REAL-TIME DIGITAL SYSTEMS DESIGN AND VERIFICATION WITH FPGAS ECE 387 – LECTURE 5

PROF. DAVID ZARETSKY

DAVID.ZARETSKY@NORTHWESTERN.EDU

AGENDA

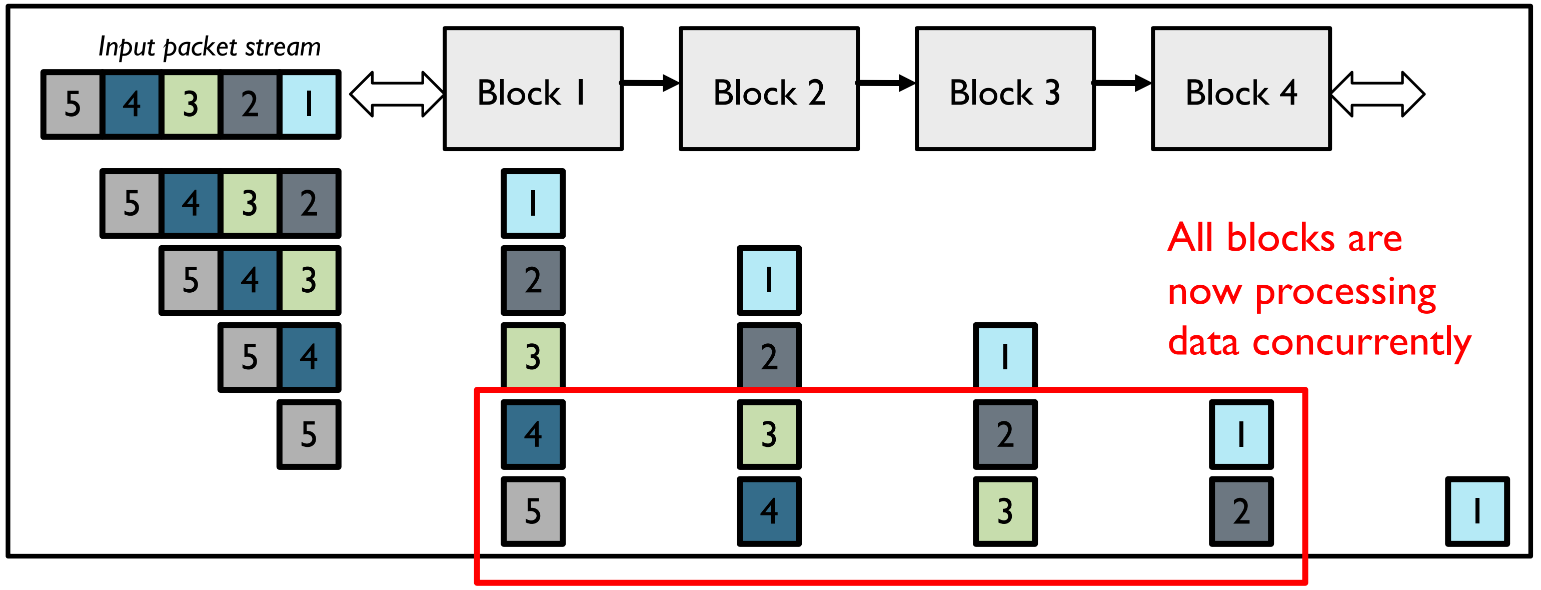
- Streaming Architectures
- FIFOs

STREAMING ARCHITECTURES IN FPGAS

- FPGAs are unique among programmable platforms in that streaming architectures can be implemented with elements of varying granularity.
- This enables different actors in a dataflow specification to be easily implemented at matched data rates, despite differing computational loads.

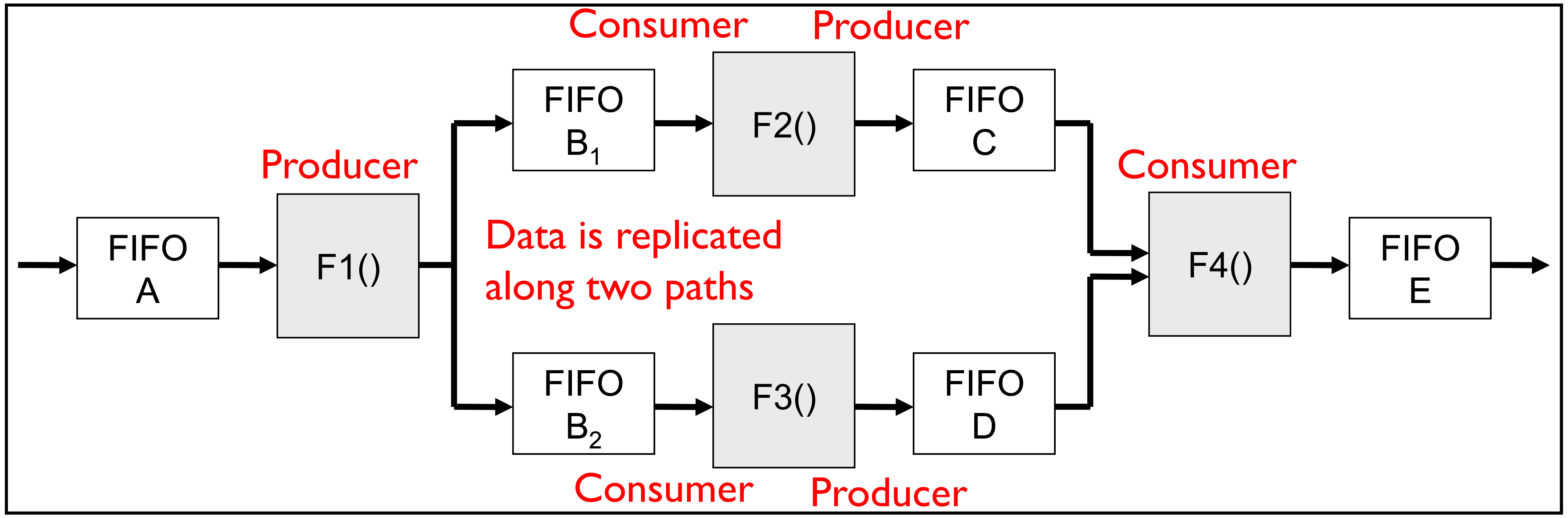
BLOCK-LEVEL PIPELINING

- Blocks or modules operate on data packets in chunks.
- Blocks are pipelined to allow faster processing
- There is an initial cost in propagating data through the pipeline until all blocks are working concurrently.

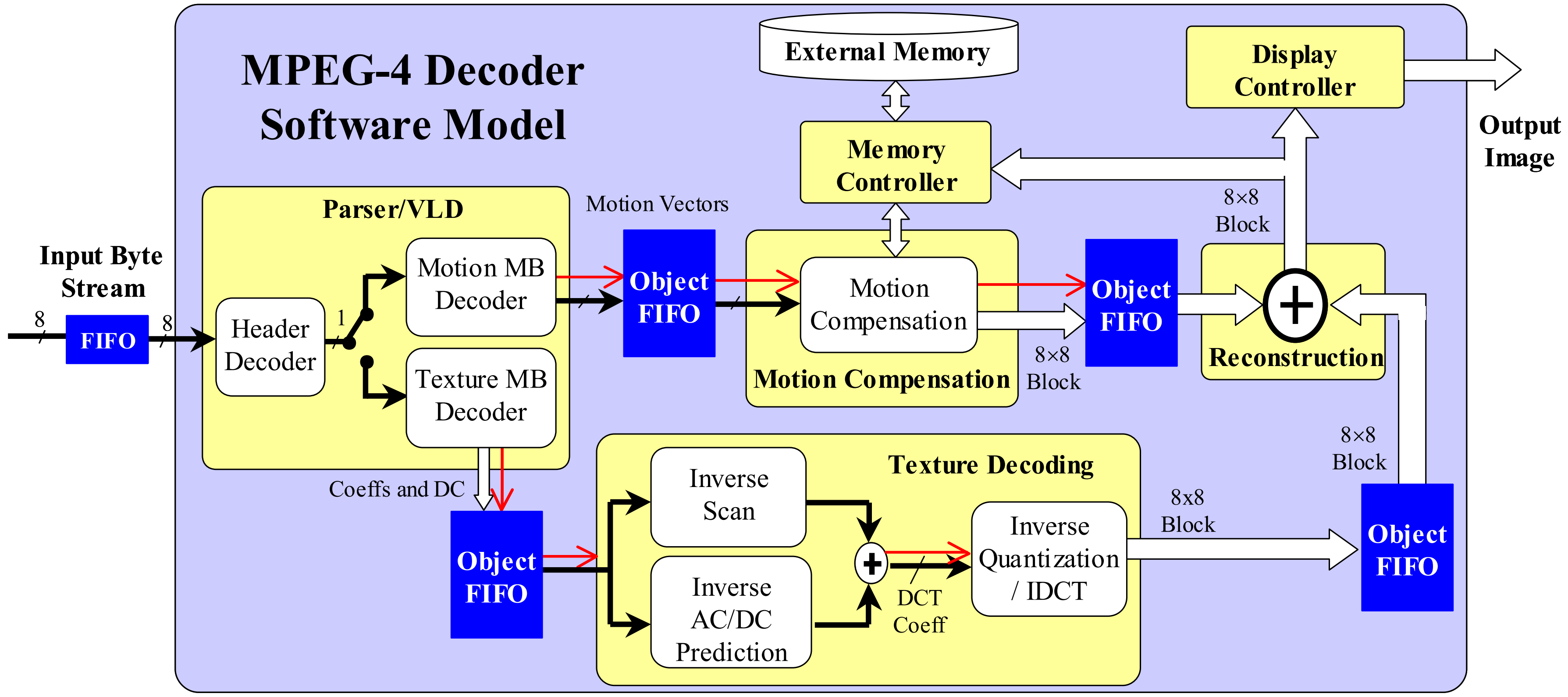


STREAMING ARCHITECTURES

- Using FIFOs, data is passed between function blocks in bursts, thus allowing blocks to be pipelined and processed concurrently.
- FIFOs can be used when the producer and consumer functions have the same data read and write schemes.

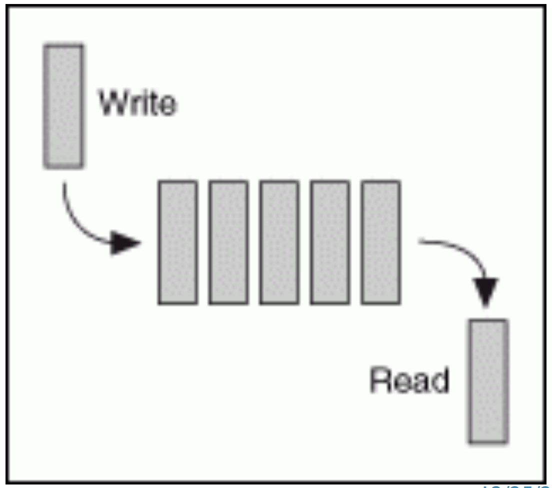


STREAMING MPEG-4 EXAMPLE



FIFOS

- A FIFO is a data structure that holds elements in the order they are received
- It provides access to those elements using a first-in, first-out access policy.
- Use FIFOs to transfer data in the following ways:
  - Between parallel processes within one clock domain
  - Across clock domains
  - Arrays passed between components (entities)
  - Between the host computer and the FPGA
  - Between peripherals and the FPGA



FIFO VS. BLOCK RAMS

- Block RAMs
  - Access data in any order
  - Requires one cycle latency to read cycles
  - Write on demand
  - Storage for full or chunk of data
  - Requires memory addressing
  - Unknown if RAM is empty/full
  - Multi-clock domains can be tricky
  - Useful for complex read/write patterns

- FIFOs
  - Access data sequentially only
  - Data read/write on demand
  - Storage for a small subset of data
  - Does not require complex addressing operations
  - Synchronization signals tell if FIFO is empty/full
  - Can pass data between multi-clock domains
  - Useful for sequential read/write patterns or if producer and consumer patterns are identical

FIFO MODEL

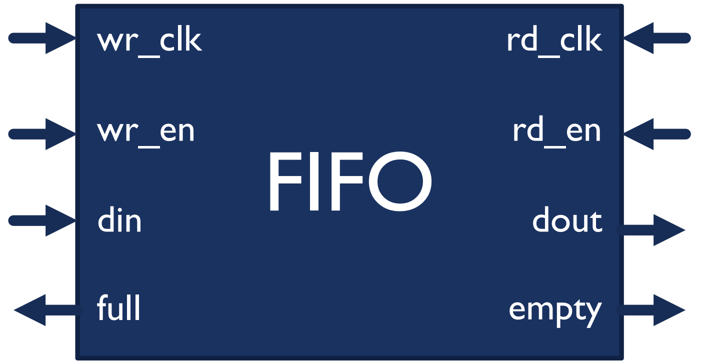
- FIFOs provide the glue-logic for streaming architectures
- Allow data to pass between different time domains
- Single-cycle read/write – data is instantaneously available
- Eliminates read and write addresses that consume a lot of resources
- If not built correctly, timing issues may emerge

```
module fifo #(
  parameter FIFO_DATA_WIDTH = 32,
  parameter FIFO_BUFFER_SIZE = 1024)
(
  input  logic reset,
  input  logic wr_clk,
  input  logic wr_en,
  input  logic [FIFO_DATA_WIDTH-1:0] din,
  output logic full,
  input  logic rd_clk,
  input  logic rd_en,
  output logic [FIFO_DATA_WIDTH-1:0] dout,
  output logic empty
);
```

FIFO READ & WRITE OPERATIONS

- FIFO Write Operations
  - If full == 0, data may be written to din
  - Data may be written instantaneously (same cycle)
  - Write enable writes the data and advances the address pointer for the next write operation
- FIFO Read Operations
  - If empty == 0, dout is valid for reading
  - Data may be acquired instantaneously (same cycle)
  - Read enable advances the address pointer for the next read operation

```
WRITE_STATE:
    if ( full == 1'b0 ) begin
        wr_en = 1'b1;
        din = data_in;
        next_state <= … ;
    end
```



```
READ_STATE:
    if ( empty == 1'b0 ) begin
        rd_en = 1'b1;
        data_out = dout;
        next_state = … ;
    end
```

FIFO WRITE

```
always_ff @(posedge wr_clk)
begin : p_write_buffer
    if ( (wr_en == 1'b1) && (full_t == 1'b0) ) begin
       fifo_buf[$unsigned(wr_addr[FIFO_ADDR_WIDTH-2:0])] <= din;
    end
end

always_ff @(posedge wr_clk, posedge reset)
begin : p_wr_addr
    if ( reset == 1'b1 )
       wr_addr <= ‘0;
    else
       wr_addr <= wr_addr_t;
end

assign wr_addr_t = (wr_en == 1'b1 && full_t == 1'b0) ? ($unsigned(wr_addr) + 'h1) : wr_addr;

assign full_t = (wr_addr[FIFO_ADDR_WIDTH-2:0] == rd_addr[FIFO_ADDR_WIDTH-2:0]) &&
                (wr_addr[FIFO_ADDR_WIDTH-1] != rd_addr[FIFO_ADDR_WIDTH-1]) ? 1'b1 : 1'b0;

assign full = full_t;
```

FIFO READ

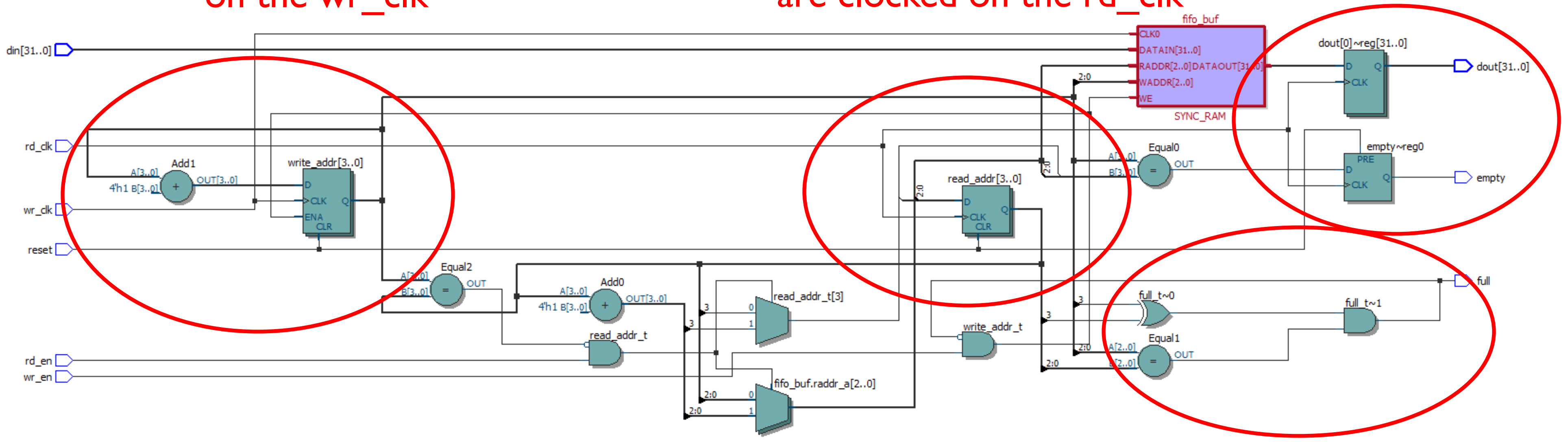
```
always_ff @(posedge rd_clk)
begin : p_rd_buffer
    dout <= fifo_buf[$unsigned(rd_addr_t[FIFO_ADDR_WIDTH-2:0])];
end

always_ff @(posedge rd_clk, posedge reset)
begin : p_rd_addr
    if ( reset == 1'b1 )
       rd_addr <= ‘0;
    else
       rd_addr <= rd_addr_t;
end

always_ff @(posedge rd_clk, posedge reset)
begin : p_empty
    if ( reset == 1'b1 )
        empty <= ‘0;
    else
        empty <= (wr_addr == rd_addr_t) ? 1'b1 : 1'b0;
end

assign rd_addr_t = (rd_en == 1'b1 && empty_t == 1'b0) ? ($unsigned(rd_addr) + 'h1) : rd_addr;

assign empty_t = (wr_addr == rd_addr) ? 1'b1 : 1'b0;
```

FIFO TECHNOLOGY VIEW

- Read and Write signals are each registered on different clock domains

Write address is clocked on the wr_clk

Read address, data and empty are clocked on the rd_clk



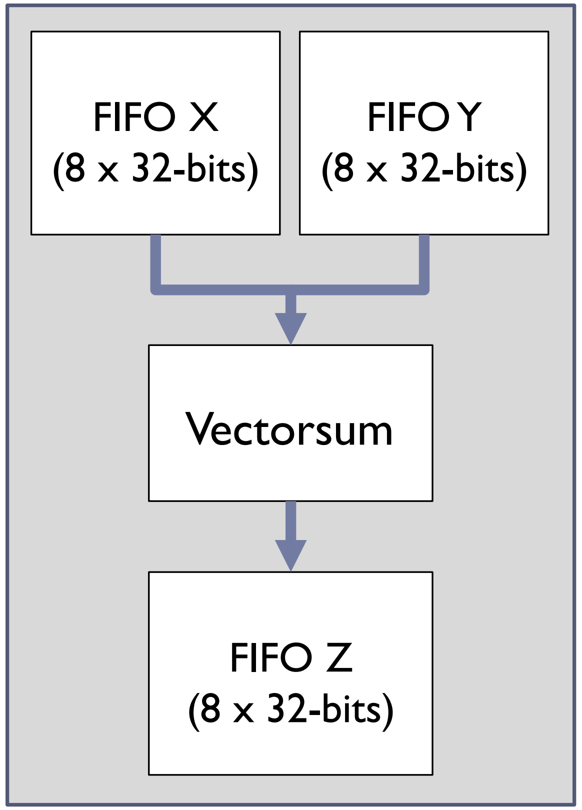
Full is based on clocked read and write addresses

STREAMING VECTORSUM

- Read arrays X and Y from FIFOs
- Compute vectorsum of n-elements
- Write output array Z to FIFO
- With FIFOs we only need to store a small amount of the local data
- Does not require memory addressing
- Does not require start/done synchronization signals

```
void vectorsum ( int *X, int *Y, int *Z, int n )
{
        for ( int i = 0; i < n; i++ )
        {
                Z[i] = X[i] + Y[i];
        }
}

int main()
{
        int X[64], Y[64], Z[64];
        for ( int i=0; i < 64; i++ ) {
                X[i] = rand();
                Y[i] = rand();
        }
         vectorsum( X, Y,  Z, 64 );
                  :
}
```

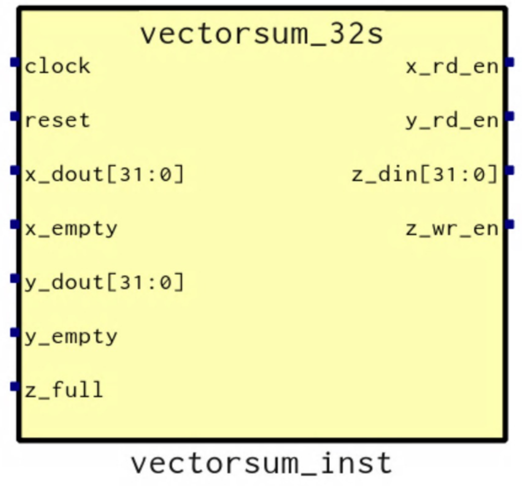
X and Y are written and read in the same order.



STREAMING VECTORSUM

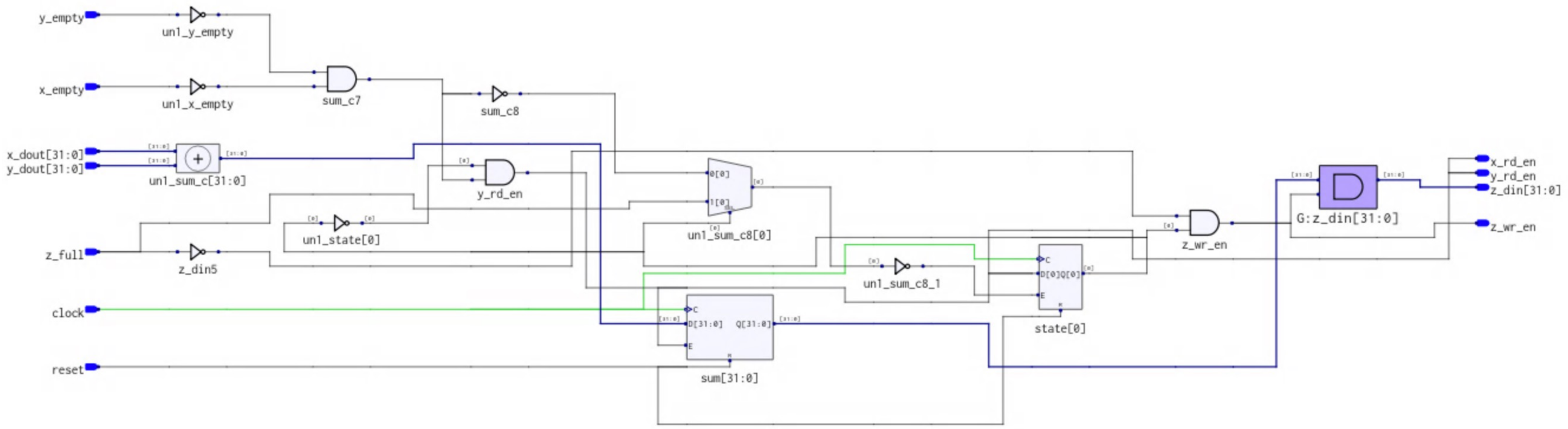
```
module vectorsum #(
    parameter DATA_WIDTH = 32
) (
  input logic clock,
  input logic reset,
  input logic [DATA_WIDTH-1:0] x_dout,
  input logic x_empty,
  output logic x_rd_en,
  input logic [DATA_WIDTH-1:0] y_dout,
  input logic y_empty,
  output logic y_rd_en,
  output logic [DATA_WIDTH-1:0] z_din,
  input logic z_full,
  output logic z_wr_en
);

typedef enum logic [0:0] {s0, s1} state_t;
state_t state, state_c;
logic [DATA_WIDTH-1:0] sum, sum_c;

always_ff @(posedge clock or posedge reset) begin
  if (reset) begin
    state <= s0;
    sum <= ‘0;
  end else begin
    state <= state_c;
    sum <= sum_c;
  end
end
```

```
always_comb begin
  z_din = 'b0;
  z_wr_en = 1'b0;
  x_rd_en = 1'b0;
  y_rd_en = 1'b0;
  sum_c = sum;
  state_c = state;

  case (state)
    s0: begin
      if (x_empty == 1'b0 && y_empty == 1'b0) begin
        sum_c = $signed(y_dout) + $signed(x_dout);
        x_rd_en = 1’b1;
        y_rd_en = 1'b1;
        state_c = s1;
      end
    end
    s1: begin
      if (z_full == 1'b0) begin
        z_din = sum;
        z_wr_en = 1'b1;
        state_c = s0;
      end
    end
  endcase
end

endmodule
```



VECTORSUM TECHNOLOGY VIEW

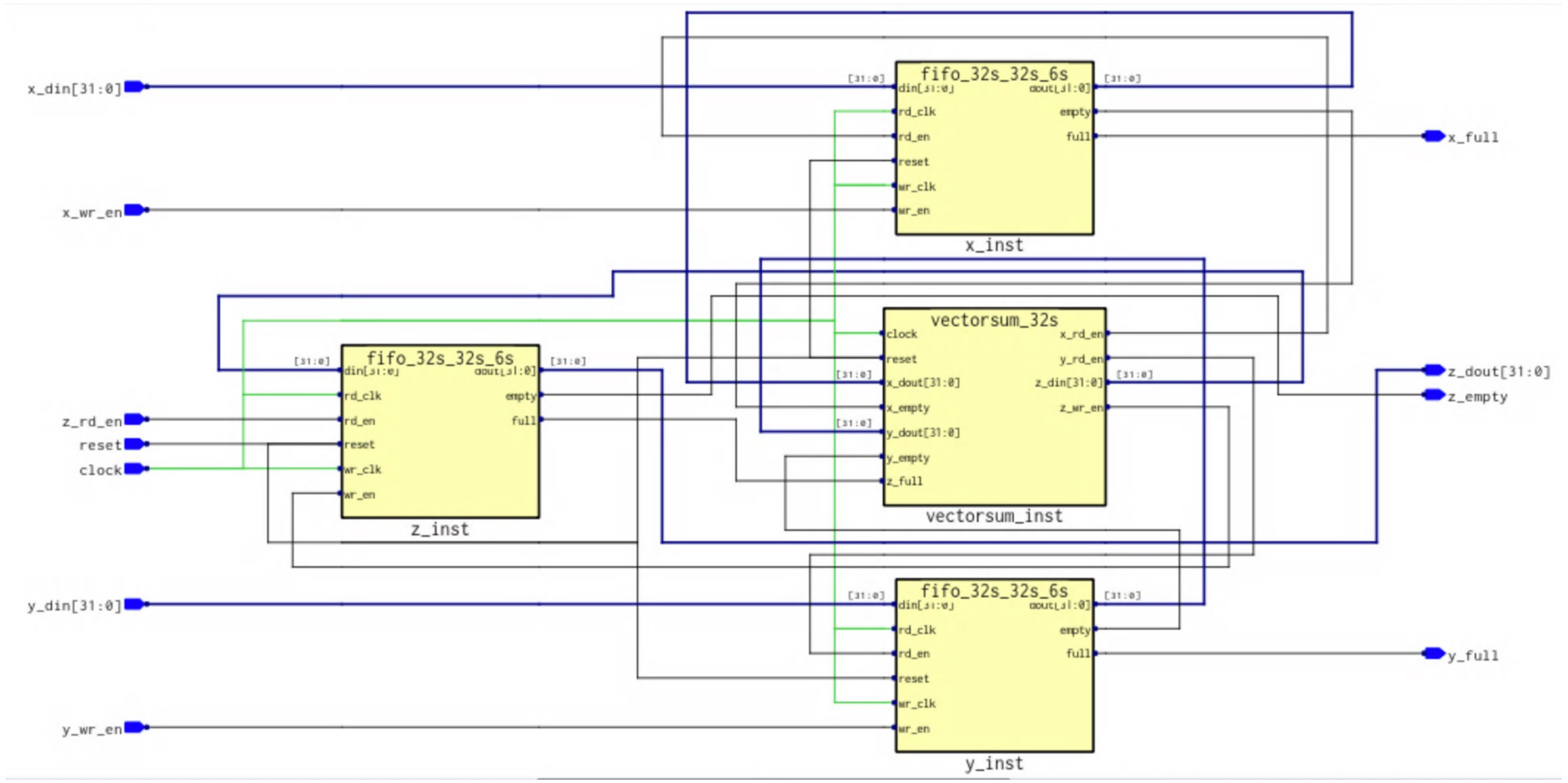
- Note: The vectorsum architecture is significantly simplified without the need for any memory address logic



VECTORSUM - WRAPPER

```
module vectorsum_top #(
  parameter DATA_WIDTH = 32,
  parameter FIFO_BUFFER_SIZE = 32)
(
  input logic clock,
  input logic reset,
  output logic x_full,
  input logic x_wr_en,
  input logic [DATA_WIDTH-1:0] x_din,
  output logic y_full,
  input logic y_wr_en,
  input logic [DATA_WIDTH-1:0] y_din,
  input logic z_rd_en,
  output logic z_empty,
  output logic [DATA_WIDTH-1:0] z_dout
);

logic [DATA_WIDTH-1:0] x_dout, y_dout, z_din;
logic x_empty, y_empty, z_full;
logic x_rd_en, y_rd_en, z_wr_en;

vectorsum #(
  .DATA_WIDTH(DATA_WIDTH)
) vectorsum_inst (
  .clock(clock),
  .reset(reset),
  .x_dout(x_dout),
  .x_rd_en(x_rd_en),
  .x_empty(x_empty),
  .y_dout(y_dout),
  .y_rd_en(y_rd_en),
  .y_empty(y_empty),
  .z_din(z_din),
  .z_full(z_full),
  .z_wr_en(z_wr_en)
);

fifo #(
  .FIFO_BUFFER_SIZE(FIFO_BUFFER_SIZE),
  .FIFO_DATA_WIDTH(DATA_WIDTH)
) x_inst (
  .reset(reset),
  .wr_clk(clock),
  .wr_en(x_wr_en),
  .din(x_din),
  .full(x_full),
  .rd_clk(clock),
  .rd_en(x_rd_en),
  .dout(x_dout),
  .empty(x_empty)
);

fifo #(
  .FIFO_BUFFER_SIZE(FIFO_BUFFER_SIZE),
  .FIFO_DATA_WIDTH(DATA_WIDTH)
) y_inst (
  .reset(reset),
  .wr_clk(clock),
  .wr_en(y_wr_en),
  .din(y_din),
  .full(y_full),
  .rd_clk(clock),
  .rd_en(y_rd_en),
  .dout(y_dout),
  .empty(y_empty)
);

fifo #(
  .FIFO_BUFFER_SIZE(FIFO_BUFFER_SIZE),
  .FIFO_DATA_WIDTH(DATA_WIDTH)
) z_inst (
  .reset(reset),
  .wr_clk(clock),
  .wr_en(z_wr_en),
  .din(z_din),
  .full(z_full),
  .rd_clk(clock),
  .rd_en(z_rd_en),
  .dout(z_dout),
  .empty(z_empty)
);

endmodule
```

VECTORSUM TOP-LEVEL TECHNOLOGY VIEW



TESTBENCH

```
`timescale 1 ns / 1 ns

module vectorsum_tb ();

localparam string X_NAME = "x.txt";
localparam string Y_NAME = "y.txt";
localparam string Z_NAME = "z.txt";
localparam DATA_WIDTH = 32;
localparam ADDR_WIDTH = 10;
localparam VECTOR_SIZE = 64;
localparam FIFO_BUFFER_SIZE = 8;
localparam CLOCK_PERIOD = 10;

logic clock = 1'b0;
logic reset = 1'b0;

logic [DATA_WIDTH-1:0] x_din;
logic x_wr_en, x_full;
logic [DATA_WIDTH-1:0] y_din;
logic y_wr_en, y_full;
logic [DATA_WIDTH-1:0] z_dout;
logic z_rd_en, z_empty;

logic x_write_done = '0;
logic y_write_done = '0;
logic z_read_done = '0;
integer z_errors = '0;
```

```
vectorsum_top #(
  .DATA_WIDTH(DATA_WIDTH),
  .FIFO_BUFFER_SIZE(FIFO_BUFFER_SIZE)
) vectorsum_top_inst (
  .clock(clock),
  .reset(reset),
  .x_full(x_full),
  .x_wr_en(x_wr_en),
  .x_din(x_din),
  .y_full(y_full),
  .y_wr_en(y_wr_en),
  .y_din(y_din),
  .z_rd_en(z_rd_en),
  .z_empty(z_empty),
  .z_dout(z_dout)
);

// clock process
always begin
  #(CLOCK_PERIOD/2) clock = 1'b1;
  #(CLOCK_PERIOD/2) clock = 1'b0;
end

// reset process
initial begin
  #(CLOCK_PERIOD) reset = 1'b1;
  #(CLOCK_PERIOD) reset = 1'b0;
end
```

```
// tb process
initial begin
  time start_time, end_time;

  @(negedge reset);
  @(posedge clock);
  start_time = $time;
  $display("@ %0t: Beginning simulation...", start_time);

  wait(z_read_done);

  end_time = $time;
  $display("@ %0t: Simulation completed.", end_time);
  $display("Total simulation cycle count: %0d",
      (end_time-start_time)/CLOCK_PERIOD);
  $display("Total error count: %0d", z_errors);

  $stop;
end
```

TESTBENCH: READ FILE INTO FIFO

```
initial begin : x_write
  integer fd, x=0, count;
  x_din = ‘0;
  x_wr_en = 1'b0;

  @(negedge reset);
  $display("@ %0t: Loading file %s...", $time, X_NAME);
  fd = $fopen(X_NAME, "r");

  while ( x < VECTOR_SIZE ) begin
    @(negedge clock);
    x_wr_en = 1'b0;
    if (x_full == 1'b0) begin
      count = $fscanf(fd, "%h", x_din);
      x_wr_en = 1'b1;
      x++;
    end
  end

  @(posedge clock);
  x_wr_en = 1'b0;
  $fclose(fd);
  x_write_done = 1'b1;
end
```

```
initial begin : y_write
  integer fd, y=0, count;
  y_din = ‘0;
  y_wr_en = 1'b0;
  @(negedge reset);
  $display("@ %0t: Loading file %s...", $time, Y_NAME);
  fd = $fopen(Y_NAME, "r");

  while ( y < VECTOR_SIZE ) begin
    @(negedge clock);
    y_wr_en = 1'b0;
    if (y_full == 1'b0) begin
       count = $fscanf(fd, "%h", y_din);
       y_wr_en = 1'b1;
       y++;
    end
  end

  @(posedge clock);
  y_wr_en = 1'b0;
  $fclose(fd);
  y_write_done = 1'b1;
end
```

TESTBENCH: READ FIFO AND COMPARE OUTPUT

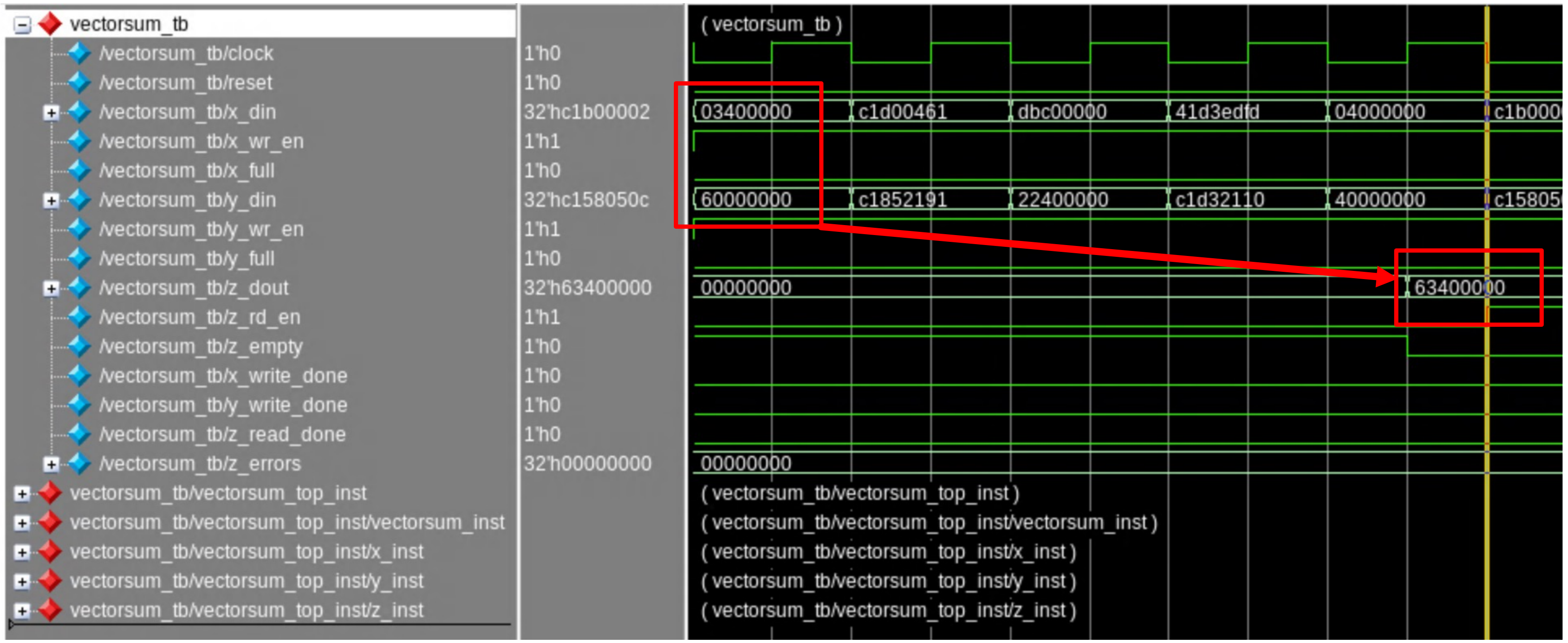
```
initial begin : z_write
   integer fd, z=0, count;
   logic [DATA_WIDTH-1:0] z_data_cmp;
   z_rd_en = 1'b0;
   z_data_cmp = '0;

   @(negedge reset);
   @(negedge clock);

   $display("@ %0t: Comparing file %s...", $time, Z_NAME);
   fd = $fopen(Z_NAME, "r");

   while ( z < VECTOR_SIZE ) begin
      @(negedge clock);
      z_rd_en = 1'b0;
      if (z_empty == 1'b0) begin
         z_rd_en = 1'b1;
         count = $fscanf(fd, "%h", z_data_cmp);
         if (z_dout != z_data_cmp) begin
            z_errors++;
            $display("@ %0t: %s(%0d): ERROR: %h != %h at address 0x%h.", $time, Z_NAME, z+1, z_dout, z_data_cmp, z);
         end
         z++;
      end
   end
   z_rd_en = 1'b0;
   z_read_done = 1'b1;
   $fclose(fd);
end
```

SIMULATION SCRIPT

```
setenv LMC_TIMEUNIT -9
vlib work
vmap work work
vlog -work work "../sv/fifo.sv"
vlog -work work "../sv/vectorsum.sv"
vlog -work work "../sv/vectorsum_top.sv"
vlog -work work "../sv/vectorsum_tb.sv"
vsim -classdebug -voptargs=+acc +notimingchecks -L work work.vectorsum_tb -wlf vectorsum.wlf

add wave -noupdate -group vectorsum_tb
add wave -noupdate -group vectorsum_tb -radix hexadecimal /vectorsum_tb/*
add wave -noupdate -group vectorsum_tb/vectorsum_top_inst
add wave -noupdate -group vectorsum_tb/vectorsum_top_inst -radix hexadecimal /vectorsum_tb/vectorsum_top_inst/*
add wave -noupdate -group vectorsum_tb/vectorsum_top_inst/vectorsum_inst
add wave -noupdate -group vectorsum_tb/vectorsum_top_inst/vectorsum_inst -radix hexadecimal /vectorsum_tb/vectorsum_top_inst/vectorsum_inst/*
add wave -noupdate -group vectorsum_tb/vectorsum_top_inst/x_inst
add wave -noupdate -group vectorsum_tb/vectorsum_top_inst/x_inst -radix hexadecimal /vectorsum_tb/vectorsum_top_inst/x_inst/*
add wave -noupdate -group vectorsum_tb/vectorsum_top_inst/y_inst
add wave -noupdate -group vectorsum_tb/vectorsum_top_inst/y_inst -radix hexadecimal /vectorsum_tb/vectorsum_top_inst/y_inst/*
add wave -noupdate -group vectorsum_tb/vectorsum_top_inst/z_inst
add wave -noupdate -group vectorsum_tb/vectorsum_top_inst/z_inst -radix hexadecimal /vectorsum_tb/vectorsum_top_inst/z_inst/*

run -all
```

MODELSIM SIMULATION

- Validation of errors and cycle count



vectorsum_tb	
/vectorsum_tb/clock	1'h0
/vectorsum_tb/reset	1'h0
/vectorsum_tb/x_din	32'hc1b00002
/vectorsum_tb/x_wr_en	1'h1
/vectorsum_tb/x_full	1'h0
/vectorsum_tb/y_din	32'hc158050c
/vectorsum_tb/y_wr_en	1'h1
/vectorsum_tb/y_full	1'h0
/vectorsum_tb/z_dout	32'h63400000
/vectorsum_tb/z_rd_en	1'h1
/vectorsum_tb/z_empty	1'h0
/vectorsum_tb/x_write_done	1'h0
/vectorsum_tb/y_write_done	1'h0
/vectorsum_tb/z_read_done	1'h0
/vectorsum_tb/z_errors	32'h00000000
vectorsum_tb/vectorsum_top_inst	
vectorsum_tb/vectorsum_top_inst/vectorsum_inst	
vectorsum_tb/vectorsum_top_inst/x_inst	
vectorsum_tb/vectorsum_top_inst/y_inst	
vectorsum_tb/vectorsum_top_inst/z_inst	

(vectorsum_tb)
03400000
c1d00461
dbc00000
41d3edfd
04000000
c1b000
60000000
c1852191
22400000
c1d32110
40000000
c15805
00000000
63400000
00000000
(vectorsum_tb/vectorsum_top_inst)
(vectorsum_tb/vectorsum_top_inst/vectorsum_inst)
(vectorsum_tb/vectorsum_top_inst/x_inst)
(vectorsum_tb/vectorsum_top_inst/y_inst)
(vectorsum_tb/vectorsum_top_inst/z_inst)

```
# Loading sv_std.std
# Loading work.vectorsum_tb(fast)
# Loading work.vectorsum_top(fast)
# Loading work.vectorsum(fast)
# Loading work.fifo(fast)
# @ 20: Loading file y.txt...
# @ 20: Loading file x.txt...
# @ 25: Beginning simulation...
# @ 30: Comparing file z.txt...
# @ 670: Simulation completed.
# Total simulation cycle count: 64
# Total error count: 0
```

NEXT…

- HW #3: Streaming architecture with Motion Detection algorithm