & -\mathbf{T}_{1,2}\mathbf{T}_{2,2}^{-1} \\ -\mathbf{T}_{2,1}\mathbf{T}_{1,1}^{-1} & \mathbf{I}_2 \end{bmatrix} \mathbf{r}_{34} = \begin{bmatrix} \tilde{\mathbf{G}}_1\mathbf{x}_1 \\ \tilde{\mathbf{G}}_2\mathbf{x}_2 \end{bmatrix}$$

where

$$\mathbf{x}_1 = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \ \mathbf{x}_2 = \begin{bmatrix} x_3 \\ x_4 \end{bmatrix},$$

$$\tilde{\mathbf{G}}_1 = \mathbf{T}_{1,1} - \mathbf{T}_{1,2}\mathbf{T}_{2,2}^{-1}\mathbf{T}_{2,1}, \ \tilde{\mathbf{G}}_2 = \mathbf{T}_{2,2} - \mathbf{T}_{2,1}\mathbf{T}_{1,1}^{-1}\mathbf{T}_{1,2}$$

The above linear matrix filtering operation has separated the two groups of transmitted signals completely. $\tilde{\mathbf{G}}_i$'s are orthogonal matrices that satisfy

$$\tilde{\mathbf{G}}_i\tilde{\mathbf{G}}_i^H = \tilde{\mathbf{G}}_i^H\tilde{\mathbf{G}}_i = g_i\mathbf{I}_2, \quad g_i = \det(\tilde{\mathbf{G}}_i), \quad i = 1,2.$$

Thus, applying the linear matrix filter

$$\begin{bmatrix} \tilde{\mathbf{G}}_1^H & \mathbf{0}_2 \\ \mathbf{0}_2 & \tilde{\mathbf{G}}_2^H \end{bmatrix}$$

onto $\mathbf{r}_{12}$ and $\mathbf{r}_{34}$, we will have the following decision statistics:

$$\tilde{\mathbf{x}}_{\text{ZFIS}} = \begin{bmatrix} (g_1^{(12)} + g_1^{(34)})\mathbf{x}_1 \\ (g_2^{(12)} + g_2^{(34)})\mathbf{x}_2 \end{bmatrix}$$

where the $g_i^{(12)}$ and $g_i^{(34)}$ denote the determine of $\tilde{\mathbf{G}}_i$ from first two or last two receive antennas. Notice that we can obtain a diversity gain of 2 which comes from two groups of antenna shown in the above equation.

4.6 Experimental Result of Adaptive Mechanism

At mentioned in Section 2.5.3, the adaptive mechanism (i.e. mode selection scheme) is based on the throughput performance in a form of tables for looking up under different channel conditions. Before building up the table, the throughput performance can be obtained by incorporating BER performance over different condition numbers with transmitted information bits and transmission time, and the expression is shown as follows:

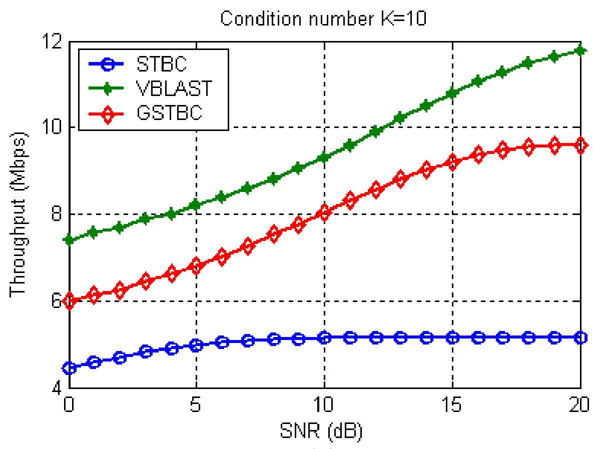
$$\text{Throughput} = \frac{\text{Successful Bits}}{\text{TransmissionTime}}(\text{b/s})$$

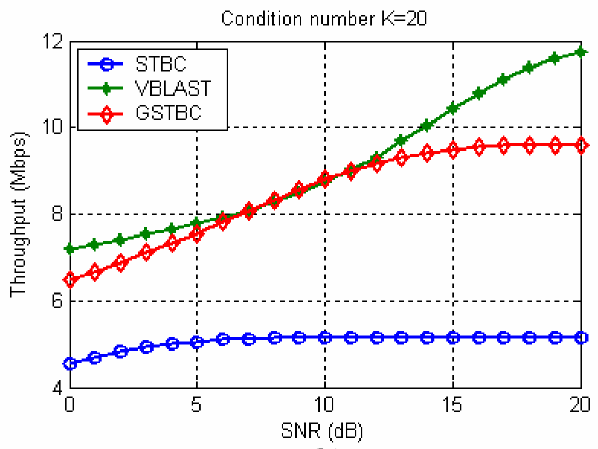
$$= \frac{\text{(1-BER)Info. Bits}}{\text{TransmissionTime}} = \text{(1-BER)ratio}$$

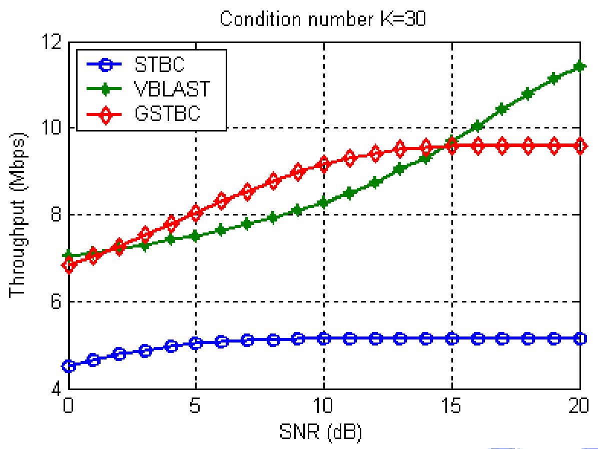
where the successful bits can be derived by BER and total transmitted information bits, and the detailed derivation for the ratio of each mode is shown in Table 4.3. After introducing the calculation of throughput, we now show the experimental results simulated with different condition numbers K; they are 10, 20, 30, 40, 50, 100, 200, 400, 1000, and 10^5. As shown in Figure 4.18, the STBC has a stable and the lowest throughput compared with others throughout all the condition numbers, which implies STBC is the most resistant scheme to channel correlations among all and has the lowest ratio (maximum throughput) listed in Table 4.3; G-STBC also has a similar behavior to STBC because these two schemes can offer some diversity gains to combat with the channel correlations; the performance of VBLAST degrades as the condition number increases, implying that VBLAST is the most sensitive scheme to channel correlations. This is because the correlation would make the split of different data streams at the receiver more difficult. The previously-mentioned ratios (maximum throughput) for each mode are 5.14, 12, and 9.6 for STBC, VBLAST, and G-STBC, respectively. In high correlation channels, especially those larger than 1000, we notice that although

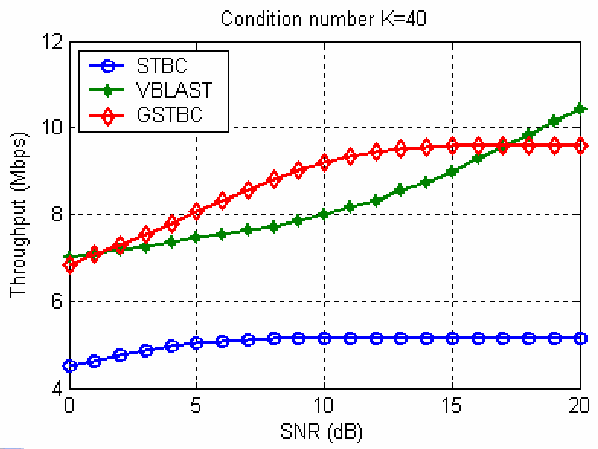
Table 4.3: Derivation of the ratio for each mode

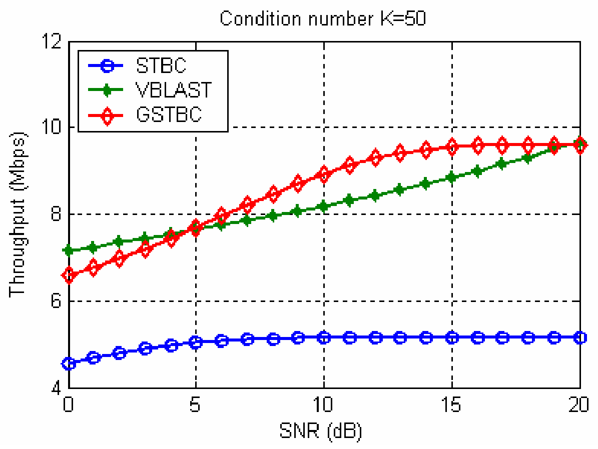
	STBC	VBLAST	GSTBC	Unit
Data Tones / (TX Ant.*Symbol)	48	48	48	complex
Spactial Streams	1	4	2	no.
ST Code Rate	3/4	1	1	no.
Transmitted Symbols	8	2	4	no.
Info. Bits	288	384	384	real (bit)
Short Preamble NO.	10	10	10	no.
Short Preamble Time	0.8	0.8	0.8	us
Long Preamble NO.	4	4	4	no.
Long Preamble Time	4	4	4	us
Symbol NO./ TX Ant	8	2	4	no.
Symbol Time	4	4	4	us
Transmission Time	56	32	40	us
Ratio	5.142857	12	9.6	Mbps

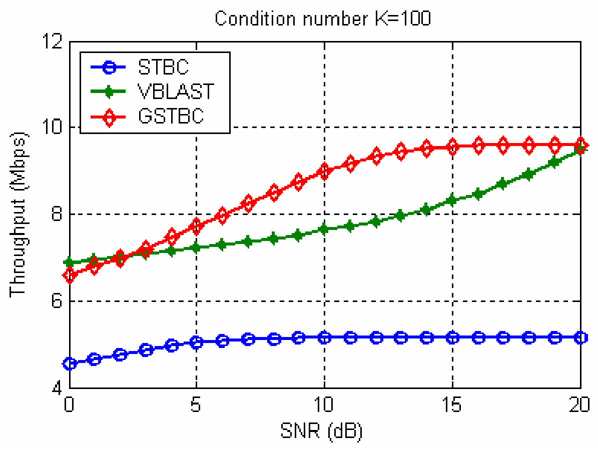

Condition number K=10
STBC
VBLAST
GSTBC
Throughput (Mbps)
SNR (dB)
(a)

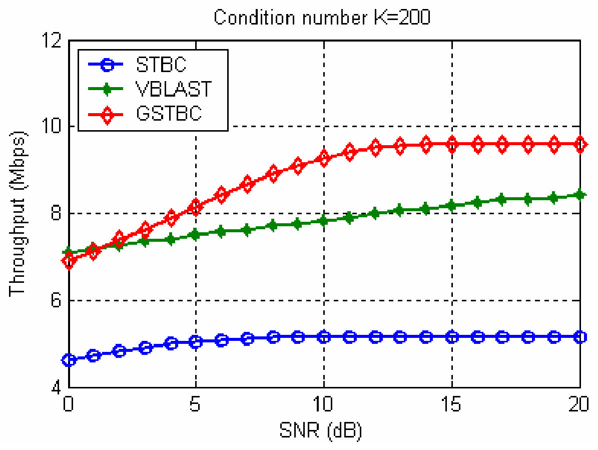

Condition number K=20
STBC
VBLAST
GSTBC
Throughput (Mbps)
SNR (dB)
(b)

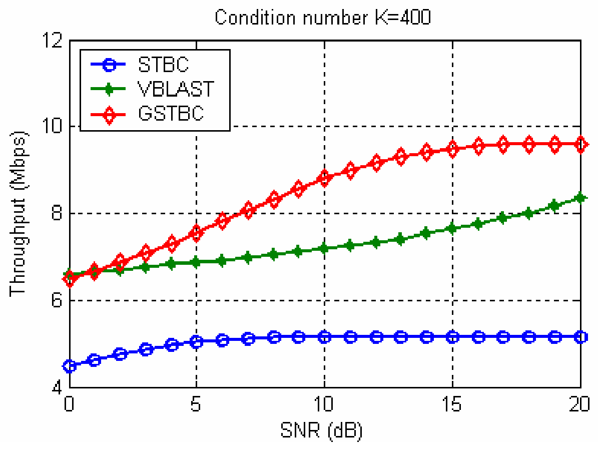

Condition number K=30
STBC
VBLAST
GSTBC
Throughput (Mbps)
SNR (dB)
(c)

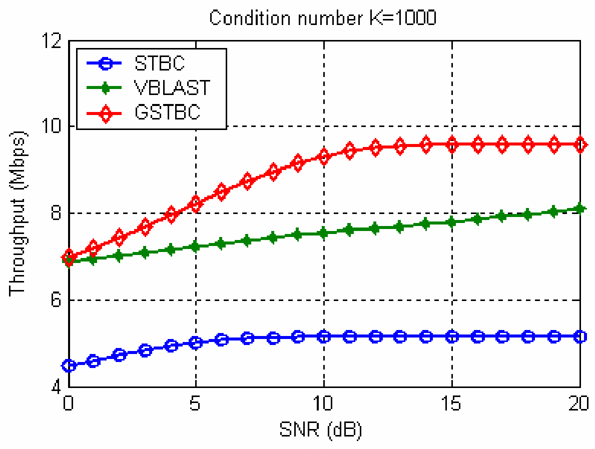

Condition number K=40
STBC
VBLAST
GSTBC
Throughput (Mbps)
SNR (dB)
(d)


Condition number K=50
STBC
VBLAST
GSTBC
Throughput (Mbps)
SNR (dB)
(e)

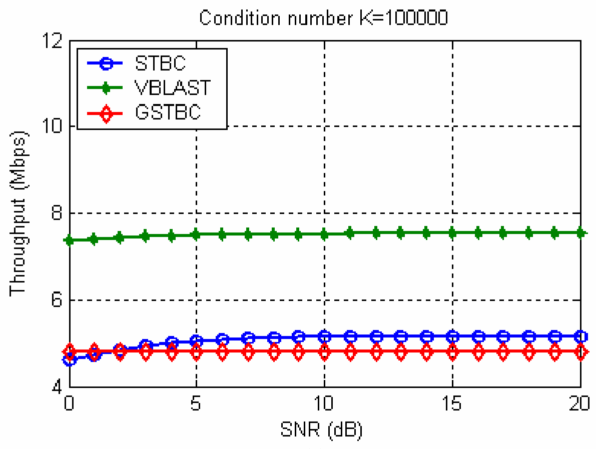

Condition number K=100
STBC
VBLAST
GSTBC
Throughput (Mbps)
SNR (dB)
(f)


Condition number K=200
STBC
VBLAST
GSTBC
Throughput (Mbps)
SNR (dB)
(g)


Condition number K=400
STBC
VBLAST
GSTBC
Throughput (Mbps)
SNR (dB)
(h)



(i)



(j)

Figure 4.18: Throughput performances under different channel conditions
(a) K =10 (b) K =20 (c) K =30 (d) K =40 (e) K =50 (f) K =100
(g) K =200 (h) K =400 (i) K =1,000 (j) K =100,000

VBLAST has the highest throughput, it saturates at a value of 7.5, which implies errors occur in almost half the data, while STBC still performs near its maximum throughput, 5.14. Therefore, we will suggest adopting the STBC when detecting a channel with the channel condition larger than 1000. In low correlation channels, although sometimes VBLAST has higher throughput than G-STBC, we still adopt G-STBC for the same reason as in high correlation channels. Further, we organize the previous figures into Table 4.4 for looking-up-table.

The selection strategy is summarized as follows

1. Determine the channel condition by calculating the condition number, and quantize the value to fall into some specific values.
2. According to the channel condition (some specific condition number), we may look up the corresponding table (Table 4.4) to choose the optimal mode.

Table 4.4: Mode selection table

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
10	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V
20	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V
30	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	V	V	V	V	V	V
40	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	V	V	V	V
50	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G
100	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G
200	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G
400	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G
1000	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G
>1000	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S

V	VBLAST
G	G-STBC
S	STBC

Moreover, in order to verify that an adaptive MIMO-OFDM system can performs better than that without incorporating the adaptive mechanism, we run the following simulation.

Thirteen random channel samples are imposed on the proposed system under the SNR of 16 dB, and each channel sample lasts a time-span for the transmission of 100 packets. The receiver can detect what the present condition number is, and quantizes the value to fit those pre-chosen condition numbers. Then, the corresponding tables are looked up and the most appropriate space-time mode will be chosen for the target of maximizing throughput. As shown in Figure 4.19, since high condition numbers, especially those larger than 1000, are scarcely randomized, the STBC cannot perform better than other modes resulting in the lowest throughput among all. Note that with the adaptive mechanism (labeled start marks) the highest throughput under each channel condition can always be obtained, when compared with those systems which use the same space-time scheme through all the channels. Thus, the maximum overall throughput can be obtained.

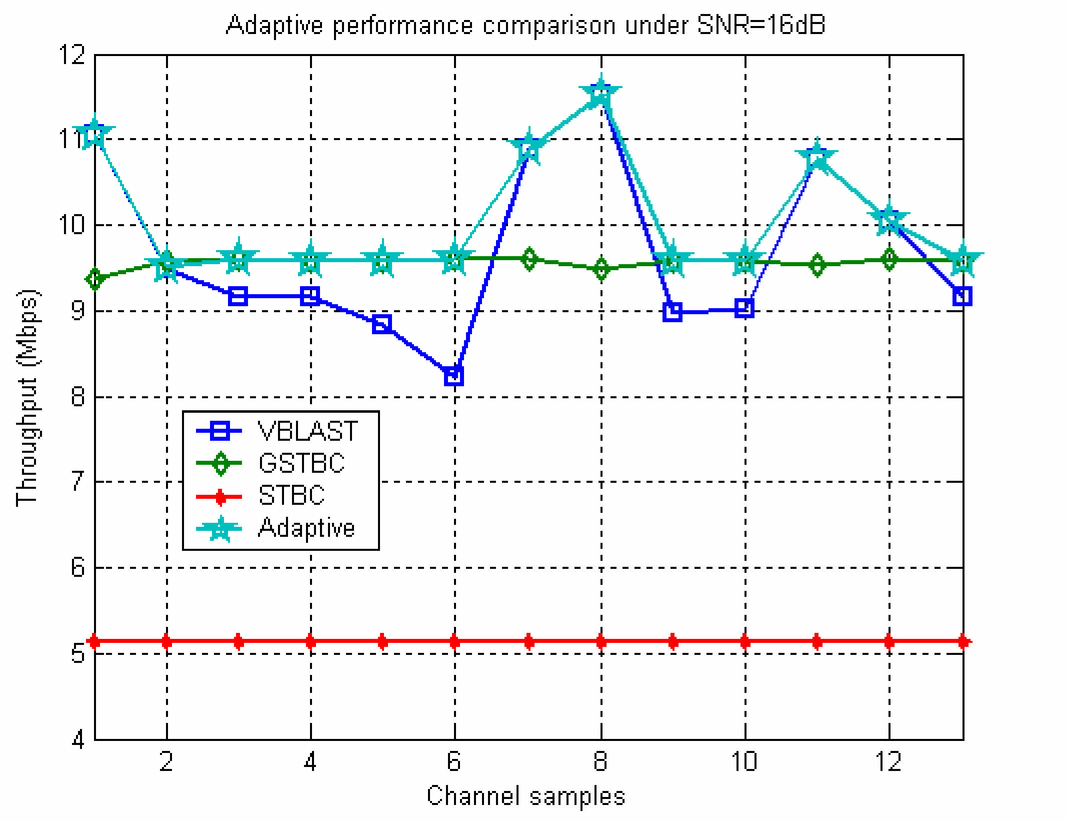
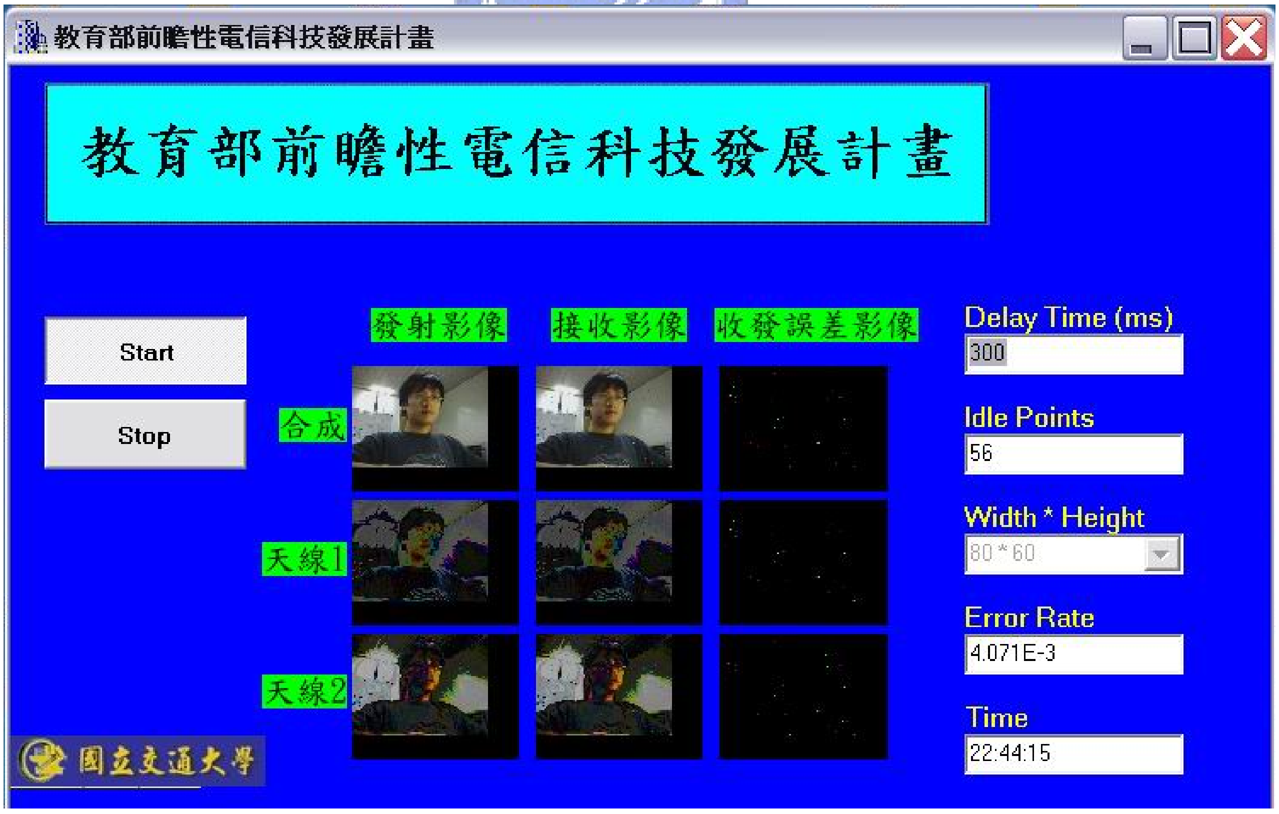


Figure 4.19: Adaptive performance comparison under SNR=16dB

4.7 Performance Evaluation

By using a web camera, we can catch the real time images continuously as the data source to verify the realized hardware system. Passing the data through DAC and ADC directly, we can obtain the pure performance containing only truncation errors and round-off errors induced during hardware realization. Through a self-developed application software in PC, we can provide an user interface to demonstrate the real time transmitted images and the received image, as shown in Figure 4.18. In the center of this figure, a 3×3 images set is located. The three columns represent transmit images, receive images, and errors images (differences between transmitted images and received images) respectively, whereas the three rows represent the synthesized images of all antennas, first two antennas, and last two antennas respectively. The real time bit error rate is also calculated and shown in the right hand side. In our system, as shown in this figure, we can achieve a pure performance about 4×10^{-3} of bit error rate.



Delay Time (ms)
300
Idle Points
56
Width * Height
80 * 60
Error Rate
4.071E-3
Time
22:44:15

Figure 4.20: Performance demonstration interface

4.8 Summary

Chapter 4 mainly describes the implementation results and the performance evaluations. We first provide some pre-implementation works, such as preamble design for MIMO structure in MATLAB, interrupt scheduling in DSP, and arrangement of data buffer insertion in FPGA. Then, we give our detailed implementation results and circuit designs on DSP and FPGA respectively, and the space-time decoding algorithms are also included. In addition, to achieve the goal of being an adaptive system, we first show the experimental results under different channel conditions. By analyzing the data, we can build up the tables for adaptive mechanism, which makes the mode selection strategies more complete. Finally, we provide some experimental results with adaptive mechanism and show that under a varying channel condition the system throughput can be improved significantly by using the strategies we proposed.