14:332:438 - Capstone Design, Digital Systems

802.16-2009 OFDM PHY - Digital Realm Transmitter Implementation

Cody Schafer

Submitted in partial fulfillment of the requirements for the
Bachelor of Science Degree

May 8, 2012

Electrical and Computer Engineering Dept.
School of Engineering
Rutgers University

Contents

1 Project Overview

We are developing a minimal 802.16-2009 OFDM Transmitter PHY. This PHY supports only QPSK-1/2 modulation-convolution rate. Additionally, any components not relevant to unlicensed bands are omitted. All optional items are omitted.

This PHY requires the cooperation of a MAC layer implementation which is compliant with the 802.16-2009 standard in order for the compliance expectations (subsection 1.2) to be met.

1.1 End Market Expectations

Currently IEEE 802.16, also known as "WiMAX", has seen little deployment in the United States and western Europe while being moderately deployed in Asian nations and Eastern European markets. To be useful in western areas, WiMAX has to effectively compete with both consumer owned Wifi hot-spots and the various cell phone networks. Given the impressively low cost of both cellular and Wifi (802.11) systems combined with the expectation that they work well in mobile devices, this physical layer (PHY) targets both simplicity (to reduce cost) and reduced power consumption.

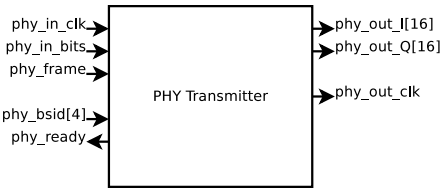

Figure 1: Overall Transmitter Interface

1.2 Degree of standards compliance and scope limitations

The 802.16 standard specifies multiple layers of a WiMAX device, including a "Service-specific convergence sublayer" [1, section 5], a MAC sublayer [1, section 6], a Security sublayer [1, section 7], and the Physical layer [1, section 8].

The PHY, while being described within section 8 of the 802.16-2009 document, is additionally subject to certain constraints and requirements stipulated within the other sections. These sections where only utilized to the extent to which they apply to design of the PHY component.

Applicable sections of [1].

- Section 8.3 - OFDM description
- Section 8.3.3.4.1 - Data Modulation: implementation is not compliant with this section, only QPSK modulation is supported.
- 8.3.3.4.3 - Rate ID encodings: not compliant with this section, only the QPSK-1/2 rate ID is supported.
- 8.3.5.1.1 - DL subchannelization: Rate ID

2 OFDM Parameters

2.1 Primitive Parameters

BW : This is the nominal channel bandwidth.

N_{used} : Number of used subcarriers.

n : Sampling factor. This parameter, in conjunction with BW and N_{used} determines the subcarrier spacing, and the useful symbol time.

G : This is the ratio of CP time to "useful" time.

2.2 Derived Parameters

N_{FFT} : Smallest power of two greater than N_{used}

Sampling Frequency : $F_s = floor(n \cdot BW/8000) \times 8000$

Subcarrier spacing : $\Delta f = Fs/N_{FFT}$

Useful symbol time : $T_b = 1/\Delta f$

CP Time : $T_g = G \cdot T_b$

OFDM Symbol Time : $T_s = T_b + T_g$

Sampling time : T_b/N_{FFT}

3 External Interface to the PHY Transmitter

The MAC interfaces with the PHY module by sending frames (with the appropriate headers and padding included) as a stream of bits. These bits are clocked via the **mac_clk_in** line, and must only be sent when the **mac_sending_frame** line is high. The state of the **mac_sending_frame** line must be low when not sending a frame, and must be lowered and raised between frames that would otherwise be abutting. The frequency of **mac_clk_in** must be less than or equal to half of the **phy_clock**.

The **phy_ready** line is a signal to the mac that it is ready for a new frame to be inputted, and should not be ignored.

The structure of individual frames detailed in section 8.3.5 of IEEE 802.16-2009. Each frame contains a header, up to 4 DL sub-frames (each with their own structure also defined in IEEE802.16-2009 section 8.3.5) and some number of UL sub-frames. Note that due to fixing certain implementation parameters, some fields are constrained further than mentioned within the standard.

Frame Header - Rate_ID is fixed at '1' indicating CPS modulation with a code rate of $\frac{1}{2}$.

Name	Width	Direction	Description
phy_out_I	16	O	The real component of the output, clocked by phy_out_clk
phy_out_Q	16	O	Imaginary component of the output, clocked by phy_out_clk
phy_out_clk	1	O	Clocks out the I and Q values produced by the PHY.
phy_in_bits	1	I	Input bitstream from a MAC device.
phy_in_clk	1	I	Clock at which the input bitstream phy_in_bits should be sampled.
phy_in_frame	1	I	Set low while the current set of bits is from the same frame. Must be set high for one clock cycle between frames.
bsid	4	I	The lower four bits of the BSID, a unique identifier.
bw	3	I	Channel bandwidth. See Table 2.

Table 1: External Interface to the PHY transmitter.

bw[3]	Bandwidth in MHz
0	1.25
1	1.5
2	1.75
3	2.0
4	2.75
5 - 7	Undefined

Table 2: Meanings of BW values.

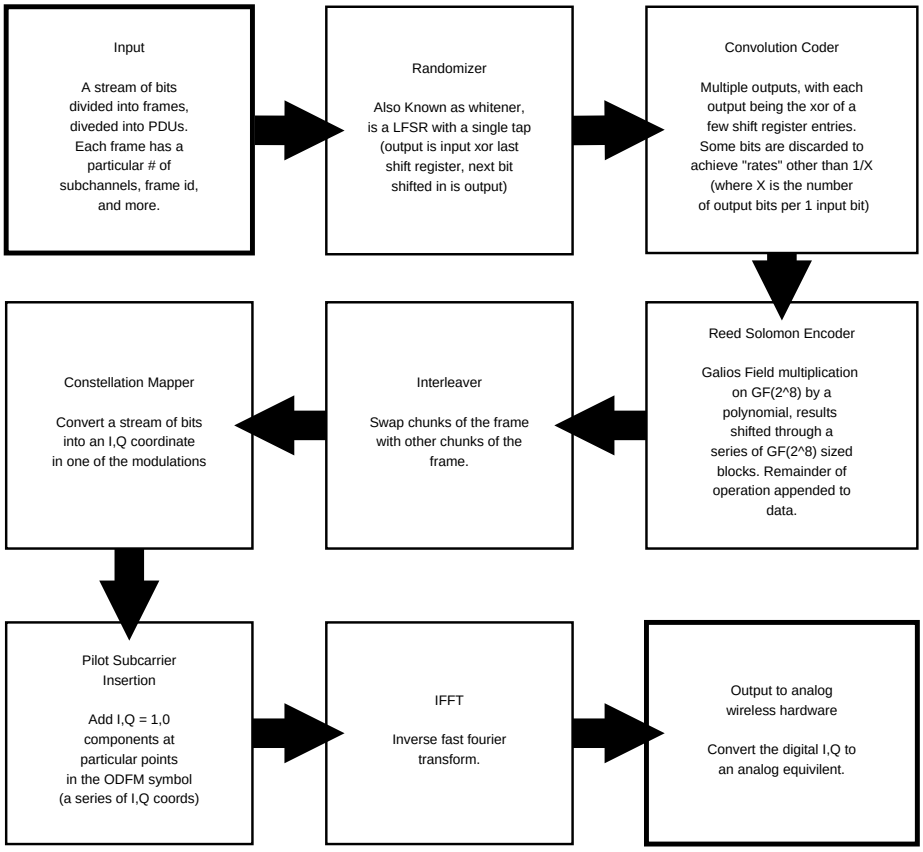

Figure 2: Basic flow of data through the transmitter

4 Internal Interfaces

This section covers the interfaces for internal structures within the PHY transmitter which are not exposed for use by the MAC or any other interfacing hardware.

Table 3 shows some of the common signals for the internal interfaces. Figure 2 shows the basic flow of information through the internal modules of the transmitter. Issues regarding timing, data path widths, buffering, and some minor data stream modifications are omitted.

Each block of the transmitter connected via direct wiring (without a buffer) is given the same clock. Each block reads its inputs on alternating clock edges such that two adjacent units read and write on different edges. This is done so that outputted data does not change while being read.

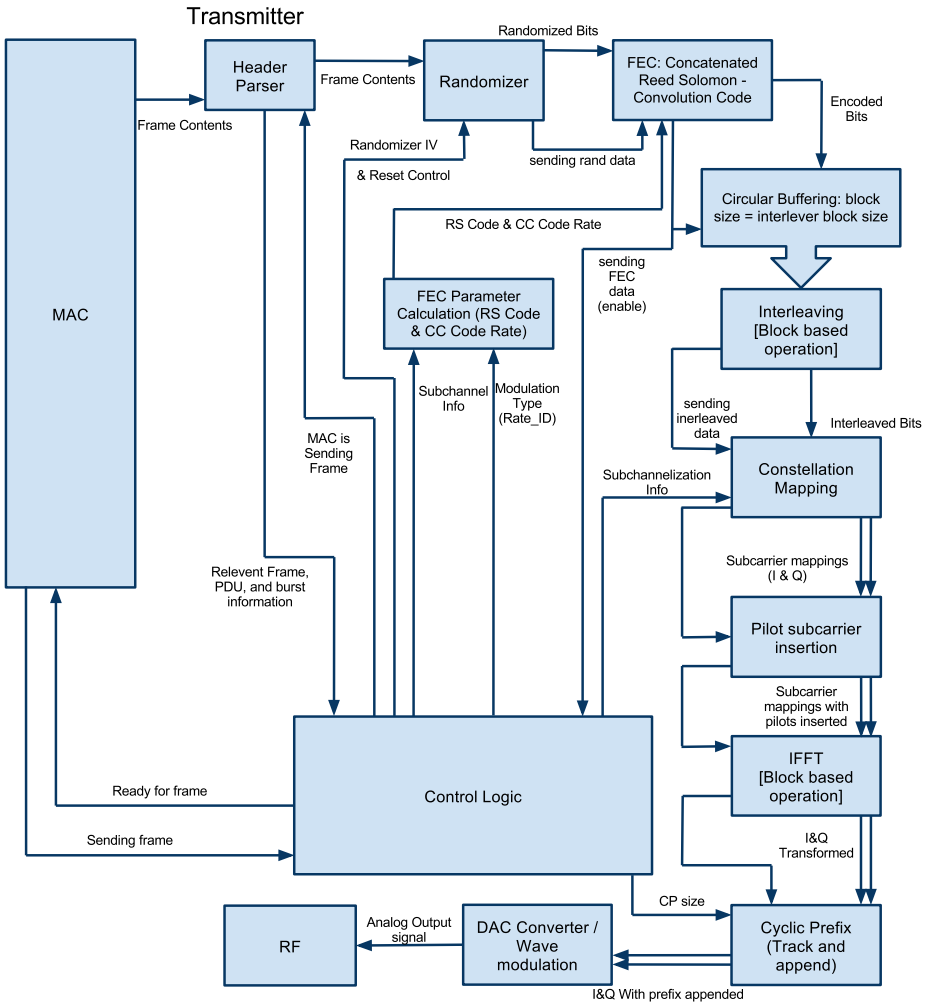

Figure 3: More detailed diagram of the transmitter

Name	Active Level	Meaning
`*_valid`	high	Indicate the source outputting the signal is also outputting data (clocked via the `phy_clock`) which should be processed by the next item in the chain.
`*_bits`	high	A serial stream of bits clocked by `phy_clock`. Only valid when the corresponding `*_valid` line is also active.
`*_flag`	high	Set active for a single clock cycle before becoming inactive again.

Table 3: Common signals used internally

5 Implemented Modules

5.1 Randomizer

Estimated gates 1000

Estimated data bitstream delay 8 clock cycles for a bit to be processed, could be cut to zero by bypassing.

The randomizer operates as a shift register with an initial value determined by the PDU (packet data unit, one half of a frame) header and whether it is a UL (uplink) or DL (downlink) PDU.

Internally, the randomizer is implemented as a shift register with a single feedback. Table 4 describes the inputs and outputs of the randomizer. Figure 4 shows the block representation of the randomizer.

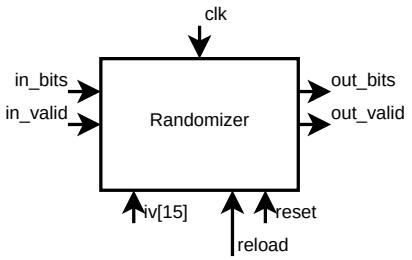

Figure 4: randomizer block diagram

5.2 Forward Error Correction

Estimated gates 2000

Estimated data bitstream delay Slightly more than $fracN_{cbps}2 + 8$

This is a Reed-Solomon and convolution coding combination which is applied per frame. Within the standard, different RS (Reed-Solomon) codes and CC (convolution code) rates are used for varying modulation types. As we have fixed the modulation and code rate to QPSK-1/2, one two of the RS code and CC rate pairs are needed, as indicated in the Table 5.

Name	Width	Direction	Description
`reset`	1	I	Resets all internal state immediately. The internal register is loaded from `iv`. Active low.
`clk`	1	I	Clock.
`in_bits`	1	I	Uncoded input bitstream to the randomizer (this is the first step in the encoding process).
`in_valid`	1	I	Indicates the input bitstream `in_bits` is valid and should be read.
`out_bits`	1	O	Output bitstream.
`out_valid`	1	O	Indicates the output bitstream is valid.
`iv`	15	I	Initialization data for the internal register.
`reload`	1	I	Indicates the internal register should be loaded with `iv`.

Table 4: Randomizer interface definition.

Modulation	Uncoded block size (bytes)	Coded block size (bytes)	Overall coding rate	RS code	CC code rate
QPSK	24	48	1/2	(32,24,4)	2/3

Table 5: Forward Error correction rates

See Table 6 for a listing of the inputs and outputs.

5.2.1 Notes on delay

Both the Reed-Solomon and convolution encoding append additional bits to the bitstream. The convolution encoder emits at most 2 bits for every one bit of input (the 1/2 rate, used with BPSK). The Reed-Solomon encoder appends the residue of its operation to the tail of the bitstream. As it operates of $GF(8)$, 8 bits will be appended. Additionally, between the RS and CC additional bits are appended to such that the bitstream is kept at a multiple of N_{cbps} bytes.

5.3 FEC to Interleaver buffer

Estimated gates 1000

Estimated data bitstream delay None, all delay is included in the calculation for the Interleaver itself.

Table 7 describes the inputs and outputs of this logic block. Transitions the data pipline from a sequence of bits to a sequence of blocks.

Name	Width	Direction	Description
`reset`	1	I	Active low, resets logic block to initial state.
`clk`	1	I	Clocking.
`in_bits`	1	I	Input bitstream, processed by randomizer.
`in_valid`	1	I	Indicates the input bitstream `in_bits` should be read.
`out_bits`	1	O	Output bitstream, has FEC applied. Will be longer than input bitstream.
`out_valid`	1	O	Indicates the output bitstream is valid (the data on it should be read and processed)
`rate_id`	3	I	The type of modulation used. See Table 15 for possible values.

Table 6: FEC interface description

Name	Width	Direction	Description
`reset`	1	I	Active low reset.
`clk`	1	I	Clock.
`fec_out_bits`	1	I	The bitstream outputed by the forward error correction unit.
`fec_out_valid`	1	I	Indicates the bitstream `fec_out_bits` is valid and should be buffered.
`block`	768	O	A block of buffered data. (768 bits = 96 bytes).
`block_valid`	1	O	Indicates the data block `block` is valid and should be processed.
`advance`	1	I	Asserted by the consumer to indicate that the current data block has been processed completely and a new data block is required.

Table 7: FEC output buffer description

Name	Width	Direction	Description
`reset`	1	I	resets the device state.
`clk`	1	I	clocking.
`in_blk`	384	I	Block of input data.
`in_blk_valid`	1	I	Indicates a block of input data on `in_blk` is valid and should be processed.
`rate_id`	3	I	The modulation type. Possible values can be found in Table 15.
`subchan_ct`	3	I	Indicates whether the number of subchannels is 1, 2, 4, 8, or 16. These are paired with values 0, 1, 2, 3, and 4, respectively. The result of other values is undefined. This input allows determination on how many bits in `in_blk` should be processed.
`out_blk`	384	O	Interleaved block which is then outputted.
`out_blk_valid`	1	O	When high, indicates that `out_blk` contains valid data which should be processed by the next item in the pipeline.

Table 8: Interleaver I/O description

5.4 Interleaver

Estimated gates 2000

Estimated data bitstream delay About 392 clocks.

All encoded data bits will be interleaved by this block interleaver. The block size is defined by the number of coded bits per allocated subchannels per OFDM symbol (N_{cbps}). N_{cbps} is determined by a combination of the number of subchannels and the modulation used, see Table 9 for a listing of possible values.

Subchannels	16	8	4	2	1
BPSK	192	96	48	24	12
QPSK	384	192	96	48	24
16-QAM	768	384	192	96	48
64-QAM	1152	576	288	144	72

Table 9: Values of N_{cbps}. Note that other than the values listed for QPSK, none of these items are supported by this device.

Interleaving is a two step type operation. The first step ensures adjacent coded bits are mapped onto nonadjacent subcarriers and the second step ensures bits are mapped alternately onto less and more significant bits of the constellation. The second step is done to avoid long runs of lowly reliable bits.

The permutations are dependent on the number of coded bits per subcarrier (N_{cpc}), which is defined by the type of modulation and the index of the coded bit before permutation (k).

For our implementation, as we are only using QPSK modulation, $N_{cpc} = 2$.

k : index of coded bit before first permutation

m_k : index after first permutation

j_k : index after second permutation

First permutation equation :

$$m_k = (N_{cbps}/12) * k_{mod12} + floor(k/12) \tag{1}$$

Second permutation equation :

$$s = ceil(N_{cpc}/2) \tag{2}$$

$$j_k = s * floor(m_k/s) + (m_k + N_{cbps} - floor(12 * m_k/N_{cbps}))_{mod(s)} \tag{3}$$

After the interleaving the first bit out of the interleaver would map to the MSB for the constellation mapping.

5.4.1 Testing and Verification

The Interleaver operation can be verified by inputting a known block of bits and observing the output. The standard document provides a few of these for our use.

The verilog code for this module was generated via a python script which directly implements the permutation equations (**??**). Additionally, a partial non-generated implementation was created, the equations for which are verified by the same script which generates interleaver code.

5.4.2 Implementation Notes

Due to the complexity of the permutation, code was written which generates the verilog implementation for all possible variations in N_{cpc}. In a further iteration of the transmitter, a multiplexer would be used to switch to the appropriate interleaver for each type of modulation. A partial implementation of this multiplexing was written.

The interleaving operation is such that bits much later in the bitstream a swapped with those much earlier in the bitstream. Due to this, the entire OFDM symbol (N_{cbps} bits) must be read into the module prior to a single 1 clock operation to swap the bytes as required. This leads to a rather high cost in delay for this module.

Name	Width	Direction	Description
`reset`	1	I	When low, the chip is reset. When high, normal operation.
`clk`	1	I	Clocks all synchronous operations.
`in_bits`	1	I	The input bitstream.
`in_valid`	1	I	Indicates that `in_bits` is valid and should be read.
`out_Q`	16	O	output of Q's. Represents an analog value
`out_I`	16	O	output of I's. Represents an analog value
`out_valid`	1	O	Indicates both `out_Q` and `out_I` are valid and should be processed or buffered immediately.
`subchan_data`	6	I	Indicates the mapping of subchannels, see Table 16 for explanation of values.

Table 10: Constellation Mapping input output description

6 Planned Modules

6.1 Constellation Mapping

Estimated gates 4000

Estimated data bitstream delay N_{cpc}, the number of bits per constellation. 4 in the case of QPSK.

The Constellation mapping module takes our bitstream and converts it to a form for an analog radio transmitter. I is used to control the 'real' portion of the final output signal while Q controls the 'imaginary' component. By placing the Q and I as axis in a 2 dimensional grid, outputs of particular positions are taken to represent particular binary numbers. In BPSK, only 2 outputs are possible, thus the number of bits in the constellation (a particular set of positions on the 2 dimensional grid) is 2. For QPSK, the number of bits per constellation is 4. For this reason, the constellation mapper needs to wait for enough bits to accumulate before it can output See Table 10 for the description of inputs and outputs.

6.2 Pilot Sub-carrier Insertion

Estimated gates 1000

Estimated data bitstream delay 1 cycle, if implemented with a bitstream width equal to N_{cbps}.

Input is (I,Q) pair, output is (I,Q) pair with pilot subcarriers inserted at some points (fixed).

Name	Width	Direction	Description
`reset`	1	I	Resets the device state.
`clk`	1	I	Clocking.
`in_blk`	$200 \cdot 2 \cdot 16$	I	Input data from the Pilot Insertion.
`in_blk_valid`	1	I	When high, indicates that the input block is valid and should be processed.
`out_blk`	$200 \cdot 2 \cdot 16$	O	Output data to the IFFT. Width of numbers is 16 bits, processing both Q and I.
`out_blk_valid`	1	O	Indicates the output block `out_blk` is valid and should be imediately copied (it will be replaced by the next block on the next clock cycle).

Table 11: Sub-carrier to IFFT Buffer input/output description.

6.3 Sub-carrier to IFFT Buffer

Estimated gates 2000

Estimated data bitstream delay 1

Insertion into this buffer is ordered but will have random jumps around different UL & DL bursts.

Block Size = blk_siz = 200 (the total number of used subcarriers)

Buffers blk_size items each of which has width 2 bits. QPSK utilizes both I and Q, amplitude and phase, each with a granularity of 2. This means that each item (a pair of I & Q) has 4 possible values and thus 2 bits are needed for each.

6.4 IFFT

Estimated gates 10,000

Estimated data bitstream delay

The IFFT must process 256 complex numbers with a depth of 16 bits for each component (16 bits for the Real component and an another 16 bits for the imaginary).

See Table 12 for a description of the inputs and outputs.

Buffers blk_size items each of which has width 2 bits. QPSK utilizes both I and Q, amplitude and phase, each with a granularity of 2. This means that

Name	Width	Direction	Description
`reset`	1	I	Resets the device state.
`clk`	1	I	Clocking.
`in_blk_valid`	1	I	When high, indicates that the input block is valid and should be processed.
`in_blk`	$200 \cdot 2 \cdot 16$	I	Input data for the IFFT. Width of numbers is 16 bits, processing both Q and I.
`out_blk`	$256 \cdot 2 \cdot 16$	O	Output data after IFFT is preformed. Width of numbers is 16 bits, processing both Q and I.
`out_blk_valid`	1	O	Indicates the output block `out_blk` is valid and should be immediately copied (it will be replaced by the next block on the next clock cycle).

Table 12: IFFT input/output description.

each item (a pair of I & Q) has 4 possible values and thus 2 bits are needed for each.

The IFFT takes in 256-bit complex samples and outputs to the Cyclic Prefix.

6.5 Cyclic Prefix

Estimated gates 2000

Estimated data bitstream delay About 1050 clocks.

The Cyclic Prefix is used to act as protection from Intersymbol Interference by duplicating a certain number of symbols defined by the CP_rate.

The Cyclic Prefix appends to the front of the symbol a copy of the last $256/CP_{rate}$ (N_{IFFT}/CP_{rate}) bits. OFDM PHY should allow for four (4) rates as defined by 2 bits in the PHY Mode ID field.

It reads 256 2 bit items from the IFFT into 2 256 blocks of memory. On output, the appropriate number of bits are read from the end of memory by offset.

At a rate of 1/4, 64 bits would need to be read from the end and at a rate of 1/32, 8 bits would need to be read.

The sample size would then increase from 256 bits to anywhere between 264 to 320 bits.

For our implementation, we will set the CP rate at 1/16 which leads to a sample size of 272 bits.

Value of param_G	Ratio of CP time to "useful" time.
0b00	1/4
0b01	1/8
0b10	1/16
0b11	1/32

Table 13: Values of OFDM parameter G as presented on param_G

Name	Direction	Width	Description
param_G	I	2	The meanings of valid values are listed in Table 13. Indicates the fraction of CP time to "useful" time.
cp_in_bits	I	1	Input bitstream.
cp_in_valid	I	1	Indicates the input bitstream cp_in_bits is valid.
cp_out_bits	O	1	Output bitstream.
cp_out_valid	O	1	Indicates the output bitstream cp_out_bits is valid.

Table 14: Cyclic Prefix Inputs and Outputs

This process would take about about 512 clocks to read in the data to fill memory and about 528 clocks to write out.

The total time would include clocks to set and check valid lines and would total to about 1050 clocks.

Table 14 lists the inputs and outputs to the cyclic prefix unit.

Appends the last Tg items of the previous frame to the start of the present frame. Reads 256 by 2 16 bit items from the IFFT, appending Tg items to the start of it. It also stores the last Tg items so that they may be appended to the next frame from the IFFT.

6.6 Frame Handler

Estimated gates 2000

Estimated bitstream delay 2 clock cycles for a bit to be processed

Is given a bitstream directly from the MAC device, expected to be able to clock in this bit stream at the rate defined by the MAC. Data and parsed configuration is sent to the Burst Handler (subsection 6.7) to be processed and outputed.

Inputs and outputs are listed in Table 17.

6.7 Burst Handler

Estimated gates 2000

rate_id[3]	Modulation RS-CC rate
0	BPSK-1/2
1	QPSK-1/2
2	QPSK-3/4
3	16-QAM-1/2
4	16-QAM-3/4
5	64-QAM-2/3
6	64-QAM-3/4
7	Reserved

Table 15: Possible values for `rate_id`. Note that All values but 1 (QPSK-1/2) are unsupported by this device.

Table 16: Possible values for `subchan_data`.

Name	Width	Direction	Description
`clk`	1	I	Clocks the logic block.
`reset`	1	I	Active low reset to initial state.
`frame_in_bits`	1	I	Input bitstream, clocked via `frame_in_clk`.
`frame_in_clk`	1	I	Positive edge indicates that data on `frame_in_bits` should be read.
`frame_in_valid`	1	I	Set high while the current series of bits is part of the same frame.
`burst_reset`	1	O	reset line to the burst handler (subsection 6.7).
`burst_bits`	1	O	Bits to the burst handler.
`burst_valid`	1	O	indicates the bitstream `burst_bits` is valid and should be processed.
`frame_num`	4	O	The frame number parsed from the header structures.
`iuc`	4	O	The UIUC or DIUC field, parsed from the header stuctures. Only valid for the currently outputed burst.
`subchan_data`	6	O	See Table 17. Table 16 describes valid values. Per burst.
`rate_id`	3	O	See Table 15. Per burst.

Table 17: Frame Handler interface description

Estimated data bitstream delay 2 clock cycles for a bit to be processed, could be cut to zero by bypassing.

Bursts are composed of multiple OFDM symbols. OFDM symbols are composed of multiple sub-channels.Table 18 shows the burst interface inputs and outputs. Across a single burst initialization of the pipeline hardware will not change.

7 Health, Safety, and Environmental Issues

7.1 Product Dangers

Potential for heat, radiation, and distraction while operating equipment (depending on what type of end-user device this implementation is integrated into. As this is only a component of an arbitrary end user device, dangers caused or directly inferred from its use are limited.

7.2 Health Hazards

As a wireless transmitter, certain power outputs at particular frequencies could be dangerous for people to be in close proximity to. These power levels and transmission frequencies are not handled within this project, rather, they fall to the real of the analog transmitter portion.

7.3 Environmental Hazards

As a wireless device, any implementation utilizing this code will need to obtain FCC approval to ensure it properly uses the frequency "space" allocated to it. Notably, the WiMAX specification provides for both licensed and unlicensed frequency range use, meaning additional approval could be required from the owner of the particular band the device operates in.

8 Conclusion

This design failed to be completed. Reexamining the plan of design for this project reveals a few issues that made effective implementation difficult. Planning on a lack of delay communication between hardware modules creates a significant amount of planning difficulty. Every module's input and output speed must be able to be predetermined and buffers (which understand how to handle delays) added. This can result in the use of numerous overly large buffers due to worse case calculations being used to determine the buffer size. Additionally, this plan is made more difficult by multiple 802.16 frames being pushed through the same hardware one after the other. As the number of outputted bits (prior to conversion to wide Q and I signals) is larger than the number of inputted bits, a delay between frame inputs from outside the system or a buffer of infinite size

Name	Width	Direction	Description
reset	1	I	Imediately resets the burst handler. Active low.
clk	1	I	Clocking.
iuc	4	I	The UIUC or DIUC field (depending on uplink or downlink state). Used by randomizer.
bsid	4	I	Used by randomizer.
frame_num	4	I	Used by randomizer.
burst_in_bits	1	I	Input bitstream.
burst_in_valid	1	I	Indicates the input bitstream burst_in_bits is valid.
rand_iv	15	O	The initialization for the randomizer's internal register.
rand_reload	1	O	Instructs the randomizer to reload it's internal register with a new one outputed on rand_iv.
burst_out_bits	1	O	Output bitstream. Sent to the randomizer for first portion of processing.
burst_out_valid	1	O	Indicates output is valid. Also attached to the randomizer.
subchan_data_in	6	I	See Table 17. Table 16 describes valid values.
subchan_data_out	6	O	Output pair of subchan_data_in, simple pass through.
subchan_ct	4	O	Indicates the number of subchannels in use. Calculated from subchan_data_in.
rate_id_in	3	I	See Table 15.
rate_id_out	3	O	Pass-through pair of rate_id_in.

Table 18: Interface definition for the Burst Handler.

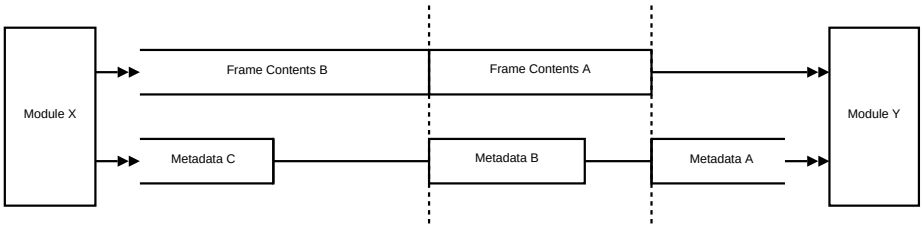

Figure 5: Proposed Bitstream Transfer Pattern

is required. Clearly, infinite flip-flops are not a possibility in any logic circuit, thus the system as a whole would need an understanding of when it was "ready" for more data.

Further along the same track, the current design does not allow for the use of internal modules for separate frames simultaneously. For example, the reed-Solomon encoder cannot be easily used for one frame while the convolution coder begins processing the next. One possible plan to allow this to be done simply is to have a signal between modules for when a frame begins or ends (rather than using a central piece of logic to manage this). Unfortunately, we need more than just knowledge of when the frame begins and ends to properly process it; a whole slew of metadata needs to be passed along while being timed properly with the motion of the bitstream through the system. Additionally this metadata would need to handle delays properly (as using solely buffering is unfeasible). From these requirements, the possibility of using a second bitstream to carry the metadata arises, as shown in Figure 5. Each module could stay as initially designed, but be wrapped in a controller which handles extraction of metadata from the bitstream as well as properly handling it's timing. It could be passed along ahead of the start of a frame such that when the actual data reaches a new module, the necessary metadata for processing it is already present. Of course, additional issues arise if the metadata bitstream can exceed the length of the shortest 802.16 frame: an additional delay would need to be inserted after a short frame to give time for the metadata to be transferred.

Using an FPGA for this is probably not economically viable or efficient. Power consumption would be much higher than could be achieved with a non-FPGA realization. Cost would also be an issue for any product with a non-inconsequential volume of production. As wireless devices are often mobile and battery powered, power consumption is a serious concern. Additionally, using this as portion of a larger FPGA design would not be feasible due to the high use of logic circuit resources by various components, primarily the (planned) IFFT.

A Code Listings

../param.v

```
`ifndef PARAM_V_
`define PARAM_V_

module p_rate_id();
        parameter w = 2;

        parameter BPSK = 0;
        parameter QPSK = 1;
        parameter QAM16 = 2;
        parameter QAM64 = 3;
endmodule

module p_cc_rate();
        parameter w = 2;

        parameter r_1_2 = 0;
        parameter r_3_4 = 1;
        parameter r_5_6 = 2;
endmodule

/* FIXME: this is just thrown together */
/* Mod     16   8   4   2   1
 * BPSK    192  96  48  24  12
 * QPSK    384  192 96  48  24
 * 16-QAM 768  384 192 96  48
 * 64-QAM 1152 576 288 144 72
 */
module p_subchan_id();
        parameter ct = 4;
        parameter w  = 2;
endmodule

module p_subchan_ct();
        parameter ct = 5;
        parameter w  = 3;
endmodule


`endif
```

../common_widths.v

```
`ifndef COMMON_WIDTHS_H_
`define COMMON_WIDTHS_H_

`define UIUC_SZ 4
`define BSID_SZ 4
`define FRAMEN_SZ 4

`define RATEID_SZ 5
`define SUBCHAN_IDX_SZ 5

`endif
```

../func/gen_rand_iv.v

```
1
2  function [14:0] gen_rand_iv;
3          input [3:0] bsid;
4          input [3:0] uiuc;
5          input [3:0] fnum;
6
7          gen_rand_iv = {
8                  fnum[0], fnum[1], fnum[2], fnum[3],
9                  1'b1,
10                 uiuc[0], uiuc[1], uiuc[2], uiuc[3],
11                 2'b11,
12                 bsid[0], bsid[1], bsid[2], bsid[3]
13                 };
14 endfunction
```

../rand.v

```
1  module randomizer(
2          input reset, clk,
3          input in_bits,
4          input in_valid,
5          output reg out_bits,
6          output reg out_valid,
7          input [14:0] rand_iv,
8          input reload);
9
10         reg [14:0] vect;
11         reg nout;
12         reg nvalid;
13
14         /* XXX: EFFICIENCY: there should be a way to do this 8 bits
                at a time,
15          * as mentioned in "OFDM Baseband Transmitter
                Implementation Compliant
16          * IEEE Std 802.16d on FPGA" */
17
18         always @ (posedge clk or posedge reset) begin
19                 if (reset) begin
20                         nout  <= 0;
21                         nvalid <= 0;
22                         vect  <= 0;
23                 end else begin
24                         if (reload) begin
25                                 nvalid <= 0;
26                                 nout  <= 0;
27                                 vect  <= rand_iv;
28                         end else if (in_valid) begin
29                                 nvalid <= 1;
30                                 nout  <= in_bits ^ (vect[13] ^
                                       vect[14]);
31                                 vect  <= { vect[13:0], vect[13] ^
                                       vect[14] };
32                         end else begin
33                                 nvalid <= 0;
34                                 nout  <= 0;
```

```
35                                      vect   <= vect;
36                              end
37                      end
38              end
39
40              always @ (negedge clk or posedge reset) begin
41                      if (reset) begin
42                              out_valid <= 0;
43                              out_bits  <= 0;
44                      end else begin
45                              out_valid <= nvalid;
46                              out_bits  <= nout;
47                      end
48              end
49
50      endmodule
51
52      /* Processes 8 (or some variation between 1 and 14 bits) at a time
            */
53      module rand_parm
54              #(parameter bits_pclk = 8)(
55              input reset, clk,
56              input [bits_pclk-1:0] in_bits,
57              input in_valid,
58              output reg [bits_pclk-1:0] out_bits,
59              output reg out_valid,
60              input [14:0] rand_iv,
61              input reload);
62
63              reg [14:0] vect;
64              reg [bits_pclk-1:0]  nout;
65              reg nvalid;
66
67              always @ (posedge clk or posedge reset) begin
68                      if (reset) begin
69                              nvalid <= 0;
70                              nout   <= 0;
71                              vect   <= 0;
72                      end else begin
73                              if (reload) begin
74                                      nvalid <= 0;
75                                      nout   <= 0;
76                                      vect   <= rand_iv;
77                              end else if (in_valid) begin
78                                      nvalid <= 1;
79                                      nout   <= in_bits ^ (vect[13 -:
                                            bits_pclk] ^ vect[14 -:
                                            bits_pclk]);
80                                      vect   <= { vect[14 - bits_pclk:0],
81                                              vect[13 -: bits_pclk] ^
                                                    vect[14 -: bits_pclk]
                                                    };
82                              end else begin
83                                      nvalid <= 0;
84                                      nout   <= 0;
85                                      vect   <= vect;
86                              end
```

```
                end
        end

        always @ (negedge clk or posedge reset) begin
                if (reset) begin
                        out_valid <= 0;
                        out_bits  <= 0;
                end else begin
                        out_valid <= nvalid;
                        out_bits  <= nout;
                end
        end

endmodule
```

../gf.v

```

`ifdef NOT_DEF
module aopm_unit(
        input clk, reset,

        output xs,
        output and_out);
endmodule


module aopm
        #(
        parameter m = 8,
        parameter x_init = 0,
        )(
        input clk, reset,

        input b, c,
        output a
        );



endmodule

/* 2^8 gfa mult for p(x) = x^8 + x^4 + x^3 + x^2 + 1,
 * bit serial methodology */
module gfa_mult
#(parameter w = 1)(
        input  reset, clk,
        input  [w-1:0] bit_in,
        output [w-1:0] bit_out
);

        reg [w-1:0] b;

        always @(posedge clk or posedge reset) begin
                if(reset) begin
                        b = {w{1'b0}};
                end else begin
```

```
41                  end
42          end
43
44          always @(negedge clk or posedge reset) begin
45                  if (reset) begin
46                  end else begin
47                  end
48          end
49
50  endmodule
51
52  `endif
53
54  /* Design from: "An efficient reconfigurable multiplier
        architecture
55  * for Galois field GF(2^m)" (Kistos et al.)
56  * After m clock cycles the right multiplication result
57  * is stored in the LFSR. */
58  module gf_mult
59          #(
60          parameter m = 8, /* in GF(2^m) */
61          parameter a = 0, /* one of the multiplicands */
62          parameter p = 0 /* the primitive poly */
63          )(
64          input clk, reset,
65
66          input b, /* the second multiplicand */
67          output [m-1:0] c /* multiplied output */
68          );
69
70          reg b_cpy;
71          reg [m-1:0] lfsr;
72
73          wire [m-1:0] partial_products = {m{b_cpy}} & a;
74          wire [m-1:0] feed_back        = {m{lfsr[m-1]}} & p;
75          wire [m-1:0] lfsr_next        = feed_back ^
                partial_products ^ {lfsr[m-1:1], 0};
76
77          always @(posedge clk or posedge reset) if (reset) begin
78                  b_cpy = 0;
79          end else begin
80                  b_cpy = b;
81          end
82
83          always @(negedge clk or posedge reset) if (reset) begin
84                  lfsr = {m{1'b0}};
85          end else begin
86                  lfsr = lfsr_next;
87          end
88
89          assign c = lfsr;
90  endmodule
91
92  `ifdef NOTDEF
93  /* a controller for gf_mult */
94  module gf_mult_ctrl
95          #(
```

```
96          parameter m = 8;
97          parameter a = 0;
98          parameter p = 0;
99          )(
100         input clk, reset,
101
102         input  i, i_valid,
103         output [m-1:0] o, o_valid
104         );
105
106         reg [$clog2(m)-1:0] ct;
107
108         always @(posedge clk or posedge reset) if (reset) begin
109                 ct = 0;
110         end else begin
111                 if (
112         end
113
114 endmodule
115 `endif
```

../rs.v

```
1
2   `include "gf.v"
3
4
5   module rs
6           #(
7           parameter w = 1,
8           parameter m = 8,
9           parameter logm = /* $clog2(m) */ 3,
10          parameter T = 8,
11          parameter log2T = /* $clog2(2*T) */ 4,
12          parameter p = 0/* the primitive polynomial */
13          )(
14          input reset, clk,
15
16          input in_bits,
17          input in_valid,
18
19          output reg out_bits,
20          output reg out_valid
21          );
22
23          localparam g = {
24                  { },
25                  { },
26          };
27
28
29          reg [1:0] state;
30          localparam S_INIT = 0; /* shifting in the first 8 bits,
31                                    no valid output */
32          localparam S_IMED = 1; /* in the middle, output continually
                valid */
```

```
33          localparam S_SHCK = 2; /* shift out the check bits, no
                input
34                                  accepted */
35          localparam S_INVL = 3; /* invalid and unused (only a place
                holder) */
36
37          wire [m-1:0] m_out [2*T-1:0];
38          reg  [m-1:0] m_reg [2*T-1:0];
39          reg  [m-1:0] x     [2*T-1:0];
40
41          wire vin  = in_bits & in_valid;
42          wire vclk = clk     & in_valid;
43
44          /* in_valid on posedge of current clock */
45          reg was_valid;
46
47          /* expected to overflow on exceeding m.
48          * XXX: probably assumed to be 8 at some point, be careful
                */
49          reg [logm-1:0] ct_8;
50
51          /* expected to overflow upon exceeding 2T */
52          reg [log2T-1:0] ct_2T;
53
54          /* the next x values. */
55          wire [m-1:0] xn [2*T-2:0];
56
57          generate
58                  genvar i;
59                  for(i = 0; i < 2*T; i = i + 1) begin : mutls
60                          gf_mult #(.p(p), .a(g[i])) gm (reset, vclk,
                                vin, m_out[i]);
61                  end
62
63                  for(i = 0; i < 2*T - 1; i = i + 1) begin : xnext
64                          assign xn[i] = x[i+1] ^ m_out[i];
65                  end
66          endgenerate
67
68          always @(posedge clk or posedge reset) if (reset) begin :
                pos_reset
69                  integer i;
70                  for(i = 0; i < 2*T; i = i + 1) begin
71                          m_reg[i] = 0;
72                  end
73
74                  was_valid = 0;
75          end else begin : pos_run
76                  integer i;
77                  for(i = 0; i < 2*T; i = i + 1) begin
78                          m_reg[i] = m_out[i];
79                  end
80
81                  was_valid = in_valid;
82          end
83
84          /* FIXME: in state S_IMED, gaps in the input (the valid_in
```

```
    line
 * temporarily going low) are not allowed. Need to keep track of
 * how far along in the block we are to allow gaps. */
always @(negedge clk or posedge reset) if (reset) begin
        out_bits = 0;
        out_valid = 0;
        state = S_INIT;
        ct_8 = 0;
        ct_2T = 0;
end else begin

        if (state == S_IMED) begin
                /* FIXME: does not allow gaps in input, see above. */
                if (~was_valid) begin
                        state = S_SHCK;
                end
                out_valid = 1;
                out_bits = x[0][m-1];
                x[0] = { x[0][m-2:0], 0 };
        end

        if (state == S_SHCK) begin
                ct_2T = ct_2T + 1;

                out_valid = 1;
                out_bits = x[0][m-1];
                x[0] = { x[0][m-2:0], 0 };
                if (~ct_2T) begin : back_to_init
                        integer i;

                        state = S_INIT;
                        out_valid = 0;

                        for(i = 0; i < 2*T; i = i + 1)
                            begin
                                x[i] = 0;
                        end
                end
        end

        if (was_valid || (state == S_SHCK)) begin
                ct_8 = ct_8 + 1;
                if (~ct_8) begin : shift_x
                        integer i;

                        /* the first 8 bits have been shifted in, advance
                        * state */
                        if (state == S_INIT) begin
                                state = S_IMED;
                        end

                        /* Needed as X is an array */
                        for(i = 0; i < 2*T; i = i + 1)
                            begin
```

```
                                        x[i] = xn[i];
                                    end
                                end
                            end
                        end

endmodule


/* reed solomon encoding */
module rs_a
        #(
        parameter w = 1
        )(
        input reset, clk,

        input [w-1:0] in_bits,
        input in_valid,

        output reg [w-1:0] out_bits,
        output reg out_valid
        );

        localparam T = 8;
        //localparam log_T = 3;
        localparam log_2T = 4;

        /* The Reed-Solomon encoding shall be derived from a
            systematic RS (N
         * = 255, K = 239, T = 8) code using GF(2^8),
         *
         * Can be viewed as multiplication by the generator:
         *  c = g * i
         *  where c x = a valid code word (a poly)
         *        g x = the generator polynomial
         *        i x = the information poly.
         *
         * Also: remainder from division:
         */

        /* Field generator polynomial
         * p(x) = x^8 + x^4 + x^3 + x^2 + 1
         */

        /* Code generator polynomial
        *  y = lambda
         * g(x) = (x + y^0)(x + y^1)(x + y^2)...(x + y^(2T - 1)), y
             = 02HE
         */

        /* Galios field operations:
         * addition: xor
         * multiplication: multiplication modulo the generator
             polynomial
         */

        reg in_data;
```

```
190         reg in_data_valid;
191
192         /* when ~in_data_valid, the number of shifts out remaining
                */
193         reg start_T;
194         reg [log_2T-1:0] shift_ct;
195
196         /* 2T flip flops each with width 'w' */
197         reg [w-1:0] b [2*T-1:0];
198
199         /* Outputs of the multipliers */
200         wire [w-1:0] gm_out [2*T-1:0];
201         reg  [w-1:0] gm_in;
202
203         generate
204                 genvar i;
205                 for (i = 0; i < 2*T; i = i + 1) begin : mult_gen
206                         gfa_mult #(.w(w)) mult (reset, clk, gm_in,
                                gm_out[i]);
207                 end
208         endgenerate
209
210         always @(posedge clk or posedge reset) begin
211                 if (reset) begin
212                         in_data <= 0;
213                         in_data_valid <= 0;
214                         start_T <= 0;
215
216                 end else begin
217                         in_data <= in_bits;
218                         if (in_data_valid && ~in_valid) begin
219                                 in_data_valid <= in_valid;
220                                 start_T <= 1;
221                         end else begin
222                                 in_data_valid <= in_valid;
223                                 start_T <= 0;
224                         end
225                 end
226         end
227
228         always @(negedge clk or posedge reset) begin
229                 if (reset) begin
230                         out_bits  <= 0;
231                         out_valid <= 0;
232
233                         gm_in <= 0;
234
235                         begin : reset_ff
236                                 integer i;
237                                 for (i = 0; i < 2*T; i = i + 1)
                                        begin
238                                             b[i] <= {w{1'b0}};
239                                 end
240                         end
241                         shift_ct <= 0;
242                 end else begin
243
```

```
244                         if (start_T) begin
245                                 shift_ct <= {log_2T{1'b1}} - 1;
246
247                                 gm_in <= 0;
248                                 shift_ct <= shift_ct - 1;
249                                 out_bits <= b[2*T-1];
250                                 out_valid <= 1;
251
252                         end else if (in_data_valid) begin
253                                 gm_in <= in_data ^ b[2*T-1];
254
255                                 out_bits <= in_data;
256                                 out_valid <= 1;
257
258                         end else if (shift_ct) begin
259                                 gm_in <= 0;
260                                 shift_ct <= shift_ct - 1;
261                                 out_bits <= b[2*T-1];
262                                 out_valid <= 1;
263                         end else begin
264                                 gm_in <= 0;
265                                 out_bits <= 0;
266                                 out_valid <= 0;
267                         end
268
269                         begin : shift_ff
270                                 integer i;
271                                 for (i = 0; i < 2*T - 1; i = i + 1)
                                        begin
272                                        b[i+1] = gm_out[i] ^ b[i];
273                                 end
274                         end
275                 end
276         end
277
278 endmodule
```

../cc.v

```
1
2  `include "param.v"
3
4  /* convolution code - this one is based on the Figure 202 of
       802.16-2009,
5   * labeled "1/2 rate". Seems to indicate 2 bit output per 1 bit
       input.
6   */
7  module cc_base(
8          input reset, clk,
9          input valid_in,
10         input cur_in,
11         output reg [1:0] z,
12         output reg valid_out
13         );
14
15         reg [5:0] state;
16         reg in_progress;
```

```
17          wire x, y;
18
19          assign x = state[0] ^ state[1] ^ state[2] ^ state[5] ^
                cur_in;
20          assign y = state[1] ^ state[2] ^ state[4] ^ state[5] ^
                cur_in;
21
22          /**
23          * This is a 1 bit width design which has similar concerns
                as the 1 bit
24          * reed solomon
25          */
26
27          always @ (posedge clk or posedge reset) begin
28                  if (reset) begin
29                          state <= 0;
30                          in_progress <= 0;
31                  end else begin
32                          if (valid_in) begin
33                                  in_progress <= 1;
34
35                                  /* shift */
36                                  state[5:0] <= { state[4:0], cur_in
                                        };
37                          end else if (in_progress) begin
38                                  /* TODO: flush out remaining data,
                                        set
39                                  * in_progress = 1 when all needed
                                        data has
40                                  * escaped. Note that in_progress is
                                         used on
41                                  * the negedge, so it needs to be
                                        cleared after
42                                  * the data is shifted out, not just
                                         read in.
43                                  */
44                          end else begin
45                                  /* XXX: do we need anything here?
                                        */
46                          end
47                  end
48          end
49
50          always @ (negedge clk or posedge reset) begin
51                  if (reset) begin
52                          z <= 0;
53                          valid_out <= 0;
54                  end else begin
55                          if (in_progress) begin
56                                  z[0] <= x;
57                                  z[1] <= y;
58                                  valid_out <= 1;
59                          end
60                  end
61          end
62  endmodule
63
```

```
64  /* Buffers the output of cc_base into something the next stage in
        the pipeline
65   * wants. Also handles the needed discards for the requested rate.
66   */
67  module cc
68          #(
69          parameter w = 1,
70          parameter in_width  = w,
71          parameter out_width = w,
72          parameter ncbps     = 768,
73          parameter buf_sz    = ncbps / 2,
74          parameter baddr_sz  = $clog2(buf_sz),
75          parameter cc_base_o_width = 2
76          )(
77          input reset, clk,
78          input [in_width-1:0] cur_in,
79          input valid_in,
80          output reg [out_width-1:0] z,
81          output reg valid_out,
82
83          input [p_cc_rate.w-1:0] cc_rate
84          );
85
86          /* cc_base connections */
87          wire [cc_base_o_width-1:0] base_out;
88          wire base_valid_out;
89
90          /* buffering/fifo */
91          reg [buf_sz-1:0] dbuf;
92          reg [baddr_sz-1:0] i_loc;
93          reg [baddr_sz-1:0] o_loc;
94
95          /* "next", to be assigned to outputs on negedge. */
96          reg [out_width-1:0] nout;
97          reg nvalid;
98
99          cc_base cc(reset, clk, valid_in, cur_in, base_out,
                base_valid_out);
100
101         always @(posedge clk or posedge reset) begin
102                 if (reset) begin
103                         i_loc <= 0;
104                         o_loc <= 0;
105                 end else begin
106                         if (base_valid_out) begin
107                                 dbuf[i_loc +: cc_base_o_width] <=
                                        base_out;
108                                 i_loc <= i_loc + cc_base_o_width;
109                         end else begin
110
111                         end
112                 end
113         end
114
115         always @(negedge clk or posedge reset) begin
116                 if (reset) begin
117                         z         <= 0;
```

```
                        valid_out <= 0;
                end else begin
                        z         <= nout;
                        valid_out <= nvalid;
                end
        end

endmodule
```

../fec.v

```
/* NOTE: unfinished code */

/* FEC is composed of 2 portions, reed solomon and
* convolution code, applied in that order. On
* non-subchannelized data, RS is bypassed */

`include "rs.v"
`include "cc.v"

module fec
        #(
        parameter w = 1
        )(
        input reset, clk,
        input [w-1:0] in_bits,
        input in_valid,
        output [w-1:0] out_bits,
        output out_valid,

        /* config */
        input enable_rs,
        input [1:0] cc_rate
        );

        /** RS **/
        reg [w-1:0] rs_in_bits;
        reg rs_in_valid;

        wire [w-1:0] rs_out_bits;
        wire rs_out_valid;

        rs #(.w(w)) rs1 (reset, clk,
                rs_in_bits,
                rs_in_valid,
                rs_out_bits,
                rs_out_valid
        );
        /** **/

        /** CC **/
        reg [w-1:0] cc_in_bits;
        reg cc_in_valid;

        wire [w-1:0] cc_out_bits;
        wire cc_out_valid;

```

```
47          cc #(.w(w)) cc1 (reset, clk,
48                  cc_in_bits,
49                  cc_in_valid,
50                  cc_out_bits,
51                  cc_out_valid,
52
53                  cc_rate
54          );
55          /** **/
56
57          always @(posedge clk or posedge reset) begin
58                  if (reset) begin
59                  end else begin
60                  end
61          end
62
63          always @(negedge clk or posedge reset) begin
64                  if (reset) begin
65                  end else begin
66                  end
67          end
68
69  endmodule
```

A.1 Interleaving

The following is a partial implimentation of the 802.16-ODFM interleaver implimented in incomplete verilog. Due to inflexibility in the use of generate blocks, this does not fully support all possible interleaver setups, leading to the development of a tool to generate the interleaver verilog code.

../interleaver.v

```
1   `include "param.v"
2
3   /* Ncbps = blk_size
4    * Mod     16   8   4   2   1
5    * BPSK    192  96  48  24  12
6    * QPSK    384  192 96  48  24
7    * 16-QAM  768  384 192 96  48
8    * 64-QAM  1152 576 288 144 72
9    */
10
11  module blk_sz_decoder
12          #(
13          parameter rate_w  = p_rate_id.w,
14          parameter subct_w = p_subchan_ct.w
15          )(
16          input [rate_w-1:0]  rate,
17          input [subct_w-1:0] subchan,
18          output
19          );
20
21  endmodule
22
```

```
23 module interleaver
24         #(
25         parameter rate_w  = p_rate_id.w,
26         parameter subct_w = p_subchan_ct.w,
27         parameter ex_blk_size  = 1, /* the external block size */
28         )(
29         input reset, clk,
30         input [blk_size-1:0] in_blk,
31         input in_blk_valid,
32         input [rate_w-1:0] rate_id,
33         input [subct_w-1:0] subchan_ct,
34         output reg [blk_size-1:0] out_blk,
35         output reg out_blk_valid
36         );
37
38         localparam max_internal_buf_sz = 1152;
39
40         localparam mod_ct = 4;
41         localparam sub_ct = 5;
42         localparam blk_szs [mod_ct-1:0][sub_ct-1:0] = {
43                 {192,   96,     48,     24,     12},
44                 {384,   192,    96,     48,     24},
45                 {768,   384,    192,    96,     48},
46                 {1152,  576,    288,    144,    72}
47         };
48
49         /* @output_buffer - data which is being shifted out. */
50         /* @input_buffer  - data which is being shifted in. */
51         reg [max_internal_buf_sz-1:0] output_buffer, input_buffer;
52
53         /* @i_blk - input to the current ir_base module */
54         reg [max_internal_buf_sz-1:0] i_blk;
55
56         /* @in_ct - number of external blocks which have been read
               into internal
57          * buffers */
58         /* @out_ct - number of external blocks which need to be
               written out
59          */
60         reg [$clog2(blk_size)-1:0] in_ct, out_ct;
61
62         /* @set_output - when 1, read the output from the ir_base
               into the
63          * output_buffer */
64         reg set_output;
65
66         /* presently, only QPSK is supported */
67         wire o_valid [mod_ct-1:0][sub_ct-1:0];
68         wire [blk_size-1:0] o_blk [mod_ct-1:0][sub_ct-1:0];
69
70         generate
71         genvar mod_n, sub_n;
72         for (mod_n = 0; mod_n < mod_ct; mod_n = mod_n + 1) begin :
               ir_mod
73         for (sub_n = 0; sub_n < sub_ct; sub_n = sub_n + 1) begin :
               ir_sub
74                 ir_base r
```

```
75                      #(
76                      .blk_size(blk_szs[mod_n][sub_n]),
77                      .subchan_ct(sub_n)
78                      )(
79                      .in_blk(i_blk),
80                      .out_blk(o_blk[mod_n][sub_n])
81                      );
82          end
83          end
84          endgenerate
85
86          always @(posedge clk or posedge reset)
87          if (reset) begin
88                  in_ct  = 0;
89                  i_blk = 0;
90                  set_output = 0;
91                  input_buffer = 0;
92          end else begin
93                  if (valid_in) begin
94                          /* store the current input data into the
                               internal
95                           * buffer */
96                          /* XXX: this multiplication always has an
                               equivalent
97                           * shift, is the synthesizer smart enough?
                                */
98                          input_buffer[ex_blk_sz * in_ct :+ ex_blk_sz
                              ] = in_blk;
99                          in_ct = in_ct + 1;
100
101                         if (in_ct >= cur_internal_blk_sz) begin
102                                 /* setup the current ir_base so
                                        that on the
103                                  * negedge we will have the
                                        interleaved results */
104                                 /* XXX: as in_blk_sz is non-
                                        constant, this is
105                                  * not valid verilog. */
106                                 i_blk = input_buffer[ex_blk_sz *
                                        in_ct :- in_blk_sz];
107                                 set_output = 1;
108                         end else begin
109                                 set_output = 0;
110                         end
111                 end
112
113         end
114
115         always @(negedge clk or posedge reset)
116         if (reset) begin
117                 out_blk = 0;
118                 out_blk_valid = 0;
119                 out_ct = 0;
120         end else begin
121                 if (out_ct) begin
122                         /* shift out data */
123                         out_ct = out_ct - 1;
```

```
124                         out_blk = outpuf_buffer[ex_blk_sz * out_ct
                                :- ex_blk_sz];
125                 end
126
127                 if (set_output) begin
128                         out_ct = in_blk_sz;
129                         outpuf_buffer[0 :+ in_blk_sz] = o_blk[
                                rate_id][subchan_ct];
130                 end
131         end
132
133
134 endmodule
135
136
137 /* Takes a particular block of data, and outputs the interleaved
        version of
138  * that block */
139 module ir_base
140         #(
141         parameter blk_size = 384,
142         parameter subch_ct = 16
143         )(
144         input [blk_size-1:0] in_blk,
145         output reg [blk_size-1:0] out_blk,
146         );
147
148         /*
149          * Let Ncpc be the number of coded bits per subcarrier,
150          * i.e., 1, 2, 4 or 6 for BPSK, QPSK, 16-QAM, or 64-
151          * QAM, respectively. Let s = ceil(Ncpc/2).
152          *
153          * The first permutation is defined by:
154          * m_k = ( N_cbps  12 )  k mod12 + floor ( k  12 ).
155          *       k = 0, 1, ..., N_cbps  1
156          *
157          * The second permutation is defined by Equation (26).
158          * j_k = s  floor ( m_k  s ) + ( m_k + N_cbps
159          *           floor ( 12  m_k  N_cbps ) ) mod ( s )
160          *       k = 0, 1, ..., N_cbps  1
161          */
162
163         /* FIXME: Only works for QPSK and BPSK, need to use Ncpc?
                */
164         generate
165                 genvar k;
166                 for (k = 0; k < blk_size; k = k + 1) begin: g1
167                         assign out_blk[k] <= in_blk[(k % 12) *
                                blk_size / 12 + floor(k / 12)];
168                 end
169         endgenerate
170 endmodule
```

The following generates verilog code for implimenting a particular (adjustable) interleaver. Changing the items in the 'param' dictionary allow generation of all possible interleavers.

../scripts/ir_gen.py

```
#! /usr/bin/env python
# vim: set fileencoding=utf8 :

#  Let Ncpc be the number of coded bits per subcarrier,
#  i.e., 1, 2, 4 or 6 for BPSK, QPSK, 16-QAM, or 64-
#  QAM, respectively. Let s = ceil(Ncpc/2).
#
#  The first permutation is defined by:
#  m_k = ( N_cbps  12 )  k mod12 + floor ( k  12 ).
#        k = 0, 1, ..., N_cbps  1
#
#  The second permutation is defined by Equation (26).
#  j_k = s  floor ( m_k  s ) + ( m_k + N_cbps
#             floor ( 12  m_k  N_cbps ) ) mod ( s )
#        k = 0, 1, ..., N_cbps  1


from math import floor, ceil
import sys

param = {
    'out_var' : 'out_blk',
    'in_var'  : 'inr',
    'mod'     : 'QPSK',
    'subchan' : 8
}


param['subchan'] = int(sys.argv[1])
param['mod']     = sys.argv[2]
which            = sys.argv[3]
warn             = sys.argv[4] == "warn"

if which == "both":
    p_std = p_quick = True
elif which == "std":
    p_std = True
    p_quick = False
elif which == "quick":
    p_std = False
    p_quick = True
else :
    raise Exception()

ncpc_ = {
    'BPSK': 1.0,
    'QPSK': 2.0,
    '16-QAM': 4.0,
    '64-QAM': 6.0
}

ncbps_ = {
        'BPSK':   [192,  96,  48,  24,  12],
        'QPSK':   [384,  192, 96,  48,  24],
        '16-QAM': [768,  384, 192, 96,  48],
        '64-QAM': [1152, 576, 288, 144, 72]
```

```
}

subchan_ct = {
        16: 0,
        8:  1,
        4:  2,
        2:  3,
        1:  4
}

ncpc = ncpc_[param['mod']]
ncbps = ncbps_[param['mod']][subchan_ct[param['subchan']]]
s = ceil(ncpc / 2)

m = []

for k in range(0, ncbps):
    m.append((ncbps / 12) * (k % 12) + floor(k / 12));

j = []

for k in range(0, ncbps):
    j.append(s * floor(m[k] / s) + ((m[k] + ncbps - \
        floor(12 * m[k] / ncbps) ) % s))

of = "{out_var}[{k}] <= {in_var}[{in_index}];"
bad = False
for k in range(0, ncbps):
    i1 = int(j[int(k)])
    i2 = ((k % 12) * ncbps / 12) + (int(k / 12))

    if (i1 != i2):
        if (warn):
            print "omg: k = {0}, {1} != {2}".format(k, i1, i2)
        bad = True

    if (p_std):
        print of.format(k = k, in_index = i1, **param)

    if (p_quick):
        print of.format(k = k, in_index = i2, **param)

if bad:
    print "omg"

#for x in range(0, ncbps, 12):
#    for y in range(0, 12):
#        k = x + y
#        ii = (y * ncbps / 12) + (x / 12)
#        print of.format(k = k, in_index = ii, **param)
```

This is the sample output of that generator run un-eddited.

../scripts/ir_gen_output.v

```
out_blk[0] <= inr[0];
out_blk[1] <= inr[16];
out_blk[2] <= inr[32];
```

```
4  out_blk[3] <= inr[48];
5  out_blk[4] <= inr[64];
6  out_blk[5] <= inr[80];
7  out_blk[6] <= inr[96];
8  out_blk[7] <= inr[112];
9  out_blk[8] <= inr[128];
10 out_blk[9] <= inr[144];
11 out_blk[10] <= inr[160];
12 out_blk[11] <= inr[176];
13 out_blk[12] <= inr[1];
14 out_blk[13] <= inr[17];
15 out_blk[14] <= inr[33];
16 out_blk[15] <= inr[49];
17 out_blk[16] <= inr[65];
18 out_blk[17] <= inr[81];
19 out_blk[18] <= inr[97];
20 out_blk[19] <= inr[113];
21 out_blk[20] <= inr[129];
22 out_blk[21] <= inr[145];
23 out_blk[22] <= inr[161];
24 out_blk[23] <= inr[177];
25 out_blk[24] <= inr[2];
26 out_blk[25] <= inr[18];
27 out_blk[26] <= inr[34];
28 out_blk[27] <= inr[50];
29 out_blk[28] <= inr[66];
30 out_blk[29] <= inr[82];
31 out_blk[30] <= inr[98];
32 out_blk[31] <= inr[114];
33 out_blk[32] <= inr[130];
34 out_blk[33] <= inr[146];
35 out_blk[34] <= inr[162];
36 out_blk[35] <= inr[178];
37 out_blk[36] <= inr[3];
38 out_blk[37] <= inr[19];
39 out_blk[38] <= inr[35];
40 out_blk[39] <= inr[51];
41 out_blk[40] <= inr[67];
42 out_blk[41] <= inr[83];
43 out_blk[42] <= inr[99];
44 out_blk[43] <= inr[115];
45 out_blk[44] <= inr[131];
46 out_blk[45] <= inr[147];
47 out_blk[46] <= inr[163];
48 out_blk[47] <= inr[179];
49 out_blk[48] <= inr[4];
50 out_blk[49] <= inr[20];
51 out_blk[50] <= inr[36];
52 out_blk[51] <= inr[52];
53 out_blk[52] <= inr[68];
54 out_blk[53] <= inr[84];
55 out_blk[54] <= inr[100];
56 out_blk[55] <= inr[116];
57 out_blk[56] <= inr[132];
58 out_blk[57] <= inr[148];
59 out_blk[58] <= inr[164];
60 out_blk[59] <= inr[180];
```

```
61  out_blk[60] <= inr[5];
62  out_blk[61] <= inr[21];
63  out_blk[62] <= inr[37];
64  out_blk[63] <= inr[53];
65  out_blk[64] <= inr[69];
66  out_blk[65] <= inr[85];
67  out_blk[66] <= inr[101];
68  out_blk[67] <= inr[117];
69  out_blk[68] <= inr[133];
70  out_blk[69] <= inr[149];
71  out_blk[70] <= inr[165];
72  out_blk[71] <= inr[181];
73  out_blk[72] <= inr[6];
74  out_blk[73] <= inr[22];
75  out_blk[74] <= inr[38];
76  out_blk[75] <= inr[54];
77  out_blk[76] <= inr[70];
78  out_blk[77] <= inr[86];
79  out_blk[78] <= inr[102];
80  out_blk[79] <= inr[118];
81  out_blk[80] <= inr[134];
82  out_blk[81] <= inr[150];
83  out_blk[82] <= inr[166];
84  out_blk[83] <= inr[182];
85  out_blk[84] <= inr[7];
86  out_blk[85] <= inr[23];
87  out_blk[86] <= inr[39];
88  out_blk[87] <= inr[55];
89  out_blk[88] <= inr[71];
90  out_blk[89] <= inr[87];
91  out_blk[90] <= inr[103];
92  out_blk[91] <= inr[119];
93  out_blk[92] <= inr[135];
94  out_blk[93] <= inr[151];
95  out_blk[94] <= inr[167];
96  out_blk[95] <= inr[183];
97  out_blk[96] <= inr[8];
98  out_blk[97] <= inr[24];
99  out_blk[98] <= inr[40];
100 out_blk[99] <= inr[56];
101 out_blk[100] <= inr[72];
102 out_blk[101] <= inr[88];
103 out_blk[102] <= inr[104];
104 out_blk[103] <= inr[120];
105 out_blk[104] <= inr[136];
106 out_blk[105] <= inr[152];
107 out_blk[106] <= inr[168];
108 out_blk[107] <= inr[184];
109 out_blk[108] <= inr[9];
110 out_blk[109] <= inr[25];
111 out_blk[110] <= inr[41];
112 out_blk[111] <= inr[57];
113 out_blk[112] <= inr[73];
114 out_blk[113] <= inr[89];
115 out_blk[114] <= inr[105];
116 out_blk[115] <= inr[121];
117 out_blk[116] <= inr[137];
```

```
118 out_blk[117] <= inr[153];
119 out_blk[118] <= inr[169];
120 out_blk[119] <= inr[185];
121 out_blk[120] <= inr[10];
122 out_blk[121] <= inr[26];
123 out_blk[122] <= inr[42];
124 out_blk[123] <= inr[58];
125 out_blk[124] <= inr[74];
126 out_blk[125] <= inr[90];
127 out_blk[126] <= inr[106];
128 out_blk[127] <= inr[122];
129 out_blk[128] <= inr[138];
130 out_blk[129] <= inr[154];
131 out_blk[130] <= inr[170];
132 out_blk[131] <= inr[186];
133 out_blk[132] <= inr[11];
134 out_blk[133] <= inr[27];
135 out_blk[134] <= inr[43];
136 out_blk[135] <= inr[59];
137 out_blk[136] <= inr[75];
138 out_blk[137] <= inr[91];
139 out_blk[138] <= inr[107];
140 out_blk[139] <= inr[123];
141 out_blk[140] <= inr[139];
142 out_blk[141] <= inr[155];
143 out_blk[142] <= inr[171];
144 out_blk[143] <= inr[187];
145 out_blk[144] <= inr[12];
146 out_blk[145] <= inr[28];
147 out_blk[146] <= inr[44];
148 out_blk[147] <= inr[60];
149 out_blk[148] <= inr[76];
150 out_blk[149] <= inr[92];
151 out_blk[150] <= inr[108];
152 out_blk[151] <= inr[124];
153 out_blk[152] <= inr[140];
154 out_blk[153] <= inr[156];
155 out_blk[154] <= inr[172];
156 out_blk[155] <= inr[188];
157 out_blk[156] <= inr[13];
158 out_blk[157] <= inr[29];
159 out_blk[158] <= inr[45];
160 out_blk[159] <= inr[61];
161 out_blk[160] <= inr[77];
162 out_blk[161] <= inr[93];
163 out_blk[162] <= inr[109];
164 out_blk[163] <= inr[125];
165 out_blk[164] <= inr[141];
166 out_blk[165] <= inr[157];
167 out_blk[166] <= inr[173];
168 out_blk[167] <= inr[189];
169 out_blk[168] <= inr[14];
170 out_blk[169] <= inr[30];
171 out_blk[170] <= inr[46];
172 out_blk[171] <= inr[62];
173 out_blk[172] <= inr[78];
174 out_blk[173] <= inr[94];
```

```
175 out_blk[174] <= inr[110];
176 out_blk[175] <= inr[126];
177 out_blk[176] <= inr[142];
178 out_blk[177] <= inr[158];
179 out_blk[178] <= inr[174];
180 out_blk[179] <= inr[190];
181 out_blk[180] <= inr[15];
182 out_blk[181] <= inr[31];
183 out_blk[182] <= inr[47];
184 out_blk[183] <= inr[63];
185 out_blk[184] <= inr[79];
186 out_blk[185] <= inr[95];
187 out_blk[186] <= inr[111];
188 out_blk[187] <= inr[127];
189 out_blk[188] <= inr[143];
190 out_blk[189] <= inr[159];
191 out_blk[190] <= inr[175];
192 out_blk[191] <= inr[191];
```

A.2 Testbenches

../t/vect/1.v

```
1  `include "common_widths.v"
2
3  module vect();
4  /* Page 633
5   * Modulation mode: QPSK, rate 3/4, Symbol Number within burst: 1,
        UIUC: 7,
6   * BSID: 1, Frame Number 1 (decimal values)
7   */
8
9  parameter uiuc = 7;
10 parameter bsid = 1;
11 parameter frame_num = 1;
12
13 parameter input_data_sz = 560;
14 reg [0:input_data_sz-1] input_data =   { '
       h45_29_C4_79_AD_0F_55_28_AD_87_B5_76_1A_9C_80,
15         'h50_45_1B_9F_D9_2A_88_95_EB_AE_B5, '
               h2E_03_4F_09_14_69_58_0A_5D };
16
17 reg [0:input_data_sz-1] randomized_data = { '
       hD4_BA_A1_12_F2_74_96_30_27_D4_88_9C_96_E3_A9,
18         'h52_B3_15_AB_FD_92_53_07_32_C0_62, '
               h48_F0_19_22_E0_91_62_1A_C1 };
19
20
21 parameter rs_data_sz = input_data_sz + 8 * 5;
22 reg [0:rs_data_sz-1] rs_encoded_data =  {'
       h49_31_40_BF_D4_BA_A1_12_F2_74_96_30_27_D4_88,
23         'h9C_96_E3_A9_52_B3_15_AB_FD_92_53, '
               h07_32_C0_62_48_F0_19_22_E0_91_62_1A_C1_00 };
24
25 parameter cc_data_sz = 3 * rs_data_sz / 4; // rs_data_sz + 8*8;
```

```
26 reg [0:cc_data_sz-1] cc_encoded_data = { '
       h3A_5E_E7_AE_49_9E_6F_1C_6F_C1_28_BC_BD_AB_57,
27         'hCD_BC_CD_E3_A7_92_CA_92_C2_4D_BC, '
               h8D_78_32_FB_BF_DF_23_ED_8A_94_16_27_A5_65,
28         'hCF_7D_16_7A_45_B8_09_CC };
29
30 reg [0:cc_data_sz-1] inteinterleaved_data = { '
       h77_FA_4F_17_4E_3E_E6_70_E8_CD_3F_76_90_C4_2C,
31         'hDB_F9_B7_FB_43_6C_F1_9A_BD_ED_0A, '
               h1C_D8_1B_EC_9B_30_15_BA_DA_31_F5_50_49_7D,
32         'h56_ED_B4_88_CC_72_FC_5C };
33
34
35 /*
36 Subcarrier mapping (frequency offset index: I value Q value)
37 -100: 1 -1, -99: -1 -1, -98: 1 -1, -97: -1 -1, -96: -1 -1, -95: -1
       -1, -94: -1 1, -93: -1 1, -92: 1 -1, -91: 1
38 1, -90: -1 -1, -89: -1 -1, -88:pilot= 1 0, -87: 1 1, -86: 1 -1,
       -85: 1 -1, -84: -1 -1, -83: 1 -1, -82: 1 1, -81:
39 -1 -1, -80: -1 1, -79: 1 1, -78: -1 -1, -77: -1 -1, -76: -1 1, -75:
        -1 -1, -74: -1 1, -73: 1 -1, -72: -1 1, -71:
40 1 -1, -70: -1 -1, -69: 1 1, -68: 1 1, -67: -1 -1, -66: -1 1, -65:
       -1 1, -64: 1 1, -63:pilot= -1 0, -62: -1 -1,
41 -61: 1 1, -60: -1 -1, -59: 1 -1, -58: 1 1, -57: -1 -1, -56: -1 -1,
       -55: -1 -1, -54: 1 -1, -53: -1 -1, -52: 1 -1,
42 -51: -1 1, -50: -1 1, -49: 1 -1, -48: 1 1, -47: 1 1, -46: -1 -1,
       -45: 1 1, -44: 1 -1, -43: 1 1, -42: 1 1,
43 -41: -1 1, -40: -1 -1, -39: 1 1, -38:pilot= 1 0, -37: -1 -1, -36: 1
        -1, -35: -1 1, -34: -1 -1, -33: -1 -1,
44 -32: -1 -1, -31: -1 1, -30: 1 -1, -29: -1 1, -28: -1 -1, -27: 1 -1,
        -26: -1 -1, -25: -1 -1, -24: -1 -1,
45 -23: -1 1, -22: -1 -1, -21: 1 -1, -20: 1 1, -19: 1 1, -18: -1 -1,
       -17: 1 -1, -16: -1 1, -15: -1 -1, -14: 1 1,
46 -13:pilot= -1 0, -12: -1 -1, -11: -1 -1, -10: 1 1, -9: 1 -1, -8: -1
        1, -7: 1 -1, -6: -1 1, -5: -1 1, -4: -1 1,
47 -3: -1 -1, -2: -1 -1, -1: 1 -1, 0: 0 0, 1: -1 -1, 2: -1 1, 3: -1
       -1, 4: 1 -1, 5: 1 1, 6: 1 1, 7: -1 1, 8: -1 1, 9:
48 1 1, 10: 1 -1, 11: -1 -1, 12: 1 1, 13:pilot= 1 0, 14: -1 -1, 15: 1
       -1, 16: -1 1, 17: 1 1, 18: 1 1, 19: 1 -1,
49 20: -1 1, 21: -1 -1, 22: -1 -1, 23: -1 1, 24: -1 -1, 25: 1 1, 26:
       -1 1, 27: 1 -1, 28: -1 1, 29: -1 -1, 30: 1 1,
50 31: -1 -1, 32: 1 1, 33: 1 1, 34: 1 1, 35: 1 -1, 36: 1 -1, 37: 1 -1,
        38:pilot= 1 0, 39: -1 1, 40: -1 -1, 41: -1
51 1, 42: -1 1, 43: -1 -1, 44: 1 -1, 45: -1 1, 46: -1 1, 47: 1 1, 48:
       -1 -1, 49: 1 1, 50: 1 -1, 51: -1 -1, 52: -1
52 -1, 53: 1 -1, 54: 1 -1, 55: 1 -1, 56: 1 -1, 57: 1 1, 58: 1 1, 59: 1
        -1, 60: 1 1, 61: -1 1, 62: 1 -1, 63:pilot=
53 1 0, 64: 1 -1, 65: -1 -1, 66: -1 -1, 67: 1 -1, 68: 1 -1, 69: 1 -1,
       70: 1 -1, 71: -1 1, 72: -1 -1, 73: -1 1, 74:
54 -1 -1, 75: 1 -1, 76: -1 1, 77: -1 -1, 78: 1 -1, 79: 1 1, 80: -1 1,
       81: 1 1, 82: -1 1, 83: 1 1, 84: -1 -1, 85: 1
55 1, 86: -1 -1, 87: 1 1, 88:pilot= 1 0, 89: 1 -1, 90: -1 -1, 91: 1 1,
        92: -1 1, 93: -1 -1, 94: -1 -1, 95: -1 -1,
56 96: 1 1, 97: 1 -1, 98: 1 -1, 99: -1 -1, 100: 1 1
57 */
58
59 endmodule
```

../t/vect/2.v

```
1  `include "common_widths.v"
2
3  module vect();
4
5  /*
6   * Modulation mode: QPSK, rate 3/4, Symbol Numbers within burst:
        1-5,
7   * UIUC: 7, BSID: 1, Frame Number: 1, subchannel index: 0b00001 (
        decimal values)
8   */
9
10 parameter uiuc = 7;
11 parameter bsid = 1;
12 parameter frame_num = 1;
13 parameter subchannel_index = 'b00001;
14 parameter input_data_sz = 80;
15
16 reg [0:input_data_sz-1] input_data =       '
       h45_29_C4_79_AD_0F_55_28_AD_87;
17
18 /* NOTEThe last hex value represents 2 bits only. */
19 reg [0:input_data_sz-1] randomized_data = '
       hD4_BA_A1_12_F2_74_96_30_27_D4;
20 //reg [159:0] randomized_data = '
       hD4_BA_A1_12_F2_74_96_30_27_D4_00_00;
21
22 /* Convolutionally encoded data (Hex) */
23 parameter cc_data_sz = input_data_sz + 5 * 8;
24 reg [0:cc_data_sz-1] convolution_encoded_data = '
       hEE_C6_A1_CB_7E_04_73_6C_BC_61_95_D3_B7_DF_00;
25
26 /* Interleaved data (Hex) */
27 reg [0:cc_data_sz-1] interleaved_data =          '
       hBC_EC_A1_F4_8A_3A_7A_4F_78_39_53_87_DF_2A_A2;
28
29 /*
30 Subcarrier mapping (frequency offset index: I value Q value)
31 1st data symbol:
32 -100: -1 1, -99: -1 -1, -98: -1 -1, -37: 1 1, -36: -1 -1, -35: -1
       1, 1: -1 -1, 2: 1 1, 3: -1 1, 64: -1 1,
33 65: 1 1, 66: 1 -1
34 2nd data symbol:
35 -100: -1 -1, -99: -1 -1, -98: 1 -1, -37: 1 1, -36: -1 1, -35: 1 1,
       1: -1 1, 2: -1 1, 3: 1 1, 64: -1 -1, 65:
36 -1 1, 66: -1 1
37 3rd data symbol:
38 -100: 1 -1, -99: -1 -1, -98: -1 1, -37: -1 1, -36: 1 -1, -35: 1 1,
       1: -1 -1, 2: -1 -1, 3: 1 -1, 64: -1 -1,
39 65: -1 1, 66: 1 1
40 4th data symbol:
41 -100: 1 1, -99: -1 -1, -98: -1 1, -37: 1 -1, -36: 1 -1, -35: 1 -1,
       1: 1 1, 2: -1 -1, 3: -1 1, 64: 1 1, 65:
42 1 -1, 66: -1 -1
43 5th data symbol:
44 -100: -1 -1, -99: 1 -1, -98: -1 -1, -37: -1 -1, -36: 1 1, -35: -1
       1, 1: -1 1, 2: -1 1, 3: -1 1, 64: -1 1,
```

```
65: 1 1, 66: -1 1
Note that the above QPSK values are to be normalized with a factor
    1  2 as indicated in Figure 205.
*/

endmodule
```

../t/t_rand.v

```
`include "rand.v"
`include "t/vect/2.v"

module rand_test();

        localparam w = 8;

        reg reset, clk, in_valid, reload;
        reg  [w-1:0] in_bits;
        wire [w-1:0] out_bits;
        wire out_valid;

        reg [14:0] rand_iv;

        reg odata[0:vect.input_data_sz-1];

        rand_parm #(w) x1(
                reset, clk,
                in_bits,
                in_valid,
                out_bits,
                out_valid,
                rand_iv,
                reload);

        task set_vect;
                input [14:0] new_iv;

                /* insert new_iv into randomizer */
                begin
                        clk = 1;
                        #1
                        reload = 1;
                        rand_iv = new_iv;
                        clk = 0;
                        #1
                        clk = 1;
                        #1
                        rand_iv = 0;
                        reload = 0;
                end
        endtask

        `include "func/gen_rand_iv.v"

        integer i, o;
        initial begin
                $dumpfile("randomizer.lxt");
```

```
49                  $dumpvars();
50
51                  clk = 0;
52                  reset = 0;
53                  in_bits = 0;
54                  in_valid = 0;
55                  reload = 0;
56                  rand_iv = 0;
57
58                  #1
59                  reset = 1;
60                  #1
61
62                  reset = 0;
63
64                  /* device is now reset, iv = 0 */
65                  set_vect(gen_rand_iv(vect.bsid, vect.uiuc,
66                          vect.frame_num));
67
68                  #1
69                  clk = 1;
70                  #1
71
72                  o = 0;
73
74                  $display("iv: %b", x1.vect);
75                  for (i = 0; i < vect.input_data_sz; i = i + w)
                        begin
76                      in_valid = 1;
77                      in_bits = vect.input_data[i +: w];
78                      clk = 0;
79                      #1;
80                      clk = 1;
81
82                      /* read on the rising edge */
83
84                      if (out_valid) begin
85                          //odata[o +: w] = out_bits;
86                          $display("r: %b => %b %b %b", vect.
                                input_data[o +: w], out_bits,
                                vect.randomized_data[o +: w],
                                x1.vect);
87                          o = o + w;
88                      end else begin
89                          $display("skip");
90                      end
91                      #1;
92
93                  end
94
95                  while (o < vect.input_data_sz) begin
96                      in_valid = 0;
97                      clk = 0;
98                      #1;
99                      if (out_valid) begin
100                         //odata[o +: w] = out_bits;
```

```
101                                     $display("r: %b => %b %b %b", vect.
                                            input_data[o +: w], out_bits,
                                            vect.randomized_data[o +: w],
                                            x1.vect);
102                                     o = o + w;
103                             end
104                             clk = 1;
105                             #1;
106                     end
107             end
108
109     endmodule
```

../t/t_cc.v

```
1   `include "t/vect/1.v"
2   `include "fec.v"
3
4   module test();
5
6           integer w = 1;
7           integer o_w = 2;
8           reg reset, clk, valid_in, cur_in;
9           wire [o_w-1:0] z;
10          wire valid_out;
11
12
13
14          cc_base x1(reset, clk, valid_in, cur_in, z, valid_out);
15
16          integer i, o;
17          initial begin
18                  $dumpfile("cc.txt");
19                  $dumpvars();
20
21
22                  clk = 0;
23                  reset = 0;
24                  cur_in = 0;
25                  valid_in = 0;
26
27                  #1
28                  reset = 1;
29                  #1
30
31                  reset = 0;
32
33                  #1
34                  clk = 1;
35                  #1
36
37                  //$display("iv: %b", x1.vect);
38                  for (i = 0; i < vect.rs_data_size; i = i + w) begin
39                          in_valid = 1;
40                          in_bits = vect.rs_encoded_data[i +: w];
41                          clk = 0;
42                          #1;
```

```
43                          clk = 1;
44
45                          /* read on the rising edge */
46
47                          if (out_valid) begin
48                                  $display("r: %b => %b : %b", vect.
                                        rs_encoded_data[o +: o_w], z,
                                        vect.cc_encoded_data[o +: o_w])
                                        ;
49                                  o = o + o_w;
50                          end else begin
51                                  $display("skip");
52                          end
53                          #1;
54
55                  end
56
57                  while (o < vect.rs_data_sz) begin
58                          in_valid = 0;
59                          clk = 0;
60                          #1;
61                          if (out_valid) begin
62                                  $display("r: %b => %b : %b", vect.
                                        rs_encoded_data[o +: o_w], z,
                                        vect.cc_encoded_data[o +: o_w])
                                        ;
63                                  o = o + o_w;
64                          end
65                          clk = 1;
66                          #1;
67                  end
68          end
69
70
71          end
72
73  endmodule
```

References

[1] IEEE Std 802.16-2009, *IEEE Standard for Local and Metropolitan Area Networks — Part16: Air Interface for Broadband Wireless Access Systems*. New York, NY, USA, May 2009.