CS 151 C Winter Quarter 2004
Design of Digital Systems Final Project
Prof. Leon Alkalai

Adaptive Differential Pulse Code Modulation Decoder/Encoder Hardware Implementation

Adam Kaplan
Serge Baltayan

March 23rd, 2004

ABSTRACT

Our project is the implementation of an Adaptive Differential Pulse Code Modulation (ADPCM) encoder/decoder pair in reconfigurable logic. We selected this particular project due to its algorithmic complexity, as well as its influence upon voice encryption techniques of cellular technology. This project is interesting because it encompasses some of the fundamental techniques of hardware design. It requires interfacing with memory, local buffering (of reads and writes), compression based on an algorithm which performs prediction, and real-time sound encoding. Our implementation assumes that the sound data exists in RAM before execution begins, but it could trivially be extended to operate on RAM as the sound is being stored (this feature would require synchronization logic and it was decided that this was beyond the scope of a single quarter project). The project was challenging because it uses many capabilities of the Xilinx XSV-50 board, including audio and RAM interfaces. It was also extremely educational for us, as we started with a high-level algorithmic specification, then took it down to a behavioral state-machine specification, and further into a full RTL specification (with separate processes for next-state, output function, and data path). Additionally, we learned that debugging at the RTL level is extremely difficult, and that it is more efficient to debug the high-level specification and re-implement the RTL code. Finally, we learned that debugging at the hardware level is much more time-consuming and difficult than debugging software (even when software simulation of the component has succeeded).

---

1. Introduction

The DVI Adaptive Differential Pulse Code Modulation (ADPCM) algorithm was first described in an IMA recommendation on audio formats and conversion practices [1]. ADPCM is a transformation that encodes 16-bit audio as 4 bits (a 4:1 compression ratio). In order to achieve this level of compression, the algorithm maintains an adaptive value predictor, which uses the distance between previous samples to store the most likely value of the next sample. The difference between samples is quantized down to a new sample using an adaptive step-size. The algorithm in [1] suggests using a table to adapt this step-size to the analyzed data. ADPCM has become widely used and adapted, and a variant of the algorithm performs voice encoding on cellular phones (allowing minimal data to be sent across the wireless network and increasing throughput).

Figure 1 demonstrates the high-level flow of the ADPCM encoder algorithm. The current sample analyzed is compared to the previous predictor value. This difference is compared to the current stepsize value, and a 4-bit encoded data value is produced (storing the delta between this sample and the predicted sample value). Simultaneously, the predictor and stepsize are updated to reflect new sample information. The decoder works in exactly the opposite way, unfolding the deltas into sample data and predictor values.

In this project, Adam Kaplan performed the coding of the ADPCM decoder (from high level algorithm to behavioral code) and the RTL partitioning and design of the decoder. Additionally, Adam simulated the high-level and low-level implementations to verify functionality. Serge Baltayan performed the coding of the ADPCM encoder (from high-level description to behavioral VHDL), and partitioned this into an RTL component. Serge designed and debugged both implementations of his encoder, and simulated the high-level and low-level components to verify functionality.

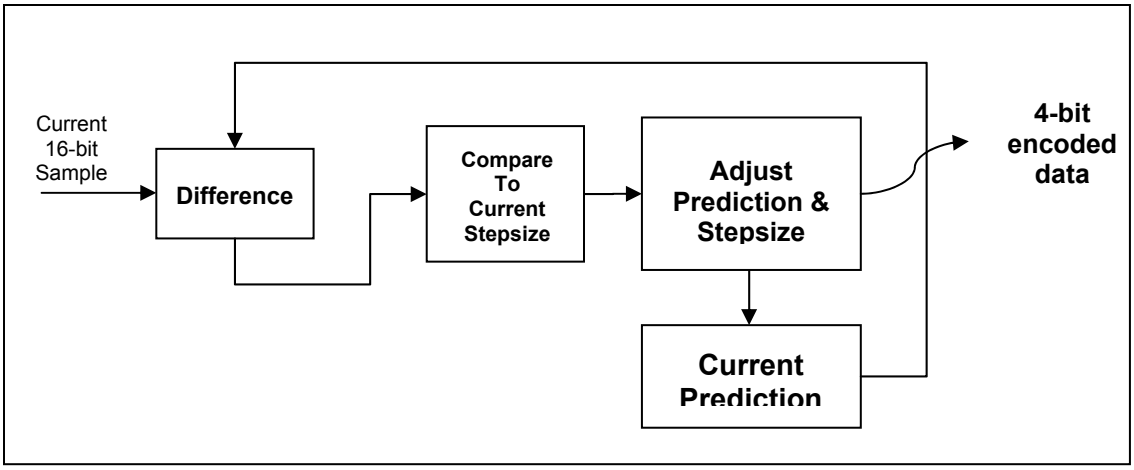

FIGURE 1. ADPCM Encoder High-Level Algorithmic Flow

Together, Serge and Adam worked on integrating the encoder and decoder designs into the Queensland audio project [2], allowing the decoder and encoder to execute on the same recorded audio data. (Since the encoder, decoder, and audio drivers together were too big for one XCV50 FPGA configuration, they were split into two configurations. One configuration included the player, recorder, and decoder, whereas the other configuration included the player, recorder, and encoder.)

2. System Level Description

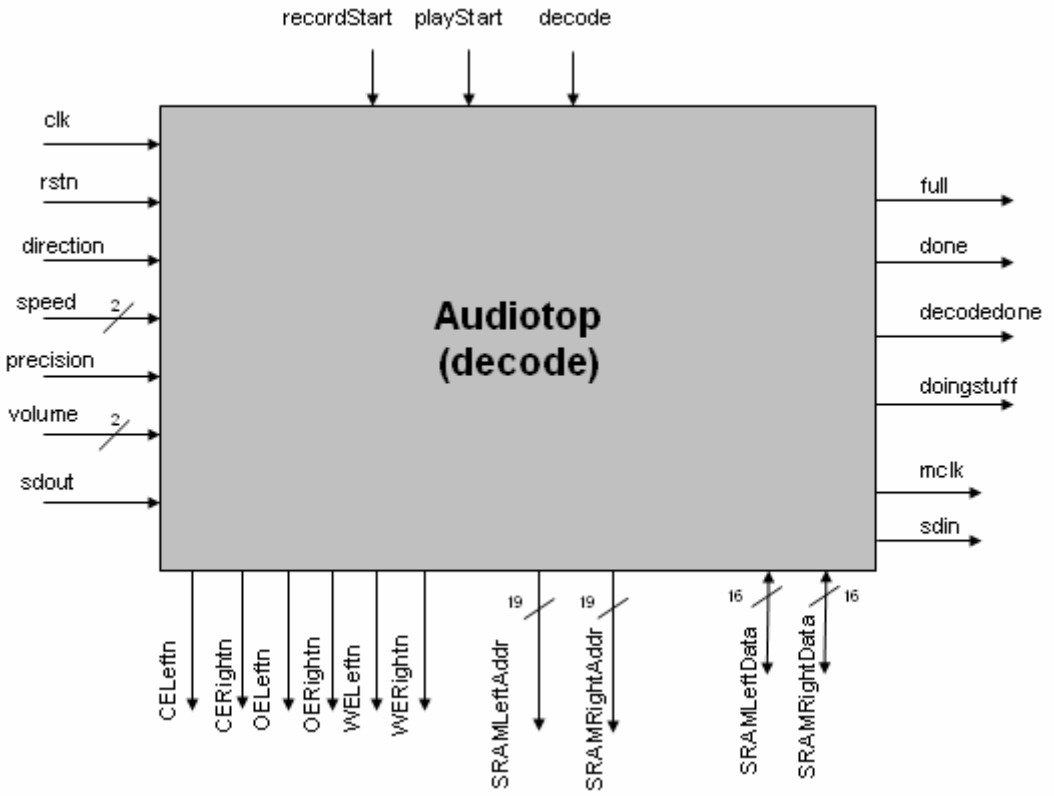

FIGURE 2. High-Level Decoder/Recorder/Player module

The device named Audiotop in Figures 2 and 5 is the highest level black box system in our design. Due to capacity constraints, Audiotop is divided into two configurations: one that consists of an encoder/recorder/player combination, and the other consisting of a decoder/recorder/player combination. It basically receives direct inputs from the user (human) via the I/O features of the Virtex XSV50 boards. The device possesses the classic ability to restart each process via a "reset" button and bring the device to an idle status. The device needs to be able to receive raw wave data in order to perform the encoding and decoding processes. When the "recorder" button is pressed, the raw sound is transmitted via an external analog signal into the 3.5mm audio input terminal on the Virtex board. The recorder next stores this analog audio as digital data in the memory banks using the ADC feature of the board. After the sound data is available on memory, the ADPCM compression encoding technique may be executed via the

"encode" button, converting the digital sound data in the upper half of RAM into encoded data one-fourth the size of the original in the lower half of RAM (since the algorithm utilizes a 4:1 compression ratio). The encoding may then be reversed by utilizing the decompression algorithm which converts the ADPCM compressed format in lower memory back into the raw sound data in higher memory. This may be performed via the "decoder" button. After the data has been decoded, the final data present in the memory banks will be the digital wav audio, playable via the "player" button. The player reads the digital audio from the Virtex memory banks and drives it to the 3.5mm output jack via a DAC.

The implementation further consists of output signals utilized in order to provide a stronger level of communication between user and device. Various indicator lights were used in order to display critical statuses of the device. During recording, a flashing indicator light is dedicated to notifying the user that the board's memory is entirely used and full, thus no longer allowing any more raw audio data to be input. Two other indicator lights are also used in aiding the user: one light to show the board busy during the encoding/decoding processes, and another to show that the device has completed the current process. Other I/O which are not in communication with the user of the device are the datastreams between the application and the RAM of the board.

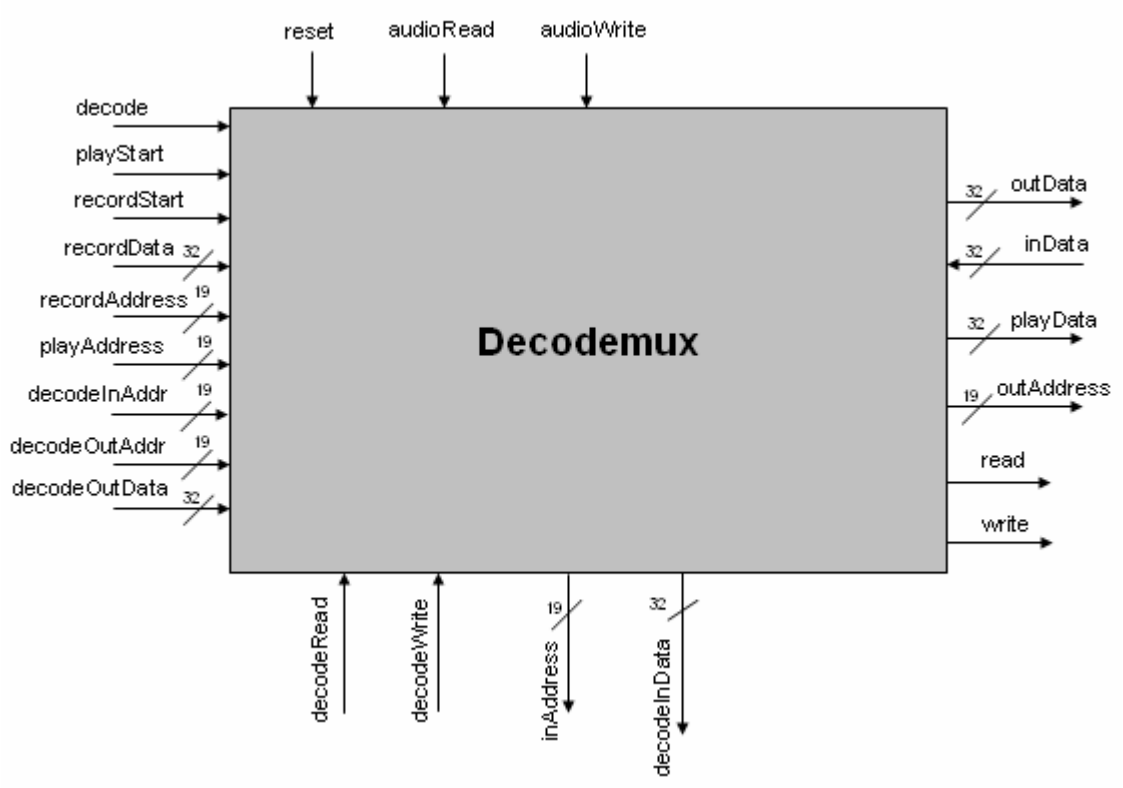

FIGURE 3. Decoder functionality assisting Multiplexer

The decodemux which will be described in the RTL section is shown in Figure 3.

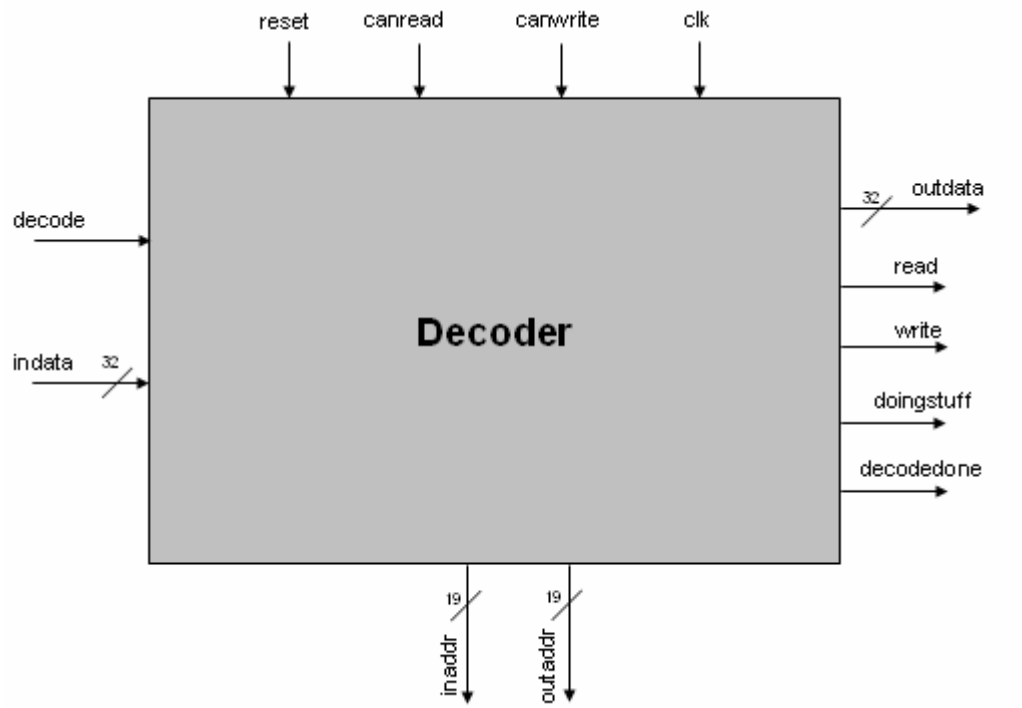

FIGURE 4. Decoder (decompression) module

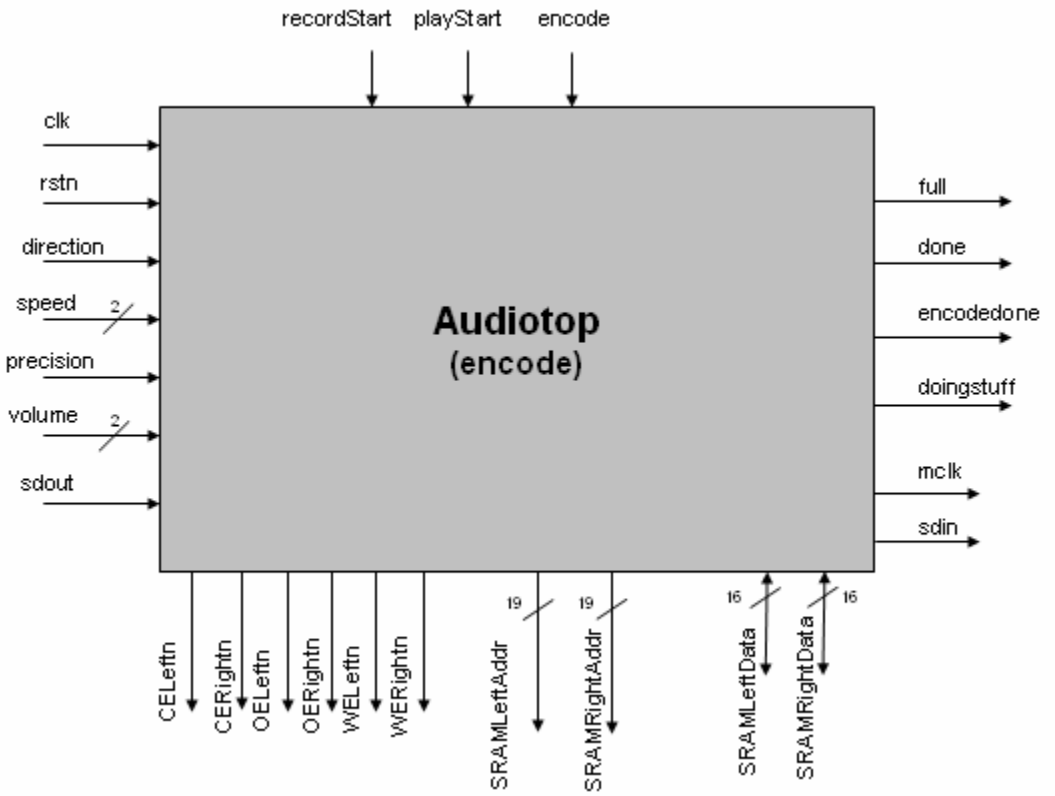

FIGURE 5. High-Level Encoder/Recorder/Player module

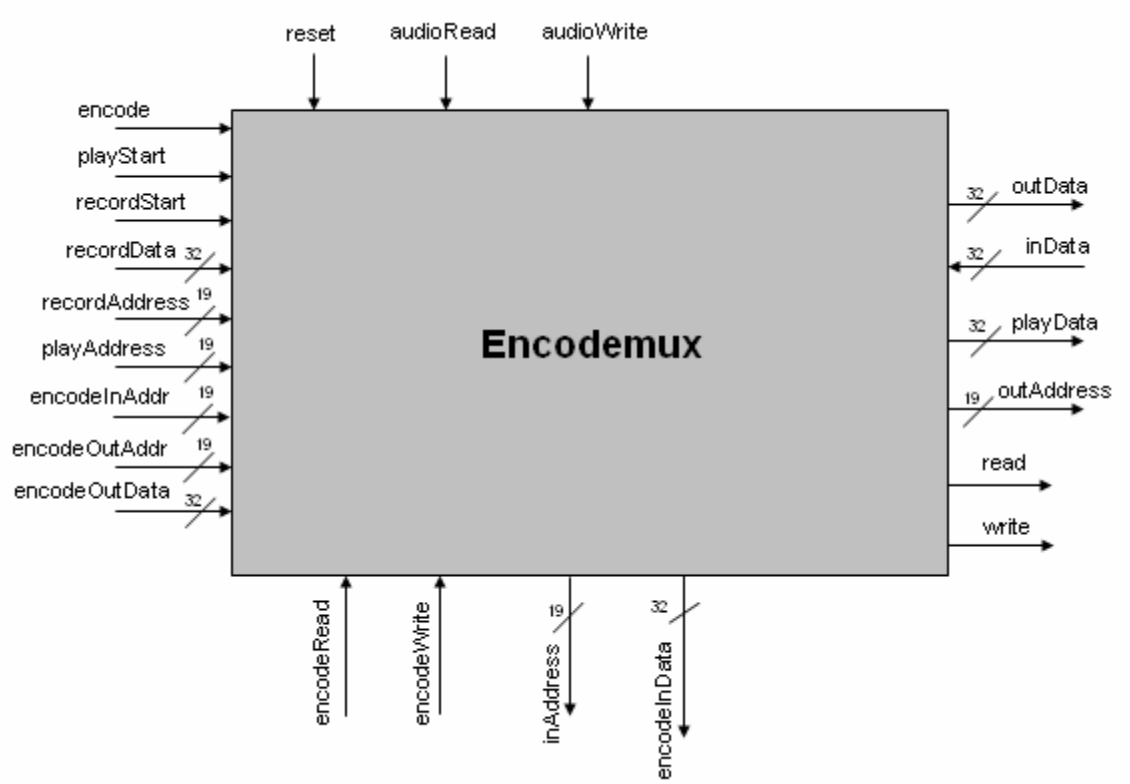

FIGURE 6. Encoder functionality assisting Multiplexer

The ADPCM encoder shown in Figure 5 behaves as follows. As every 16-bit sample is read from input, a difference is taken between that sample and the previous value prediction. This difference, along with the current quantization step (read from a stepsize table) is used to generate the next value prediction, the next index into the stepsize table, and the encoded difference (named *delta*). Delta is a 4-bit value, which is the ADPCM encoding of the 16-bit sample.

The ADPCM decoder is much simpler than the encoder. Every 4-bit delta value is read from the input, and used as an index into the index table. The index is incremented by the index table value, and used to read the stepsize table to find the next stepsize. Meanwhile, the value prediction (of the sample) is reconstructed using the current stepsize along with the delta value. The sample's value prediction, a 16 bit value, is output as the decoded sound data.

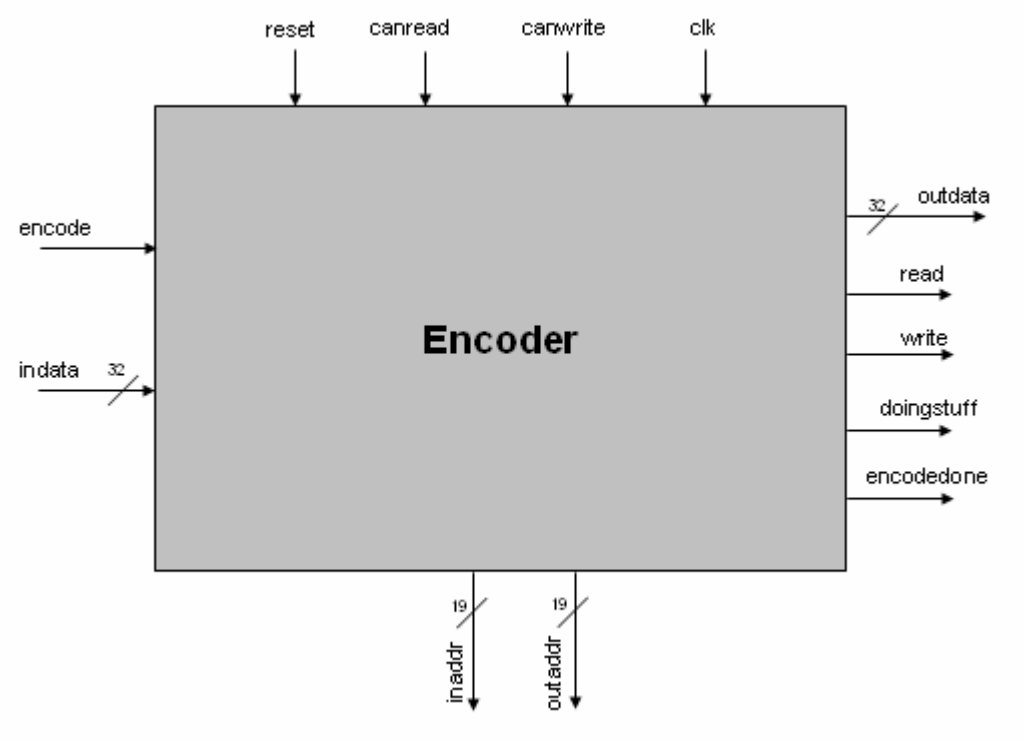

FIGURE 7. Encoder (compression) module

DECODER BEHAVIORAL PSUEDO-VHDL

```
architecture Behavioral of decoder is
       signal valpred : integer;
       signal stepsize_index : integer;
begin
       process (clk)
              type indextabletype is array (15 downto 0) of integer;
              type stepsizetabletype is array (88 downto 0) of integer;
              variable indextable : indextabletype;
              variable stepsizetable : stepsizetabletype;
              variable outp : std_logic_vector(15999 downto 0);
              variable inp : std_logic_vector(3999 downto 0);
              variable sign : bit;
              variable delta : std_logic_vector (3 downto 0);
              variable step : std_logic_vector (15 downto 0);
              variable Vvalpred : integer;
              variable vpdiff : integer;
              variable Vindex : integer;
              variable inputbuffer : std_logic_vector (7 downto 0);
              variable bufferstep : bit;

       begin

              if (clk'EVENT AND clk = '1' and decode = '1') then
                     inputbuffer := "00000000";
                     indextable := (-1, -1, -1, -1, 2, 4, 6, 8,
                                    -1, -1, -1, -1, 2, 4, 6, 8);

                     stepsizetable := (7, 8, 9, 10, 11, 12, 13, 14, 16, 17,
                                       19, 21, 23, 25, 28, 31, 34, 37, 41, 45,
                                       50, 55, 60, 66, 73, 80, 88, 97, 107, 118,
                                       130, 143, 157, 173, 190, 209, 230, 253, 279,
```

```
                                307, 337, 371, 408, 449, 494, 544, 598, 658,
                                724, 796, 876, 963, 1060, 1166, 1282, 1411,
                                1552, 1707, 1878, 2066, 2272, 2499, 2749, 3024,
                                3327, 3660, 4026, 4428, 4871, 5358, 5894, 6484,
                                7132, 7845, 8630, 9493, 10442, 11487, 12635,
                                13899, 15289, 16818, 18500, 20350, 22385, 24623,
                                27086, 29794, 32767);


                inp := indata;
                Vvalpred := valpred;
                Vindex := stepsize_index;

                step := conv_std_logic_vector(stepsizetable(Vindex), 16);
                bufferstep := '0';

for each 4 bit value in memory
                -- step 1: get the delta value

                if (bufferstep = '1') then
                        delta := inputbuffer(3 downto 0);
                else
                        inputbuffer := inp (7 downto 0);
                        inp := to_stdlogicvector(to_bitvector(inp) srl 8);
                        delta := inputbuffer(7 downto 4);
                end if;

                bufferstep := NOT(bufferstep);

                -- step 2: find new index value

                Vindex := Vindex + indextable(conv_integer(delta));
                if (Vindex < 0) then Vindex := 0; end if;
                if (Vindex > 88) then Vindex := 88; end if;

                -- step 3: separate sign and magnitude

                if (delta(3) = '1') then
                        sign := '1';
                else
                        sign := '0';
                end if;

                delta := '0' & delta(2 downto 0);

                -- step 4: compute difference and new predicted value

                vpdiff := conv_integer("000" & step(15 downto 3));

                if (delta(2) = '1') then
                        vpdiff := vpdiff + conv_integer(step);
                end if;
                if (delta(1) = '1') then
                        vpdiff := vpdiff + conv_integer("0" & step(15 downto 1));
                end if;
                if (delta(0) = '1') then
                        vpdiff := vpdiff + conv_integer("00" & step(15 downto 2));
                end if;

                if (sign = '1') then
                        Vvalpred := Vvalpred - vpdiff;
                else
                        Vvalpred := Vvalpred + vpdiff;
                end if;

                -- step 5: clamp output value

                if (Vvalpred > 32767) then Vvalpred := 32767; end if;
                if (Vvalpred < -32768) then Vvalpred := -32768; end if;

                -- step 6: update step value
```

```
                        step := conv_std_logic_vector(stepsizetable(Vindex), 16);

                        -- step 7: output value
                        outp(15 downto 0) := conv_std_logic_vector(Vvalpred, 16);
                        outp := to_stdlogicvector(to_bitvector(outp) rol 16);
                        -- buffer and write out 16 bits at a time

                end loop;
                -- write any leftover 16-bit data (flush buffer)
                valpred <= Vvalpred;
                stepsize_index <= Vindex;
                outdata <= outp;

                end if;
        end process;
end Behavioral;
```

ENCODER BEHAVIORAL PSEUDO-VHDL

```
architecture Behavioral of encoder is
        signal valpred : integer;
        signal stepsize_index : integer;
begin
        process (clk)
                type indextabletype is array (15 downto 0) of integer;
                type stepsizetabletype is array (88 downto 0) of integer;
                variable indextable : indextabletype;
                variable stepsizetable : stepsizetabletype;

                variable inp : std_logic_vector(15999 downto 0);  -- port sending in the
wave
                variable outp : std_logic_vector(3999 downto 0);  -- port outputting the
ADPCM data
                variable val : std_logic_vector(15 downto 0);
                variable sign : bit;
                variable delta : std_logic_vector(3 downto 0);
                variable diff : integer;
                variable step : std_logic_vector(15 downto 0);
                variable Vvalpred : integer;
                variable vpdiff : integer;
                variable Vindex : integer;
                variable outputbuffer : std_logic_vector (7 downto 0);
                variable bufferstep : bit;

        begin

                if (clk'EVENT AND clk = '1' and encode = '1') then
                        outputbuffer := "00000000";
                        indextable := (-1, -1, -1, -1, 2, 4, 6, 8,
                                       -1, -1, -1, -1, 2, 4, 6, 8);

                        stepsizetable := (7, 8, 9, 10, 11, 12, 13, 14, 16, 17,
                                          19, 21, 23, 25, 28, 31, 34, 37, 41, 45,
                                          50, 55, 60, 66, 73, 80, 88, 97, 107, 118,
                                          130, 143, 157, 173, 190, 209, 230, 253, 279,
                                          307, 337, 371, 408, 449, 494, 544, 598, 658,
                                          724, 796, 876, 963, 1060, 1166, 1282, 1411,
                                          1552, 1707, 1878, 2066, 2272, 2499, 2749, 3024,
                                          3327, 3660, 4026, 4428, 4871, 5358, 5894, 6484,
                                          7132, 7845, 8630, 9493, 10442, 11487, 12635,
                                          13899, 15289, 16818, 18500, 20350, 22385, 24623,
                                          27086, 29794, 32767);
```

```
                inp := indata;
                Vvalpred := valpred;
                Vindex := stepsize_index;

                step := conv_std_logic_vector(stepsizetable(Vindex), 16);
                bufferstep := '1';

for each 16 bit value inp in memory

                val := inp(15 downto 0);
                inp := to_stdlogicvector(to_bitvector(inp) srl 16);

                -- step 1: compute difference with previous value

                diff := conv_integer(val) - Vvalpred;
                if (diff < 0)  then
                        sign := '1';
                else
                        sign := '0';
                end if;

                if (sign = '1') then
                        diff := 0 - diff ;
                end if;

                -- step 2: divide and clamp

                delta := "0000";
                vpdiff := conv_integer("0000" & step(15 downto 3));

                if (diff >= conv_integer(step)) then
                        delta := "0100";
                        diff := diff - conv_integer(step);
                        vpdiff := vpdiff + conv_integer(step);
                end if;
                --
                step := ("0" & step(15 downto 1));
                if (diff >= conv_integer(step)) then
                        delta(1) := '1';
                        diff := diff - conv_integer(step);
                        vpdiff := vpdiff + conv_integer(step);
                end if;
                --
                step := ("0" & step(15 downto 1));
                if (diff >= conv_integer(step)) then
                        delta(0) := '1';
                        vpdiff := vpdiff + conv_integer(step);
                end if;
                --

                --

                -- step 3: update previous value

                if (sign = '1') then
                        Vvalpred := Vvalpred - vpdiff;
                else
                        Vvalpred := Vvalpred + vpdiff;
                end if;

                -- step 4: clamp previous value to 16 bits

                if (Vvalpred > 32767) then
                        Vvalpred := 32767; end if;
                if (Vvalpred < -32768) then
                        Vvalpred := -32768; end if;

                -- step 5: assemble value, update index and step values
```

```
                        if(sign = '1') then
                                delta(3) := '1';
                        end if;

                        Vindex := Vindex + indextable(conv_integer(delta));
                        if (Vindex < 0) then
                                Vindex := 0; end if;
                        if (Vindex > 88) then
                                Vindex := 88; end if;
                        step := conv_std_logic_vector(stepsizetable(Vindex), 16);

                        -- step 6: output value

                        if (bufferstep = '1') then
                                outputbuffer := (delta(3 downto 0) & "0000");
                        else
                                outp(7 downto 0) := ("0000" & delta(3 downto 0)) or
                                outputbuffer;
                                outp := to_stdlogicvector(to_bitvector(outp) rol 8);
                        end if;
                                bufferstep := NOT(bufferstep);

                        end loop;

                        -- output last step if needed
                                if (bufferstep = '1') then
                                outp(7 downto 0) := outputbuffer;
                                end if;

                        valpred <= Vvalpred;
                        stepsize_index <= Vindex;
                        outdata <= outp;
                        -- write out 8 bits at a time (buffer fills 4 bits @time)
                end if;
        end process;
end Behavioral;
```

3. RTL Description

3.1 Encoder/Decoder MUX RTL

These are simple collections of multiplexers, which specify which components will be speaking to RAM at a given time. The decoder and encoder MUXes are controlled by the one-bit decoder and encoder signals, respectively. In the decoder MUX design, if the decode signal is high (the decode button is pressed) RAM switches from being in player/recorder control to being hooked up to the decoder unit. This functionality is identical for the encoder MUX. Please refer to the Appendix for the brief implementation of these components.

3.2 Encoder RTL

The encoder is partitioned into a controller and datapath, which share signals as shown in Figure 8. The control signals sent by the controller to the datapath are 35 bits, which include 25 bits driving register loads and arithmetic, and 10 bits used to drive selectors for five muxes.

The state diagram for the encoder controller is shown in Figure 9. The reason for the number of states is due to the number of dependent assignments between clock cycles. Although much of this could have been done in combinational logic,

we attempted to accommodate the 50 MHz clock requirement of the Queensland player [2]. Therefore, we decided on a tradeoff of more states (and more cycles) with a smaller cycle time. Only one register assignment is permitted per variable per state. The state numbers come from the ADPCM reference implementation [4]. Where we have broken up states into multiple cycles, we use letters to indicate sequential control transfer. (For instance, step 2 has been broken into st2a, st2b, st2c, and st2/3.)

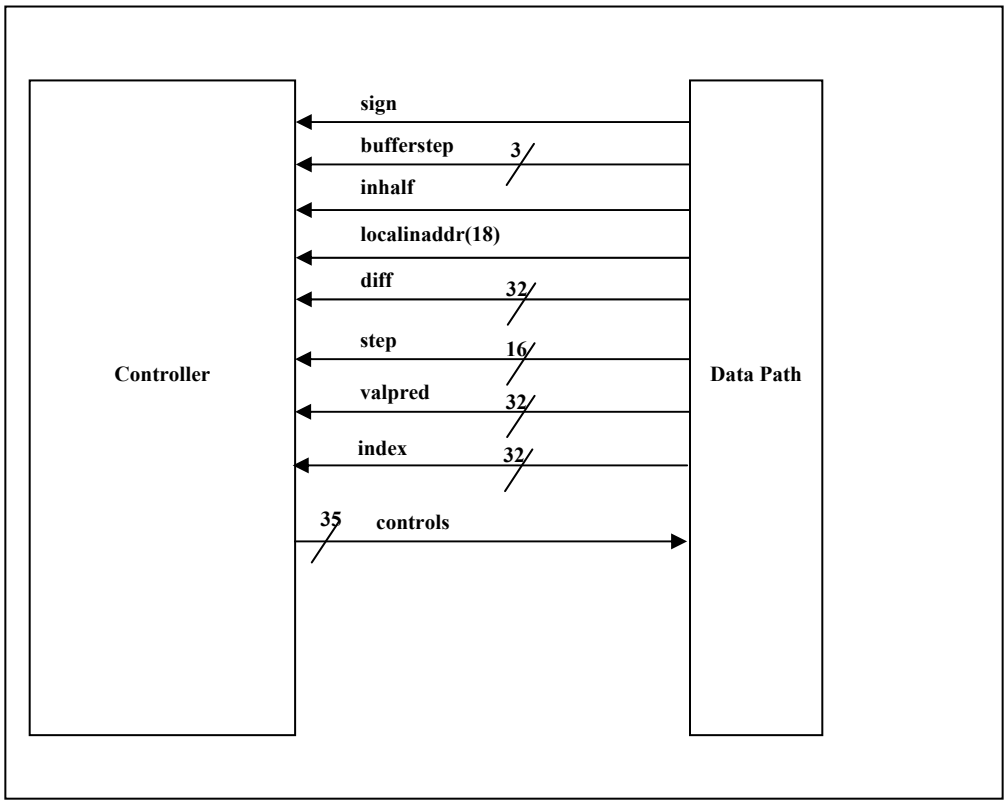

FIGURE 8. ADPCM Encoder RTL Partitioning

The encoder datapath, as shown in Figure 10 as well as the RTL implementation below, is comprised of several registers (index (integer), valpred (integer), inputbuffer (32 bit), index table (16 integers), step (16 bit), inhalf (1 bit), bufferstep (3 bits), localinaddress (19 bits), localoutaddress (19 bits), val (16 bits), sign (1 bit), diff (integer), delta (4 bits), and vpdiff (integer)). Additionally, two tables exist in the FPGA: index table and stepsize table. The total storage size is 3631 bits. Twelve muxes are needed to control which values get loaded into registers. Seven adders and three counters (one counter for each incrementing memory pointer, and one for the bufferstep) are needed to perform the RTL operations. In this design, some space is saved via the use of alignment to represent shifted values (rather than using actual shift operators).

The full RTL implementation of the ADPCM encoder follows:

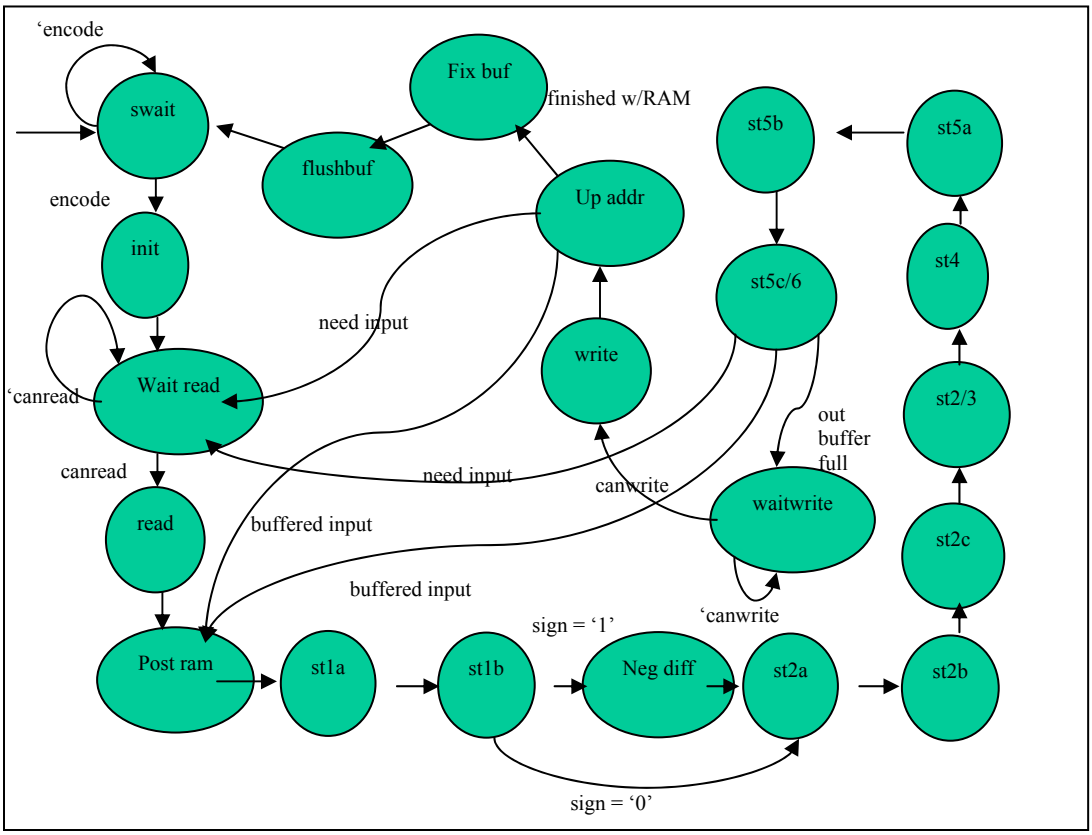

FIGURE 9. ADPCM Encoder Controller State Machine

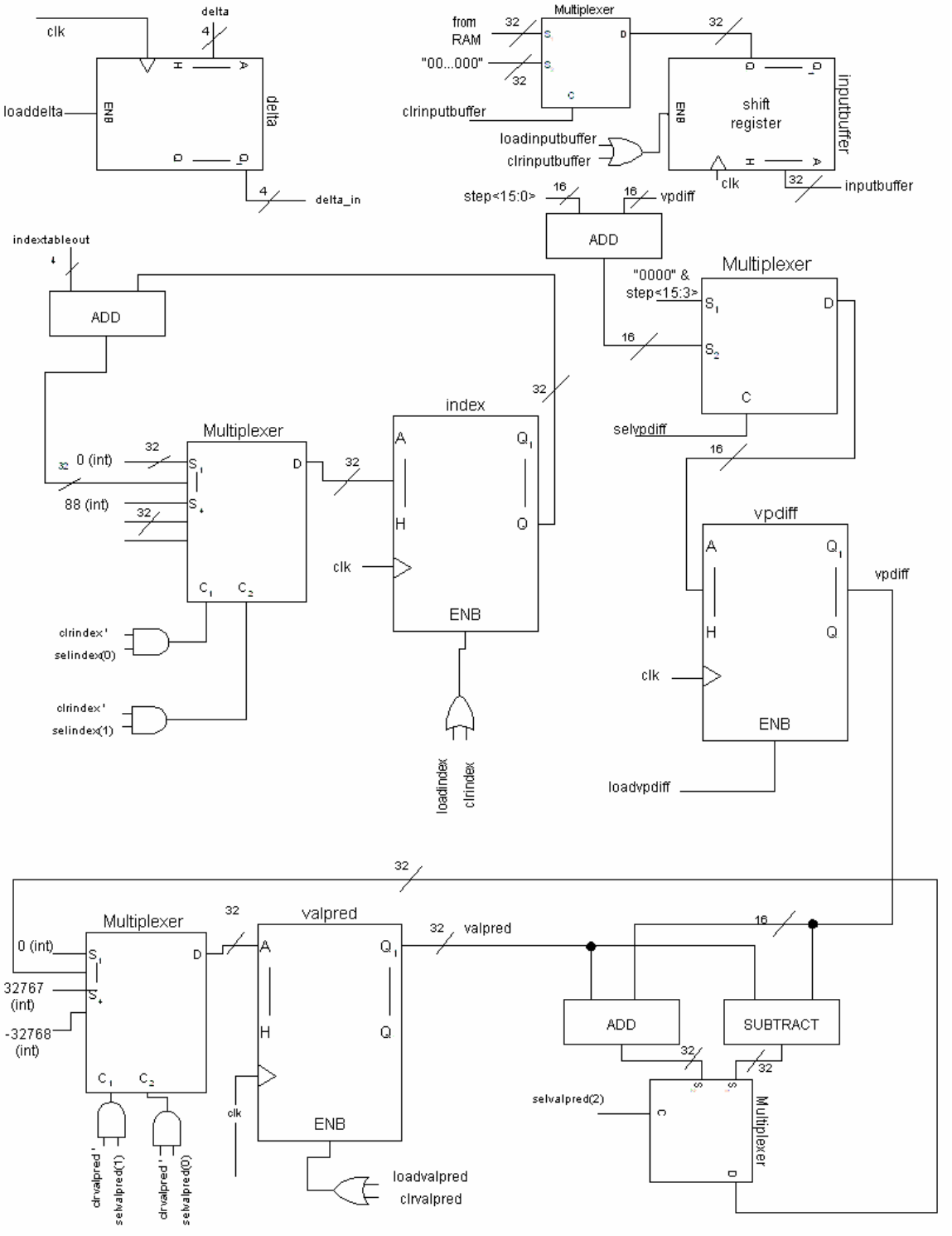

FIGURE 10a. ADPCM Encoder Datapath

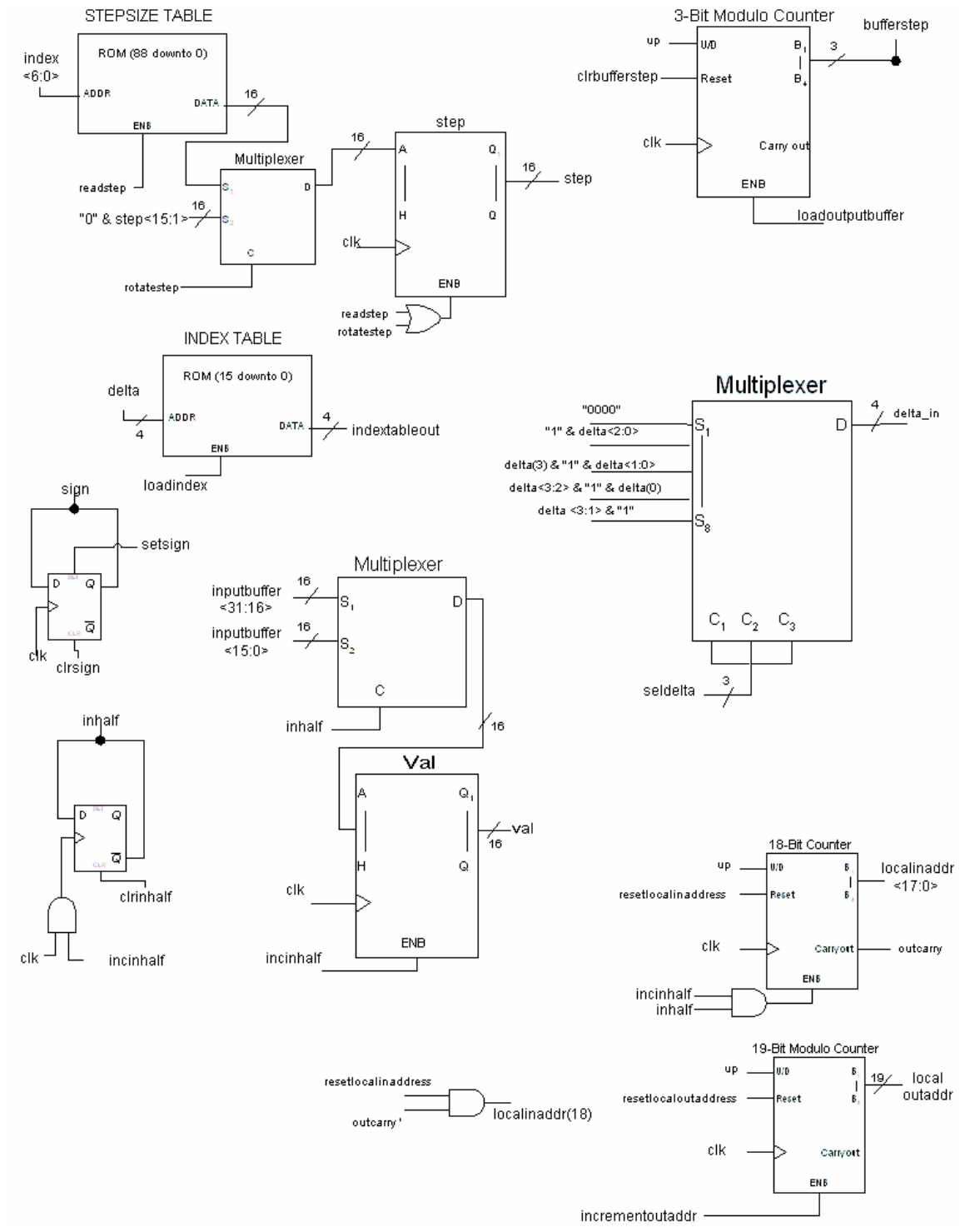

FIGURE 10b. ADPCM Encoder Datapath (cont)

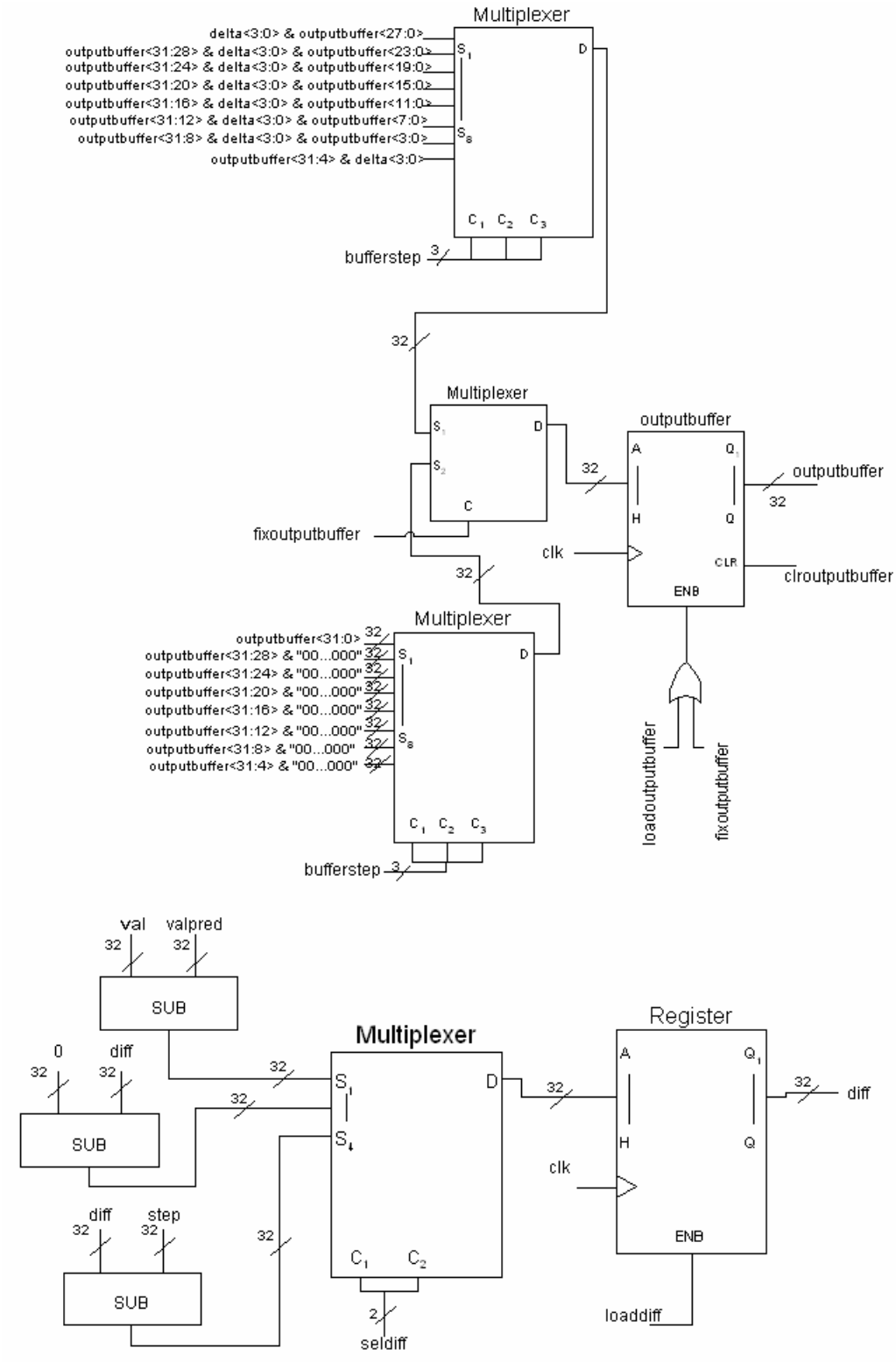

FIGURE 10c. ADPCM Encoder Datapath (cont)

```
architecture rtl of encoder is

        --------buffer signals-------------
        signal inputbuffer : std_logic_vector(31 downto 0);
        signal outputbuffer: std_logic_vector(31 downto 0);


        -------types------------------------

        type stateType is (swait, init, waitread, readstate, postram, step1a, step1b,
        negativediff, step2a, step2b, step2c, step2if3withstep3, step4, step5a, step5b,
        step5cwithstep6, waitwrite, stepwrite, upaddr, fixbuf, flushbuf);

        type indextabletype is array (15 downto 0) of integer;

        type stepsizetabletype is array (88 downto 0) of integer;


        type diffseltype is (valvalpred, negdiff, substep);
        type deltaseltype is (clear, three, two, one, zero);
        type vpdiffseltype is (shiftstep, incstep);
        type indexseltype is (inc, low, high);
        type valpredseltype is (high, low, plus, minus);
        ------ctrl signals-----------------

        signal state : stateType := swait;
        signal nextstate : stateType := swait;

        signal clrindex : bit;
        signal clrvalpred : bit;
        signal clrinputbuffer : bit;
        signal clroutputbuffer : bit;
        signal readstep : bit;
        signal clrbufferstep : bit;
        signal clrinhalf : bit;
        signal resetlocalinaddress : bit;
        signal resetlocaloutaddress : bit;
        signal loadinaddr : bit;
        signal loadinputbuffer : bit;
        signal incinhalf : bit;
        signal seldiff : diffseltype;
        signal loaddiff : bit;
        signal setsign : bit;
        signal clrsign : bit;
        signal seldelta : deltaseltype;
        signal loaddelta : bit;
        signal selvpdiff : vpdiffseltype;
        signal loadvpdiff : bit;
        signal selvalpred : valpredseltype;
        signal loadvalpred : bit;
        signal rotatestep : bit;
        signal selindex : indexseltype;
        signal loadindex : bit;
        signal loadoutputbuffer : bit;
        signal loadoutaddr : bit;
        signal loadoutdata : bit;
        signal incrementoutaddr : bit;
        signal fixoutputbuffer : bit;

        -------var signals-----------------

        signal indextable : indextabletype;
        signal stepsizetable : stepsizetabletype;

        signal sign : bit;
        signal delta : std_logic_vector (3 downto 0);
        signal step : std_logic_vector (15 downto 0);
        signal valpred : integer;
        signal index : integer;
        signal bufferstep : bit_vector(2 downto 0);
        signal val : std_logic_vector (15 downto 0);
```

```
        signal diff : integer;
        signal vpdiff : integer;
        signal localinaddr : std_logic_vector(18 downto 0);
        signal localoutaddr : std_logic_vector(18 downto 0);
        signal inhalf : bit;

begin

        indextable <= (-1, -1, -1, -1, 2, 4, 6, 8,
                       -1, -1, -1, -1, 2, 4, 6, 8);

        stepsizetable <= (7, 8, 9, 10, 11, 12, 13, 14, 16, 17,
                       19, 21, 23, 25, 28, 31, 34, 37, 41, 45,
                       50, 55, 60, 66, 73, 80, 88, 97, 107, 118,
                       130, 143, 157, 173, 190, 209, 230, 253, 279, 307,
                       337, 371, 408, 449, 494, 544, 598, 658, 724, 796,
                       876, 963, 1060, 1166, 1282, 1411, 1552, 1707, 1878, 2066,
                       2272, 2499, 2749, 3024, 3327, 3660, 4026, 4428, 4871, 5358,
                       5894, 6484, 7132, 7845, 8630, 9493, 10442, 11487, 12635, 13899,
                       15289, 16818, 18500, 20350, 22385, 24623, 27086, 29794, 32767);


    -------------------------------------------------------
    -----------change our state ----------------------------
    -------------------------------------------------------
        process (clk, reset, nextstate)
        begin
                if (reset = '0') then
                        state <= swait;
                elsif (clk'EVENT and clk = '1') then
                        state <= nextstate;
           end if;
        end process;

        -------------------------------------------------------
        ------------------next state----------------------------

        controller_ns : process (state)       -- fills in next state
        begin
                case state is
                        when swait => doingstuff <= '1';
                                        if (encode = '0') then
                                              nextstate <= swait;
                                        else
                                              nextstate <= init;
                                        end if;
                        when init => nextstate <= waitread;
                        when waitread => if (canread = '1') then
                                              nextstate <= readstate;
                                        end if;
                        when readstate => nextstate <= postram;
                        when postram => nextstate <= step1a;
                        when step1a => nextstate <= step1b;
                        when step1b => if (sign = '0') then
                                              nextstate <= step2a;
                                       else
                                              nextstate <= negativediff;
                                       end if;
                        when negativediff => nextstate <= step2a;
                        when step2a => nextstate <= step2b;
                        when step2b => nextstate <= step2c;
                        when step2c => nextstate <= step2if3withstep3;
                        when step2if3withstep3 => nextstate <= step4;
                        when step4 => nextstate <= step5a;
                        when step5a => nextstate <= step5b;
                        when step5b => nextstate <= step5cwithstep6;
                        when step5cwithstep6 =>
                                       if (bufferstep = "111") then
                                              nextstate <= waitwrite;
                                       else
                                              if (inhalf = '0') then
```

```
                                        nextstate <= waitread;
                                    else
                                        nextstate <= postram;
                                    end if;
                            end if;
                when waitwrite => if (canwrite = '1') then
                                        nextstate <= stepwrite;
                                   end if;
                when stepwrite => nextstate <= upaddr;
                when upaddr =>
                            if (localinaddr(18) = '0') then
                                    nextstate <= fixbuf;
                            else
                                    if (inhalf = '0') then
                                        nextstate <= waitread;
                                    else
                                        nextstate <= postram;
                                    end if;
                            end if;
                when fixbuf =>
                            if(bufferstep = "000") then
                                    nextstate <= swait;
                                    encodedone <= '1';
                            else
                                    nextstate <= flushbuf;
                            end if;
                when flushbuf => nextstate <= swait;
                                  encodedone <= '1';
        end case;
end process;


-------------------------------------------------------------
----------------------output function-----------------------

controller_op : process (state)

variable vclrindex : bit;
variable vclrvalpred : bit;
variable vclrinputbuffer : bit;
variable vclroutputbuffer : bit;
variable vreadstep : bit;
variable vclrbufferstep : bit;
variable vclrinhalf : bit;
variable vresetlocalinaddress : bit;
variable vresetlocaloutaddress : bit;
variable vloadinaddr : bit;
variable vloadinputbuffer : bit;
variable vincinhalf : bit;
variable vloaddiff : bit;
variable vsetsign : bit;
variable vclrsign : bit;
variable vloaddelta : bit;
variable vloadvpdiff : bit;
variable vloadvalpred : bit;
variable vrotatestep : bit;
variable vloadindex : bit;
variable vloadoutputbuffer : bit;
variable vloadoutaddr : bit;
variable vloadoutdata : bit;
variable vincrementoutaddr : bit;
variable vwrite : std_logic;
variable vread : std_logic;
variable vfixoutputbuffer : bit;

begin

        vwrite := '0';
        vread := '0';

        case state is
```

```
when swait =>
       vclrindex := '1';
       vclrvalpred := '1';

when init =>
       vclrinputbuffer := '1';
       vclroutputbuffer := '1';
       vreadstep := '1';
       vclrbufferstep := '1';
       vclrinhalf := '1';
       vresetlocalinaddress := '1';
       vresetlocaloutaddress := '1';

when waitread => -- nothing

when readstate =>
       vread := '1';
       vloadinaddr := '1';
       vloadinputbuffer := '1';

when postram => vincinhalf := '1';


when step1a =>
       seldiff <= valvalpred;
       vloaddiff := '1';


when step1b =>
       if (diff < 0) then
              vsetsign := '1';
       else
              vclrsign := '1';
       end if;

when negativediff =>
       seldiff <= negdiff;
       vloaddiff := '1';


when step2a =>
       seldelta <= clear;
       vloaddelta := '1';
       selvpdiff <= shiftstep;
       vloadvpdiff := '1';

when step2b =>
       if (diff >= conv_integer(step)) then
              seldelta <= two;
              vloaddelta := '1';
              seldiff <= substep;
              vloaddiff := '1';
              selvpdiff <= incstep;
              vloadvpdiff := '1';
       end if;
       vrotatestep := '1';

when step2c =>
       if (diff >= conv_integer(step)) then
              seldelta <= one;
              vloaddelta := '1';
              seldiff <= substep;
              vloaddiff := '1';
              selvpdiff <= incstep;
              vloadvpdiff := '1';
       end if;
       vrotatestep := '1';

when step2if3withstep3 =>
       if (diff >= conv_integer(step)) then
              seldelta <= zero;
```

```
                        vloaddelta := '1';
                        selvpdiff <= incstep;
                        vloadvpdiff := '1';
                end if;
                if(sign ='1') then
                        selvalpred <= minus;
                        vloadvalpred := '1';
                else
                        selvalpred <= plus;
                        vloadvalpred := '1';
                end if;

        when step4 =>
                if( valpred > 32767) then
                        selvalpred <= high;
                        vloadvalpred := '1';
                elsif (valpred < -32768) then
                        selvalpred <= low;
                        vloadvalpred := '1';
                end if;

        when step5a =>
                if(sign = '1') then
                        seldelta <= three;
                        vloaddelta := '1';
                end if;
                selindex <= inc;
                vloadindex := '1';

        when step5b =>
                if (index < 0) then
                        selindex <= low;
                        vloadindex := '1';
                elsif (index > 88) then
                        selindex <= high;
                        vloadindex := '1';
                end if;

        when step5cwithstep6 =>
                vreadstep := '1';
                vloadoutputbuffer := '1';

        when waitwrite => -- nothing

        when stepwrite =>
                vwrite := '1';
                vloadoutaddr := '1';
                vloadoutdata := '1';

        when upaddr => vincrementoutaddr := '1';

        when fixbuf => vfixoutputbuffer := '1';

        when flushbuf =>
                vwrite := '1';
                vloadoutaddr := '1';
                vloadoutdata := '1';

end case;

clrindex <= vclrindex;
clrvalpred <= vclrvalpred;
clrinputbuffer <= vclrinputbuffer;
clroutputbuffer <= vclroutputbuffer;
readstep <= vreadstep;
clrbufferstep <= vclrbufferstep;
clrinhalf <= vclrinhalf;
resetlocalinaddress <= vresetlocalinaddress;
resetlocaloutaddress <= vresetlocaloutaddress;
loadinaddr <= vloadinaddr;
loadinputbuffer <= vloadinputbuffer;
```

```
        incinhalf <= vincinhalf;
        loaddiff <= vloaddiff;
        setsign <= vsetsign;
        clrsign <= vclrsign;
        loaddelta <= vloaddelta;
        loadvpdiff <= vloadvpdiff;
        loadvalpred <= vloadvalpred;
        rotatestep <= vrotatestep;
        loadindex <= vloadindex;
        loadoutputbuffer <= vloadoutputbuffer;
        loadoutaddr <= vloadoutaddr;
        loadoutdata <= vloadoutdata;
        incrementoutaddr <= vincrementoutaddr;
        write <= vwrite;
        read <= vread;

end process;


-----------------------datapath---------------------------------
datapath : process (clk)
---
---
begin

        if (clk'EVENT and clk = '1') then
                if (clrindex = '1') then
                        index <= 0;
                end if;

                if (clrvalpred = '1') then
                        valpred <= 0;
                end if;

                if (clrinputbuffer = '1') then
                        inputbuffer <= (others => '0');
                end if;

                if (clroutputbuffer = '1') then
                        outputbuffer <= (others => '0');
                end if;

                if (readstep = '1') then
                        step <= conv_std_logic_vector(stepsizetable(index), 16);
                end if;

                if (clrbufferstep = '1') then
                        bufferstep <= "000";
                end if;

                if (clrinhalf = '0') then
                        inhalf <= '0';
                end if;

                if (resetlocalinaddress = '1') then
                                localinaddr(17 downto 0) <= (others => '0');
                                localinaddr(18) <= '1';
                end if;

                if (resetlocaloutaddress = '1') then
                                localoutaddr <= (others => '0');
                end if;

                if (loadinaddr = '1') then
                        inaddr <= localinaddr;
                end if;

                if (loadinputbuffer = '1') then
                        inputbuffer <= indata;
                end if;
```

```
if (incinhalf = '1') then
        case inhalf is
                when '0' =>
                        -- inhalf is high
                        -- have just read
                        inhalf <= '1';
                        val <= inputbuffer(31 downto 16);
                when '1' =>
                        -- input half is low
                        -- will read next time
                        localinaddr <= localinaddr + 1;
                        inhalf <= '0';
                        val <= inputbuffer(15 downto 0);
        end case;
end if;

if (setsign = '1') then
        sign <= '1';
end if;

if (clrsign = '1') then
        sign <= '0';
end if;

if (loaddiff = '1') then
        case seldiff is
                when valvalpred =>
                        diff <= conv_integer(val) - valpred;
                when negdiff =>
                        diff <= 0 - diff;
                when substep =>
                        diff <= diff - conv_integer(step);
        end case;
end if;

if (loaddelta = '1') then
        case seldelta is
                when clear =>  delta <= "0000";
                when three =>  delta(3) <= '1';
                when two   =>  delta(2) <= '1';
                when one   =>  delta(1) <= '1';
                when zero  =>  delta(0) <= '1';
        end case;
end if;

if (loadvpdiff = '1') then
        case selvpdiff is
        when shiftstep =>
                vpdiff <= conv_integer("0000" & step(15 downto 3));
        when incstep =>
                vpdiff <= vpdiff + conv_integer(step);
        end case;
end if;

if (rotatestep = '1') then
        step <= ("0" & step(15 downto 1));
end if;

if (loadvalpred = '1') then
        case selvalpred is
                when high =>   valpred <= 32767;
                when low =>    valpred <= -32768;
                when minus =>  valpred <= valpred - vpdiff;
                when plus =>   valpred <= valpred + vpdiff;
        end case;
end if;


if (loadindex = '1') then
        case selindex is
                when inc  =>
```

```
                                index <= index +
                                      indextable(conv_integer(delta));
                        when low  =>   index <= 0;
                        when high =>   index <= 88;
                end case;
        end if;

        if (loadoutputbuffer = '1') then
                case bufferstep is
                        when "000" =>
                                        bufferstep <= "001";
                                        outputbuffer(31 downto 28) <= delta;
                        when "001" =>
                                        bufferstep <= "010";
                                        outputbuffer(27 downto 24) <= delta;
                        when "010" =>
                                        bufferstep <= "011";
                                        outputbuffer(23 downto 20) <= delta;
                        when "011" =>
                                        bufferstep <= "100";
                                        outputbuffer(19 downto 16) <= delta;
                        when "100" =>
                                        bufferstep <= "101";
                                        outputbuffer(15 downto 12) <= delta;
                        when "101" =>
                                        bufferstep <= "110";
                                        outputbuffer(11 downto 8) <= delta;
                        when "110" =>
                                        bufferstep <= "111";
                                        outputbuffer(7 downto 4) <= delta;
                        when "111" =>
                                        bufferstep <= "000";
                                        outputbuffer(3 downto 0) <= delta;
                end case;
        end if;

        if (fixoutputbuffer = '1') then
                case bufferstep is
                when "000" => -- do nothing...no more to write
                when "001" =>
                        outputbuffer(27 downto 0) <= (others => '0');
                when "010" =>
                        outputbuffer(23 downto 0) <= (others => '0');
                when "011" =>
                        outputbuffer(19 downto 0) <= (others => '0');
                when "100" =>
                        outputbuffer(15 downto 0) <= (others => '0');
                when "101" =>
                        outputbuffer(11 downto 0) <= (others => '0');
                when "110" =>
                        outputbuffer(7 downto 0) <= (others => '0');
                when "111" =>
                        outputbuffer(3 downto 0) <= (others => '0');
                end case;
        end if;


        if (loadoutaddr = '1') then
                outaddr <= localoutaddr;
        end if;

        if (loadoutdata = '1') then
                outdata <= outputbuffer;
        end if;

        if (incrementoutaddr = '1') then
                localoutaddr <= localoutaddr + 1;
        end if;
    end if;
end process;
end rtl;
```

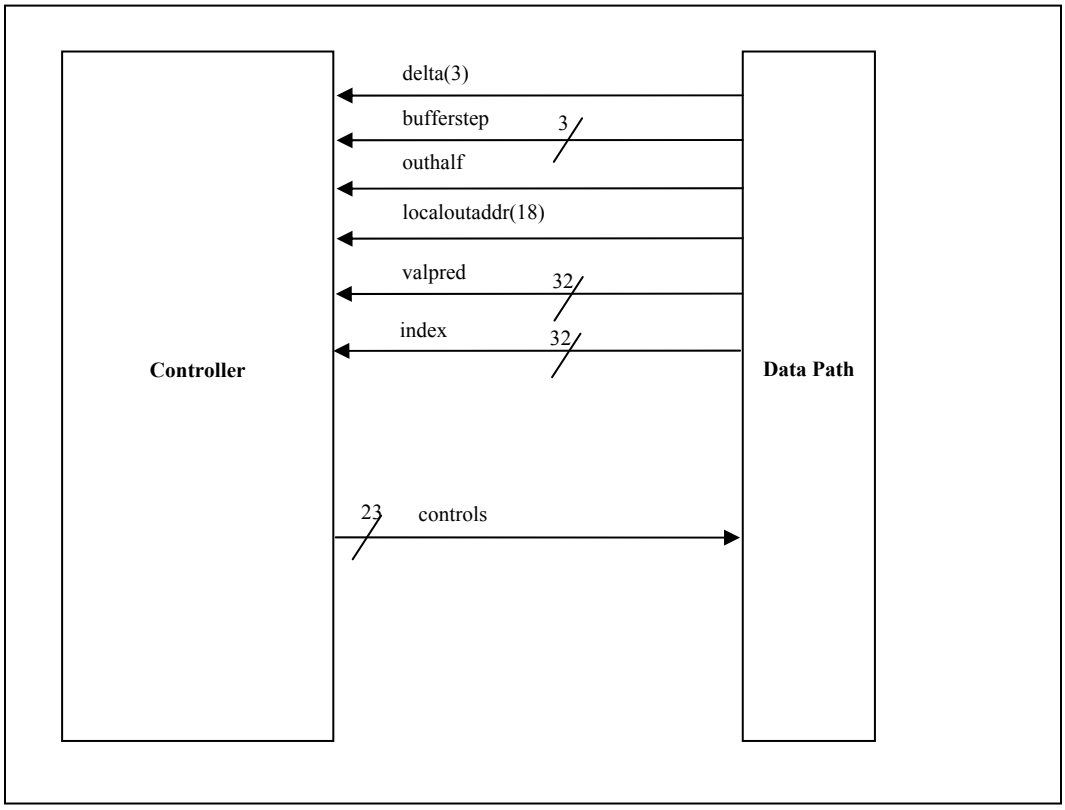

FIGURE 11. ADPCM Decoder RTL Partitioning

3.3 Decoder RTL

The decoder is likewise partitioned into a controller and datapath, which share signals as shown in Figure 11. The control signals sent by the controller to the datapath are 23 bits, which include 19 bits driving register loads and arithmetic, and 4 bits used to drive selectors for two (2-bit selected) muxes.

The state diagram for the encoder controller is shown in Figure 12. The decoder has vastly fewer states than the encoder, as fewer comparison-based computations are needed. Again, we leaned toward more clock cycles with a smaller cycle time. In this state machine, there was no need to further decompose steps from the reference implementation. In fact, in some cases (e.g. step5/6) states could be combined via extracted parallelism.

The decoder datapath, shown in Figure 13, is comprised mostly of storage, much like the encoder datapath. We store the stepsize and index tables (89 and 16 integers, respectively), and in registers we store index (integer), valpred (integer), input and output buffers (32 bits each), step (16 bits), bufferstep (3 bits), local input and output addresses (19 bits each), delta (4 bits), sign (1 bit), and outhalf (1 bit). The total in-FPGA storage size in this case is 3551 bits, dominated again by the table storage. Seven multiplexers are needed to alternate storage into these registers, and an additional three multiplexers are required for selection along the

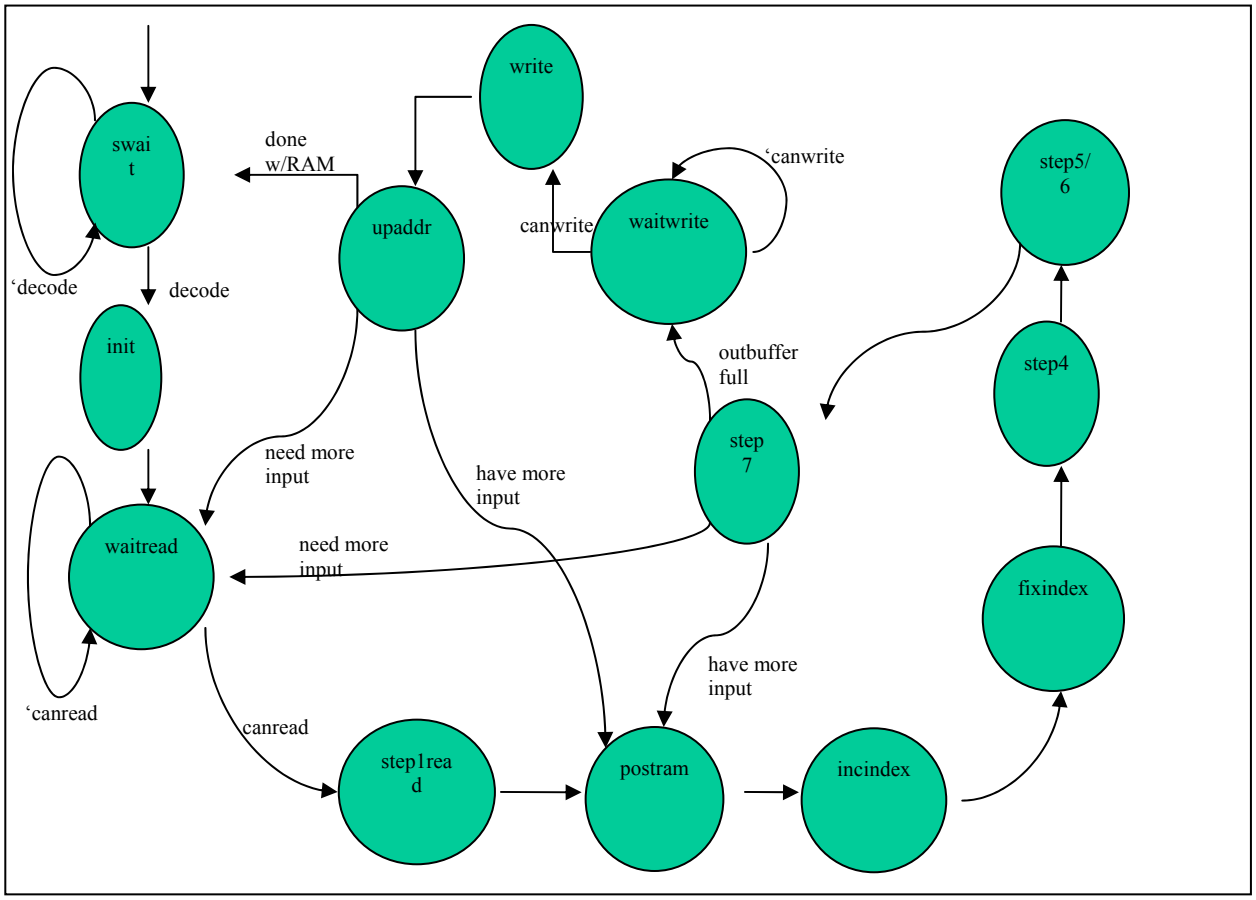

FIGURE 12. ADPCM Decoder Controller State Machine

computation path of valpred. Three counters are used to store values which are incremented by 1 during computation, and six adders are needed for arithmetic operations in the RTL implementation. Additionally, one comparator is used (to compare the bufferstep in order to conditionally increment the input address).

The full RTL implementation of the ADPCM encoder follows:

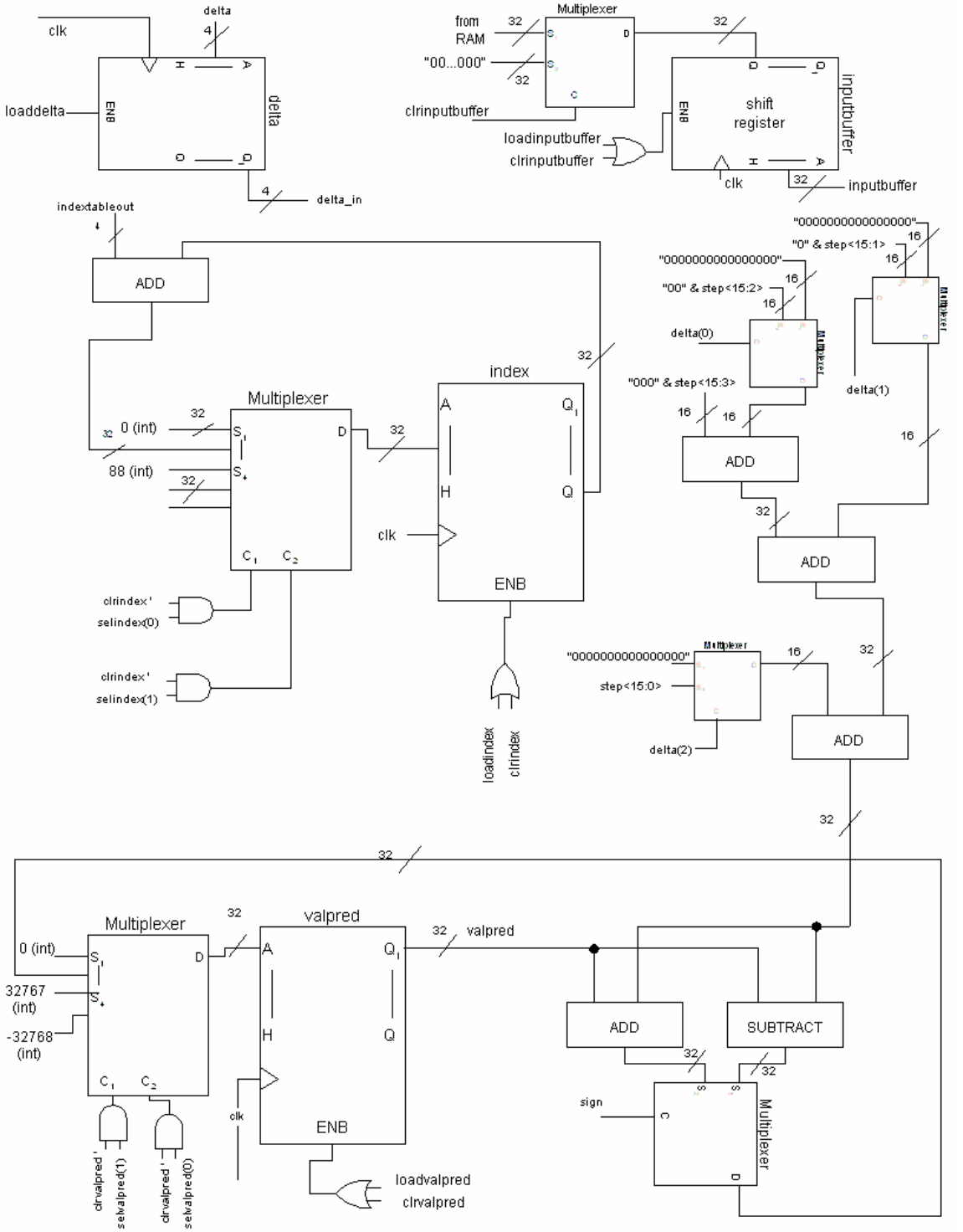

FIGURE 13a. ADPCM Decoder Datapath

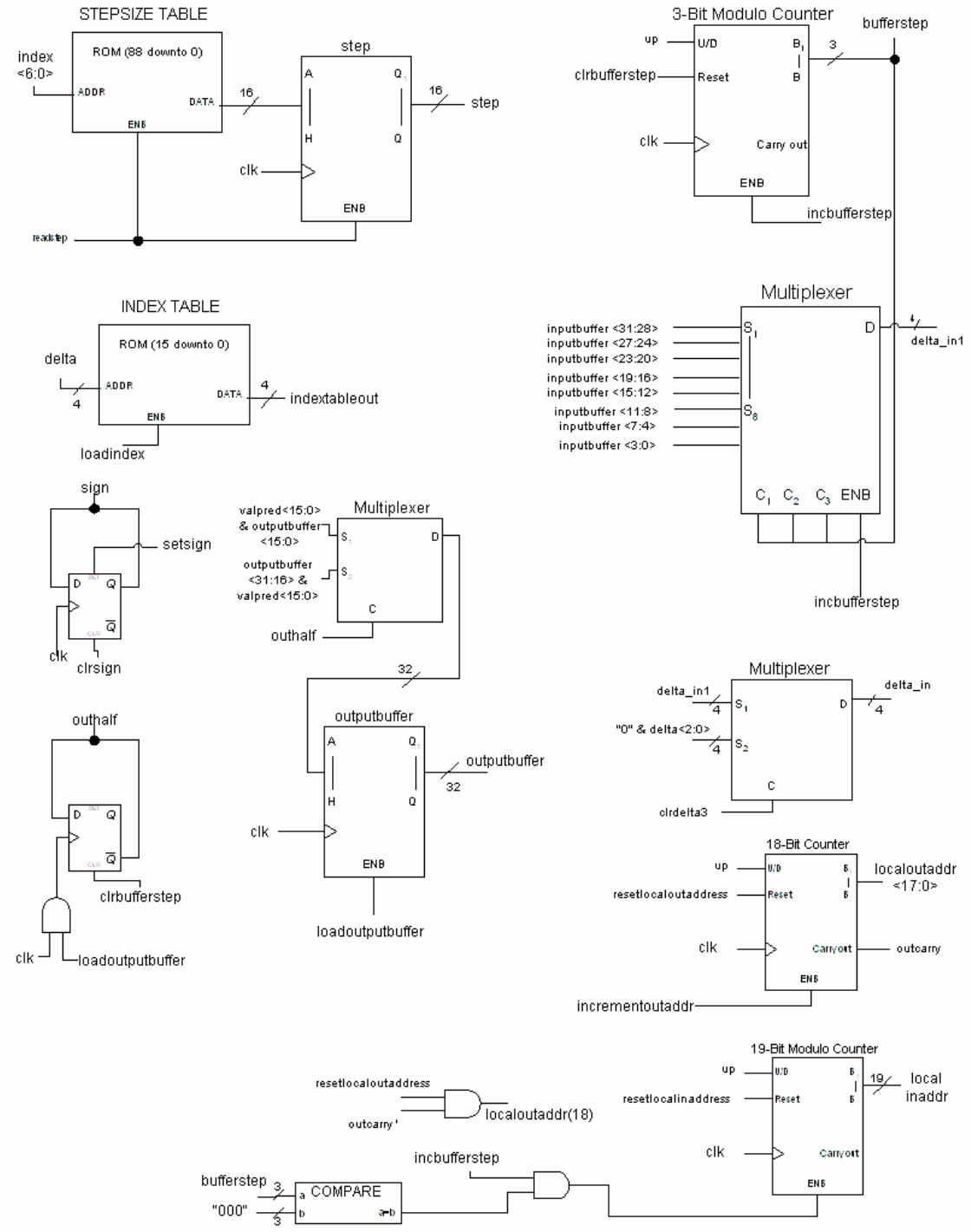

FIGURE 13b. ADPCM Decoder Datapath (cont)

```
architecture rtl of decoder is
        ----buffer signals---
        signal inputbuffer : std_logic_vector(31 downto 0);
        signal outputbuffer : std_logic_vector(31 downto 0);

        -------types---------
        type stateType is (swait, init, waitread, step1read, postram, incindex,
        fixindexstep3, step4, step56, step7, waitwrite, stepwrite, upaddr);
        type indextabletype is array (15 downto 0) of integer;
        type stepsizetabletype is array (88 downto 0) of integer;

        type indexseltype is (high, low, increment);
        type valpredseltype is (high, low, outoperation);

        ------ctrl signals---
        signal state : stateType := swait;
        signal nextstate : stateType := swait;

        signal clrindex : bit;
        signal clrvalpred : bit;
        signal clrinputbuffer : bit;
        signal readstep : bit;
        signal clrbufferstep : bit;
        signal resetlocalinaddress : bit;
        signal resetlocaloutaddress : bit;
        signal loadinaddr : bit;
        signal loadinputbuffer : bit;
        signal incbufferstep : bit;
        signal selindex : indexseltype;
        signal loadindex : bit;
        signal setsign : bit;
        signal clrsign : bit;
        signal clrdelta3 : bit;
        signal selvalpred : valpredseltype;
        signal loadvalpred : bit;
        signal loadoutputbuffer : bit;
        signal loadoutaddr : bit;
        signal loadoutdata : bit;
        signal incrementoutaddr : bit;

        -------var signals---
        signal indextable : indextabletype;
        signal stepsizetable : stepsizetabletype;
        signal sign : bit;
        signal delta : std_logic_vector (3 downto 0);
        signal step : std_logic_vector (15 downto 0);
        signal valpred : integer;
        signal index : integer;
        signal bufferstep : bit_vector(2 downto 0);
        signal localinaddr : std_logic_vector(18 downto 0);
        signal localoutaddr : std_logic_vector(18 downto 0);
        signal outhalf : bit;


begin

        indextable <= (-1, -1, -1, -1, 2, 4, 6, 8,
                       -1, -1, -1, -1, 2, 4, 6, 8);

        stepsizetable <= (7, 8, 9, 10, 11, 12, 13, 14, 16, 17,
                        19, 21, 23, 25, 28, 31, 34, 37, 41, 45,
                        50, 55, 60, 66, 73, 80, 88, 97, 107, 118,
                        130, 143, 157, 173, 190, 209, 230, 253, 279, 307,
                        337, 371, 408, 449, 494, 544, 598, 658, 724, 796,
                        876, 963, 1060, 1166, 1282, 1411, 1552, 1707, 1878, 2066,
                        2272, 2499, 2749, 3024, 3327, 3660, 4026, 4428, 4871, 5358,
                        5894, 6484, 7132, 7845, 8630, 9493, 10442, 11487, 12635, 13899,
                        15289, 16818, 18500, 20350, 22385, 24623, 27086, 29794, 32767);
```

```
    process (clk, reset, nextstate)
    begin
            if (reset = '0') then
                    state <= swait;
            elsif (clk'EVENT and clk = '1') then
            state <= nextstate;
            end if;
    end process;


------------------next state-----------------------------
    controller_ns : process (state)
    -- fills in next state

    begin
            decodedone <= '0';
            case state is
                    when swait => doingstuff <= '1';
                                  if (decode = '0') then
                                          nextstate <= swait;
                                  else
                                          nextstate <= init;
                                  end if;
                    when init => nextstate <= waitread;
                    when waitread =>
                                  if (canread = '1') then
                                          nextstate <= step1read;
                                  end if;
                    when step1read => nextstate <= postram;
                    when postram => nextstate <= incindex;
                    when incindex => nextstate <= fixindexstep3;
                    when fixindexstep3 => nextstate <= step4;
                    when step4 => nextstate <= step56;
                    when step56 => nextstate <= step7;
                    when step7 =>
                                  if(outhalf = '1') then
                                          if (bufferstep = "000") then
                                                  nextstate <= waitread;
                                          else
                                                  nextstate <= postram;
                                          end if;
                                  else
                                          nextstate <= waitwrite;
                                  end if;
                    when waitwrite => if(canwrite = '1') then
                                          nextstate <= stepwrite;
                                     end if;
                    when stepwrite => nextstate <= upaddr;
                    when upaddr =>    if (localoutaddr(18) = '0') then
                                          nextstate <= swait;
                                          decodedone <= '1';
                                      else
                                          if (bufferstep = "000") then
                                                  nextstate <= waitread;
                                          else
                                                  nextstate <= postram;
                                          end if;
                                      end if;
                    end case;
    end process;


    ---------------output function -------------
    controller_op : process (state)

    variable vclrindex : bit;
    variable vclrvalpred : bit;
    variable vclrinputbuffer : bit;
    variable vreadstep : bit;
    variable vclrbufferstep : bit;
    variable vresetlocalinaddress : bit;
```

```
variable vresetlocaloutaddress : bit;
variable vloadinaddr : bit;
variable vloadinputbuffer : bit;
variable vincbufferstep : bit;
variable vloadindex : bit;
variable vsetsign : bit;
variable vclrsign : bit;
variable vclrdelta3 : bit;
variable vloadvalpred : bit;
variable vloadoutputbuffer : bit;
variable vloadoutaddr : bit;
variable vloadoutdata : bit;
variable vincrementoutaddr : bit;

variable vread : std_logic;
variable vwrite : std_logic;

begin
		-- clear local vars by default
		vread := '0';
		vwrite := '0';
		vclrindex := '0';
		vclrvalpred := '0';
		vclrinputbuffer := '0';
		vreadstep := '0';
		vclrbufferstep := '0';
		vresetlocalinaddress := '0';
		vresetlocaloutaddress := '0';
		vloadinaddr := '0';
		vloadinputbuffer := '0';
		vincbufferstep := '0';
		vloadindex := '0';
		vsetsign := '0';
		vclrsign := '0';
		vclrdelta3 := '0';
		vloadvalpred := '0';
		vloadoutputbuffer := '0';
		vloadoutaddr := '0';
		vloadoutdata := '0';
		vincrementoutaddr := '0';

		case state is
			when swait => vclrindex := '1'; vclrvalpred := '1';

			when init =>
					vclrinputbuffer := '1';
					vreadstep := '1';
					vclrbufferstep := '1';
					vresetlocalinaddress := '1';
					vresetlocaloutaddress := '1';

			when waitread => -- nothing
			when step1read =>
					vread := '1';
					vloadinaddr := '1';
					vloadinputbuffer := '1';
			when postram => vincbufferstep := '1';

			when incindex => selindex <= increment;
					  vloadindex := '1';

			when fixindexstep3 =>
					if(index < 0) then
						vloadindex := '1';
						selindex <= low;
					elsif (index > 88) then
						vloadindex := '1';
						selindex <= high;
					end if;
					if (delta(3) = '1') then
						vsetsign := '1';
```

```
                                        else
                                                vclrsign := '1';
                                        end if;
                                        vclrdelta3 := '1';

                                when step4 =>
                                        vloadvalpred := '1';
                                        selvalpred <= outoperation;


                                when step56 => if (valpred > 32767) then
                                                selvalpred <= high;
                                                vloadvalpred := '1';
                                        elsif (valpred < -32768) then
                                                selvalpred <= low;
                                                vloadvalpred := '1';
                                        end if;
                                        vreadstep := '1';

                                when step7 =>  vloadoutputbuffer := '1';
                                when waitwrite => -- nothing
                                when stepwrite =>
                                        vwrite := '1';
                                        vloadoutdata := '1';
                                        vloadoutaddr := '1';

                                when upaddr => vincrementoutaddr := '1';
                        end case;

                        read <= vread;
                        write <= vwrite;
                        clrindex <= vclrindex;
                        clrvalpred <= vclrvalpred;
                        clrinputbuffer <= vclrinputbuffer;
                        readstep <= vreadstep;
                        clrbufferstep <= vclrbufferstep;
                        resetlocalinaddress <= vresetlocalinaddress;
                        resetlocaloutaddress <= vresetlocaloutaddress;
                        loadinaddr <= vloadinaddr;
                        loadinputbuffer <= vloadinputbuffer;
                        incbufferstep <= vincbufferstep;
                        loadindex <= vloadindex;
                        setsign <= vsetsign;
                        clrsign <= vclrsign;
                        clrdelta3 <= vclrdelta3;
                        loadvalpred <= vloadvalpred;
                        loadoutputbuffer <= vloadoutputbuffer;
                        loadoutaddr <= vloadoutaddr;
                        loadoutdata <= vloadoutdata;
                        incrementoutaddr <= vincrementoutaddr;

        end process;


-------------------------datapath ----------------------------

        datapath : process (clk)

        variable stepr3 : integer;
        variable stepr2 : integer;
        variable stepr1 : integer;
        variable stepr0 : integer;
        variable inc1 : integer;
        variable inc2 : integer;
        variable vpdiff : integer;
        variable outop : integer;
        variable valpredvectortemp : std_logic_vector(15 downto 0);

        begin

                if (clk'EVENT and clk = '1') then
```

```
if (clrindex = '1') then
       index <= 0;
end if;

if (clrvalpred = '1') then
       valpred <= 0;
end if;

if (clrinputbuffer = '1') then
       inputbuffer <= (others => '0');
end if;

if (readstep = '1') then
       step <= conv_std_logic_vector(stepsizetable(index), 16);

end if;

if (clrbufferstep = '1') then
       bufferstep <= "000";
end if;

if (resetlocalinaddress = '1') then
       localinaddr <= (others => '0');
end if;

if (resetlocaloutaddress = '1') then
       localoutaddr(17 downto 0) <= (others => '0');
       localoutaddr(18) <= '1';
end if;

if (loadinaddr = '1') then
       inaddr <= localinaddr;
end if;

if (loadinputbuffer = '1') then
       inputbuffer <= indata;
end if;

if (incbufferstep = '1') then
       case bufferstep is
              when "000" =>  bufferstep <= "001";
                             delta <= inputbuffer (31 downto 28);
                             localinaddr <= localinaddr + 1;
              when "001" =>  bufferstep <= "010";
                             delta <= inputbuffer (27 downto 24);

              when "010" =>  bufferstep <= "011";
                            delta <= inputbuffer (23 downto 20);

              when "011" =>  bufferstep <= "100";
                            delta <= inputbuffer (19 downto 16);


              when "100" =>  bufferstep <= "101";
                             delta <= inputbuffer (15 downto 12);

              when "101" =>  bufferstep <= "110";
                             delta <= inputbuffer (11 downto 8);

              when "110" =>  bufferstep <= "111";
                             delta <= inputbuffer (7 downto 4);

              when "111" =>  bufferstep <= "000";
                             delta <= inputbuffer (3 downto 0);

       end case;
end if;

if (loadindex <= '1') then
       case selindex is
       when increment =>
```

```
                index <= index + indextable(conv_integer(delta));
        when high =>
                index <= 88;
        when low =>
                index <= 0;
        end case;
end if;

if (setsign = '1') then
        sign <= '1';
end if;

if (clrsign = '1') then
        sign <= '0';
end if;

if (clrdelta3 = '1') then
        delta(3) <= '0';
end if;

if (loadvalpred = '1') then
        case selvalpred is
        when outoperation =>
                stepr3 := conv_integer("000" & step(15 downto 3));
                stepr2 := conv_integer("00" & step(15 downto 2));
                stepr1 := conv_integer("0" & step(15 downto 1));
                stepr0 := conv_integer(step);
                if (delta(0) = '1') then
                        inc1 := stepr3 + stepr2;
                else
                        inc1 := stepr3;
                end if;
                if (delta(1) = '1') then
                        inc2 := inc1 + stepr1;
                else
                        inc2 := inc1;
                end if;
                if (delta(2) = '1') then
                        vpdiff := inc2 + stepr0;
                else
                        vpdiff := inc2;
                end if;
                if (sign = '1') then
                        outop := valpred - vpdiff;
                else
                        outop := valpred + vpdiff;
                end if;
                valpred <= outop;

        when high =>
                valpred <= 32767;

        when low =>
                valpred <= -32768;

        end case;
end if;

if (loadoutputbuffer <= '1') then
        case outhalf is
        when '0' =>
                valpredvectortemp :=
                     conv_std_logic_vector(conv_signed(valpred, 16),
                     16);
                outputbuffer (31 downto 16) <=
                     valpredvectortemp(15 downto 0);
                outhalf <= '1';
        when '1' =>
                valpredvectortemp :=
                     conv_std_logic_vector(conv_signed(valpred, 16),
                     16);
```

```
                                        outputbuffer (15 downto 0) <=
                                             valpredvectortemp(15 downto 0);
                                        outhalf <= '0';
                                end case;
                        end if;

                        if (loadoutaddr = '1') then
                                outaddr <= localoutaddr;
                        end if;

                        if (loadoutdata = '1') then
                                outdata <= outputbuffer;
                        end if;

                        if (incrementoutaddr = '1') then
                                localoutaddr <= localoutaddr + 1;
                        end if;
                end if;
        end process;

end rtl;
```

4. VHDL Simulations

In this course, we first developed high-level behavioral VHDL (almost exactly the same as shown earlier) from the reference C implementation of ADPCM [4], which we obtained from the MediaBench benchmark suite [3]. We then brought this VHDL down to a low-level RTL abstraction, but found this too difficult to debug. We settled for an implementation in the middle, decomposed into states but with all datapath action included in the state transition function. We found this the easiest to debug and simulate. (Refer to the Appendix for this code.)

Our approach to this project was to debug our implementation on the chip much like we debug software. Unfortunately, chip-level debugging requires more time and more tools than we had at our disposal. For instance, we were unable to successfully write and read test data to/from RAM on many of the FPGAs. We subsequently found out that this was due to RAM failure on most of the XSV boards. We suggest that in the future, this class be simulation-only, as it is very difficult to bring a non-trivial FPGA project to completion in the last half of a given quarter. If FPGAs must be used, we propose that this course have a dedicated lab separate from CS152B students, with newer and higher-capacity boards.

To perform a functional validation of our design, we ran on simple test data. In the simulator, we repeated the 16-bit input data 0x00FF repeatedly (both as sound data for the encoder to read and ADPCM data for the decoder to read). We were able to demonstrate perfect control functionality, as the predictor values of the encoder and decoder converged on the repeated input (shown in Figures 16 and 19 respectively). For reference, we have included Figures 14 and 17 (which respectively show the initialization phases of the decoder and encoder), and Figures 15 and 18 (which respectively show the conclusion of the decoder and encoder, as they complete their read of and write to memory).

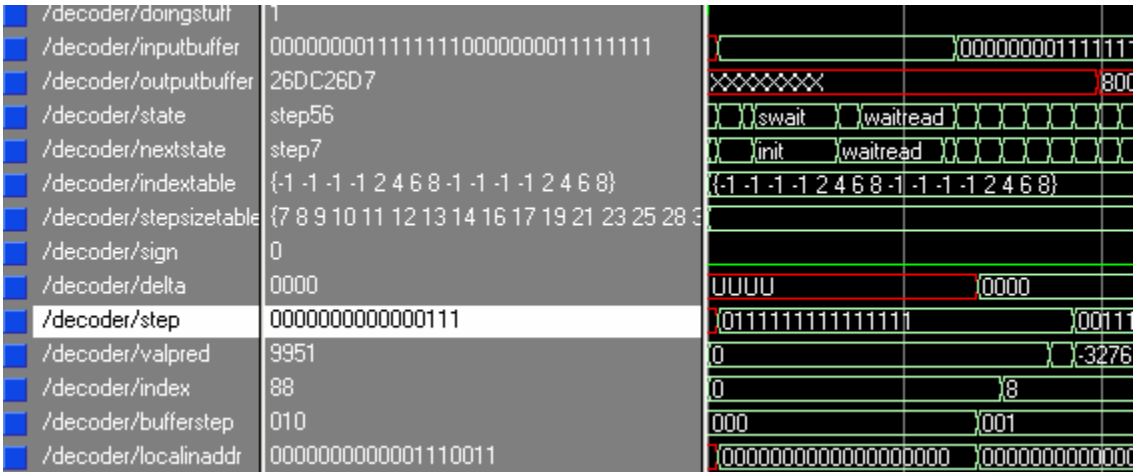

FIGURE 14. Decoder initialization

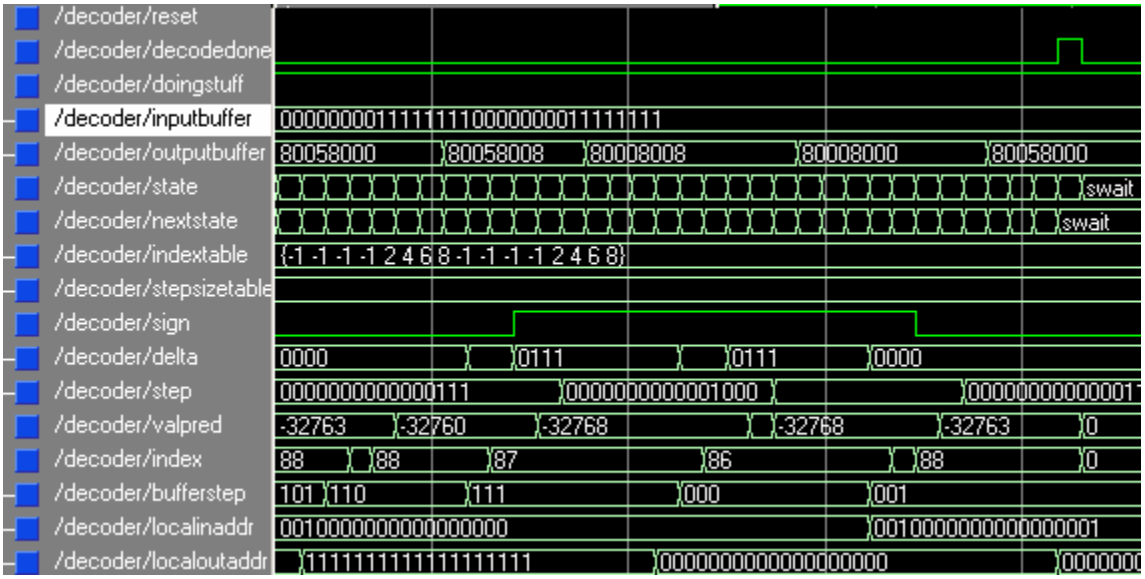

FIGURE 15. Decoder conclusion

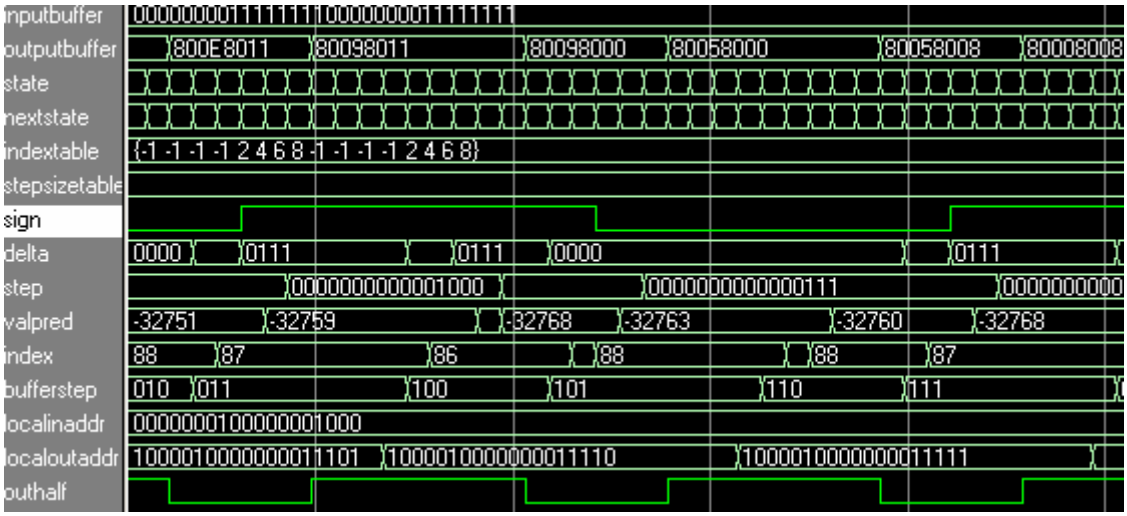

FIGURE 16. Decoder convergence

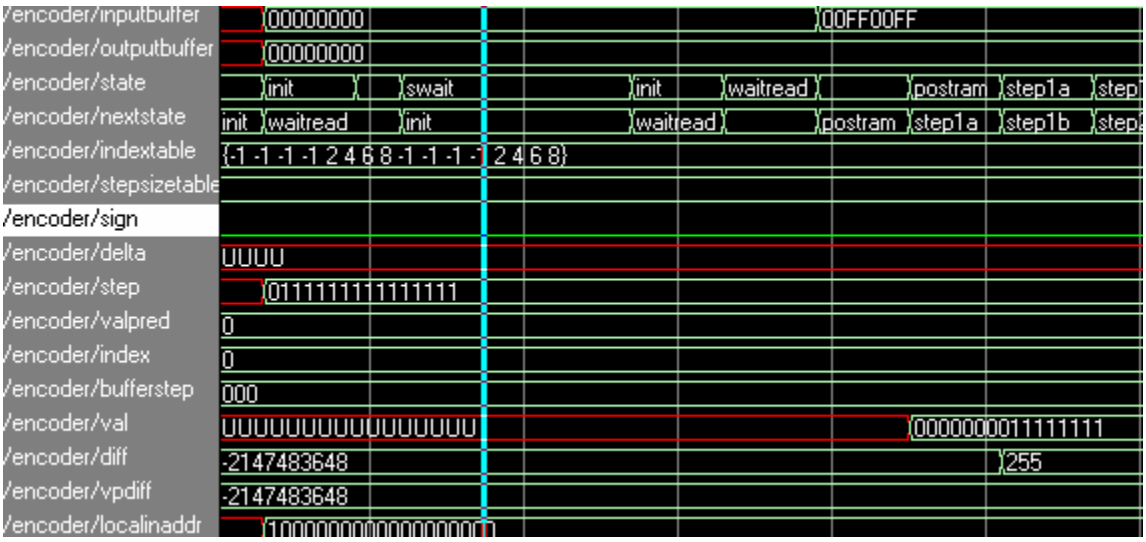

FIGURE 17. Encoder Initialization

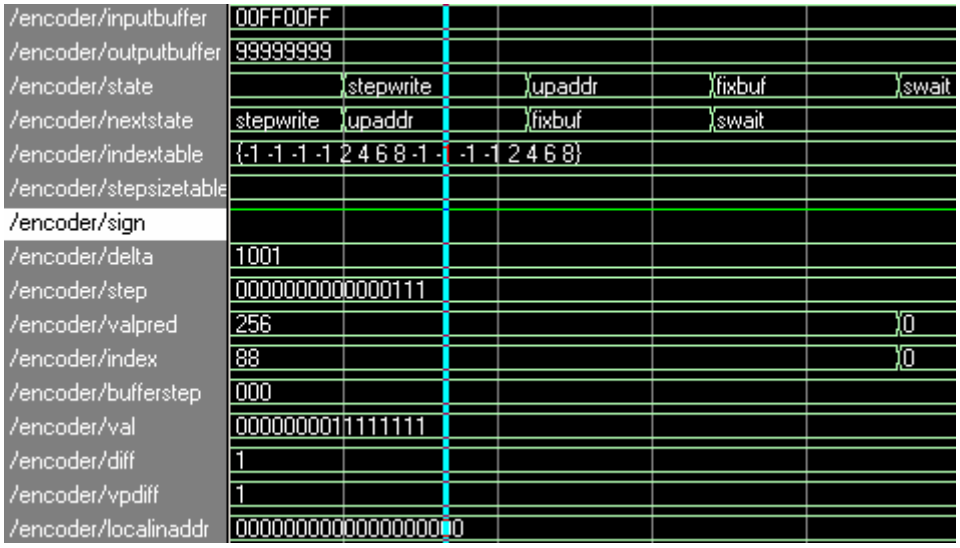

FIGURE 18. Encoder Conclusion

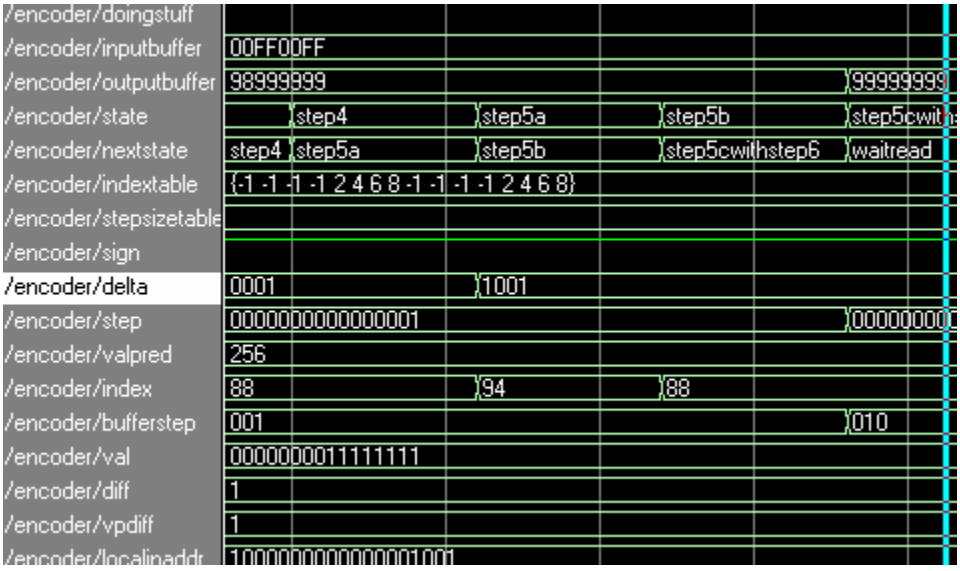

FIGURE 19. Encoder Convergence

We were able to synthesize the decoder in 615 out of 768 slices at 53.9 MHz. This took up roughly 80% of the FPGA. The encoder was synthesized in 766 out of 768 slices at 53.9 MHz. This took up 99% of the FPGA.

5. Conclusions

Overall, this course felt like boot camp for hardware designers. We became very familiar and comfortable with the VHDL language as well as the simulation tools. Moreover, we gained the vital skill to break down any behavioral system into a register transfer level system with unique data, as well as control subsystems. Aside from these obvious lessons taught by a hardware design course, we also learned how to be

decompose a sequential algorithmic specification into parallel hardware. We gained much experience scrutinizing designs in order to allow the maximum number of code to run in parallel, increasing the efficiency of our design in terms of speed and throughput. Furthermore, we became very familiar with an industry standard audio/data compression algorithm, used in many dialogue voice processing applications. Interestingly enough, we recently learned that the implementation of ADPCM on an FPGA has been the subject of multiple masters theses (e.g. [5]), indicating the non-triviality of this project. After the knowledge and experience gained during this project, we feel as if we are nearly prepared to succeed in an industry hardware design environment.

6. References

[1] IMA Digital Audio Focus and Technical Working Groups, "Recommended Practices for Enhancing Digital Audio Compatibility in Multimedia System: Revision 3.00," IMA Compatibility Proceedings, Vol. 2, Number 2, October 1992.

[2] P. Sutton, "XSV Board 1.0 – VHDL Interfaces and Example Designs: Audio Project," http://www.itee.uq.edu.au/~peters/xsvboard/audio/audio.htm

[3] C. Lee, M. Potkonjak and W. H. Maggione-Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems," in Proceedings of International Symposium on Microarchitecture, 1997.

[4] J. Jansen, ADPCM Reference Implementation, http://www.cwi.nl/ftp/audio/adpcm.zip

[5] Sampth Gangi, "ASIC Implementation of a custom ADPCM encoder," Master's Project, University of South Florida Electrical Engineering Department, December 2002.

Appendix

Attached is all source code that we used in simulation and synthesis: