



U N I V E R S I T Y O F
SOUTH CAROLINA

CSCE 491

Capstone System Engineering Project

Project Specifications

Receiver: Parts 1, 2, and 3
Transmitter: Parts 1, 2

Version 051019-C

Dr. James P. Davis, Associate Professor
Department of Computer Science and Engineering
University of South Carolina
Columbia, SC 29208 USA
jimdavis@ceee.org

© 2005 James P. Davis
All Rights Reserved

Table of Contents

Table of Contents.....	2
1. Introduction.....	4
1.1. Premise of the Design Project.....	4
1.2. The Project Structure.....	5
2. Project Assignment.....	6
2.1. Preliminaries.....	6
2.2. Bit-Shifting Interfaces for Receiver and Transmitter Blocks.....	9
3. MAC Receiver Block Specifications.....	9
3.1. Shift Controller block.....	11
3.1.1. Receiver Shifter block.....	11
3.1.2. Frame Sequencer Controller.....	12
3.1.3. Word Selector.....	14
3.1.4. Bootstrapping the ATN Sequencing and Word Selection using the Shifter.....	14
3.2. Frame Control Decoder block.....	14
3.2.1. Interface Signals:.....	14
3.2.2. Functionality:.....	15
3.2.3. Test Data:.....	16
3.3. Address Decoder block.....	16
3.3.1. Interface Signals:.....	16
3.3.2. Functionality:.....	16
3.4. Sequence Control Decoder block.....	17
3.4.1. Interface Signals:.....	17
3.4.2. Functionality:.....	18
3.5. Frame Body Decoder block.....	19
3.5.1. Interface Signals:.....	19
3.5.2. Functionality:.....	19
3.6. FCS Decoder block.....	20
3.6.1. Interface Signals:.....	20
3.6.2. Functionality:.....	20
4. MAC Transmitter Block Specifications.....	21
4.1. Introduction.....	21
4.2. Design Scope and Assumptions.....	21
4.3. Functional Description.....	22
4.3.1. Frame Preparation.....	22
4.3.2. Distributed Control Function.....	22
4.3.3. Transmit Frame Buffer.....	23
4.3.4. Transmit Control Block.....	25
4.3.5. Build Frame Block.....	26
4.3.6. Transmit Frame Block.....	28
4.3.7. Medium Allocation Block.....	30
4.4. Behavioral and RTL Description.....	31
4.4.1. State sequence 1.....	31
4.4.2. State Sequence 2.....	33

4.4.3. State Sequence 3	34
4.4.4. State Sequence 4	34
4.4.5. Sequence Diagram 1	35
4.4.6. Sequence Diagram 2	36
4.4.7. Flowchart 1	37
4.4.8. Flowchart 2	38
4.5. Performance Estimation	39
5. Distributed Coordination Facility (DCF) Mode	40
5.1. Block description of DCF	40
5.2. Backoff Generator	41
5.3. Abstract Behavior for DCF Operation	42
5.3.1. Control Flow for DCF	43
5.3.2. Control Flow for NAV Update	44
5.4. Time Scale description of DCF	45
5.5. Performance Estimation	47
6. CRC Coding Algorithm Description	47
6.1. Introduction	47
6.2. Statement of Problem	47
6.2.1. Design Objectives	47
6.2.2. Assumptions and Constraints	48
6.3. Functional Description	48
6.3.1. Serial Implementation	48
6.3.2. Parallel Implementation	50
6.4. CRC Generation and Checking Example Scenario	52
6.5. Architectural Description	55
6.5.1. Transmitter	55
6.5.2. Receiver	56
6.6. Behavioral and RTL Description	57
6.6.1. Serial Implementation	57
6.6.2. Parallel Implementation	60
6.7. Performance Estimation	62
7. Exception Handling & Exception Codes	62
7.1. MAC Receiver Exceptions	62
7.2. MAC Transmitter Exceptions	64
8. References	66

1. Introduction

This document provides a specification for a semester-oriented digital systems design project. It is targeted to senior undergraduates who must complete a design project as part of their degree requirements for graduation from a 4-year ABET® accredited university engineering degree program in Computer Science and Engineering, Computer Engineering, or Electrical and Computer Engineering.

The contents of this document are based on the author’s work in electronics systems design for a number of global electronics firms, and on the teaching of a senior projects course taught at the University of South Carolina over a four-year period. Any questions regarding the content of this document or the project design course should be directed to the author. Any errors and omissions are solely the responsibility of the author.

1.1. Premise of the Design Project

Wireless networking has made significant inroads into the infrastructure of electronic commerce and consumer services in the past few years. A significant portion of this growth is in the area of wireless local area networks (WLAN). One important of WLAN technology is that it enables the quick setup of networking topologies without the need for physical cabling, acquiring licenses to operate, or extensive network planning. Such networks are referred to as “ad hoc” wireless networks, because they form in a somewhat pseudo-random and unplanned manner, as opposed to top-down and centrally controlled wired or wireless telephony networks. They can be easily setup, allow mobile users to communicate at relatively high speeds, and be torn down at the end of the engagement.

The predominant wireless LAN standard in the United States is the IEEE Std. 802.11 and its Addendums (particularly 802.11b, 802.11g, 802.11i and the upcoming 802.11n). These portions of the standard deal with provision of ever-greater levels of network traffic throughput, and 802.11i corrects limitations with the original security standard for wireless traffic.

The most common form of implementation of this WLAN functionality in products and embedded mobile devices is through the use of embedded microprocessor cores. In effect, a product vendor may develop or purchase a design “model” or a whole chipset supporting the network protocol, using this as the base for building its own products. Most WLAN products have some form of microprocessor running software for the 802.11 protocol, and providing interfacing to the device platform’s internal communication bus, memory, and other electronic hardware.

In this project, we will be exploring an alternate implementation strategy. Rather than rely on the use of a microprocessor to provide the protocol support, we will create our own “model” of the protocol in digital logic, with the goal of creating a direct implementation in a programmable logic device. This type of computer chip allows systems builders to create high-performance, low-power design solutions, while also retaining some of the programmability that is afforded to a microprocessor.

The rationale for why a company might adopt this type of product strategy is becoming more compelling in the marketplace. Over the past few years, WLANs have become widespread, and adoption of the 802.11 standard is much greater than its designers anticipated. In fact, many of the handset vendors in the telephone market are now adding 802.11 support, since the benefits of higher-speed data transmission from the same device that is used for standard wireless telephony makes great market sense.

There are increasing demands on the processing capability of microprocessor-based networking devices coming from three different sources: (1) the recent adoption and proliferation of MPEG-4 video on demand services, thus requiring greater data compression capability; (2) the increased need to provide greater levels of security by data encryption over WLANs; and (3) the increased need to support higher levels of error correction over increasingly crowded and “noisy” wireless channels. This processing demand pushes the computing ability of standard microprocessors to their limits. It is for this reason—better computing throughput, and lower power consumption—that we adopt a strategy of creating the core functionality of the 802.11 WLAN protocol directly in silicon, rather than write software to program a microprocessor.

1.2. The Project Structure

We will start our process of analysis, architecture and design in this design project by quickly ramping the architecture for various components of the 802.11 Protocol. The background material for this project is mostly contained in this document, and is also supported in the materials from the Gast text [1], and augmented by the various Lecture Notes specifically created for the accompanying class lectures given to prepare the students for the actual project work.

Here’s how the design project works: (1) we start with this specification, which provides the basic information for modeling pieces of the 802.11 MAC Layer using flowHDL in order to get used to using the tools, (2) you’ll work on these together as a team, to complete the modules within the required time frames, (3) we’ll incrementally add functionality and relax some of the simplifying assumptions as we work through the Transmitter and Receiver blocks, and (4) we’ll integrate the blocks and simulate them together, and (4) we’ll synthesize a circuit from these design descriptions using the code generating capabilities of the flowHDL tool set.

During the class lectures, we discuss the design process, your progress through the design project milestones and deliverables, and any issues involving the 802.11 specification or other project matters that come up—in addition to the digital design materials we’ll cover in lectures as well.

This project will be a steep introduction to Register-level architecture and design modeling using flowHDL. You’ll have to spend the time working with flowHDL to get the hang of it (every tool had its own behaviors, and you’ll have to spend the time “logging the flight hours”). In addition, you’ll have to spend time getting to know the IEEE 802.11 problem domain. However, this is a design project, so the only way to get the grade is to do it. Since this is a 3 hours course, you can expect to spend roughly 8-10 hours a week over the lifetime of the project during the semester working on these design tasks—*analysis, modeling, verification, refinement, and extension.*

2. Project Assignment

The project deliverables are broken out according to a set of project milestones, each milestone involving the creation of some portion of the design. As a block of functionality is completed, it is integrated and tested with the blocks created in prior weeks. This requires some “finesse” and control of the design activities—because you can easily foul up earlier work by introducing new work that has not been properly integrated and tested.

In the following sections we present the deliverables for the Receiver Team, followed by a discussion of deliverables for the Transmitter team. It is assumed that the team size for each component consists of two team members, and that a Transmitter Team and a Receiver Team are working together.

2.1. Preliminaries

These different functional blocks comprise the scope of the project, Receiver and Transmitter components, as listed in the two tables, below:

Receiver Block Name	Document Section	Brief Description	Time Frame
Receiver Shifter	3.1	Brings data in from the PHY Layer, 4 bits at a time, and creates a 16-bit internal word.	1/2 week
Shift Controller	3.2	Controls data-shift, performs “triage” on new word, and passes received 16-bit word to one of the decoder blocks in the receiver.	1 week
Frame Control Decoder	3.3	Decodes information in the Frame Control Word field, extracting subtype and other relevant information.	1/2 week
Address Decoder	3.4	Decodes MAC address information to extract addresses and determine of frame is for this receiver.	1 week
Sequence Control Decoder	3.5	Decodes frame and fragment numbers in Data frames and checks frame ordering.	1 week
Frame Body Decoder	3.6	Decode the frame body and buffer the data from frame fragments.	1/2 week
CRC Decoder	3.7	Decode the CRC information from the FCS field in the frame.	1 week
Receiver Exception Handler	3.8	Respond to any exceptions generated by the other Receiver blocks, flag any frame errors, and reset all threads.	1/2 week

Transmitter Block Name	Document Section	Brief Description	Time Frame
Transmitter Shifter	4.1	Shifts a constructed frame, a 16-bit word at a time, out to the PHY Layer 4-bits at a time.	1/2 week
Frame Builder	4.2	Construct a frame of specific type and sub-type.	1 week
CRC Generator	4.3	Generate CRC bits for the FCS field.	1 week
Media Allocation Controller	4.4	Contend for control of the network channel by timer countdown and checking carrier sense.	1 week
Transmitter Exception Handler	4.4	Respond to any exceptions generated by the other Transmitter blocks, flag any timeout or retry count errors, and reset all threads.	1/2 week
MAC Master Controller	4.5	Manage the 4-frame transactions, and coordinate the Transmitter and Receiver blocks.	1 week

These blocks listed in the tables are to be analyzed (according to this specification, and referencing the 802.11 specification in [1]), designed and verified in the methods discussed in this document and in the class lectures. Each person will need to be responsible for carrying out some function on the design team, although you can work together with your teammates to complete the designs (which is recommended). As discussed before, you'll need to label your designs with all of the team members' names or the team name (this is done by adding names to the design in flowHDL through the Design Information pull down menu option).

NOTE 1: You will need to come up with specific test data for your block, at least 4-6 test scenarios that you will use for your debugging and verification of functions. For each test situation, you will need to use the simulator the step through the design. Once you have gotten a good simulation run for your block (meaning you get the correct data in the output waveform for each test), you will then save the result as a Batch File, labeling as Test_1, etc. You can accelerate creation of test data by modeling different threads that are part of the test bench, and driving the testing from these threads (as we have discussed in class). Create specific threads to control the testing of your design, but remember, you'll have to debug the test threads as well. See the flowHDL Manuals for more information (or ask me and we'll discuss further in class).

NOTE 2: The project "deliverables" are as discussed for the various homework assignments, including flowHDL source files: diagrams, Bus Table, Memory Table (if used), printouts of batch files, and screen capture of waveforms from simulation, showing the data for each test simulation waveforms. Please follow the requirements for the assignment deliverables, as posted on the web page. We will define the specific delivery dates for each block as part of the lectures, and I'll post it on the web page.

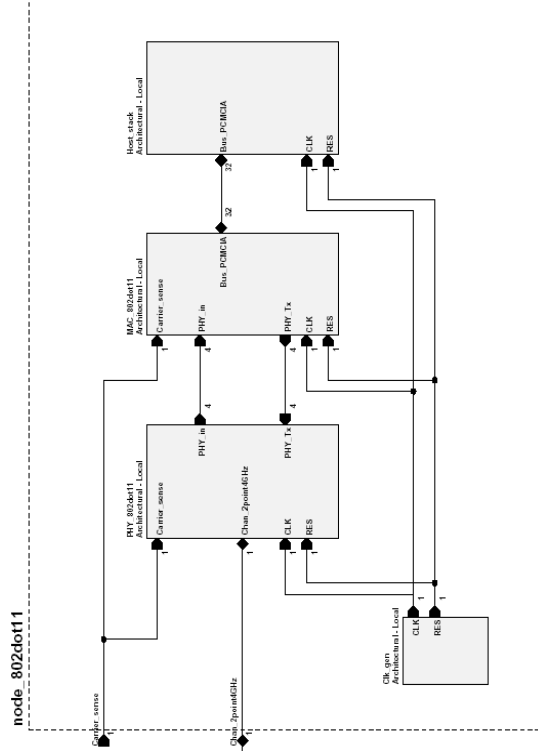


Figure 1. 802.11 Block Diagram

NOTE 3: You will work using a single flowHDL file for your entire Receiver or Transmitter design, of which you'll print out and hand in different pieces of the design as it evolves. Save it often when you are working with it. Coordinate with your teammates who will keep the "golden" version of the design file. Also, keep backup copies that you can revert to, if you ultimately have problems. (Things like random power outages in the Swearingen building have caused student to lose several days of project work—so don't let this happen to you. Also, no software is bug-free, and flowHDL has some intermittent problems—so save into different versions, labeled with date and time, to avert disaster.)

As you work on your various blocks, different team members will create and verify blocks in their own flowHDL files. You can copy/paste these thread elements from one design file into another, using an open copy of flowHDL: (1) save your file first, (2) copy the desired elements of a given thread, (3) open the "master" design file that is your team's golden version, (4) create a new sheet and open it on the canvas, (5) paste the copied design elements onto the new sheet (position leftmost corner using the left mouse). I've verified that this works, just be careful about grabbing the tread box—you don't want to inadvertently select it.

NOTE 4: I will grade the project according to the following criteria, not necessarily in this order: (1) conformance to specification, (2) originality and use of the methodology, (3) correctness of functionality and timing, (4) performance (i.e., throughput).

2.2. Bit-Shifting Interfaces for Receiver and Transmitter Blocks

This one we have already spent some time with in the lecture, so it is not covered further in this specification. Please refer to the accompanying Lecture Notes for more details on this deliverable.

The IEEE 802.11 Frame Format is shown below:

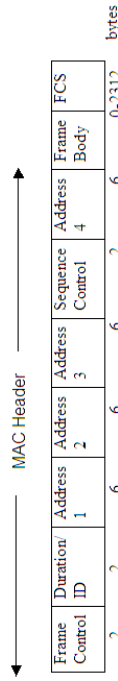


Figure 2. MAC Frame Structure

The frame begins with a MAC header. The start of the header is the frame control field then a field that contains the duration information for the network allocation vector followed by the three addressing fields. The next field contains frame sequence information. The final field of the MAC header is the fourth address field. Following the MAC header is the frame body. The final field in the MAC frame is the frame check sequence

3. MAC Receiver Block Specifications

This section contains the specifications for each block in the 802.11 MAC Receiver function block. Each block is described in terms of input/output behaviors, and a description of tasks the block is to perform.

NOTE 1: For signal interface definitions for declaration in the flowHDL Bus Table, assume all signals are Binary and created using Registers. We may go back and optimize the design by incorporating some non-registered elements later. Other properties are defined below.

NOTE 2: The most important function in most of the blocks is to check the integrity of the incoming frame. When we find errors (or, *exceptions*) in the frame's contents, we set the `X_ERR` flag and load the `X_ERRCODE` register with the appropriate error code—to be defined later—where the “X” prefix is the block name. All blocks will thus have these two exception signals defined as outputs; each block has their own specific signals. Also note that what we do is simply ignore the frame by “flushing” it from the system if it doesn't meet our criteria (i.e., if it has errors due to transmission noise, incorrect framing, etc.).

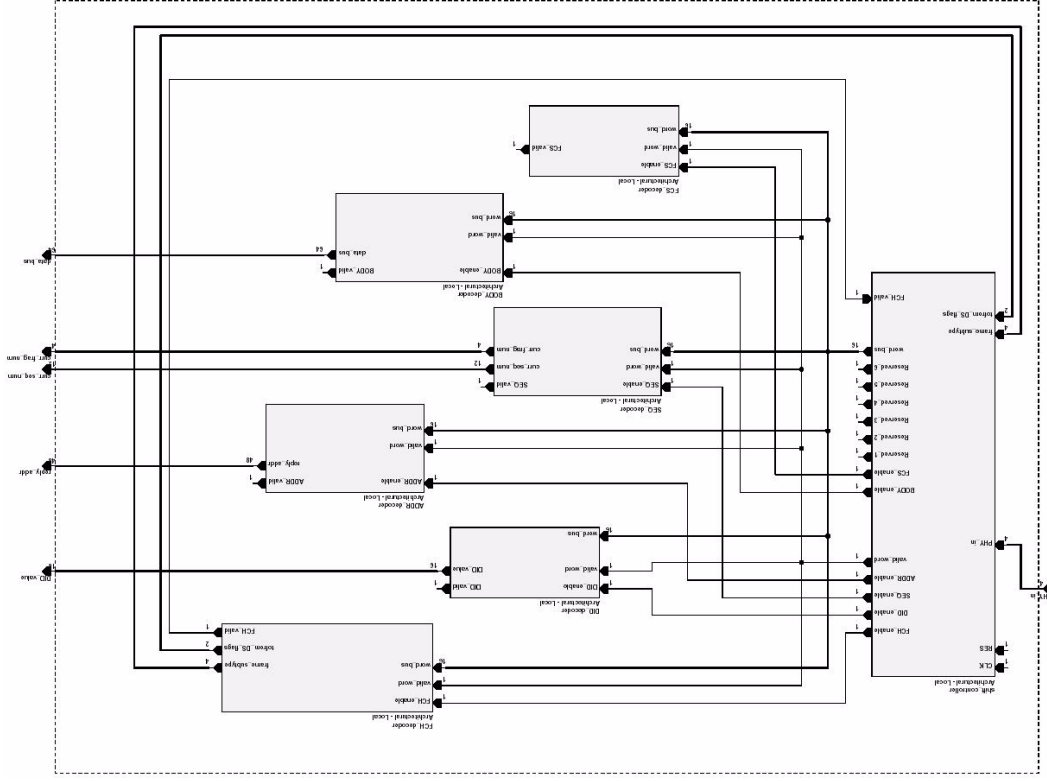


Figure 3. MAC Receiver Block Diagram

3.1. Shift Controller block

The Shift Controller block contains a set of sub-blocks that work together to carry out the function of retrieving data from the PHY Layer and passing this data, 16-bit at a time, to the various other Receiver blocks responsible for decoding the contents of the frame data stream. The list of sub-blocks is as follows:

- 1) **Shifter:** Responsible for shifting in the 4-bits and constructing 16-bit words, making them available for other Actors downstream in the processing pipeline. **Word Counter:** Responsible for determining which of the downstream Actors gets the newly shifted Frame word. To do so, it must keep track of two things: (1) where in the frame structure it is currently processing, and (2) when it is processing the 1st word of the frame (i.e., the header) or some other Frame 16-bit word. This separation and partitioning of responsibilities might indicate that further partitioning and assigning responsibilities is required as we continue to refine our architecture.
- 3) **Decoder_Selector:** Responsible for selecting one of the target Actors to get the latest word. Since each word is passed to only one Actor, this will be some decoding process performed by the hardware.
- 4) **Frame_Header_Decoder:** Responsible for reading the Frame Control Header and decoding information from the header word: (1) Frame Subtype – used to identify the type of frame being received (and used to initialize the Word Counter actor), and (2) To_DS and From_DS flags, indicating whether there will be 3 or 4 addresses contained in a Data frame (if this is the frame subtype being processed.)

3.1.1. Receiver_Shifter_block

The Receiver_shifter block takes in 4-bits on each read cycle, and after some number of clock cycles, provides a 16-bit word to the rest of the MAC Receiver block. This block handshakes with the PHY block, in that PHY does the following:

- 1) Test to see if the shifter is busy (MAC_shift_busy), and waits until it is not busy.
- 2) Once not busy, it sets the flag PHY_go, and posts its data on the PHY_in bus.
- 3) PHY block waits until Receiver_shifter block posts the signal MAC_shift_done.
- 4) PHY goes to the top of its loop to start this process again.

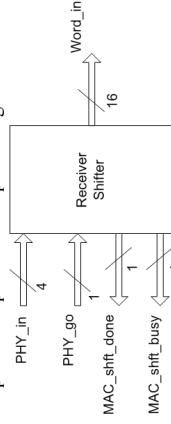


Figure 4. Shifter block interface definition.

The Receiver_shifter block sits in a poll loop waiting for PHY_go to be set. Once set by PHY block, it sets its MAC_shift_busy flag, and starts its shift operation. Once it is done, it clears

MAC_shift_busy and sets MAC_shift_done. When this thread starts, it also sets MAC_shift_done = 0 and busy = 0. See the set of lecture notes on this topic.

3.1.2. Frame Sequencer_Controller

We want to keep track of the state of processing of the current frame being read in from the PHY Layer. We need to do this because, as we read in successive words from the Shifter, we need to know which of the Receiver's field decoding blocks should process this new work. Rather than hardcoding this information into the Receiver, we'll define an *augmented transition network* (ATN) defining this sequence. We do this because, by not hardcoding the sequencing, we can easily add support for additional frame types/subtypes without impacting other parts of the design. *Ease of reuse and extensibility* are key design goals for the project.

Therefore, the "lifecycle" of a Frame corresponds to the sequence of states that each frame might pass through between the time the Shifter recognizes it and when it is fully read in from the Physical Layer and decoded by the collection of downstream blocks in the MAC Receiver's pipeline. The ATN can be directly mapped into an ASM model for the processing thread. This state sequence comprising the frame processing lifecycle is shown below.

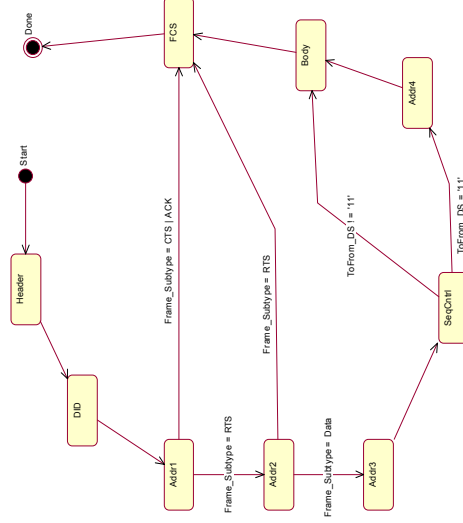


Figure 5. ATN Frame Sequencer Statechart Diagram

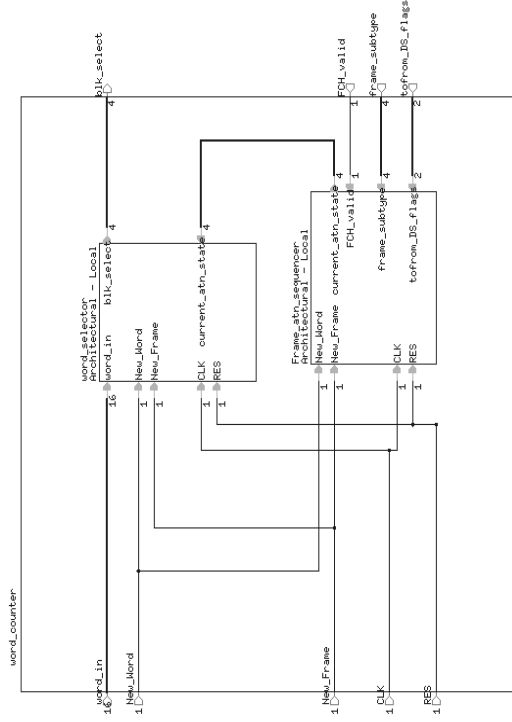


Figure 6. Word Counter Block Diagram

Note that each of the 4 kinds of frames is represented as a path through the state diagram. You'll create an ASM thread that will support this sequencing. It will output a bus value denoting the "present state" of the frame being processed. This "present state" information is used to decode and select the downstream Receiver block that should operate on the current 16-bit shifted word.

The ATN sequencer thread must know two things in order to function properly: (1) when has a new word been received, and (2) when the new word received is associated with the start of a new frame. The ATN state machine is initialized to its starting sequence state on detecting that the current new word also indicates start of a new frame. Start of a new frame is implied in two cases: (1) the first word after a power-up reset (the ^RES signal being asserted, and all threads are in the reset states), or (2) we have just completed processing of a frame, and we are transitioning from the FCS state around to the start with the FCH state.

Note that the clocking of this thread could be different than the other threads. Since we want to transition between states in this thread only when we have a new word, we won't advance to a new state on each clock cycle. We can manage this in one of two ways: (1) place a loop on each state in the ATN Sequencer thread to check for "new word" signal, or, (2) define the new word signal as the clock for the thread (in which case, we are using a signal other than the predefined CLK as the synchronizing signal).

3.1.3. Word Selector

The word selection process reads the "present state" register indicating the state of the frame processing, i.e., what field for which currently receiving data, and uses this to decode the enable section logic for selecting a single downstream, block.

This is best handled using a Case structure in the ASM thread to decode the value of the ATN state from the ATN Sequencer thread, and map this to selection of an individual enable line that is read by the specific decoder block to be enabled. Note that, for address fields, each address should generate its own sequence value (since each address is a separate state in the ATN sequencer). But there is only one Address Decoder block; therefore, you will want to perform a logic OR on all 4 of the address enable lines and generate the single ADDR_Enable line that will activate the Addr_Decoder block (which is discussed in a later section).

This thread can operate as a combinational logic block, in that every clock cycle it can be reading the value of the ATN state (although it may not be changing every cycle). A point to consider is how many possible enable signals can be asserted. We'll be implementing one decoder block for each frame field: Control Word, DID, Address, Frame Sequence Count, Frame Body, and FCS. Only one block should be enabled for a given word—unless you are carrying out the CRC computation in parallel (see the section on CRC decoding).

3.1.4. Bootstrapping the ATN Sequencing and Word Selection using the Shifter

Note that, for the first word in a new frame, we don't yet know what kind of frame we are dealing with, so we won't know which block to select. However, since all frames start with a Frame Control header word, we will have a default assumption that the first word of the frame will indicate the ATN sequence state for "FCH" (frame control header word).

On reading the "new frame" signal as True, the ATN Sequencer thread will set this as the "present state" of the frame, and the Word Selector will enable the Frame Control Decoder block to enable the FCH_Decoder. This block will then decode the actual type and subtype, which is provided back to the ATN Sequencer, where it is used to govern all subsequent state transitions in this thread.

Note that it is the Shifter block that will set the "new frame" flag, which will be used by the ATN Sequencer to start the state sequencing of the new frame. We'll set the value of "new frame" for the first received word after a hard reset (the ^RES signal) in this thread.

3.2. Frame Control Decoder block

3.2.1. Interface Signals:

SHFTOUT_BUS	Input	16 bits
Enable_FCD	Input	1 bit
FrameByteCount	Input	12 bits
MoreFrag-Bit	Output	1 bit
Retry_Bit	Output	1 bit

FCH_Subtype Output 4 bits
 FCD_ERR Output 1 bit
 FCD_ERRCODE Output 4 bits

You'll have to define other Internal signals for doing your design.

Note that the 12-bit FrameByteCounter bus allows us to report frame sizes up to 4K in length (much larger than we need, since the max frame size is ~2082 bits. Since we're limiting the Data frame size to 2K (instead of the specified 2312 bytes from 802.11b), and we have 4 6-byte addresses (possibly), a 4 byte FCS, a 2 byte Header, 2-byte Duration ID (which we are ignoring in the current design iteration) and a 2-byte Sequence Control Word.

This bus is optional, in that we have a means to keep track of where we are in the transmission process by using other contextual information. This was explained in the previous section.

3.2.2. Functionality:

- 1) Wait for Enable_FCD to go high (using a Wait on Enable Loop in the flowdiagram, as in the Shift block I have done).
- 2) "Latch" the data from SHFTOUT_BUS into internal register.
- 3) Do frame checking on the following parts of the Header:

- a) Check the protocol version (default is 'b00', anything other than that should cause the flag setting ERR = 1).
- b) Check the correspondence of Frame Type and Subtype. We are only dealing with the RTS, CTS, Data, and ACK types of frames, so we'll check that the types and subtypes match each other and are in this set (see the handout notes from last week.)
- c) Check the byte counts for the Type/Subtypes. Each of the frames has a fixed byte count (as given in the notes for RTS, CTS, ACK). The Data frame will always be transmitting a 2K data block, so you have to figure a total byte count based on that, plus headers, addresses, etc. Note that, in order to check this property, you'll also need to know whether the frame is a fragment—as computed by the Sequence Controller Block (see below, and in the handout notes from last week).

- 4) Strip the subtype from the header and place into FCH_Subtype register (if no ERR occurred), also load the bit values from the MoreFragments and Retry bit fields in the header, and place into the output registers for use by the Sequence Control Decoder block.

- 5) Set the FCD_ERR and FCD_ERRCODE registers with values (ERR = 0 means "no error").

- 6) Additional Byte Count Checks:

- a) We need to incorporate some additional coordinated byte count checking, having to do with the total and fragmented byte counts associated with Data frame subtypes.

- b) New Input bus: FrameByteCount 12 bits: This is the total frame byte count provided by the PHY layer (after it strips off its own headers and information). We need this byte count because we otherwise don't know when we have End of Frame (EOF).

- c) Reading the SCD_FragFlag, and checking if the frame is a fragmented one, we need to make sure that the Data byte count for the frame is less than the 2K limit. If the bit is set and the data size is greater than 2K, then we have an ERR condition (so we need to define a new ERRCODE value for this one, say 'b1010').

- d) Reading SCD_FragFlag and reading the FragBit from the header, we could have a 2K data size if this was the last fragment in a sequence of 1 fragments per frame sequence; this is a weird boundary condition, but we'll test for it nonetheless (boundary conditions are "killers" for a design). So, again, if this is the last fragment, we'll need to make sure that the SCD_FragFlag is also set (or else this is an ERR with new ERRCODE = 'b1011'), and we'll need to check that the byte count for data is not greater than or equal to 2K. If not the last fragment, it must be less than 2K. If it's the last fragment, it could be less than or equal to 2K.

- e) Note that we are not maintaining a running byte count checking in the FCD block, since it is only concerned with checking consistency between frame header and other parts of the current frame in process. The FBD block will likely maintain sequence-fragment byte counting across frames, so as to be able to assemble a complete frame in sequence to pass up to the upper layers of the 802 protocol stack.

3.2.3. Test Data:

You will need to come up with specific test data for your block, at least 8 test scenarios.

3.3. Address Decoder block

3.3.1. Interface Signals:

SHFTOUT_BUS	Input	16 bits
Enable_AD	Input	1 bit
FCH_Subtype	Input	4 bits (from Frame Control Decoder block)
SCD_FragFlag	Input	1 bit (from Sequence Control Decoder block)
SCD_Countier	Input	16 bits (from Sequence Control Decoder block)
FrameByteCount	Input	12 bits
SenderAddr	Output	48 bits
ADR_ERR	Output	1 bit
ADR_ERRCODE	Output	4 bits

3.3.2. Functionality:

- 1) Wait for Enable_AD to go high (wait loop)
- 2) "Latch" SHFTOUT_BUS data into internal register. Note that since this bus is 16 bits in length, we'll need 3 clock cycles to fully latch an address. So you might define a 48 bit internal bus, and latch different parts into each 16 bit slice on each pass through. The signal Enable_AD will remain high for the duration of the 3 word transfer, then will go low afterwards. We will assume there is a "strobe" signal, which we won't model for now, that keeps the stuff moving. So, you should be able to latch the data on 3 successive clock cycles. Maybe we could check that the data isn't the same on a subsequent cycle, just to make sure that we have new data.
- 3) You'll need to pick up 1,2,3 addresses, depending on the type/subtype of frame you are receiving. This is given in the supplemental notes. So, you'll need some outer loop on the addresses, based on reading and latching the value of FCH_Subtype, produced by the Frame Control Decoder block.
- 4) We will use the sequence of the following MAC addresses for our various wireless stations. Each project team should select the address according to their project team number (assigned in class):
 - Assume the MAC address for your station's MAC is between 'h4FFFFFFF044'.
 - Assume the IBSS address is 'h4FFFFFFEE11'.
- 5) Checking on integrity of addresses: we need to check that the addresses are of valid format, that they are in correct location, and that they have the correct number of bits. For that, we'll need to strip off the lower 46 bits (the valid address) and make sure the upper 2 bits are zeros (since we are not using these two bits in our implementation of 802.11).
- 6) Take the Sender address and store it in the output register. Since we are handling many frames from a sender, such as with fragments, we'll also need to check that we are receiving data from a sender for whom we have a valid address. This will be true if the Sequence Control Decoder block tells you that this is not the first frame or fragment in a sequence from a "sender". For instance, if the SCD_Counter input is non-zero, then we should have a match on the Sender address.
- 7) (NOTE: We are assuming "full ordering" of frames in the IBSS network, so therefore we expect to receive all of the frames from one sender in the order in which they are sent.)
- 8) Set the ADR_ERR and ADR_ERRCODE registers with values (ERR = 0 means "no error").

3.4. Sequence Control Decoder block

3.4.1. Interface Signals:

SHFTOUT_BUS Input 16 bits
 Enable_SCD Input 1 bit

FCH_Subtype Input 4 bits (from Frame Control Decoder block)
 MoreFrag-Bit Input 1 bit (from Frame Control Decoder block)
 Retry_Bit Input 1 bit (from Frame Control Decoder block)
 SenderAddr Output 48 bits (from Address Decoder block)
 FrameByteCount Input 12 bits
 SCD_FragFlag Output 1 bit
 SCD_Counter Output 16 bits
 SCD_ERR Output 1 bit
 SCD_ERRCODE Output 4 bits

2.4.2. Functionality:

- 1) Wait for Enable_SCD signal to go high.
- 2) Latch the SHFTOUT_BUS containing the SequenceControl field from incoming frame. This is subdivided into a Sequence_No (12-bits) and Fragment_No (4-bits). You'll need to store each of these; either in one internal register or in separate registers.
- 3) Check the contents of these and make sure they are consistent. More-frags and fragment number are out of sync: You'll need to read the MoreFrag-Bit and Retry_Bit registers loaded by the Frame Control Decoder block. This will be checked against ERRCODE conditions 'b0101', 'b0110', 'b0111'. If any of these is inconsistent, then set ERR = 1 and write the appropriate code into ERRCODE.
- 4) You'll need to keep track of the fragments associated with a particular frame sequence, of which there can be at most 16 fragments per frame sequence (any more would also be an error, so add this condition and assign a new ERRCODE to it). You'll need to update this information on the SCD_Counter for each frame or fragment, value taken from the appropriate field read off the SHFTOUT_BUS. This will be used by the Frame Body Decoder to reassemble the fragments into a complete frame.
- 5) You'll also need to track the frame sequence count, to make sure that (1) the frame count appropriately "wraps" to zero when the count reaches the maximum value in 12 bits (so this is 2**12 - 1 before wrapping to zero). If it wraps to zero before reaching this, then you can assume an error in the sequence, since we are assuming total ordering of frames received from a sender, and we are assuming only a single sender at this point.
- 6) You'll need to pass the frame sequence number to the Frame Body Decoder block as well. It will be used to insure that the appropriate data packet is being passed to higher 802 levels in the protocol stack. So, you'll need to make sure the frame sequence numbers are in sequence. The best way is to keep track of the most recent fragment and sequence numbers and compare them against the new ones to insure they are in order. This will require some tricks with the macro functions, like testing that NewFragNo = OldFragNo + 1, possibly using the INCRNC macro, and then testing the result in a Condition. Note: that you'll need to check the sender address on all of this frame

sequence and fragment counting to insure that we are counting relative to a given sender (although we will only be testing with a single sender in the IBSS network).

- 7) You'll generate SCD_ERR and SCD_ERRCODE values and output.

3.5. Frame Body Decoder block

3.5.1. Interface Signals:

SHFTOUT_BUS	Input	16 bits
Enable_FBD	Input	1 bit
FCH_Subtype	Input	4 bits (from Frame Control Decoder block)
SCD_FragFlag	Input	1 bit (from Sequence Control Decoder block)
SCD_Counter	Input	16 bits (from Sequence Control Decoder block)
FrameByteCount	Input	12 bits
SenderAddr	Output	48 bits
FBD_ERR	Output	1 bit
FBD_ERRCODE	Output	4 bits

3.5.2. Functionality:

- 1) Wait for Enable_FBD to go high (wait loop)
- 2) "Latch" SHFTOUT_BUS data into internal register. Note that since this bus is 16 bits in length, we'll need many clock cycles to fully latch all of the data. So you might define a 64 bit internal bus, and latch different parts into each 16-bit slice on each pass through. The signal Enable_FBD will remain high for the duration of the n word transfer, and then will go low afterwards. We will assume there is a "strobe" signal, which we won't model for now, that keeps the stuff moving. So, you should be able to latch the data on some successive clock cycles. You'd take the contents of your 64 bit register and write the data into a memory buffer 2K in size (so that means you'll need to use the Memory Table to define a memory array labeled "ReceiveBodyBuffer". We may have to look at the signal types since I think memory requires some form of multi-valued logic (i.e., not Binary signals).
- 3) You'll also need to use the FCH_Subtype to determine whether you actually have memory data to store, since some of the frame types won't have any frame "body" (i.e., RTS, CTS, ACK won't have any data). Note also, if you get information from the Sequence Control Decoder that you are receiving frame fragments, you'll have to assemble them back into a complete 2K packet of data, that can then be sent to higher levels of the 802 protocol stack.
- 4) Checking for errors will consist of simply insuring that the data is properly stored. There are no specific ERR conditions at this point, so you'll always return ERR = 0, ERRCODE = 'b0000', for now (unless we find something where we need to do this.)

- 5) Note that you can get an indication from the CRC Decoder block that they have detected a checksum error in the frame, in which case you'll have to flush the buffer. However, it may be possible to check this prior to processing the memory data in the first place. (This will be something we discuss in class).

- 6) I think we'll need to define a state machine that simply counts the fragments, and we'll need to alert the FCS Decoder block that we have a complete frame Data set in the buffer so that the CRC check can be run against it. Note that CRC checking is done per frame, regardless of whether it is a fragment or a full sequenced frame (i.e., full 2K of data).

3.6. FCS Decoder block

This block will check the 32 bits in the FCS (frame check sequence) segment of the frame, last 32 bits, to insure that the data integrity is intact. Usually, this would be done first, since bad frame would indicate that the whole frame is corrupted and thus we'd want to flush it completely--allowing the sender to timeout and then go into a retry sequence to re-send the frame to out receiver station.

However, we will attempt to perform the MAC Receiver functions in parallel, taking advantage of the hardware capabilities, with the assumption that processing on bad frames is infrequent, and thus we will gain significantly in throughput with a low likelihood of framing errors (although this may be an invalid assumption in areas of high wireless LAN traffic or low signal to noise -SNR). But we'll operate in this manner for the project as posed.

3.6.1. Interface Signals:

SHFTOUT_BUS	Input	16 bits
Enable_FCS	Input	1 bit
FCH_Subtype	Input	4 bits (from Frame Control Decoder block)
MoreFrag-Bit	Input	1 bit (from Frame Control Decoder block)
Retry_Bit	Input	1 bit (from Frame Control Decoder block)
SCD_FragFlag	Input	1 bit (from Sequence Control Decoder block)
SCD_Counter	Input	16 bits (from Sequence Control Decoder block)
FrameByteCount	Input	12 bits
FCS_ERR	Output	1 bit
FCS_ERRCODE	Output	4 bits

3.6.2. Functionality:

Wait for Enable_FCS signal to go high (wait loop).

Latch the SHFTOUT_BUS data into a 32 bit internal register containing the CRC value for the frame being received. You'll need to read the input register twice, over two cycles, and write the received data into low and high-16 bit slices of the 32 bit register.

Once you have the CRC value, you can check the data. We'll assume you have access to the memory buffer data captured by the Frame Body Decoder block, which would be a 2K block (for a full frame sequence) or less than 2K (for a frame fragment). You'll have to check fragment's CRC values as well as frame sequence CRC values, since they will arrive as separate packets.

If the CRC check doesn't match the data received, then we'll flag an error. Note that the CRC check is done against the full frame, including the header, address information, Data, and other parts (see the handout notes from the 802.11b IEEE Specification). We'll need to give some thought as to how we modify this specification so that we can get access to the full frame for this purpose. We'll discuss in class.

The CRC algorithm and approach we'll use is specified in the later section of this document.

4. MAC Transmitter Block Specifications

4.1. Introduction

The IEEE 802.11 MAC supplies the functionality required to provide a reliable delivery mechanism for user data over noisy, unreliable wireless media. The transmitter has been modeled into blocks based on their functionality. A top down approach was used to define these blocks. The four main blocks modeled are: Transmit control block (TCB), Build header, Medium allocation control and Transmit Frame.

Various design decisions have to be made at each level, such as whether a frame to be transmitted should be fragmented or not, depending on the noise in the medium. Good design decisions as how to obtain the data from the buffer and transmitting it over the medium have to be made in order to reduce hardware cost, in terms of resource usage, as well as number of clock cycles required to operate each block.

Collisions between two simultaneously transmitting stations can be avoided by checking the medium before each transmission. Whenever we don't receive an acknowledgement for a frame, we need to initiate a retransmit using retry logic in the Transmitter. All these issues are discussed in the following sections.

4.2. Design Scope and Assumptions

The goal of the problem is to build frames, obtain high throughput, reliable data delivery, and to obtain continuous network connections over the medium. The objective is to design a MAC transmitter for reliable transmission (using minimum number of clock cycles) of data between two stations. It includes construction of frames, Collision Avoidance, transmission Timers & Retry logic. The following assumptions are made about the use of the MAC Transmitter block.

1. The MSDU "payload" (Data Length is 2k bytes for frame, and 128 bytes per fragment) is available in a buffer.

2. Protocol version of each frame is '00'.
3. ToDS of each frame is '0'.
4. FromDS of each frame is '0'.
5. Power Management Subfield of a frame is '0'.
6. MoreData Subfield of a frame is '0'.
7. WEP subfield of a frame is '0'.
8. Order Subfield of a frame is '1'.
9. All the addresses are available, and we use the range specified earlier in this document.
10. Frame check sequence (Trailer in a frame) is to be generated, as discussed in the next section of this document.

4.3. Functional Description

The MAC Transmitter block has been partitioned into four main sub-blocks, as shown in the following figure, which can be further sub-divided. Each sub-block is explained below.

The transmitter block prepares and streams the data back to the physical layer for transmission to other stations. It is important to minimize the latency between receiving a correct frame and transmitting the next as it diminishes the chance that another station will enter broadcast range and creating a collision.

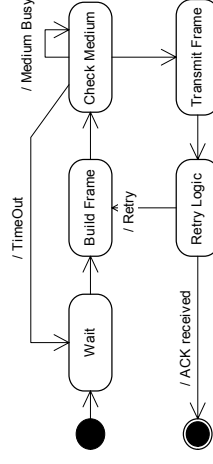


Figure 1: Transmitter Sequence

4.3.1. Frame Preparation

The preparation of the frame is a key component in the transmission of the frame. The build frame block initializes the header, duration ID, necessary address fields, and the sequence control fields. Additionally, the build frame block enables the transmit CRC calculation for final insertion into the frame. This block may especially exploits the benefits of parallelism inherent in reconfigurable computing. All of the operations described may simultaneously occur. This advantage lowers the turnaround time before the transmitter is shifting out the next frame in sequence, and maintains the speed of the design.

4.3.2. Distributed Control Function

Since many wireless networks will be ad hoc, it is necessary to regulate the use of the medium without a centralized governing station. The Distributed Control Function (DCF) ensures reliable transmissions through two methods.

First, prior to transmitting a frame, the transmitter senses the medium. If the medium is not idle, the transmitter enters a random length back-off period, within limits set by the protocol. Following the back-off period, the transmitter senses the medium again, and if idle, begins the transmission. Second, a retry algorithm is implemented in the transmitter. Following the completion of shifting the frame bits out to the physical layer, the transmitter begins waiting on the correct response, as indicated below.

Transmitted Frame	Expected Frame
RTS	CTS
DATA	ACK
ACK	-----

Table 7: Expected Next Frame

If the transmitter times out without receiving the correct response frame, the previous frame is retransmitted, with the retry bit of the header indicating a retried frame. The transmitter returns to the wait state again to receive the correct return frame.

4.3.3. Transmit Frame Buffer

To successfully shift out the many components of different frame types, a specific block has responsibility for selecting the various frame fields to include with the transmitted frame. This block represents much of the same functionality as the frame sequencer in the receiver block, as it maintains the sequence of the transmission.

At its simplest, this decoder may be considered a multiplexer; however, the select decoding logic is not a simple multiplexer, and the functionality must be implemented using the design tools manually.

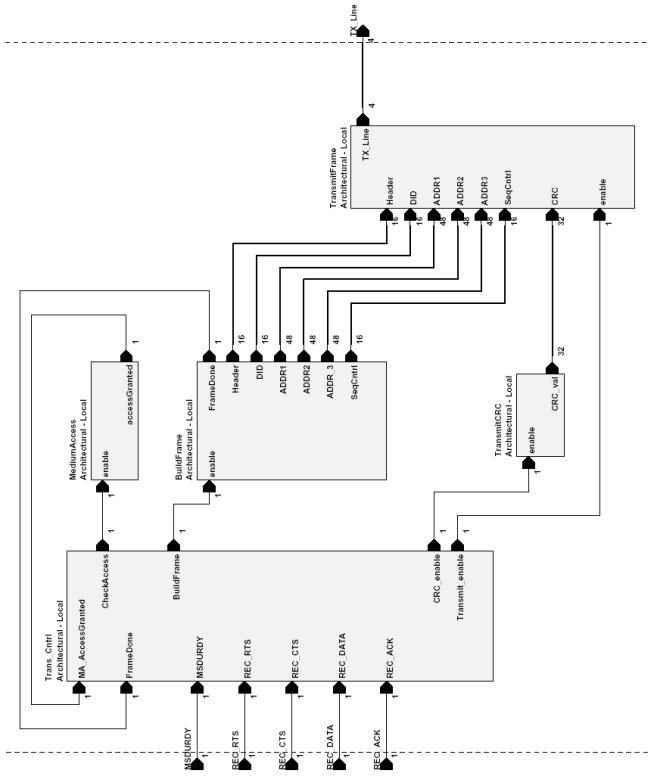


Figure 2: Top level Block diagram

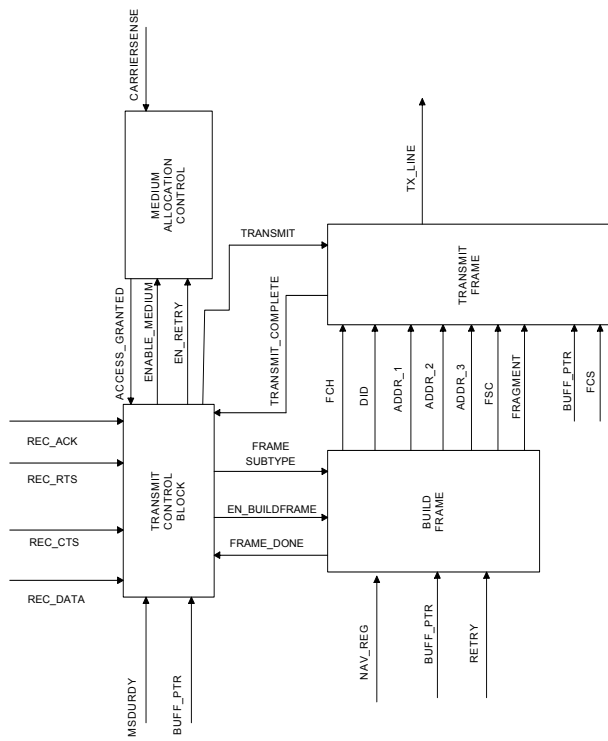


Figure 3: Alternate Top level Block diagram

4.3.4. Transmit Control Block

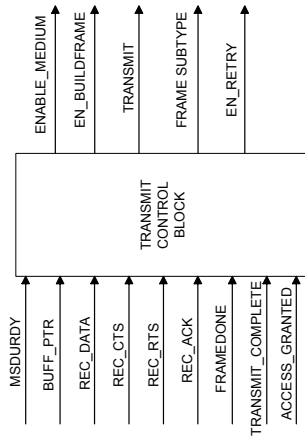


Figure 4: Transmit Control Block

The Transmit control block (TCB) coordinates with various blocks and provides efficient transmission of frames. It is enabled by five signals MSDURDY, REC_DATA, REC_CTS, REC_RTS, REC_ACK.

Name	Bits	Type
MSDURDY	1	Input
REC_DATA	1	Input
REC_CTS	1	Input
REC_RTS	1	Input
REC_ACK	1	Input
FRAME_DONE	1	Input
ACCESS_GRANTED	1	Input
TRANSMIT_COMPLETE	1	Input
BUF_PTR	24	Input
TRANSMIT	1	Output
EN_BUILDFRAME	1	Output
FRAMESUBTYPE	4	Output
EN_RETRY	1	Output
ENABLE_MEDIUM	1	Output

When the MSDURDY enables it, it copies the buffer pointer and the offset from the buffer management block. The Build Frame block (BFB) is enabled with the EN_BUILDFRAME signal and FRAMESUBTYPE is set to RTS frame, and the block waits for the FRAME_DONE signal from the BFB.

The TCB then enables the medium allocation control (MACtrl) with the ENABLE_MEDIUM signal. It waits for the ACCESS_GRANTED signal from the MACtrl.

After receiving this signal, it enables the Transmit Frame block (TFB) with the TRANSMIT signal to send the RTS frame. A timer is started and TCB then waits for REC_CTS signal. If the watchdog timer exceeds the time limit, TCB sends EN_RETRY signal to MACtrl.

After receiving the REC_CTS, TCB enables BFB to build the data frame. It waits for the FRAME_DONE signal from Build Frame block. It enables the TFB with TRANSMIT signal to send the data frame. A timer is started and TCB then waits for REC_ACK signal. If the timer expires, TCB sends EN_RETRY signal to MACtrl.

4.3.5. Build Frame Block

The build frame block is enabled by the EN_BUILDFRAME signal from the Transmit Control block. It receives the FRAMESUBTYPE of the frame that is to be built from the TCB. There are four sub-blocks: Frame Control Header, DID, Generate Address, and Frame sequence control,

each of which builds different parts of the header. All these blocks are concurrently enabled by the subtype signal.

The Frame Control Header generator block sets the values as shown in the table below. The values are stored in the register and then sent to the Transmit Frame block through FCH signal.

The DID generator block selects the appropriate pre-defined value for the inter-frame spacing delay (either SIFS or DIFS) and assigns it as the DID value. The value is stored in the register and then sent to the Transmit Frame block after the DID signal is activated.

The Address generator block takes BUFF_PTR input from the buffer, and extracts and stores the packet destination address in a register. It looks up its own MAC address and stores in the appropriate ADDR_x field, according to the type-subtype pair for the frame being constructed. For Data frames, the BSS address (which we assume is a hardcoded value, since our design does not go through the Association process for constructing an ad-hoc network) is also assigned to the appropriate address. The generated addresses are sent to the Transmit Frame block through the three signals: ADDR₁, ADDR₂, and ADDR₃.

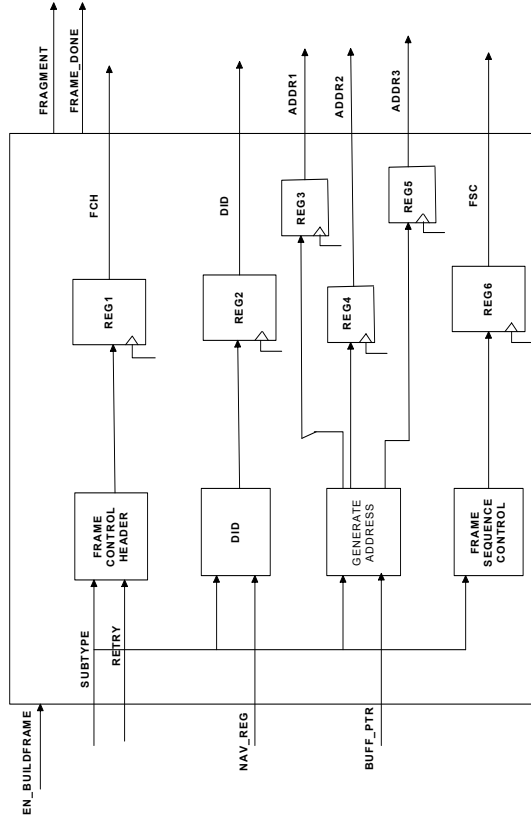


Figure 5: Build Frame

Name (No. Of Bits)	Value
Protocol version (2)	00

Type (2)	Input
Subtype (4)	Input
ToDS (1)	0
FromDS (1)	0
MoreFragments (1)	0
Retry (1)	Input
PowerManagementsystem (1)	0
MoreData (1)	0
WEP (1)	0
Order (1)	1

The Frame Sequence Control generator block assembles the frame sequence only for data frames. It checks whether the frame has to be fragmented or not, by comparing the size of the frame with the dot11 Threshold value. If the frame is to be fragmented, this block sends a FRAGMENT signal to the Transmit Frame block. This block appropriately increments a sequence counter value for insertion into the frame being constructed. The values are stored in the register and then sent to the Transmit Frame block through FCS signal.

NAME	BITS	TYPE
SUBTYPE	4	Input
RETRY	1	Input
NAV_REG	16	Input
BUFF_PTR	24	Input
FRAGMENT	1	Output
FCH	16	Output
DID	16	Output
ADDR1, ADDR2, ADDR3	48	Output
FSC	16	Output
FRAME_DONE	1	Output

This block also maintains a series of counters; one counter is associated with each unique destination address in the network. In a full implementation of the 802.11 MAC standard, we would not restrict the design to any specific limit on the number of stations in the network. However, in this project, we'll assume a hardcoded limit of 4 possible stations for which we can transmit and receive data.

We'll define a set of registers into which we can store the set of possible remote station MAC addresses to which we can transmit. Since we need to maintain a separate frame/fragment sequence for each station association, we'll keep a sequence counter for each MAC address that is passed down through an MSDU packet.

4.3.6. Transmit Frame Block

The Transmit Frame block has a multiplexer and a 32-bit shift register (the Transmit Shifter). The buses FCH, DID, ADDR1, ADDR2, ADDR3, FCS, and DATA are all given as inputs to the multiplexer. It also has a 3-bit select line. The output of the multiplexer is sent to the 32-bit shift register associated with the Transmit Shifter. Using multiplexer and shift register, Transmit Frame block transmits the different parts of the frame through TX_LINE signal, which is passed to the PHY Layer.

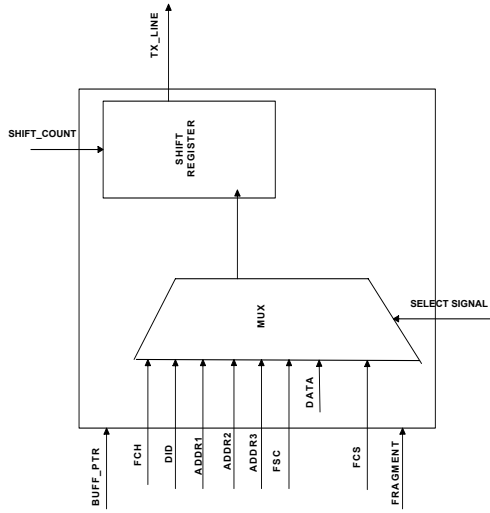


Figure 6: Transmit Frame

BUFF_PTR contains a buffer pointer and offset pointer. Buffer pointer is offset for every 512 words in memory for each MSDU. Using BUFF_PTR as an index, data of one word length is read from the buffer RAM and shifted out to the TX_LINE through the shift register. If the fragment signal is high data is read and shifted as 128k, otherwise 2k data is shifted.

```

Offsetpointer ← Bufferpointer;
While Offsetpointer < (Bufferpointer + 512) do {
  Read location and memory;
  Write it to shift register;
}

```

Name	Size	Type
FCH	16	Input
DID	16	Input

ADDR_1	48	Input
ADDR_2	48	Input
ADDR_3	48	Input
FSC	16	Input
BUFF_PTR	24	Input
FCS	32	Input
FRAGMENT	1	Input
TRANSMIT	1	Input
TRANSMIT_COMPLETE	1	Output
TX_LINE	1	Output

4.3.7. Medium Allocation Block

The Medium Allocation block manages the process of gaining control of the channel in order to transmit a frame. It includes functionality for collision avoidance as well as for transmission retries. The retry logic block maintains counters for the number of retries for (1) the allocation process of gaining access to the channel, and (2) the transmission itself, which is coupled to receiving indication from the MAC's Receiver block that the corresponding frame was received in response to a frame just sent. This includes the following sequencing of frame pairings:

- 1) CTS received in response to the transmitted RTS.
- 2) Data frame (full frame or fragment) received in response to CTS.
- 3) Data frame (fragment only) received in response to an ACK (except for the ACK sent in response to having previously received the final fragment)
- 4) ACK received in response to a data frame (full or fragment).

The medium allocation block is enabled by the ENABLE_MEDIUM and the ENABLE_RETRY signals. When it is enabled by the ENABLE_MEDIUM signal, the Collision Avoidance Block checks for availability of the medium by checking the CARRIER_SENSE signal from the Physical layer and the value of the NAV, which is stored in the NAV Register.

If the medium is available, it sends the ACCESS_GRANTED signal to the Transmit Control Block. If the medium is not available, it enters the Backoff process, which is determined by the exponential backoff algorithm. After counting this value down, it checks for availability of the medium once again; it also increments the retry counter.

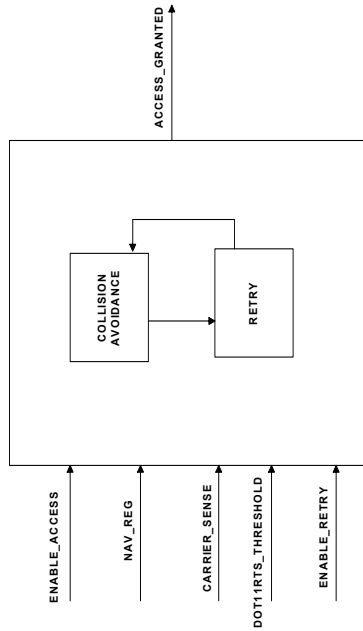


Figure 7: Medium Allocation Control block

When this block is enabled by the ENABLE_RETRY, it increments the retry counter, then it checks for availability of the medium and the process continues as described above. If the Retry count exceeds its limit, the frame is discarded and not retransmitted, and an exception is returned to the MAC Controller.

Name	Size	Type
ENABLE_ACCESS	1	Input
NAV_REG	16	Input
DOT11RTS_THRESHOLD		Input
CARRIERSENSE	1	Input
ENABLE_RETRY	1	Input
ACCESS_GRANTED	1	Output

4.4. Behavioral and RTL Description

4.4.1. State sequence 1

The master state machine starts operation when the Transmit Control block is enabled by assertion of the MSDURDY signal.

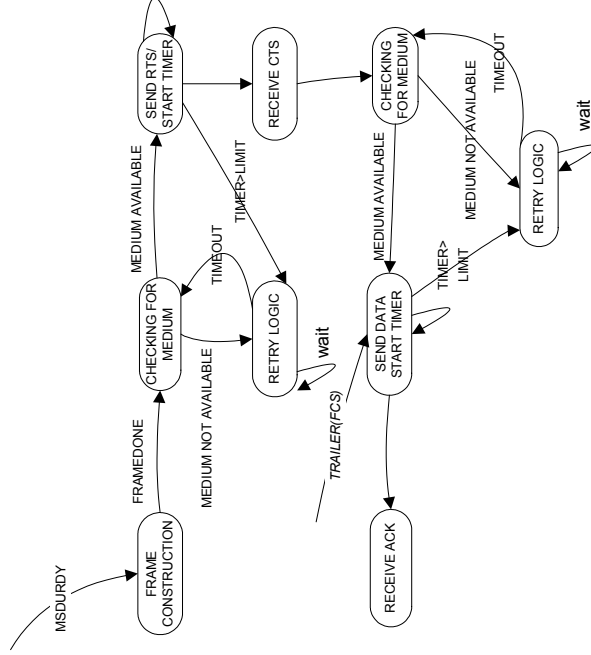


Figure 8: State Diagram 1

When the MSDURDY is sent to the TCB, it enters into the 'Frame Construction' state. After the 'FrameDone' signal is received, it enters the 'Checking for Medium' state. If the channel is available, it enters into 'Send RTS' state; otherwise, it enters the 'Retry Logic' state.

If medium is not available, it waits for the time indicated by the exponential backoff algorithm, and again checks for medium once this value has been counted down. It also increments the value in Allocation retry counter.

In the 'Send RTS' state, a timer is started. If a CTS frame is not received (as indicated by the posting of such by the MAC's Receiver block) before the timer exceeds the limit (based on the dot11RTSThreshold value), it transitions to the 'Retry Logic' state. It counts down the DIFS value, then again enters the 'Checking for Medium' state. It then transitions to the 'Send RTS' state.

If a CTS frame is received before the timer expires, it enters for the 'Checking for Medium' state in order to send data frame. If medium is available, it enters the 'Send Data' state and starts a countdown timer that "brackets" the entire transmit-acknowledge cycle between this Transmitter and the remote station to which it is sending its frame.

If a corresponding ACK frame is not received by the MAC's Receiver block before the timer expires (using the dot11RTS_Threshold value), it enters into the 'Retry Logic' state, which causes transition to the 'Checking for Medium' state. It again enters the 'Send Data' state, but first increments the Retry count register, and sets the Retry bit in the frame header's FCW.

If ACK is received before the timer expires, it enters the 'Frame Successfully Transmitted' state, indicating that the portion of the transmit cycle has completed.

4.4.2. State Sequence 2

The transition states when the Transmit Control block is enabled by REC_RTS signal.

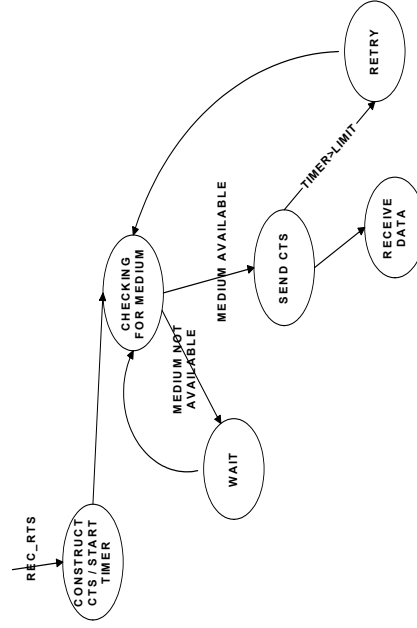


Figure 9. State Sequence 2.

When REC_RTS is sent to the TCB, it enters the 'Construct CTS' state. After the CTS is constructed, it starts a timer and enters the 'Checking for Medium' state. If the channel is available (i.e., carrier sense is not set), it transitions to the 'Send CTS' state; otherwise, it waits for the designated time increment indicated by the exponential backoff algorithm and again enters 'Checking for Medium' state.

In the 'Send CTS' state, after transmitting the frame, it waits for indication from the MAC's Receiver block that a data frame has been correctly received from this station. If data frame is received within the timer limit (dot11RTSThreshold), it transitions to the 'Received Data' state; otherwise, if the transmit timeout occurs, it enters the 'Retry' state. Again it enters 'Checking

for Medium' state, and the process repeats until either a successful frame is received from the remote station (indicating that the transmitted frame was received on the remote end successfully) or it once again times out.

4.4.3. State Sequence 3

The transition states when the Transmit Control block is enabled by REC_DATA signal.

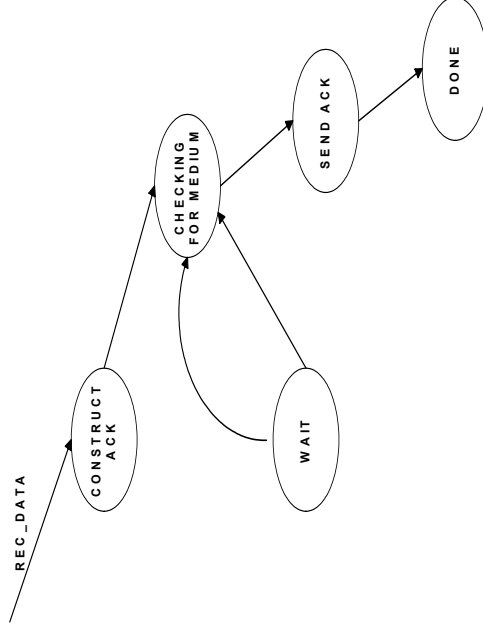


Figure 10. State Diagram 3

When REC_DATA is sent to the TCB it enters the construct ACK state. After the ACK is constructed it enters the checking for medium state. If medium is available it enters the send ACK state otherwise it enters wait state and again enters checking for medium state.

4.4.4. State Sequence 4

State diagram for a send data state in the transmit control block.

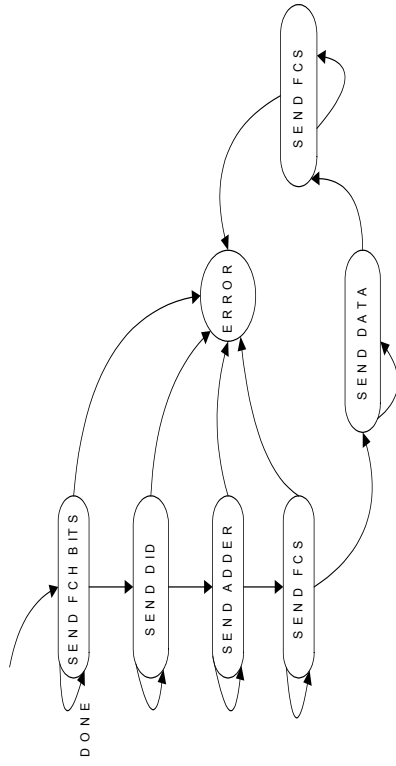


Figure 11: State Sequence 4

4.4.5. Sequence Diagram 1

Sequence diagram for the Top-level functional decomposition when MSDURDY is received is shown below.

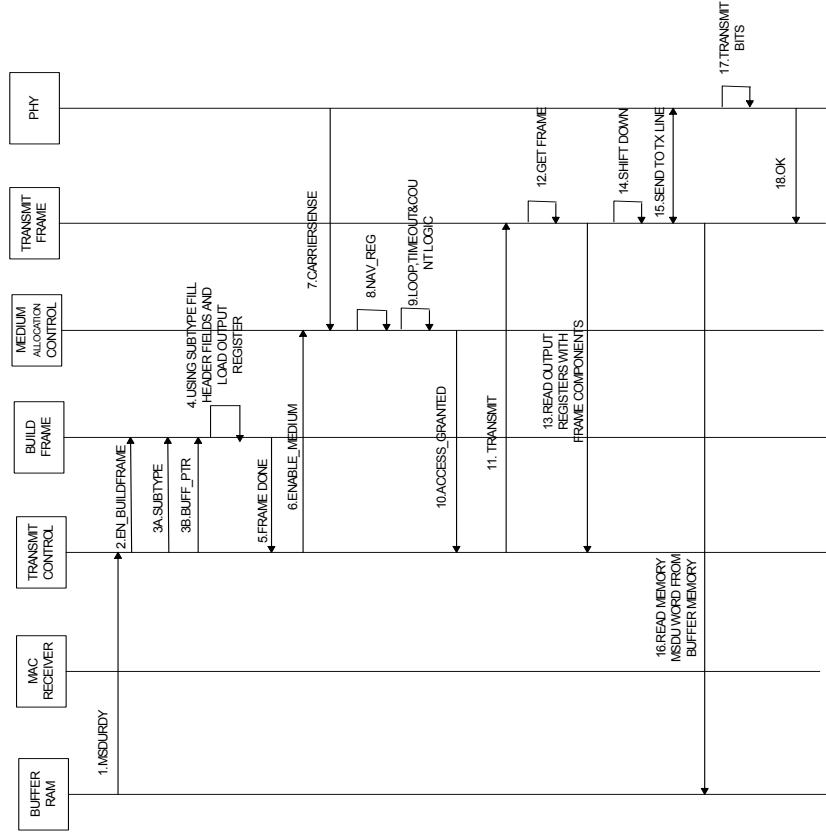


Figure 12: Sequence Diagram 1

4.4.6. Sequence Diagram 2

Sequence diagram for transmitting the frame after gaining access to the medium is shown in the figure below.

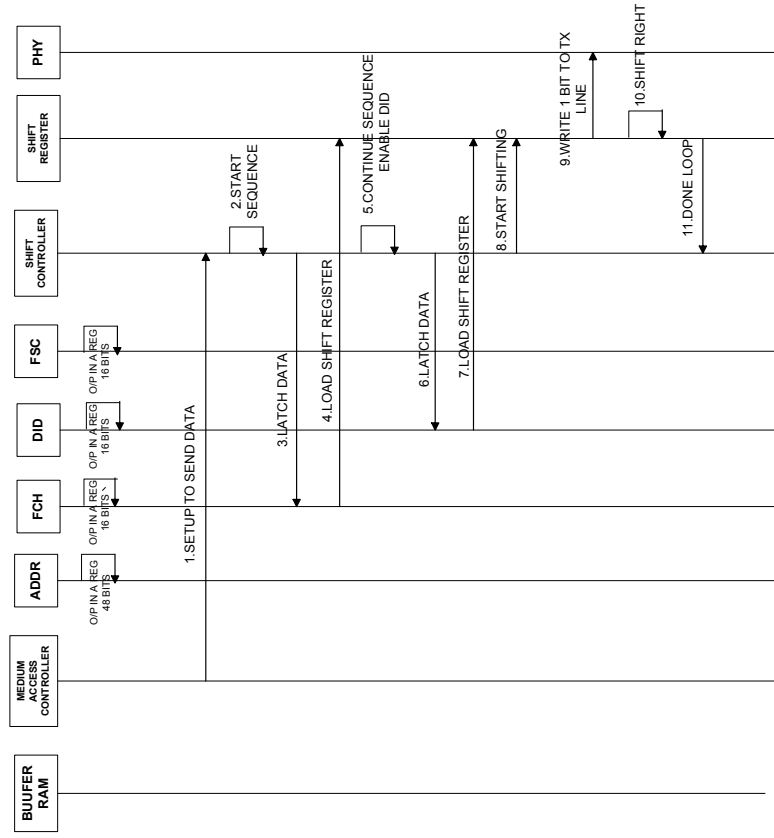


Figure 13: Sequence Diagram 2

4.4.7. Flowchart 1

The flowchart for collision avoidance process is shown below.

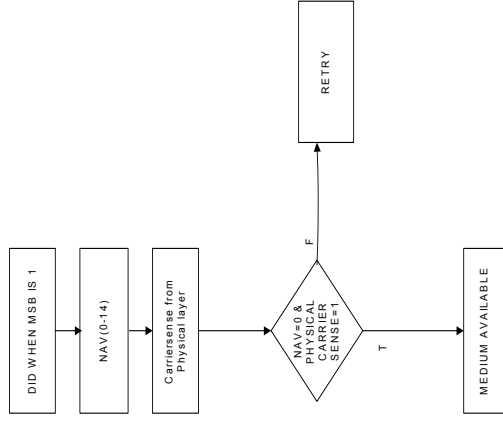


Figure 14. Retry Management process.

4.4.8. Flowchart 2

The flowchart for RETRY process is shown below.

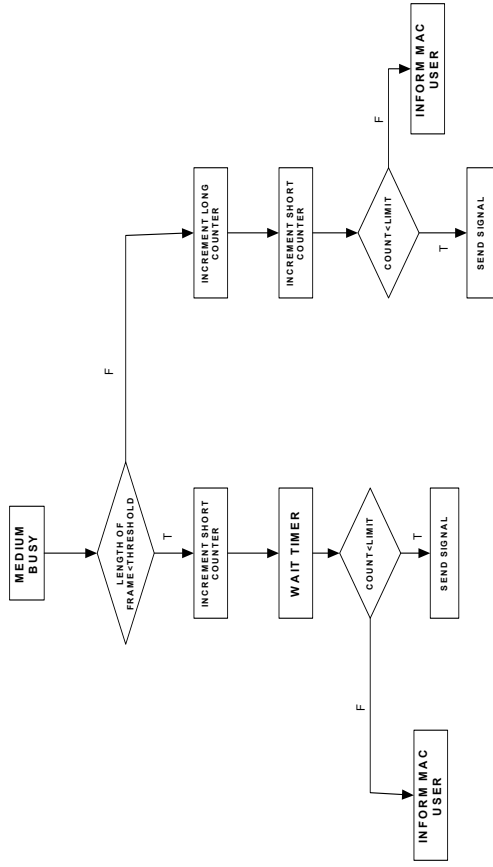


Figure 15. Retry management process flowchart.

4.5. Performance Estimation

On examining the block diagram and the sequence of data flow, we can see that the sequence of events in transmitting a frame occurs in the following order: (1) the frame is built; (2) the request for medium allocation is processed; and, (3) the data is shifted out onto the serial link to the PHY Layer.

If the medium allocation block and the build frame block are enabled concurrently (i.e., we attempt to gain access of the medium while also building the frame), we assume that we can get better performance. The build frame block in this model was designed in such way that the various parts of the header are built concurrently. In the Build frame Block, we do not access or process the MSDU data; therefore, the number of clock cycles taken by this block is small.

The data is being read from the Buffer Management block and shifted into the Transmit Frame block. We can save only a minimum number of clock cycles if we do this concurrently. This performance gain also depends on the probability of gaining access to the medium on the first attempt; however, each attempt to allocate the medium involves counting down one of the inter-frame spacing delays (either SIFS or DIFS), so the process of constructing the frame is can be completely absorbed in the time taken to count down the delays.

5. Distributed Coordination Facility (DCF) Mode

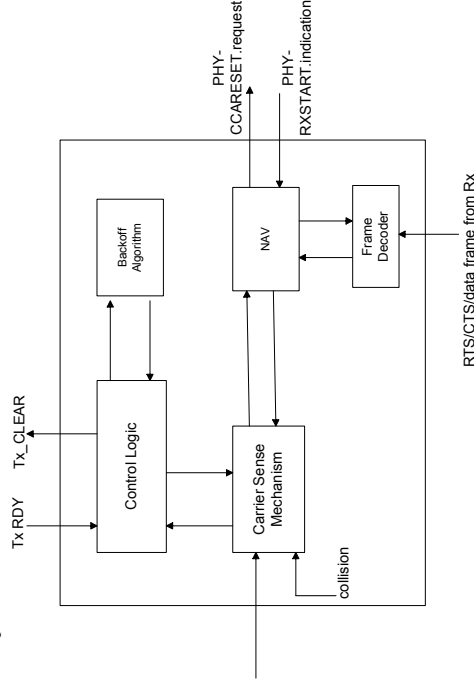
This section gives the functional description of DCF mode. The functional aspects of DCF affect the Receiver, Transmitter and the MAC Controller; therefore, we discuss it as a separate section (so that both teams will review it and consider the aspects that affect their design block).

This DCF operation is activated when it receives a TxRDY signal from the Transmitter. DCF sets up a mechanism for automatic medium sharing between different stations. It sends TxCLEAR signal to the Transmitter so that it can go ahead and transmit the ready frame. First, we give the overall block representation of the DCF operation. Then we go into details of each block at the RTL level, also, present a control flow for each block.

5.1. Block description of DCF

The figure below shows the RTL level block description of whole operation of DCF. It contains three main blocks namely Control Logic, Backoff Algorithm, Carrier Sense Mechanism and Network Allocation Vector (NAV).

Input: TxRDY, PHY carrier sense, RTS/CTS/Data frame
 Output: TxCLEAR, PHY-CCRESET



When this block receives TxRDY signal from the Transmitter, it accesses the carrier sense line to determine the state of the channel. Carrier Sense Mechanism combines the NAV state and the station's transmitter status with physical carrier sense to determine the busy/idle state of the

medium. The NAV register value predicts the time for when future traffic on the medium will commence, based on the information provided by the output of the Receiver. The Receiver checks if the frame is destined to the current station, and also extracts the duration information from the DID field. The duration information is used to update the NAV value.

The value of NAV can be thought to indicate the idle/busy state of the medium. Depending on the output of the Carrier Sense Mechanism, the Transmitter may invoke the *exponential backoff* algorithm. The Controller sends TXCLEAR to the Transmitter clearing it to send the ready frame.

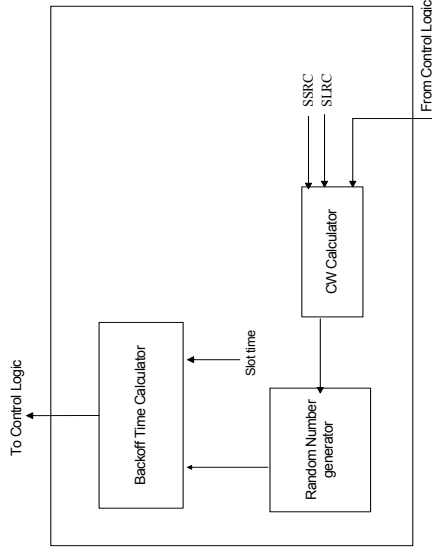
5.2. Backoff Generator

The figure below shows the functional representation of the *exponential backoff* algorithm block. It contains the following components: (1) CW Calculator, (2) Random Number Generator and (3) Backoff Time Calculator. The function of this block is to calculate a random backoff time, which is decremented while the medium is idle. This block is called on by the control logic when it senses the medium as initially busy.

Input: Signal from Control Logic, SSRC, SLRC,
Output: Signal to the Control Logic

A signal from Allocate Medium master block activates this sub-block. The operation starts with the calculation of CW. Its output is given to Random Generator, which generates a number in the interval [0,CW]. Finally, the backoff time is calculated. The abbreviation "CW" stands for "contention window", and indicates the amount of time that the Transmitter assumes will be the time it has to wait in order for the channel to be free. Remember, that the cumulative counting-down is based on SIFS/DIFS timers, as well as NAV, in addition to the Backoff value.

The Backoff value is only added when the medium is busy (as detected by Carrier sense). On each subsequent attempt to access the medium, the value for CW is extended, and a value within the interval [0, CW] is selected for the station to use for the current attempt to allocate the channel.



5.3. Abstract Behavior for DCF Operation

The following figure gives an abstract representation of the state sequencing of the MAC Transmitter operating in DCF mode. In our model, we are always operating in DCF mode; however, if we were implementing the complete 802.11b protocol, we'd have a state register that indicated which mode (DCF or PCF) we were operating.

The DCF operation starts when a frame is ready to be sent. The figure indicates the following set of states and transitions:

- 1) **Ready to transmit:** This is the first state in the cycle. This state occurs when a frame is ready to be sent. This state is also reached when a collision is detected i.e. Ack is not received by the sending station.
- 2) **Transmitting:** When in Ready to transmit state, if the PHY carrier sense is idle and NAV = 0 and retries not exceeded and backoff duration satisfied, this state occurs.
- 3) **Collision Detection (no ACK):** This state exists when a sending station does not receive an ACK from the receiving station.
- 4) **Transmission Complete:** This state occurs when the transmission of the data frame is successful and no collision is detected.

- 5) **Retries exceeded:** This state occurs when the number retries exceed dot11ShortRetryThreshold for frames shorter than RTS threshold, and number of retries exceed aLongRetryLimit for frames longer than RTS threshold. After this state the frame is discarded.

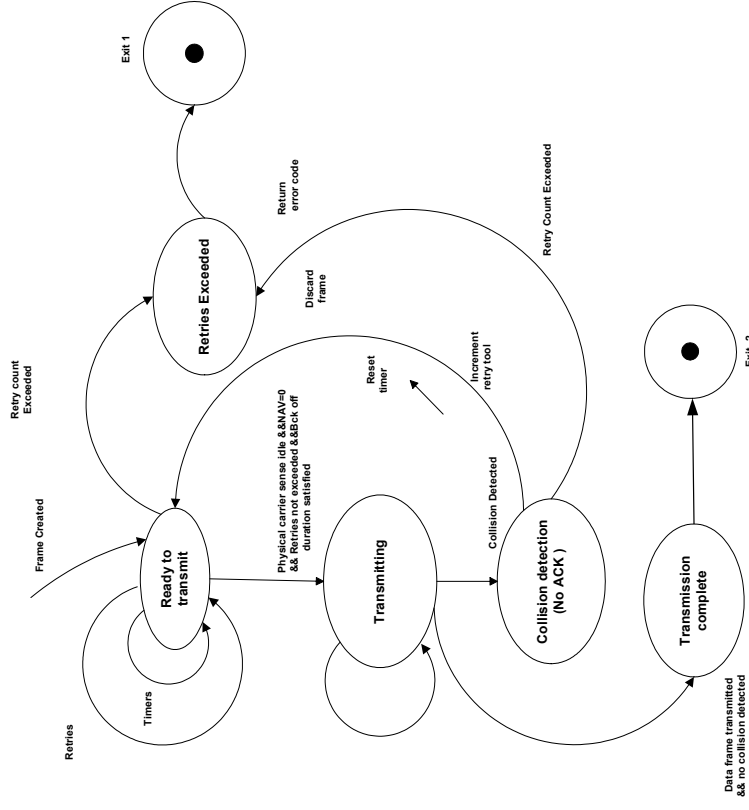


Figure 1. DCF Abstract State Sequencing.

5.3.1. Control Flow for DCF

The figure gives the behavioral description of the DCF. As said this block is activated when it receives TXRD signal from the Transmitter. Then it checks the NAV value. If it is zero, it checks the PHY carrier sense. If the medium is idle, frame is sent and ACK is expected. If there is no ACK received collision is assumed to have occurred. In that case random backoff time is calculated, and set to decrement, when the medium is idle. When this value reaches zero frame is retransmitted and the process goes over.

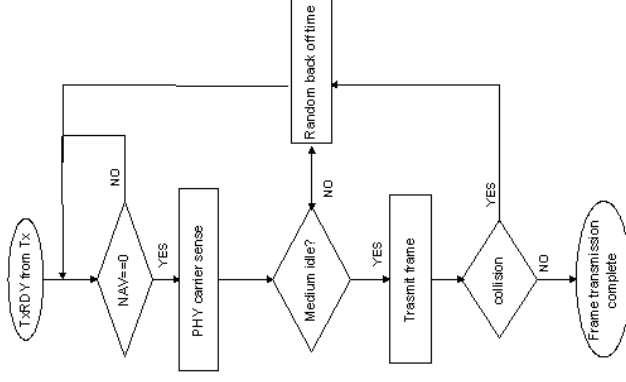


Figure 2. DCF Control Flow

5.3.2. Control Flow for NAV Update

The NAV Register is updated by the Receiver based on the DID value and used by the Transmitter for counting down delay timer associated with allocation of the media. Setting the value for the NAV Register is dependent on whether the new value to be used is less than the current value being used by the Transmitter. Additional logic must be defined in the Receiver, and coordination must be carried out between the Receiver and Transmitter.

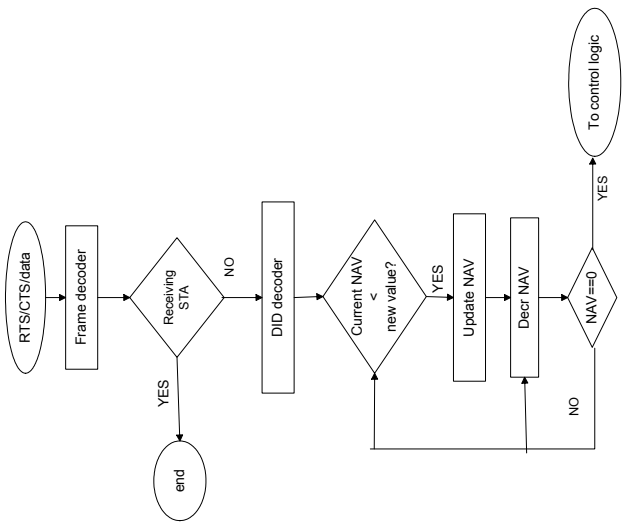


Figure 3. NAV Update algorithm.

NAV sets up a virtual carrier sense mechanism as follows. An RTS/CTS/Data frame is decoded and the receiver address (RA) field of the frame is checked with the station address. If the frame is destined for this station, the NAV won't be updated. If not, the duration ID is extracted from the frame. This duration is checked with current value of NAV. If greater, the existing NAV value is updated to the new value. The NAV value is then decremented, as before, until it reaches zero. During this time, data exchange between the sending and receiving stations has been completed. So, when NAV reaches zero, the medium is assumed to be idle and transmission is carried out.

5.4. Time Scale description of DCF

This figure gives description of DCF on a time scale as shown below. As we see when a STA becomes ready to transmit it senses the medium for a period of DIFS. If it senses that the medium was busy during this interval, the station defers transmission to later time that depends on the backoff time.

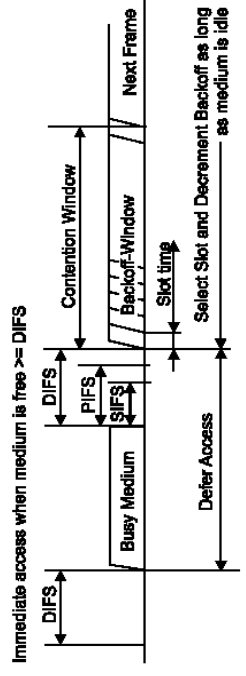


Figure 4. DCF Relationships (from [3]).

The backoff time is calculated using the following relationship

$$\text{Backoff Time} = \text{Random}() * \text{aSlotTime}$$

Random() is a value in the interval [0,CW], where CW take a value between aCW_{min} and aCW_{max} .

$$aCW_{min} < CW < aCW_{max}$$

The values of aCW_{min} and aCW_{max} are defined for a given type of PHY Layer (i.e., by the PHY Layer's transmission modulation technique). The value of CW is incremented exponentially, as shown below.

For example if we take Frequency-Hopping Spread Spectrum as our PHY transmission scheme, values are 63 and 1023 for aCW_{min} and aCW_{max} , respectively.

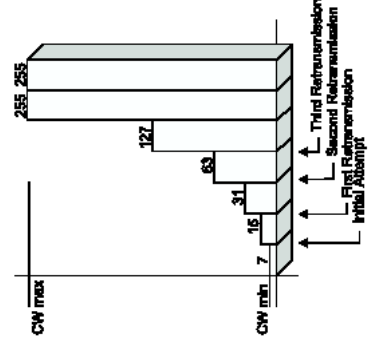


Figure 5. Exponential rise of CW (from [3]).

5.5. Performance Estimation

The performance of the wireless network as a whole can be viewed as a function of the backoff time calculations. The performance is improved if the backoff time of each station is generated to a unique value. In such a scenario the probability of collision with other stations (except those mobile stations that are moving into the WLAN cell) is far reduced.

The random backoff time is calculated by multiplying slot time with a random number, which is generated using random number generator using the contention window width.

The total delay in the generation of a frame can be estimated by the following:

$$T_{\text{delay}} = \text{DIFS} + \text{busy period} + \text{DIFS} + \text{backoff time} + \text{time to send frame}$$

6. CRC Coding Algorithm Description

6.1. Introduction

In networking systems a significant role of the MAC layer is to convert the potentially unreliable physical link between two machines into an apparently very reliable link. This is achieved by including redundant information in each transmitted frame. Depending on the nature of the link and the data, one can include just enough redundancy to make it possible to detect errors and then arrange for the retransmission of damaged frames. The cyclic redundancy check or CRC is a widely used parity bit based error detection scheme in serial data transmission applications.

All basic error detection techniques operate around the idea of trying to enable a receiver of a message transmitted through a noisy (error-introducing) channel to determine if the message was sent free of errors. In order to determine if an error is present, the transmitter appends a calculated value, called the checksum, to the end of the message. The receiver then has the ability to calculate the checksum in the same way as the transmitter, based on the data that was sent. If the checksum calculated by the receiver is the same, as that was, appended to the end of the message, then the data was sent free of error.

6.2. Statement of Problem

The basic objective of the problem is to implement a CRC (Cyclic Redundancy Check) Error Detection algorithm. At the transmitter, a CRC will be calculated and appended to the end of the data stream. At the receiver, we will verify the CRC (a checksum) appended to the end of the data stream in order to check for errors. The bit stream received will use the CRC at the end of the bit stream to check if error occurred during transmission.

6.2.1. Design Objectives

Our design objective is to design a CRC for control frame subtypes used in the data transmission for 802.11. The control frame subtypes are

- Request to send (RTS): 20 Bytes
- Clear to send (CTS): 14 Bytes
- Data 1: 2k MSDU
- Data 2: 128bytes MSDU
- Acknowledgement (ACK): 14Bytes

There are two types of implementation for the CRC algorithm.

- 1) *Serial Implementation* - uses bit-by-bit operation. This method is not optimal, as IEEE 802.11 transceivers communicate the data stream to the MAC layer over a 4-bit bus.
- 2) *Parallel Implementation* - uses multiple bits of stream per clock cycle. This speeds up the operation. Therefore a parallel implementation is preferred.

6.2.2. Assumptions and Constraints

The assumptions for this specification are

- 1) The specifications used for CRC 802.3 are same as CRC 802.11.
- 2) Four bytes are transmitted in a parallel data transmission.
- 3) Zero Propagation Delay for the XOR gates.

6.3. Functional Description

6.3.1. Serial Implementation

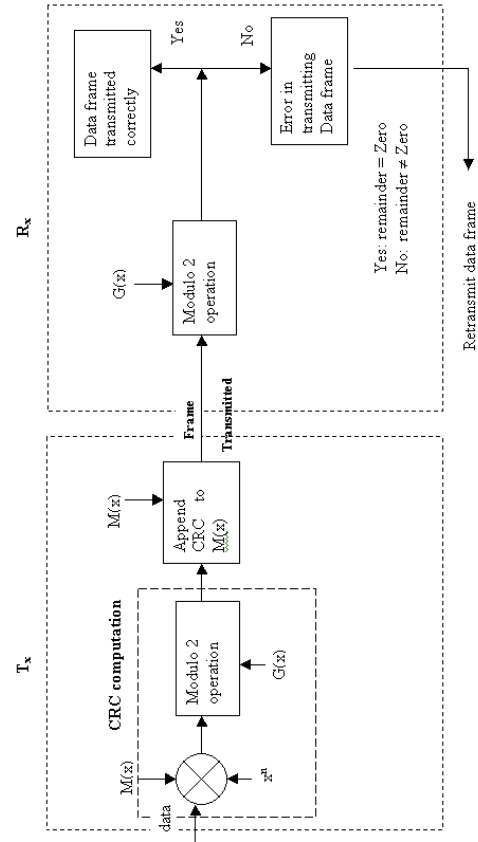


Figure 8. CRC Block Diagram

The block diagram shown above explains the basic function of the CRC algorithm. The Block diagram has two parts. The first part deals with the description on the transmitter side and the second part deals with the description on the receiver side. The basic function of the CRC is to ensure error free transmission and reception of data. The operations generally carried out in CRC are modulo-2 operations i.e., all the operations (addition, multiplication, division) XOR operator.

6.3.1.1. CRC/ Transmitter

The CRC is generated using a generator polynomial. The standard CRC-32 polynomial used is as follows:

$$G(X) = X^{32} + X^{26} + X^{23} + X^{22} + X^{16} + X^{12} + X^{11} + X^8 + X^5 + X^4 + X^2 + X + 1$$

In general, the n -bit CRC is calculated by representing the data stream as a polynomial, $M(x)$. Then we multiply $M(x)$ by x^n (where n is the degree of the polynomial $G(x)$), and divide the result by a generator polynomial $G(x)$, carried out by using a Modulo-2 operation. The resulting remainder of the above operation is CRC. The CRC thus generated is appended to the data stream. The whole of the above operation can be mathematically represented as

$$CRC = \text{Remainder of } (M(x) * x^n / G(x))$$

The RTL description of the above logic is discussed in the following section, after first discussing an example scenario.

6.3.1.2. CRC/ Receiver

The complete transmitted polynomial $T(x)$ is then divided by the same generator polynomial $G(x)$ at the receiver end. If the resulting solution has no remainder, then there are no errors in transmission; otherwise, the receiver sends no acknowledgment and the data frame has to be retransmitted. This mode of operation is supported by both the DCF (Distributed Coordination Function) and PCF (Point Coordination Function) modes in the IEEE 802.11 standard.

6.3.2. Parallel Implementation

To speed up the whole process of CRC generation, parallel implementation is preferred. This implementation operates on multiple bits of data stream per clock cycle. (In our implementation we use 4 bits at a time). The whole implementation could be made simpler by storing the results of the most of the calculations in a table or a storage array. This is referred to as table-driven implementation.

A table can be created to store all possible values of the Poly XORed with the Shifted Bits of the message. For example, in our case since we are processing 4-bits per clock cycle, there are 16 possible values of the Poly XORed with the 4-bits shifted out of the register. Thus we created a table that can store these 16 calculated values in a byte addressable storage array. These precomputed results are retrieved during CRC computation. We can basically state the above procedure in the form of an algorithm given below:

```

While (more message bits exist) {
    TOP = top (Register);
    Register = (Register << 28) | next_augmessage;
    Register = Register XOR precomputed_table [TOP];
}

```

The basic operations we use in the implementation of the algorithm include an OR, a Left Shift, an XOR, and a Table Lookup. Therefore, the implementation consists of two modes: *Table Generation* and *CRC Processing*. We discuss these modes as follows.

6.3.2.1. Table Generation

During table generation, we calculate all possible values of the 8-bit polynomial 'divided' by any 4-bits that will be shifted out of the main register during the calculation of the checksum. These

sixteen 8-bit values are then stored in a byte-addressable table to be used throughout the second mode of the implementation.

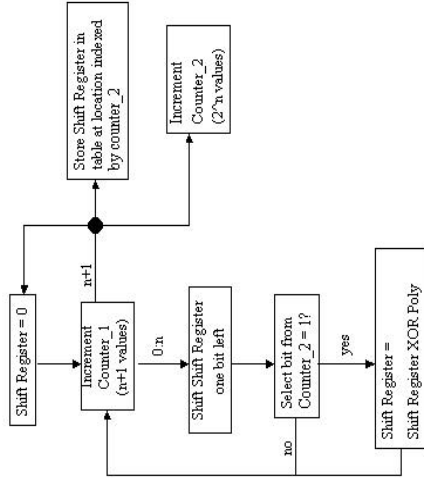


Figure 9. CRC Table Generation Algorithm.

Dividing each n -bit value by the polynomial creates the table. The remainder is stored in the table. The division part of the process is taken care by the boxes on the left. The Counter_2 is used to increment through each n -bit number. The shift register contains the remainder throughout the process of division. A multiplexer (MUX) is used to select between Counter_1 and Counter_2.

6.3.2.2. CRC/CRC Processing

The CRC processing mode consists of the following steps, as shown in Figure 3, below:

- 1) Appending the checksum to the message; and,
- 2) Receiving the message and checking whether the message was correctly received.

In the above diagram, instead of performing the costly division, the values from the table are accessed.

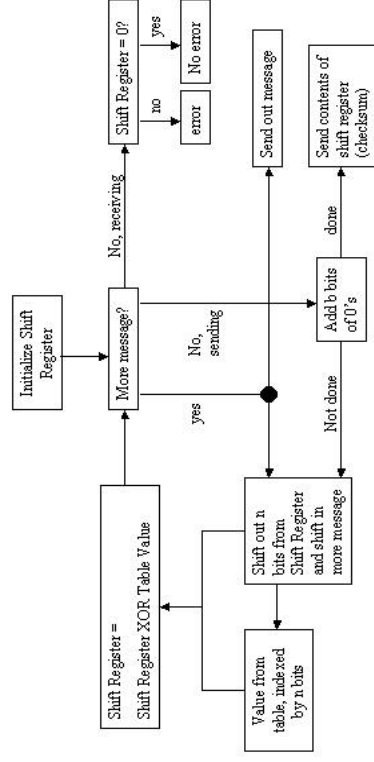


Figure 10. CRC Processing Algorithm.

6.4. CRC Generation and Checking Example Scenario

In the illustrative example that follows, we have the CRC Generator block read the frame to be transmitted and generate the appropriate CRC bits. This is followed by the mirror-image computation in a remote MAC Receiver block receiving the data stream and performing the CRC computation in order to test the integrity of the data. In this example, we introduce an error in the data stream that occurred as a result of noise in the wireless channel.

Transmitted Frame: 1101011011
 Generator: 10011
 Message after appending 4 zero bits: 11010110000

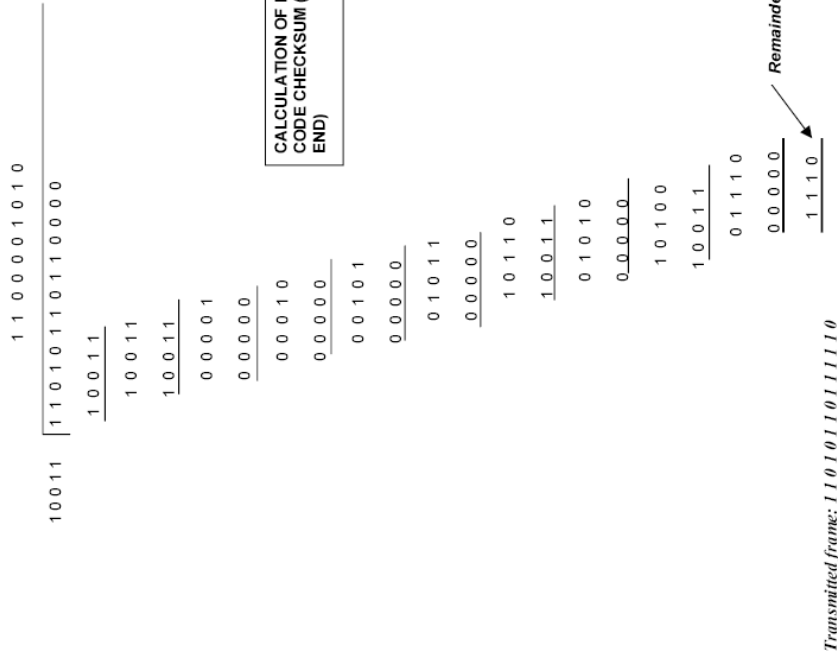


Figure 11. CRC Generation Scenario.

CRC Detection

Received Frame: 1101011001
 Generator: 10011
 Message after appending 4 zero bits: 11010010000

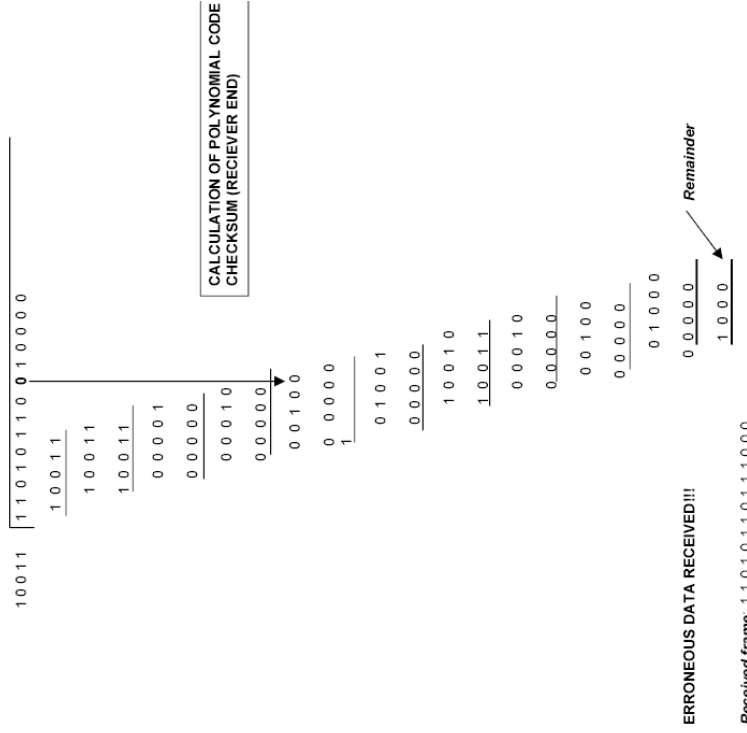


Figure 12. CRC Checking Scenario.

6.5. Architectural Description

6.5.1. Transmitter

The block diagram showing the Input and Output signals to the CRC Encoder at the transmitter end is given below.

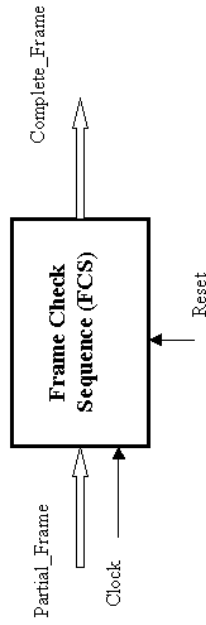


Figure 13. CRC Transmitter block.

The Description of the signals is contained in the table below.

Name Of The Signal	Type Of The Signal	Description
Partial_Frame	Input for the frame check sequence	It contains the information regarding the header and body. The length depends on the type of the frame body
Complete_Frame	Output of the frame check sequence	It contains the information of CRC appended to the frame (FCS)
Clock	Input	1 bit signal
Reset	Input	1 bit signal

Partial_Frame consists of these control frame subtypes:

- 1) Request to send (RTS): 20 Bytes
- 2) Clear to send (CTS): 14 Bytes
- 3) Data 1: 2k MSDU
- 4) Data 2: 128bytes MSDU
- 5) Acknowledgement (ACK): 14Bytes

6.5.2. Receiver

The block diagram showing the Input and Output signals to the CRC Decoder at the receiver end is given below

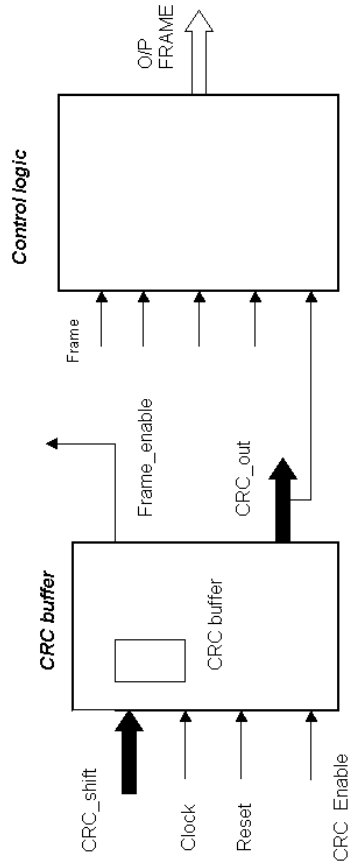


Figure 14. CRC Receiver decoder block.

The Description of the signals is presented in the table below:

Signal Name	Signal Type	Bit Width	Description
CRC_shift	input	16 – bits	Data received from the 16 bit shift register
CRC_Enable	input	1-bit	Used to enable CRC decoder
Clock	input	1-bit	Used for synchronization
Reset	input	1-bit	Reset all the signals
CRC_out	output	32-bits	The data is combined into 32 bits after 2 clock cycles
Frame_enable	output	1-bit	This signal is sent to frame encoder to confirm that the data received is error free.

The frame received consists of Frame Control, Duration/ID, Addresses, Sequence Control, Frame Body and Frame Check Sequence (CRC). The headers are separated at the receiver end by

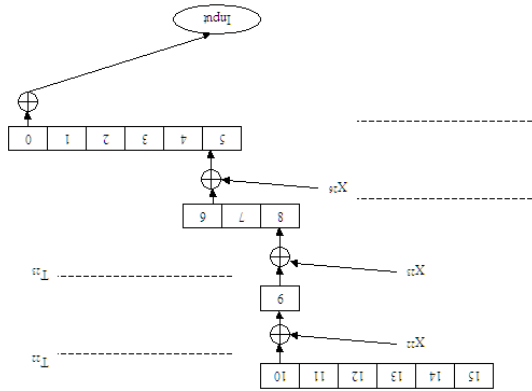


Figure 16. CRC Parallel Processing.

6.6.2. Parallel Implementation

The behavioral and RTL description mainly consists of state diagrams representing the table creation and CRC error detection.

6.6.2.1. Table Creation

The state diagram below represents the table creation mode. Initially, the message bits are read and XOR'ed during the first state depending on the index. When the message bit is zero, only shift operation is done as shown in the states following the state shift. After all the four bits are shifted, the value of the register XORed with the Poly will be stored in the register.

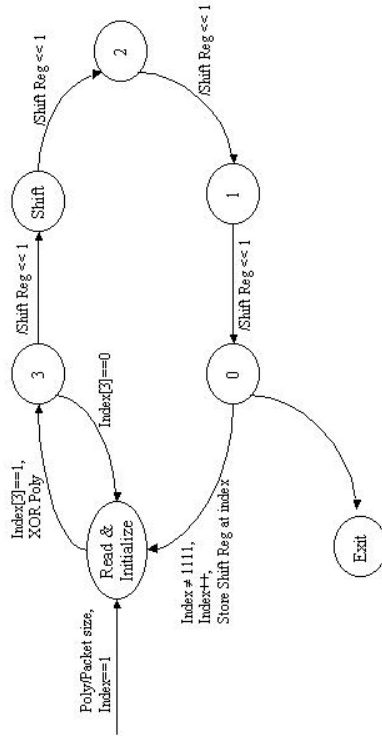


Figure 17. CRC Table Generation.

6.6.2.2. CRC Error Detection

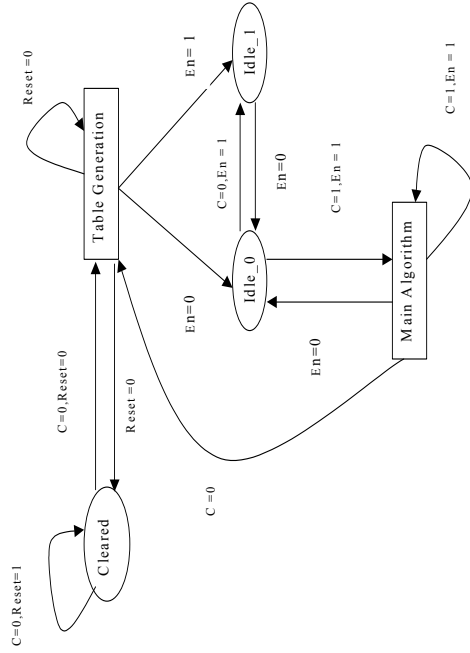


Figure 18. CRC Error Detection.

The state diagram for CRC Error Detection is shown in the figure below.

- 1) In the Cleared state, C = 0 indicates that the table is being constructed and Reset=1 indicates that the control returns back to this state.
- 2) When Reset = 0 and C = 0, the control gets transferred to the state where the table is constructed.
- 3) Depending on the Enable signal, the data packets will be sent for error detection through the input data. When En = 1, the data packets will be sent for error detection.
- 4) C = 1 means that the state is ready to check the errors on incoming data packets.
- 5) During the 'Main Algorithm' state, the error detection process takes place where in the CRC is computed and checked with that obtained in the transmitter side.
- 6) When C = 0, the control is transferred to the 'Table Generation' state.
- 7) During any of these operations if 'Reset' becomes zero the control will be transferred to 'Cleared' state.

6.7. Performance Estimation

The Performance of IEEE 802.11 design depends on the efficiency with which the data can be transferred. Here we use a 16-bit data bus to achieve our purpose.

The performance of data path explained for the CRC generation can be analyzed by counting the number of clock cycles required for the whole process. In the data path, 32 clock cycles are required for the execution of the whole process as we assume that there is no propagation delay for the XOR gates.

On the MAC Receiver side, a 16-bit serial-parallel shifter is used. So, for a 32-bit long data word, two clock cycles are required for the completion of the whole process. Hence a buffer is required for the successful implementation of the logic. In order to improve performance, we can use a 32-bit serial parallel shifter, so the whole CRC frame can be transmitted in a single clock cycle.

The performance can be enhanced further, by using a parallel implementation for a CRC-32 instead of a serial implementation. The performance estimation for CRC-32 discussed here is done for serial implementation in accordance with the requirements of different blocks involved in the design.

7. Exception Handling & Exception Codes

7.1. MAC Receiver Exceptions

The following list of exception codes, shown in the table, are likely to be generated from within the various blocks of the 802.11 MAC Layer Receiver block and its sub-blocks. The bus 'ErrCode' encodes the exception code in a 4-bit value (shown in binary), where the MSB is on the left, LSB is on the right.

#	Code	Exception	Description
1	0000	No Exception	This should be the default value for this bus, and is the value that the Receiver block would make available to the MAC Control block (not part of this project) on completion of processing of the current received frame.
2	0001	CRC Error	Generated by the FCS_Decoder block, this exception indicates that an error in the CRC calculation was detected. It is sufficient to note the error, not where the error actually occurred.
3	0010	Protocol Version Error	Generated by FCH_Decoder block, indicates that a value for the protocol version in the Frame Control word is a value other than the base supported (namely, '00').
4	0011	Frame Type/Subtype Mismatch	Generated by FCH_Decoder, indicates that the value for the subtype doesn't match that of the specified type encountered in the control word. Could indicate a CRC error (which would be found later in processing) or possible intrusion attempt using frame spoofing.
5	0100	Frame Address Sync	Generated in Addr_Decoder block, covers the case where the

	Error	RA is equal to the TA. This is an erroneous case, signifying that the station is sending frames to itself. Might indicate spoofing. Only occurs for RTS and Data frames involving 2 addresses.
6	0101 Fragmentation Sync Error	Generated in the Seq_Control_Decoder block, where we have a Data frame fragment, and the Fragment No. is defined in proper sequence, but is less than '1111'; however, we have the MoreFragments bit reported by the FCH_decoder block set to zero.
7	0110 Erroneous Fragment Error	Generated by the Seq_Control_Decoder block, where the previous frame had a fragment number ('b0000 - b111') with MoreFragments bit set, but the current frame has its MoreFragments bit set but no Fragment No. following the sequence (either zeros or fragment number other than that expected).
8	0111 Duplicate Sequence Number Error	Generated in the Seq_Control_Decoder block, where the Sequence No. is zero, or some non-zero value, for more than one non-fragmented frame in sequence. This implies that the Sequence No. has the same value as that of the previous received frame (must be from the same sender), there is no fragmentation indicated (i.e., the MoreFragments bit is not set) and the Retry bit is also not set.
9	1000 Sequence Sync Error	Generated in the Seq_Control_Decoder block, where the current frame's Sequence No. is some value other than that of the previous frame's, plus one, or if zero (following the wrap of Sequence No. values from 2**12 minus 1, wrapping to zero as the next value in frame sequence). This would be for a given frame sequence from a specific sender address.
10	1001 Address Format Error	Generated by the Addr_Decoder block. The addresses received for TA or RA do not have their MSB 2 bits as zero, as provided by the specification, for the various frame subtypes we are using.
11	1010 Byte Count Error	For each frame of a given type and subtype, we come up short of the total number of words that should be shifted into the Receiver block. Each of these has a pre-defined number of words, so we should maintain a word count to insure that all blocks should be getting their appropriate data. This might manifest itself by the Sequencer getting out of sync with the shifter, in that some of the decoder blocks never get enabled, because there isn't any more data from the PHY Layer to shift and pass on to them. The frame is too short. This could be detected in several places in the design, but perhaps the best place is in the shifter (hence, the word counter that we haven't used up to this point).
12	1011 Retry Sync Error	Detected in the Seq_Control_Decoder block, based on the posting of the RetryFlag by the FCH_decoder block. If the Sequence No. is the next one in the sequence
13	1100 Duplicate Frame Error	Detected in the Seq_Control_Decoder block, based on checking the Sequence No., or Frag No. (if we are fragmenting), and it is the same as the previous frame received from the given sender (and we are the targeted receiver), and the RetryBit detected by FCH_Decoder is not set to '1' indicating this is NOT a retry frame. It could be a reflection of the frame, arriving at a later time than the original frame's image. Presuming we received the original correctly, the Receiver would have already signaled the Transmitter to send an appropriate ACK response. Note this is valid only for Data

14	1101 Retry Frame Encountered	frames. Detected in the Seq_Control_Decoder block, based on checking the Sequence No., or Frag No. (if we are fragmenting), and it is the same as the previous frame received from the given sender (and we are the targeted receiver), and the RetryBit detected by FCH_Decoder is set to '1' indicating this IS a retry frame. In this case, we have already seen a good version of this frame, but a delay in response failed to arrive back to the sender. You may have found some other cases.
15	1110 Future use.	You may have found some other cases.
16	1111 Future use.	You may have found some other cases.

The behavior of the design should be such that, when one block asserts the X_ERR flag and posts its X_ERRCODE, processing in all the blocks should go into an exception handling mode, and control in each thread should return to its base poll loop after any cleanup. This is to conserve power as much as possible, since the assumption of the protocol is that the frame-sender will eventually timeout waiting for a response from the receiver, and will send a retry transmission.

Note that, when we are comparing current versus previous frame Sequence or Fragment numbers, we must be sure that these previously stored numbers are for: (1) a given remote transmitter MAC address; and, (2) for frames that had no exceptions in their processing during reception in our Receiver. In the case of detecting a framing error, we want to not keep the extracted Sequence and/or Fragment number, and we also don't want to save the DID value in the NAV_Register (also when the frame is not addressed to this station).

Note also that determining that a frame is not destined for the receiving station is an exception to normal processing, but it isn't an error. However, we would still stop normal processing of the frame in the same manner. We simply save the DID value in the NAV_Register and prepare to receive the next frame data stream.

7.2. MAC Transmitter Exceptions

The exceptions generated by logic in the Transmitter are indicated in the following table. Your design model needs to incorporate the appropriate logic to generate these, and also to handle them in the MAC Controller block (which would be coordinated with your test harness).

#	Code	Exception	Description
1	0000	No Exception	This should be the default value for this bus, and is the value that the Transmitter block would make available to the MAC Control block on completion of transmitting the current frame.
2	0001	Allocation Retry Count Exceeded	Generated in the Media Allocation control block. Indicates that the transmitter has attempted to allocate the channel, has

3	0010	Frame Transmit Retry Count Exceeded	Completed all of its allowed retries to do so, and yet still cannot allocate the medium for transmission. This results in a failure code being passed back up the network protocol stack.
4	0011	Allocation Timeout	Generated in the Allocate Media block, upon counting down the watchdog timer (a separate thread), indicating that our station has not been able to allocate the medium after counting down the Allocation Timer controlled by the amount of time set in the <i>exponential backoff</i> algorithm. On receipt of this exception, the Media Allocation control block will set up to re-allocate the medium, after first incrementing its Allocation Retry count, and then recalculating the <i>exponential backoff</i> and slot timer value. Checking for this might be optional, depending on how your design manages the allocation timers and the retry process.
5	0100	Transmit Timeout	Generated in the Transmit Frame block, upon counting down the watchdog timer (a separate thread), indicating that our station has not received a good frame response from the remote station with whom we are communicating (e.g., not CTS for a sent RTS within this time window). On receipt of this exception, the Transmit Frame block will set up to retransmit the frame, after first incrementing the Retry count in the frame's header control word.
6	0101	Invalid Station Address	Generated by the Frame Build block, upon checking the remote station address passed down in the MSDU buffer and finding that this address is not the correct format for 802.11 MAC station addresses.
7	0110	Unknown Station Address	Generated by the Frame Build block, upon checking the remote station address passed down in the MSDU buffer and finding that this address is not in the association list of station addresses in the current BSS.
8	0111 - 1111	Available for future expansion	During the course of the project, you may identify additional exceptions that you'd like to manage in your design. Use these unassigned exception codes for this purpose. You'll need to document your code assignments in your project report.

The basic flow of control in the thread for exception processing (for the Receiver and Transmitter blocks) will look much the same. The exception processing thread should wait in a poll loop on power-up reset until the master exception flag is thrown. Once detected, this thread checks which flags have been set and by which function units.

Poll on exception flag from each of the Receiver's threads (OR them all together) to respond if any of the frame decoding blocks detects an exception or frame error.

If one or more threads throw an exception, then "handle" the exceptions.

Questions: Given the set of exceptions and error codes discussed in the Specifications and the Lectures, what is the prioritization of these exceptions? How should they be parsed for processing by this block? What exceptions result in telling the Receiver's decoder threads to

ignore the remainder of the thread? What exceptions result in telling the controller about the status of the Received frame operation?

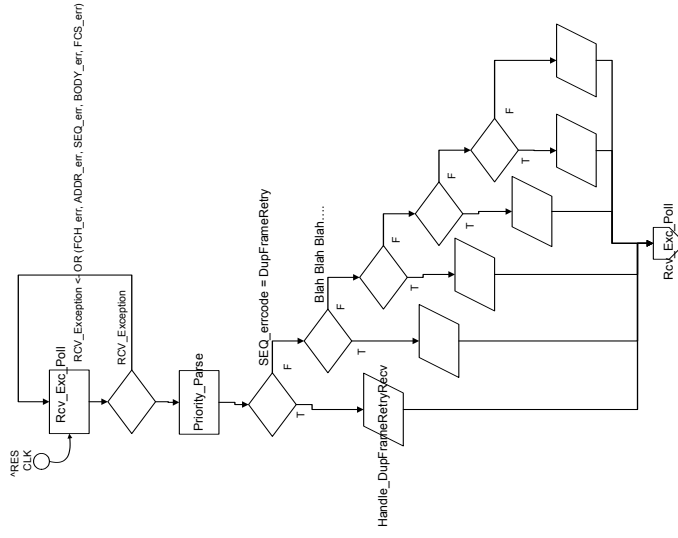


Figure 19. Exception Handling Loop – General Form.

You will want to provide some thought (and some documentation) as to your design team's exception handling "policy"; a couple of paragraphs in your final project submission is appropriate.

8. References

- [1] Matthew S. Gast, 802.11 Wireless Networks: The Definitive Guide, O'Reilly and Associates, 2002.
- [2] Petric, A., and B. O'Hara, *The IEEE 802.11 Handbook: A Designers Companion*, IEEE Press, 1999.

- [3] Institute of Electrical and Electronics Engineers, *Supplement to IEEE Standard for Wireless Networking, 802.11b*, IEEE, New York, 1999.
- [4] Kurose, J.F., and K.W. Ross, *Computer Networking: A Top-Down Approach Featuring the Internet, 2nd ed.*, Addison Wesley Publishers, Inc., 2003.
- [5] S. Dhulipala, D.C. Kovuri, D.L. Kadiri and K.D. Bhatt, "Architecture for Cyclic Redundancy Check Block for IEEE 802.11 WLAN Standard", *CSCE 613 Final Project*, Spring 2002.
- [6] Sheridan, L. "Design and Analysis of a Custom Computing Architecture for the 802.11b Wireless Network Protocol", *Honors Thesis*, South Carolina Honors College, University of South Carolina, 2004.
- [7] *flowHDL Users Manual*, © 1996 Knowledge Based Silicon Corporation.
- [8] *flowHDL Reference Manual*, © 1996 Knowledge Based Silicon Corporation.
- [9] Brenner, P., A Technical Tutorial on the 802.11 Protocol, BreezeCom Wireless Communications White Paper, 1997, at http://www.sss-mag.com/pdf/802_11tut.pdf