A Division of Labor

This report was written as a combined effort by Zack Anderson and Russell Ryan. Anderson wrote the following sections:

Abstract, Demodulation (minus the Short PN Sequence Generator), Serial Interface, Deinterleaving, Data Post-processing and Display, General (testing) Methodology, Demodulation Uncertainty, Deinterleaver Timing, Conclusion

and Ryan wrote: Introduction, Design Overview, Short PN Sequence Generator, Viterbi Decoder Module, Radio Data Acquisition, Lack of Information, Short PN Code Generator (testing), Viterbi Decoder Synthesis.

B Code Listing

- test_labkit.v
- demod.v
- correlator.v
- short_pn.v
- walsh_gen.v
- viterbi.v
- deinterleaver_derepeater.v
- pm_compare.v
- acs_node.v
- uart.v
- uart_rec.v
- cdma_rec.py
- acs_assign.v
- acs_declare.v
- pm_max.v

B.1 test_labkit.v

```
//labkit_test.v (excerpt)
//  This labkit excerpt demonstrates the test code used for verifying the hardware op
//  synthesized modules.
...
   // RS-232 Interface
```

```
  //assign rs232_txd = 1'b1;
  assign rs232_rts = 1'b1;
...


  //generate the serial clock using a DCM
  wire sclk, serial_clk;
  DCM serial_clk_dcm (.CLKIN(clock_27mhz), .CLKFX(sclk));
  // synthesis attribute CLKFX_DIVIDE of serial_clk_dcm is 1
  // synthesis attribute CLKFX_MULTIPLY of serial_clk_dcm is 4
  // synthesis attribute CLK_FEEDBACK of serial_clk_dcm  is "NONE"
  BUFG serial_clk_buf (.I(sclk), .O(serial_clk));


  //test registers
  reg halfrate;
//reg transmit;

//SERIAL CLOCK
wire serial_clky;
assign system_clk = clock_27mhz;
wire oversampled_serial_clk;
serial_clock clocker(serial_clk, serial_clky, oversampled_serial_clk);

//RESET
wire reset;
debounce bouncy(0, system_clk, !button0, reset);

//UART TX
wire [7:0] data_in;
wire transmit;
uart uart_send(serial_clky, reset, data_in, transmit, rs232_txd, done);

//UART RX
wire byte_ready;
wire [7:0] byte_out;
uart_rec receiver(system_clk, reset, user4[0], byte_ready, byte_out);

//WALSH
wire walsh_output_bit;
walsh_gen walsh(halfrate, reset, switch[5:0], walsh_output_bit);

//DEINTERLEAVER
wire load;
wire [127:0] buffered_interleaved_frame;
wire di_done;
```

```
 wire fifo_clk;
 wire [63:0] deinterleaved_frame_out;
 deinterleaver_derepeater deinterlvr(system_clk, reset, load, buffered_interleaved_fr
 fifo_buffer framebuffer(fifo_clk, reset, fifo_in, buffered_interleaved_frame, serial

 //DEMODULATOR
 wire synchronized;
 demod demodulator(symbol_clk, system_clk, reset, user4[0], synchronized, demod_data_

 //TESTCODE SECTION
 assign led[6:0] = 127;
 //DEINTERLEAVER TEST CODE
 /*
   reg [7:0] curr_bity;
 debounce loady(0, system_clk, !button2, load);
 debounce tryy(0, system_clk, !button3, tryagain);
 assign fifo_in = (byte_out == 49) ? 1 : 0;

 assign led[7] = 1;
 assign led[0] = 1;
 assign led[1] = 1;
 assign led[2] = 1;
 assign led[3] = 1;
 assign led[4] = 1;
 assign led[5] = 1;
 assign led[6] = 1;

 assign data_in = switch[0] ? (deinterleaved_frame_out[curr_bity-1] ? 49 : 48) : (buf
 reg [3:0] anothercounter;
 always @ (posedge serial_clky)
 begin
  if (tryagain && ( (!switch[0] && (curr_bity < 128)) || (switch[0] && (curr_bity < 6
begin
anothercounter <= anothercounter + 1;
if (anothercounter == 0)
begin
halfrate <= ~halfrate;
if(halfrate)
begin
transmit <= 1;
curr_bity <= curr_bity + 1;
end
end
else
transmit <= 0;
end
```

```
else if (reset)
begin
curr_bity <= 0;
transmit <=0;
end
else if (load)
curr_bity <= 0;
else
transmit <=0;
end

 assign fifo_clk = (curr_bity < 126) ? byte_ready : 0;
 */
 /*

 reg interteststate;
 parameter waiting = 0;
 parameter donee = 1;
 reg send;
 reg [63:0] tempo;
 always @ (posedge system_clk)
 case (interteststate)
waiting:
if (di_done) //start shifting out data
begin
send <= 1;
interteststate <= donee;
tempo <= deinterleaved_frame_out;
end
donee:
if (tryagain)
interteststate <= waiting;
default: interteststate <= waiting;
endcase

 reg [3:0] anothercounter;
 always @ (posedge serial_clky)
 if (send)
begin
anothercounter <= anothercounter + 1;
if (anothercounter == 0)
begin
halfrate <= ~halfrate;
if(halfrate)
transmit <= 1;
end
```

```
else
transmit <= 0;
end

//shift out at halfrate speed 64 bits
assign data_in = tempo[7:0];
always @ (posedge halfrate)
if (send)
begin
tempo <= (tempo >> 8);
end
*/

 //UART LOOPBACK TEST CODE
   reg bitybity;
 reg trany;
debounce loady(0, system_clk, !button2, load);
parameter PILOT_AQUIRED = 90;
 assign  data_in = bitybity ? PILOT_AQUIRED : byte_out;
 assign transmit = bitybity ? trany : byte_ready;
 assign led[7] = synchronized;
 always @ (serial_clky)
begin
if ((synchronized || load)  && (!byte_ready))
begin
bitybity <= 1;
trany <= 1;
end
else
begin
bitybity <= 0;
trany <= 0;
end
end


 // WALSH GENERATOR TEST CODE
 /*
 reg halfrate;
 reg transmit;
 assign data_in = walsh_output_bit ? 49 : 48;
 reg [3:0] anothercounter;
 always @ (posedge serial_clky)
begin
anothercounter <= anothercounter + 1;
if (anothercounter == 0)
```

```
begin
halfrate <= ~halfrate;
if(halfrate)
transmit <= 1;
end
else
transmit <= 0;
end
*/
// END WALSH GENERATOR TEST CODE

endmodule
```

B.2 demod.v

```
//////////////////////////////////////////////////////////////////////////////////
//
// IS-95 DEMODULATOR
// Performs maximal ratio combining
// Chooses appropriate correlator
// Loads and splits up USRP data
// Instantiates correlators, walsh generator, and PN generators
// Converts MRC output to baseband data
//
//////////////////////////////////////////////////////////////////////////////////
module demod(symbol_clk, system_clk, reset, uart_line, synchronized, demod_data_out);

input symbol_clk, system_clk, reset, uart_line; //new usrp data means it has all been
output demod_data_out; // that's right: over two-thousand bits go in, and only one co
output synchronized;  // other modules can discover frame boundary by ANDing this val

// start_of_PN from the PN generator
reg synchronized;
reg demod_data_out;
parameter THRESHOLD = 5;  // to trigger a positive on the correlator

// due to pseudonoise having avg of 0, the accumulator

// should only reach this when it is correctly synchronized

// INSTANTIATE UART RX
wire byte_ready;
wire [7:0] byte_out;
uart_rec rx(system_clk, reset, uart_line, byte_ready, byte_out);
```

```
////////////////////////////////////////////////////////////////////////////////
// RECEIVER & SPLIT UP USRP DATA BUS
// note that there are two data samples for each chip to meet Nyquist constraints
// data format: header | usrp_a_sample_I | usrp_a_sample_Q | usrp_b_sample_I | usrp_b

reg [511:0] Ia;
reg [511:0] Ib;
reg [511:0] Qa;
reg [511:0] Qb;
reg state;
parameter WAIT = 0;
parameter LOAD_DATA = 1;
reg [8:0] counter; // to obtain all 256 bytes

always @ (posedge byte_ready) // clocked on uart rx'ed byte
begin
case(state)
WAIT:
if (byte_out == 8'hAA) // check for header
begin
counter <= 0;
state <= LOAD_DATA;
end

LOAD_DATA:
begin
if (counter < 64) begin
Ia[503:0] <= Ia[511:8];
Ia[511:504] <= byte_out;
counter <= counter + 1;
end
else if (counter < 128) begin
Qa[503:0] <= Qa[511:8];
Qa[511:504] <= byte_out;
counter <= counter + 1;
end
else if (counter < 192) begin
Ib[503:0] <= Ib[511:8];
Ib[511:504] <= byte_out;
counter <= counter + 1;
end
else if (counter < 256) begin
Qb[503:0] <= Qb[511:8];
Qb[511:504] <= byte_out;
counter <= counter + 1;
```

```
end
else
state <= WAIT;
end

endcase
end


////////////////////////////////////////////////////////////////////////////////
// INSTANTIATE NECCESSARY MODULES

// walsh function
wire [63:0] walsh_output_bus;
walsh_gen walsh32(system_clk, reset, 32, walsh_output_bit, walsh_output_bus);

// short pseudonoise generator
wire [63:0] pn_i_buff;
wire [63:0] pn_q_buff;
reg shift;
short_pn pseudonoisegen(system_clk, reset, shift, pn_i, pn_q, pn_i_buff, pn_q_buff);

// correlators
wire signed [7:0] Pilot_Ia;
wire signed [7:0] Pilot_Qa;
wire signed [7:0] Sync_Ia;
wire signed [7:0] Sync_Qa;
wire signed [7:0] Pilot_Ib;
wire signed [7:0] Pilot_Qb;
wire signed [7:0] Sync_Ib;
wire signed [7:0] Sync_Qb;
correlator corr_a(system_clk, reset, Ia, Qa, pn_i_buff, pn_q_buff, walsh_output_bus,
correlator corr_b(system_clk, reset, Ib, Qb, pn_i_buff, pn_q_buff, walsh_output_bus,


////////////////////////////////////////////////////////////////////////////////
// PERFORM MAXIMAL RATIO COMBINING (for each correlator)

reg signed [16:0] demod_a;
reg signed [16:0] demod_b;

always @ (posedge symbol_clk)
begin
demod_a <= (Pilot_Ia * Sync_Ia) + (Pilot_Qa * Sync_Qa);
demod_b <= (Pilot_Ia * Sync_Ia) + (Pilot_Qa * Sync_Qa);
end
```

```
////////////////////////////////////////////////////////////////////////////////
// SYNCHRONIZED DECISION AND PN OFFSETTER
// If already synchronized, the system ensures that the in-use correlator still says
// the PN sequence is synchronized. If it is still searching for the right PN window,
// then a positively correlated value will make this block latch onto that correlator
// and tell the other modules valid data is coming out. Other modules must operate on
// coherent frames, so thus the PN's signal notifying of a frame boundary is used alo
// with the synchronized signal from this block.

reg correlator_choice;

always @ (posedge symbol_clk)
begin
if (!synchronized) // if we are still searching for the signal
begin
if (Pilot_Ia >= THRESHOLD) // use "A" correlator
begin
correlator_choice <= 0;
synchronized <= 1;
shift <= 0;
end
else if (Pilot_Ib >= THRESHOLD) // use "B" correlator
begin
correlator_choice <= 1;
synchronized <= 1;
shift <= 0;
end
else // if still uncorrelated, we do not have the right PN sequence
shift <= 1; // shift PN sequence
end
else
if ((correlator_choice ? Pilot_Ib : Pilot_Ia) < THRESHOLD) // if correlator is no long
begin
synchronized <= 0;   // then we are unsynchronized
shift <= 1;
end
end

//always @ (posedge system_clk)
//if (shift == 1)
// shift <= 0;


////////////////////////////////////////////////////////////////////////////////
```

```
// MAKE BIT-OUTPUT DECISION BASED ON MRC
// based on the assumption that a value greater than zero is a logic 1
// and less than or equal to zero is a logic 0
// note that this is not a "true" QPSK implementation, but this is how
// CDMA handles data. 19.2Ksps should come from this portion.

always @ (posedge symbol_clk)
begin
if (synchronized)
begin
if ((correlator_choice ? demod_b : demod_a) > 0)  // mux to select currently chosen c
// if MRC is greater than zero, then 1
demod_data_out <= 1;
else // if MRC is less than or equal to zero
demod_data_out <= 0;
end
else
demod_data_out <= 0;  // lower the line
// perhaps tell computer that we are not yet synched?
end

endmodule
```

B.3 correlator.v

```
//////////////////////////////////////////////////////////////////////////////////
//
// IQ CORRELATOR
// Accumulates data over 64 samples
// Highly correlated pilot samples will have a spike towards the 0-valued symbol
//
//////////////////////////////////////////////////////////////////////////////////
module correlator(clk, reset, Ia, Qa, PN_I, PN_Q, Walsh, Pilot_I, Pilot_Q, Sync_I, Sy

input clk, reset;
input [511:0] Ia; // 64 samples, 8bits each
input [511:0] Qa; // 64 samples, 8bits each
input [63:0] PN_I;
input [63:0] PN_Q;
input [63:0] Walsh;
output [7:0] Pilot_I;
output [7:0] Pilot_Q;
output [7:0] Sync_I;
output [7:0] Sync_Q;
reg signed [7:0] Pilot_I;
reg signed [7:0] Pilot_Q;
```

```
reg signed [7:0] Sync_I;
reg signed [7:0] Sync_Q;

// pilot accumulators
reg signed [7:0] temp_pI;
reg signed [7:0] temp_pQ;
// data accumulators
reg signed [7:0] temp_dI;
reg signed [7:0] temp_dQ;

reg signed [7:0] byte_Ii;
reg signed [7:0] byte_Qi;
reg [5:0] i;

always @ (posedge clk) // rate at which each new 64-bit window comes in
begin
// initialize accumulator values
temp_pI = 0;
temp_pQ = 0;
temp_dI = 0;
temp_dQ = 0;

// sum 64 samples
for (i = 0; i < 64; i = i +1) begin
byte_Ii = Ia[i*8 +: 8]; // select next byte of data from I signal input
byte_Qi = Qa[i*8 +: 8]; // select next byte of data from Q signal input

// first perform pilot channel correlation (since Walsh = 0)
temp_pI = temp_pI + ((PN_I[i] ? -1 : 1) * byte_Ii)
+ ((PN_Q[i] ? -1 : 1) * byte_Qi);

temp_pQ = temp_pQ + ((PN_I[i] ? -1 : 1) * byte_Qi)
- ((PN_Q[i] ? -1 : 1) * byte_Ii);

// next perform sync channel correlation (using Walsh = 32)
// note that this correlator function can be used for any other
// channel - all that needs to be changed is the Walsh code
temp_dI = temp_dI + ((PN_I[i] ? -1 : 1) * byte_Ii * (Walsh[i] ? -1 : 1))
+ ((PN_Q[i] ? -1 : 1) * byte_Qi * (Walsh[i] ? -1 : 1));

temp_dQ = temp_dQ + ((PN_I[i] ? -1 : 1) * byte_Qi * (Walsh[i] ? -1 : 1))
- ((PN_Q[i] ? -1 : 1) * byte_Ii * (Walsh[i] ? -1 : 1));
end

// Output accumulated, normalized sums
// we divide by 128 because there are 64 samples and since both the I and Q are
```

```
// being factored into each variable, we divide by another two
Pilot_I <= temp_pI;
Pilot_Q <= temp_pQ;
Sync_I <= temp_dI;
Sync_Q <= temp_dQ;

end

endmodule
```

B.4 short_pn.v

```
////////////////////////////////////////////////////////////////////////////////
//
// SHORT PN SEQUENCE GENERATOR
//
// This module implements two linear feedback shift registers
// to accomplish generation of the In-Phase and Quadrature
// Pseudonoise sequences.
//
////////////////////////////////////////////////////////////////////////////////
module short_pn(clock, chip_clock, reset, shift, pn_i, pn_q, pn_i_shift, pn_q_shift,
  input clock;
input chip_clock;
  input reset;
input shift;

output [63:0] pn_i_shift;
output [63:0] pn_q_shift;
output [63:0] pn_i_buff;
output [63:0] pn_q_buff;
  output pn_i;
output pn_q;

  //PI(x) = x15 + x13 + x9 + x8 + x7 + x5 + 1
//PQ(X) = x15 + x12 + x11 + x10 + x6 + x5 + x4 + x3 + 1
//parameter PN_I_CHAR_POLY = 16'b1010001110100001;
//parameter PN_Q_CHAR_POLY = 16'b1001110001111001;
parameter PN_I_CHAR_POLY = 16'h85C4;
parameter PN_Q_CHAR_POLY = 16'h9E38;

parameter PN_I_INIT = 16'h4000;
parameter PN_Q_INIT = 16'h4000;
//parameter PN_I_INIT = 16'hFFFF;
//parameter PN_Q_INIT = 16'hFFFF;
```

```
wire pni_zeroes;
wire pnq_zeroes;

reg [14:0] pn_i_lfsr;
reg [14:0] pn_q_lfsr;

reg [63:0] pn_i_buff;
reg [63:0] pn_q_buff;

reg [63:0] pn_i_shift;
reg [63:0] pn_q_shift;


reg [6:0] j;
reg shift_pending;

wire next_i_bit;
wire next_q_bit;

wire pn_i;
wire pn_q;

// wire [14:0] pn_i_lfsr_temp;
// wire [14:0] pn_q_lfsr_temp;


//If the last 14 digits generated are zeroes, raise pni_zeroes or pnq_zeroes
assign pni_zeroes = ((pn_i_lfsr & 16'hFFFE) == 0) ? 1'b1 : 1'b0;
assign pnq_zeroes = ((pn_q_lfsr & 16'hFFFE) == 0) ? 1'b1 : 1'b0;

assign next_i_bit = pn_i_lfsr[14] ^ pni_zeroes;
assign next_q_bit = pn_q_lfsr[14] ^ pnq_zeroes;
//assign next_i_bit = pn_i_lfsr[14];
//assign next_q_bit = pn_q_lfsr[14];

assign pn_i = next_i_bit;
assign pn_q = next_q_bit;

//assign pn_i_lfsr_temp = (next_i_bit == 1'b1) ? pn_i_lfsr ^ PN_I_CHAR_POLY : pn_i_lf
//assign pn_q_lfsr_temp = (next_q_bit == 1'b1) ? pn_q_lfsr ^ PN_Q_CHAR_POLY : pn_q_lf

initial j = 0;
```

```
always @ (posedge clock) begin
  if(reset) begin
pn_i_lfsr <= PN_I_INIT;
pn_q_lfsr <= PN_Q_INIT;

pn_i_buff <= 64'd0;
pn_q_buff <= 64'd0;

pn_i_shift <= 64'd0;
pn_q_shift <= 64'd0;

//zeroes <= 4'd0;
j <= 0;
shift_pending <= 0;

end

if(shift || ~chip_clock) shift_pending <= 1;

if((chip_clock || shift_pending || shift) && ~reset) begin

//if we didn't get here via chip_clock, then we're processing a shift
if(~chip_clock)
shift_pending <= 0;




/*
pn_i_lfsr[14:0] <= pn_i_lfsr_temp[15:1]; //shift right
pn_i_lfsr[15] <= next_i_bit;
//pn_i_lfsr[15] = 0;

pn_q_lfsr[14:0] <= pn_q_lfsr_temp[15:1]; //shift right
pn_q_lfsr[15] <= next_q_bit;
//pn_q_lfsr[15] = 0;

pn_i_shift[63:1] <= pn_i_shift[62:0];
pn_i_shift[0] <= next_i_bit;

pn_q_shift[63:1] <= pn_q_shift[62:0];
pn_q_shift[0] <= next_q_bit;
*/

pn_i_lfsr[0] <= pn_i_lfsr[14];
pn_i_lfsr[4:1] <= pn_i_lfsr[3:0] ;
```

```
pn_i_lfsr[5] <= pn_i_lfsr[4] ^ pn_i_lfsr[14];
pn_i_lfsr[6] <= pn_i_lfsr[5];
pn_i_lfsr[7] <= pn_i_lfsr[6] ^ pn_i_lfsr[14];
pn_i_lfsr[8] <= pn_i_lfsr[7] ^ pn_i_lfsr[14];
pn_i_lfsr[9] <= pn_i_lfsr[8] ^ pn_i_lfsr[14];
pn_i_lfsr[12:10] <= pn_i_lfsr[11:9];
pn_i_lfsr[13] <= pn_i_lfsr[12] ^ pn_i_lfsr[14];
pn_i_lfsr[14] <= pn_i_lfsr[13];

pn_q_lfsr[0] <= pn_q_lfsr[14];
pn_q_lfsr[2:1] <= pn_q_lfsr[1:0] ;
pn_q_lfsr[3] <= pn_q_lfsr[2] ^ pn_q_lfsr[14];
pn_q_lfsr[4] <= pn_q_lfsr[3] ^ pn_q_lfsr[14];
pn_q_lfsr[5] <= pn_q_lfsr[4] ^ pn_q_lfsr[14];
pn_q_lfsr[6] <= pn_q_lfsr[5] ^ pn_q_lfsr[14];
pn_q_lfsr[9:7] <= pn_q_lfsr[8:6];
pn_q_lfsr[10] <= pn_q_lfsr[9] ^ pn_q_lfsr[14];
pn_q_lfsr[11] <= pn_q_lfsr[10] ^ pn_q_lfsr[14];
pn_q_lfsr[12] <= pn_q_lfsr[11] ^ pn_q_lfsr[14];
pn_q_lfsr[14:13] <= pn_q_lfsr[13:12];

pn_i_shift[63:1] <= pn_i_shift[62:0];
pn_i_shift[0] <= next_i_bit;

pn_q_shift[63:1] <= pn_q_shift[62:0];
pn_q_shift[0] <= next_q_bit;

if(j == 0) begin
  pn_i_buff <= pn_i_shift;
pn_q_buff <= pn_q_shift;
end

if(j == 63) begin
j <= 0;
end
else begin
j <= j + 1;
end

//end



end
end
```

```
endmodule
```

B.5 walsh_gen.v

```
‘timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
//
// WALSH CODE GENERATOR
//
// This module produces a stream of bits corresponding to the walsh code of a particu
// input index in the walsh table. Output  starts with the LSB of the walsh code and
// through the 64 bits and then loops around. Entries are represented using 1 -> 0 an
// -1 -> 1 notation.
//////////////////////////////////////////////////////////////////////////////////
module walsh_gen(clk, reset, walsh_index, walsh_output_bit, walsh_output_bus);

input clk, reset;
input [5:0] walsh_index;
output walsh_output_bit; // allow for serial out
output [63:0] walsh_output_bus; // or parrallel out
reg walsh_output_bit;
reg [63:0] walsh_output_bus;

reg [5:0] curr_bit_num; // tracks which bit of the walsh entry is currently being gen
// note that this will automatically overflow after the 64th bit
reg [5:0] a; // placeholder
reg [63:0] temp_walsh;

assign walsh_output_bit = ((a[0]^a[1])^(a[2]^a[3]))^(a[4]^a[5]);

always @ (posedge clk)
begin
if (reset)
curr_bit_num <= 0;
else
begin
if (curr_bit_num == 0) // latch prll out
walsh_output_bus <= temp_walsh;
a <= curr_bit_num & walsh_index;

curr_bit_num <= curr_bit_num + 1;
temp_walsh[curr_bit_num] <= walsh_output_bit;
end
end

endmodule
```

B.6 viterbi.v

```
//////////////////////////////////////////////////////////////////////////////////
//
// VITERBI DECODER MODULE
//
// This module implements the FSM and instantiation of all the modules used
// for Viterbi decoding.
//
//////////////////////////////////////////////////////////////////////////////////
module viterbi(clock, reset, input_symbols, ready, output_data, done, state);
    input clock;
    input reset;
    input [63:0] input_symbols;
input ready;
    output [31:0] output_data;
output done;
output [2:0] state;

reg acs_clock;
wire acs_enable;
assign acs_enable = (acs_clock == 1'b1) ? 1'b1 : 1'b0;
reg [6:0] i; //for counting from 0 to 63
reg [6:0] j; //for counting from 0 to 31
reg [2:0] state;
reg [2:0] next;

parameter STATE_IDLE  = 3'd0;
parameter STATE_ACS  = 3'd1;
parameter STATE_TRACEBACK  = 3'd2;
parameter STATE_TRANSFER = 3'd3;

wire symbol0;
wire symbol1;
assign symbol0 = input_symbols[(i<32)? (i<<1) : 0];
assign symbol1 = input_symbols[(i<32)? (i<<1+1) : 1];

reg signed [7:0] acs_path_metrics[0:31][0:255];
reg [7:0] acs_prev_states[0:31][0:255];
reg acs_transitions[0:31][0:255];
reg signed [7:0] acs_rel_metrics[0:31][0:255];
reg [7:0] bestState;
reg [31:0] output_data;
reg done;

//Instantiate 256 Add-Compare-Select Units
```

```
‘include "acs_declare.v"

//Includes tons of pm_compare modules which continuosly calculate the max of acs_path

wire [7:0] acs_path_metric_max;
‘include "pm_max.v"
assign acs_path_metric_max = o8val0out;


always @ (posedge clock) begin
if(reset) begin
acs_clock <= 0;
done <= 0;
output_data <= 0;
bestState <= 0;
state <= STATE_IDLE;
next = STATE_IDLE;
i <= 0;
end
else begin

case(state)
STATE_IDLE: begin
done <= 0;
if(ready) begin
next = STATE_ACS;
i <= 0;
acs_clock <= 1’b1;
end
end

STATE_ACS: begin
i <= i + 1;
if(i >= 32) begin
next = STATE_TRACEBACK;
acs_clock <= 1’b0;
j <= 31;
bestState <= acs_path_metric_max;

end
else if(i < 32) begin

//acs_assign.v achieves the following loop
‘include "acs_assign.v"

//on the 31st loop, get ready for the traceback
```

```
// by taking the max to find the bestState to start
// traceback from
//if(acs_path_metric[j] > acs_path_metric[bestState])
// bestState = j;
//bestState <= acs_path_metric_max;

//for(j=0; j<256; j=j+1) begin
// acs_rel_metrics[i][j] <= acs_rel_metric[j];
// acs_path_metrics[i][j] <= acs_path_metric[j];
// acs_transitions[i][j] <= acs_transition[j];
// acs_prev_states[i][j] <= acs_prevstate[j];
//end

end

end

STATE_TRACEBACK: begin

//$display(bestState);
//for(j=31; j>=0; j=j-1) begin
$display(acs_transitions[j][j]);
output_data[j] <= acs_transitions[j][bestState];
bestState <= acs_prev_states[j][bestState];
if(j == 0) begin
next = STATE_TRANSFER;
end
else
j <= j-1;



//end


end

STATE_TRANSFER: begin
next = STATE_IDLE;
done <= 1;
end

endcase

state <= next;
```

```
end
end


endmodule
```

B.7 deinterleaver_derepeater.v

```
//////////////////////////////////////////////////////////////////////////////////
//
// DEINTERLEAVER AND DEREPEATER MODULE
//
// This module combines both the deinterleaving and derepeating into one module
// in order to optimize for performance. Repeated symbols are placed in a "don't
// care" bit which is never used for the output. Since data is latched in and out,
// module is made to run at the system clock. Due to a one-cycle extra delay,
// this must (and is) faster than the incoming datarate of 1.228Mcps.
//
//////////////////////////////////////////////////////////////////////////////////

module deinterleaver_derepeater(clk, reset, load, frame_in, deinterleaved_frame_out,

input clk, reset, load;
input [127:0] frame_in;
output done;
output [63:0] deinterleaved_frame_out;
//output [5:0] mapped_addr;
reg [63:0] deinterleaved_frame_out = 0;
reg done = 0;
// temp variables for data latching
reg [63:0] temp_output = 0;
reg [127:0] frame_input;
//output [6:0] counter;
// basically to allow "for-loop" like activity through the "array" loaded into ROM
reg [6:0] counter = 1;
// instantiate ROM
wire [5:0] mapped_addr;
sync_deinterleaver loadtable(clk, counter, mapped_addr);

reg state;
parameter START = 0;
parameter IDLE = 1;
reg clocker = 0;
reg clock2 = 0;

always @ (posedge clk)
```

```
begin
if (clock2)
clocker <= ~clocker;
clock2 <= ~clock2;
end

always @ (posedge clocker)
begin
if (reset)
begin //reset everything to zero
counter <= 1;
temp_output <= 0;
deinterleaved_frame_out <= 0;
frame_input <= 0;
done <= 0;
state <= IDLE;
end
else
begin
case(state)

START: begin
if (counter == 127) //after 128 bits have been shuffled, and counter overflows
begin
temp_output[63] <= frame_input[127];
state <= IDLE;
end
// put current symbol from frame into correct position of output
temp_output[mapped_addr] <= frame_input[counter];
counter <= counter + 1;
end

IDLE: begin
//latch output
deinterleaved_frame_out <= temp_output;
if (load)
begin
state <= START;
frame_input <= frame_in;
done <= 0;
counter <= 1;
end
else
// tell other modules deinterleaver is done
done <= 1;
end
```

```
default: state <= IDLE;

endcase
end
end

endmodule
```

B.8 pm_compare.v

```
//////////////////////////////////////////////////////////////////////////////////
//
// Path Metric 2-Way Compare Module
//
// This module takes two 7-bit path metrics and compares them.
// It also takes two values, representing the number of each path metric.
// The output path metric and value are the pair of values with the greater path metr
//
//////////////////////////////////////////////////////////////////////////////////
module pm_compare(pm1, val1, pm2, val2, pm_out, val_out);
    input [7:0] pm1;
    input [7:0] val1;
    input [7:0] pm2;
    input [7:0] val2;
output [7:0] pm_out;
    output [7:0] val_out;

wire pm1gtpm2;
assign pm1gtpm2 = (pm1 > pm2) ? 1'b1 : 1'b0;

assign val_out = pm1gtpm2 ? val1 : val2;
assign pm_out = pm1gtpm2 ? pm1 : pm2;

endmodule
```

B.9 acs_node.v

```
//////////////////////////////////////////////////////////////////////////////////
//
// ADD COMPARE SELECT MODULE
//
// This module implements the ACS functionality
// instantiated 256 in the Viterbi algorithm processing.
//////////////////////////////////////////////////////////////////////////////////
```

```
module acs_node(clock, reset, symbol_in0, symbol_in1, bm_in0, bm_in1, out_prev_state,
    input clock;
    input reset;

    input symbol_in0;
    input symbol_in1;
    input signed [7:0] bm_in0;
    input signed [7:0] bm_in1;

output [7:0] out_prev_state;
    output signed [7:0] out_rel_metric;
output out_transition;
output signed [7:0] out_path_metric;

//All parameters are filled upon instantiation of the 256 ACS nodes
parameter state = 8'd0;

parameter state_b0_codeword  = 8'd0; //int
parameter state_b0_prevstate  = 8'd0; //int
parameter state_b0_input  = 1'd0; //int

parameter state_b1_codeword  = 8'd0; //int
parameter state_b1_prevstate  = 8'd0; //int
parameter state_b1_input  = 1'd0; //int

//Output Registers
wire signed [7:0] out_path_metric;
wire signed [7:0] out_rel_metric;
wire [7:0] out_prev_state;
wire out_transition;

//Work Registers
reg signed [7:0] b0_temp_metric;
reg signed [7:0] b1_temp_metric;

assign out_path_metric = (b0_temp_metric > b1_temp_metric) ? b0_temp_metric : b1_temp
assign out_rel_metric = (b0_temp_metric > b1_temp_metric) ? ((state_b0_input == 1'b1)
assign out_prev_state = (b0_temp_metric > b1_temp_metric) ? state_b0_prevstate : stat
assign out_transition = (b0_temp_metric > b1_temp_metric) ? state_b0_input : state_b1


always @ (posedge clock) begin
  if(reset) begin

b0_temp_metric <= 0;
b1_temp_metric <= 0;
```

```
end
  else begin

//Perform Add Operation
if(state_b0_codeword[0] == 1'b1 && state_b0_codeword[1] == 1'b1)
b0_temp_metric <= bm_in0 - symbol_in0 - symbol_in1;
else if(state_b0_codeword[0] == 1'b0 && state_b0_codeword[1] == 1'b1)
b0_temp_metric <= bm_in0 + symbol_in0 - symbol_in1;
else if(state_b0_codeword[0] == 1'b1 && state_b0_codeword[1] == 1'b0)
b0_temp_metric <= bm_in0 - symbol_in0 + symbol_in1;
else if(state_b0_codeword[0] == 1'b0 && state_b0_codeword[1] == 1'b0)
b0_temp_metric <= bm_in0 + symbol_in0 + symbol_in1;


if(state_b1_codeword[0] == 1'b1 && state_b1_codeword[1] == 1'b1)
b1_temp_metric <= bm_in1 - symbol_in0 - symbol_in1;
else if(state_b1_codeword[0] == 1'b0 && state_b1_codeword[1] == 1'b1)
b1_temp_metric <= bm_in1 + symbol_in0 - symbol_in1;
else if(state_b1_codeword[0] == 1'b1 && state_b1_codeword[1] == 1'b0)
b1_temp_metric <= bm_in1 - symbol_in0 + symbol_in1;
else if(state_b1_codeword[0] == 1'b0 && state_b1_codeword[1] == 1'b0)
b1_temp_metric <= bm_in1 + symbol_in0 + symbol_in1;


end



end



endmodule
```

B.10 uart.v

```
//////////////////////////////////////////////////////////////////////////////////
//
// UART TRANSMITTER MODULE
//
// This module takes 10x the baudrate to transmit a single byte. New data should
// note be transmitted for at least this time.
//
//////////////////////////////////////////////////////////////////////////////////
module uart(serial_clk, reset, data_in, transmit, uart_line, done);
```

```
input serial_clk, transmit, reset;
input [7:0] data_in;
output uart_line, done;
//register outputs
reg uart_line, done;

//SHIFT REGISTER

//shift register variables
reg load, shift;
reg [7:0] curr_trans_byte;
//shifter
always @ (posedge serial_clk)
begin
if (reset || load)
curr_trans_byte <= data_in;
else if (shift)
curr_trans_byte <= curr_trans_byte >> 1; //LSB first
end

//OUTPUT MUX
// mux control register
reg [1:0] mux_control;
wire [2:0] mux_ins;
//setup inputs into one bus
assign mux_ins[0] = curr_trans_byte[0]; //LSB first
assign mux_ins[1] = 1;
assign mux_ins[2] = 0;
//tie uart output line to mux
assign uart_line_out = mux_ins[mux_control];
//register the output
always @ (posedge serial_clk) uart_line <= uart_line_out;

//FSM
//variables
reg [1:0] state;
reg [3:0] bit_count;
//constants
parameter IDLE = 0;
parameter START = 1;
parameter TRANSMIT = 2;
parameter END = 3;
//fsm controler
always @ (posedge serial_clk)
begin
```

```
if (reset)
begin
state <= IDLE;
end
else
case(state)
IDLE: begin //note that IDLE also acts as the stop bit
mux_control <= 1; //set uart line high
shift <= 0;
if (transmit)
state <= START;
end

START: begin
mux_control <= 2; //set uart line low
state <= TRANSMIT;
load <= 1;
bit_count <= 0;
end

TRANSMIT: begin
mux_control <= 0; //set uart line to shift register
load <= 0;
if (bit_count < 7)
begin
shift <= 1;
bit_count <= bit_count + 1;
end
else
state <= IDLE;
end

default:  begin
state <= IDLE;
shift <= 0;
bit_count <= 0;
load <= 1;
mux_control <= 1;
end
endcase
end

endmodule
```

B.11 uart_rec.v

```
////////////////////////////////////////////////////////////////////////////////
//
// UART RECEIVER MODULE
//
////////////////////////////////////////////////////////////////////////////////
module uart_rec(system_clk, reset, ser_in, byte_ready, byte_out);

input system_clk, reset, ser_in;
output [7:0] byte_out;
output byte_ready;
reg [7:0] byte_out;
reg byte_ready;
reg [10:0] counter;
//parameter BAUDRATE = 1288000;
parameter BAUDRATE = 9600;
reg [7:0] byte_out_temp;
reg oversampled_serial_clk;

//CLOCK GENERATOR

always @ (posedge system_clk)
begin
//if (counter == (27000000/(BAUDRATE * 32)))
if (counter == 87)
begin
 oversampled_serial_clk = ~oversampled_serial_clk;
 counter <= 0;
end
else
counter <= counter + 1;
end

//DEBOUNCING
reg store1;
reg store2;
reg rec; //debounced received bit
always @ (posedge oversampled_serial_clk)
begin
store1 <= ser_in;
store2 <= store1;
rec <= store2;
end

//FSM
```

```
//variables
reg [2:0] state;
reg [3:0] waiter;
reg [3:0] byte_count;
//constants
parameter IDLE = 0;
parameter CENTER = 1;
parameter STALL = 2;
parameter SAMPLE = 3;
parameter STOP = 4;
//fsm controler
always @ (posedge oversampled_serial_clk) //oversampled serial clock is 16x the incom
begin
if (reset)
begin
state <= IDLE;
end
else
case(state)
IDLE: begin
waiter <= 0;
byte_ready <= 0;
byte_count <= 0;
if (rec == 0)
state <= CENTER;
end

CENTER: begin
if (waiter == 7) //wait until center of start bit
begin
if (rec == 0)
begin
waiter <= 0;
state <= STALL; //wait until center of first data bit
end
else
state <= IDLE; //false alarm
end
else
waiter <= waiter + 1; //otherwise wait for center
end

STALL: begin //wait for one baudrate
if (byte_count < 8)
begin
if (waiter == 14) //check for one baud tick
```

```
begin
state <= SAMPLE;
waiter <= 0;
byte_count <= byte_count + 1;
end
else
waiter <= waiter + 1;
end
else
begin
state <= STOP;
byte_out <= byte_out_temp;
waiter <= 0;
end
end

SAMPLE: begin //shift bit into temp register
byte_out_temp[6:0] <= byte_out_temp[7:1];
byte_out_temp[7] <= rec;
state <= STALL;
end

STOP: begin //wait for one baudrate
byte_ready <= 1;
if (waiter == 14) //check for one baud tick
begin
state <= IDLE;
waiter <= 0;
end
else
waiter <= waiter + 1;
end

default:  begin
state <= IDLE;
end
endcase
end
endmodule
```

B.12 cdma_rec.py

```
#This python module implements the PC post-processing
#step of the IS-95A receiver.
```

```
import serial


def display_parse(frame):
        print "Network ID: " + str(ord(frame[0]))
        print "System ID: " + str(ord(frame[1]))
        print "Paging channel data rate: " + str(ord(frame[2]))
        print "PN Offset: " + str(ord(frame[3])) + "\n"


def main():
        print "\nCDMA FORWARD CHANNEL RECEIVER DATA INTERPRETER"
        print "\nconnecting to FPGA..."
        #Instantiate serial port
        ser=serial.Serial(0)
        ser.baudrate=9600
        if ser.isOpen():
                print "\nCONNECTED!"
                print "\nwaiting for data..."
                while 1:
                        x = ser.read()
                        if x == 'Q':
                                "exiting"
                                break
                        if x == 'U':
```

```
                                frame = ser.read(4)

                                print "\nDATA RECEIVED:"

                                print "\nprocessing data...\n"

                                display_parse(frame)

                        print "waiting for valid data..."

        else:

                print "\nunable to connect to FPGA"

main()
```

B.13 acs_assign.v

This file is over 20 pages long and is included as a digital attachment to this document.

B.14 acs_declare.v

This file is over 20 pages long and is included as a digital attachment to this document.

B.15 pm_max.v

This file is over 20 pages long and is included as a digital attachment to this document.