



CSCE 491 Project Specification Part 1

Dr. James P. Davis
jimdavis@cse.sc.edu

We will start our process of analysis, architecture and design in this course 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] (see reference at end of this specification), but also the various Lecture Notes I have created for the class.

Here's how we will do this: (1) I've created the specification, which you'll start working with and modeling pieces of it using flowHDL in order to get used to using the tools, (2) you'll work on these together, either individually or as teams, to complete them within the required time frames, (3) we'll incrementally add functionality and relax some of the simplifying assumptions, and (4) we'll integrate the blocks and simulate them together, then synthesize a circuit. During the class lectures, we'll discuss the design process, your progress through the design projects, and any issues involving the 802.11 specification or other project matters that come up—in addition to the VLSI 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 next few weeks working on these designs—analysis, design, verification, refinement and extension.

1. Project Assignment

1.1 Serial to Parallel Bit Shifter and Control Block— This one we have already spend consider time with, so it is not covered further in this specification.

1.2 Frame Control Decoder block

1.3 Address Decoder block

1.4 Frame Body Decoder

1.5 Sequence Control Decoder

1.6 CRC Decoder block

These blocks are to be analyzed (according to this specification, and referencing the 802.11 specification in [1]), designed and verified. We'll refine these in later iteration passes. 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 (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 testbench data by using flowHDL's abilities to specify individual threads as being part of the model or part of the testbench. 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. We will define the specific delivery dates for each block later during class, and I'll post it on the Assignments web page.

NOTE 3: You will work using a single flowHDL file for your entire 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 with design threads—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. IEEE 802.11 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 ERR flag and load the ERRCODE register with the appropriate error code—to be defined separately. All blocks will thus have these two signals defined as outputs. 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.).

2.1 Frame Control Decoder block

2.1.1 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
ERR	Output	1 bit
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.

2.1.2 Functionality:

2.1.2.1 Wait for Enable_FCD to go high (using a Wait on Enable Loop in the flowdiagram, as in the Shift block I have done).

2.1.2.2 "Latch" the data from SHFTOUT_BUS into internal register.

2.1.2.3 Do frame checking on the following parts of the Header:

-- Check the protocol version (default is 'b00', anything other than that should cause the flag setting ERR = 1).

-- 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.)

-- 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 properly, 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).

2.1.2.4 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.

2.1.2.5 set the ERR and ERRCODE registers with values (ERR = 0 means "no error").

2.1.2.6 Additional Byte Count Checks

- 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.

- 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).

- new checks:

- 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').
- 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.
- 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.

2.1.3 Test Data:

You will need to come up with specific test data for your block, at least 4-5 test scenarios (see above).

2.2 Address Decoder block

2.2.1 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_Counter	Input	16 bits (from Sequence Control Decoder block)
FrameByteCount	Input	12 bits
SenderAddr	Output	48 bits
ERR	Output	1 bit
ERRCODE	Output	4 bits

2.2.2 Functionality:

2.2.2.1 Wait for Enable_AD to go high (wait loop)

2.2.2.2 "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 make sure that we have new data.

2.2.2.3 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.

-- Assume the MAC address for our Receiver is 'h4FFFFFFFF044'.

-- Assume the IBSS address is 'h4FFFFFFEE11'.

2.2.2.4 Checking on integrity of addresses: we need to check that the address 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).

2.2.2.5 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.

(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.)

2.2.2.6 set the ERR and ERRCODE registers with values (ERR = 0 means "no error").

2.3 Frame Body Decoder block

2.3.1 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
ERR	Output	1 bit
ERRCODE	Output	4 bits

2.3.2 Functionality:

2.3.2.1 Wait for Enable_FBD to go high (wait loop)

2.3.2.2 "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).

2.3.2.3 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.

2.3.2.4 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.)

2.3.2.5 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).

2.3.2.6 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).

2.4 Sequence Control Decoder block

2.4.1 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
ERR	Output	1 bit
ERRCODE	Output	4 bits

2.4.2 Functionality:

2.4.2.1 Wait for Enable_SCD signal to go high

2.4.2.2 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.

2.4.2.3 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.

2.4.2.4 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 (anymore 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 SFHTOUT_BUS. This will be used by the Frame Body Decoder to reassemble the fragments into a complete frame.

2.4.2.5 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.

2.4.2.6 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 $\text{NewFragNo} = \text{OldFragNo} + 1$, 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).

2.4.2.7 You'll generate ERR and ERRCODE values and output.

2.5 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.

2.5.1 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 Cotrol Decoder block)
SCD_Counter	Input	16 bits (from Sequence Cotrol Decoder block)
FrameByteCount	Input	12 bits
ERR	Output	1 bit
ERRCODE	Output	4 bits

2.5.2 Functionality:

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

2.5.2.2 Latch the SFHTOUT_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.

2.5.2.3 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.

2.5.2.4 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.

2.5.2.5 The CRC algorithm and approach we'll use is specified in the following specification completed for last semester's CSCE 613 class, and is attached as a separate document [2].

3. References

[1] Matthew S. Gast, 802.11 Wireless Networks: The Definitive Guide, O'Reilly and Associates, 2002..

[2] 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.

[3] *flowHDL Users Manual*, © 1996 Knowledge Based Silicon Corporation.

[4] *flowHDL Reference Manual*, © 1996 Knowledge Based Silicon Corporation.

