RTL Register-Based Memory Implementations

This Application Note describes how to build and test a high speed register SRAM or FIFO given RTL code. With a small memory requirement, you can synthesize to a non-SRAM-based Actel family, such as the XL or ACT 3 families. This note covers the following three scenarios:

1. Single-port SRAM
2. Dual-port SRAM
3. FIFO

Each scenario gives the RTL code, verilog test bench, and the synthesis results in area and speed. In addition, the FIFO scenario gives the VHDL RTL code.

Register-based Single-port SRAM

This is only a viable solution if the SRAM is relatively small. Of course, this is a design dependent decision. To quickly determine whether or not this idea has a chance of working, use the following formula.

```
Total available registers >= user registers +
SRAM bits + 0.6(SRAM bits)decode logic
```

Verilog RTL CODE:

The intent was to make the source code easy to customize, therefore parameters were used. To modify the width or depth, simply modify the listed parameters in the code. However, the code does assume that the user wants to use posedge clk and negedge reset. Simply modify the always blocks if that is not the case.

```
`timescale 1 ns/100 ps
//################################################
//# Behavioral single-port SRAM description :
//#    Active High write enable (WE)
//#    Rising clock edge (Clock)
//################################################

module reg_sram(Data, Q, Clock, WE, Address);

parameter width = 8;
parameter depth = 8;
parameter addr = 3;

input Clock, WE;
input [addr-1:0] Address;
input [width-1:0] Data;
output [width-1:0] Q;
wire [width-1:0] Q;
reg [width-1:0] mem_data [depth-1:0];

always @(posedge Clock)
   if(WE)
                mem_data[Address] = #1 Data;


assign Q = mem_data[Address];

endmodule
```

Verilog RTL Synthesis Results:

The above RTL synthesized to 66 registers with a total of 102 logic modules, which utilized 51% of a 1415A. It uses 22 IOs. It runs at 220MHz in -2 speed grade.

Verilog RTL Simulation:

Below is the verilog self-checking testbench. This testbench does rely upon the default parameters given below.

```
`timescale 1 ns/100 ps
module test;

parameter width = 8;    // bus width
parameter addr_bits = 3;     // # of addr lines
parameter numvecs = 21; // actual number of vec-
tors
parameter Clockper = 1000; // 100ns period

reg [width-1:0] Data;
reg [addr_bits-1:0] Address;
reg Clock, WE;
reg [width-1:0] data_in  [0:numvecs-1];
reg [width-1:0] data_out [0:numvecs-1];

wire [width-1:0] Q;

integer i, j, numerrors;

reg_sram u0(.Data(Data), .Q(Q), .Clock(Clock),
.WE(WE), .Address(Address));

initial
begin
        // sequential test patterns entered at
neg edge Clock

   data_in[0]=8'h00; data_out[0]=8'hxx;
   data_in[1]=8'h01; data_out[1]=8'h01;
   data_in[2]=8'h02; data_out[2]=8'h02;
   data_in[3]=8'h04; data_out[3]=8'h04;
   data_in[4]=8'h08; data_out[4]=8'h08;
   data_in[5]=8'h10; data_out[5]=8'h10;
   data_in[6]=8'h20; data_out[6]=8'h20;
   data_in[7]=8'h40; data_out[7]=8'h40;
   data_in[8]=8'h80; data_out[8]=8'h80;
   data_in[9]=8'h07; data_out[9]=8'h01;
   data_in[10]=8'h08; data_out[10]=8'h02;
   data_in[11]=8'h09; data_out[11]=8'h04;
```

```
   data_in[12]=8'h10; data_out[12]=8'h08;
   data_in[13]=8'h11; data_out[13]=8'h10;
   data_in[14]=8'h12; data_out[14]=8'h20;
   data_in[15]=8'h13; data_out[15]=8'h40;
   data_in[16]=8'h14; data_out[16]=8'h80;
   data_in[17]=8'haa; data_out[17]=8'haa;
   data_in[18]=8'h55; data_out[18]=8'haa;
   data_in[19]=8'h55; data_out[19]=8'h55;
   data_in[20]=8'haa; data_out[20]=8'h55;

end

initial
begin
   Clock = 0;
   WE = 0;
   Address = 0;
   Data = 0;
   numerrors = 0;
end

always#(Clockper / 2)  Clock = ~Clock;

initial
begin
   #2450 WE = 1;
   #8000 WE = 0;
   #8000 WE = 1;
   #1000 WE = 0;
   #1000 WE = 1;
   #1000 WE = 0;
end

initial
begin
   #1450;
   for (j = 0; j <= width; j = j + 1)
      #1000 Address = j;
   for (j = 1; j <= width; j = j + 1)
      #1000 Address = j;
   Address = 0;
end

initial
begin
   $display("\nBeginning Simulation...");

   //skip first rising edge
   for (i = 0; i <= numvecs-1; i = i + 1)
   begin
      @(negedge Clock);
      // apply test pattern at neg edge
      Data = data_in[i];

      @(posedge Clock)
      #450; //45 ns later
      // check result at posedge + 45 ns
      $display("Pattern#%d time%d: WE=%b; Ad-
dress=%h; Data=%h; Expected Q=%h; Actual Q=%h",
i, $stime, WE, Address, Data, data_out[i], Q);
      if ( Q !== data_out[i] )
         begin
         $display("  ** Error");
         numerrors = numerrors + 1;
         end
   end
   if (numerrors == 0)
      $display("Good!  End of Good Simula-
tion.");
   else
      if (numerrors > 1)
         $display(
           "%0d ERRORS!  End of Faulty Simula-
tion.",numerrors);
      else
         $display(
          "1 ERROR!  End of Faulty Simulation.");

   #1000 $finish; // after 100 ns later
end

endmodule
```

RTL Simulation Results:

The simulation results for the gate-level and the RTL should of course be the same, and should match the below report:

```
Beginning Simulation...
Pattern#0 time1950: WE=0; Address=0; Data=00; Expected Q=xx; Actual Q=xx
Pattern#1 time2950: WE=1; Address=0; Data=01; Expected Q=01; Actual Q=01
Pattern#2 time3950: WE=1; Address=1; Data=02; Expected Q=02; Actual Q=02
Pattern#3 time4950: WE=1; Address=2; Data=04; Expected Q=04; Actual Q=04
Pattern#4 time5950: WE=1; Address=3; Data=08; Expected Q=08; Actual Q=08
Pattern#5 time6950: WE=1; Address=4; Data=10; Expected Q=10; Actual Q=10
Pattern#6 time7950: WE=1; Address=5; Data=20; Expected Q=20; Actual Q=20
Pattern#7 time8950: WE=1; Address=6; Data=40; Expected Q=40; Actual Q=40
Pattern#8 time9950: WE=1; Address=7; Data=80; Expected Q=80; Actual Q=80
Pattern#9time10950: WE=0; Address=0; Data=07; Expected Q=01; Actual Q=01
Pattern#10time11950: WE=0; Address=1; Data=08; Expected Q=02;Actual Q=02
Pattern#11time12950: WE=0; Address=2; Data=09; Expected Q=04;Actual Q=04
Pattern#12time13950: WE=0; Address=3; Data=10; Expected Q=08;Actual Q=08
Pattern#13time14950: WE=0; Address=4; Data=11; Expected Q=10;Actual Q=10
Pattern#14time15950: WE=0; Address=5; Data=12; Expected Q=20;Actual Q=20
Pattern#15time16950: WE=0; Address=6; Data=13; Expected Q=40;Actual Q=40
Pattern#16time17950: WE=0; Address=7; Data=14; Expected Q=80;Actual Q=80
Pattern#17time18950: WE=1; Address=0; Data=aa; Expected Q=aa;Actual Q=aa
Pattern#18time19950: WE=0; Address=0; Data=55; Expected Q=aa;Actual Q=aa
Pattern#19time20950: WE=1; Address=0; Data=55; Expected Q=55;Actual Q=55
Pattern#20time21950: WE=0; Address=0; Data=aa; Expected Q=55;Actual Q=55
Good!  End of Good Simulation.
L111 "reg_sram.vt": $finish at simulation time 229500
729 simulation events + 12571 accelerated events + 82600 timing check
events
```

Register-based Dual-Port SRAM

Verilog RTL CODE:

This code was designed to imitate the behavior of the Actel DX family dual-port SRAM.

```
`timescale 1 ns/100 ps
//################################################
//# Behavioral dual-port SRAM description :
//#    Active High write enable (WE)
//#    Active High read enable (RE)
//#    Rising clock edge (Clock)
//################################################

module reg_dpram(Data, Q, Clock, WE, RE, WAd-
dress, RAddress);

parameter width = 8;
parameter depth = 8;
parameter addr = 3;

input Clock, WE, RE;

input [addr-1:0] WAddress, RAddress;
input [width-1:0] Data;
output [width-1:0] Q;
reg [width-1:0] Q;
reg [width-1:0] mem_data [depth-1:0];

// ################################################
// # Write Functional Section
// ################################################
always @(posedge Clock)
begin
   if(WE)
                  mem_data[WAddress] = #1 Data;
end

//################################################
//# Read Functional Section
//################################################
always @(posedge Clock)
begin
   if(RE)
      Q = #1 mem_data[RAddress];
end

endmodule
```

Verilog RTL Synthesis Results:

The above RTL synthesized to 72 registers with a total of 98 logic modules, which utilized 49% of a 1415A. It uses 26 IOs. It runs at 125MHz in -2 speed grade.

Verilog RTL Simulation:

The verilog self-checking testbench is similar to the previous testbench.

```
`timescale 1 ns/100 ps
module test;

parameter width = 8;    // bus width
parameter addr = 3;     // # of addr lines
parameter numvecs = 20; // actual number of vec-
tors
parameter Clockper = 1000; // 100ns period
```

```
reg [width-1:0] Data;
reg [addr-1:0] WAddress, RAddress;
reg Clock, WE, RE;
reg [width-1:0] data_in  [0:numvecs-1];
reg [width-1:0] data_out [0:numvecs-1];

wire [width-1:0] Q;

integer i, j, k, numerrors;

reg_dpram u0(.Data(Data), .Q(Q), .Clock(Clock),
.WE(WE),
      .RE(RE), .WAddress(WAddress), .RAd-
dress(RAddress));

initial
begin
       // sequential test patterns entered at
neg edge Clock

   data_in[0]=8'h00; data_out[0]=8'hxx;
   data_in[1]=8'h01; data_out[1]=8'hxx;
   data_in[2]=8'h02; data_out[2]=8'hxx;
   data_in[3]=8'h04; data_out[3]=8'hxx;
   data_in[4]=8'h08; data_out[4]=8'hxx;
   data_in[5]=8'h10; data_out[5]=8'hxx;
   data_in[6]=8'h20; data_out[6]=8'hxx;
   data_in[7]=8'h40; data_out[7]=8'hxx;
   data_in[8]=8'h80; data_out[8]=8'hxx;
   data_in[9]=8'h07; data_out[9]=8'h01;
   data_in[10]=8'h08; data_out[10]=8'h02;
   data_in[11]=8'h09; data_out[11]=8'h04;
   data_in[12]=8'h10; data_out[12]=8'h08;
   data_in[13]=8'h11; data_out[13]=8'h10;
   data_in[14]=8'h12; data_out[14]=8'h20;
   data_in[15]=8'h13; data_out[15]=8'h40;
   data_in[16]=8'h14; data_out[16]=8'h80;
   data_in[17]=8'haa; data_out[17]=8'h80;
   data_in[18]=8'h55; data_out[18]=8'haa;
   data_in[19]=8'haa; data_out[19]=8'h55;
end

initial
begin
   Clock = 0;
   WE = 0;
   RE = 0;
   WAddress = 0;
   RAddress = 0;
   Data = 0;
   numerrors = 0;
end

always#(Clockper / 2)  Clock = ~Clock;

initial
begin
   #2450 WE = 1;
   #8000 WE = 0;
         RE = 1;
   #8000 RE = 0;
         WE = 1;
   #1000 RE = 1;

end

initial
begin
   #1450;
   for (j = 0; j <= width; j = j + 1)
      #1000 WAddress = j;
   WAddress = 0;
end

initial
begin
   #9450;
   for (k = 0; k <= width; k = k + 1)
      #1000 RAddress = k;
   RAddress = 0;
end

initial
begin
   $display("\nBeginning Simulation...");

   //skip first rising edge
   for (i = 0; i <= numvecs-1; i = i + 1)
   begin
      @(negedge Clock);
      // apply test pattern at neg edge
      Data = data_in[i];

      @(posedge Clock)
      #450; //45 ns later
      // check result at posedge + 45 ns
      $display("Pattern#%d time%d: WE=%b; Wad-
dr=%h; RE=%b; Raddr=%h; Data=%h; Expected Q=%h;
Actual Q=%h", i, $stime, WE, WAddress, RE, RAd-
dress, Data, data_out[i], Q);

      if ( Q !== data_out[i] )
         begin
         $display("  ** Error");
         numerrors = numerrors + 1;
         end
   end
   if (numerrors == 0)

      $display("Good!  End of Good Simula-
tion.");
   else
      if (numerrors > 1)

         $display(
           "%0d ERRORS!  End of Faulty Simula-
tion.",numerrors);
      else
         $display(
          "1 ERROR!  End of Faulty Simulation.");

   #1000 $finish; // after 100 ns later
end
endmodule
```

Register-based FIFO

To quickly determine whether or not this idea has a chance of working, use the following formula.

```
Total available registers >= user registers +
fifo bits + 1.2*(fifo bits) for fifo control log-
ic
```

Verilog RTL CODE:

This code was designed to imitate the behavior of the Actel DX family dual-port SRAM based fifo.

```
`timescale 1 ns/100 ps
//##############################################
//# Behavioral description of FIFO with :
//#    Active High write enable (WE)
//#    Active High read enable (RE)
//#    Active Low asynchronous clear (Aclr)
//#    Rising clock edge (Clock)
//#    Active High Full Flag
//#    Active Low Empty Flag
//##############################################

module reg_fifo(Data, Q, Aclr, Clock, WE, RE,
FF, EF);

parameter width = 8;
parameter depth = 8;
parameter addr = 3;

input Clock, WE, RE, Aclr;
input [width-1:0] Data;
output FF, EF;//Full & Empty Flags
output [width-1:0] Q;
reg [width-1:0] Q;
reg [width-1:0] mem_data [depth-1:0];
reg [addr-1:0] WAddress, RAddress;
reg FF, EF;

// ############################################
// # Write Functional Section
// ############################################
// WRITE_ADDR_POINTER
always@(posedge Clock or negedge Aclr)
begin
   if(!Aclr)
      WAddress = #2 0;
   else if (WE)
      WAddress = #2 WAddress + 1;
end

// WRITE_REG
always @(posedge Clock)
begin
   if(WE)
                mem_data[WAddress] = Data;
end

//##############################################
//# Read Functional Section
//##############################################
// READ_ADDR_POINTER
always@(posedge Clock or negedge Aclr)
begin
   if(!Aclr)
      RAddress = #1 0;
   else if (RE)
      RAddress = #1 RAddress + 1;
end

// READ_REG
always @(posedge Clock)
begin
   if(RE)
      Q = mem_data[RAddress];
end

//##############################################
//# Full Flag Functional Section : Active high
//##############################################
always@(posedge Clock or negedge Aclr)
begin
        if(!Aclr)
                FF = #1 1'b0;
        else if ( (WE & !RE) && ( (WAddress ==
RAddress-1) ||
      ( (WAddress == depth-1) && (RAddress ==
1'b0) ) ) )
      FF = #1 1'b1;
   else
      FF = #1 1'b0;
end

//##############################################
//# Empty Flag Functional Section : Active low
//##############################################
always@(posedge Clock or negedge Aclr)
begin
        if(!Aclr)
                EF = #1 1'b0;
        else if ( (!WE & RE) && ( (WAddress ==
RAddress+1) ||
      ( (RAddress == depth-1) && (WAddress ==
1'b0) ) ) )
      EF = #1 1'b0;
   else
      EF = #1 1'b1;
end

endmodule
```

VHDL RTL CODE:

```
-- ******************************************
-- Behavioral description of dual-port FIFO with :
--      Active High write enable (WE)
--      Active High read enable (RE)
--      Active Low asynchronous clear (Aclr)
--      Rising clock edge (Clock)
--      Active High Full Flag
--      Active Low Empty Flag
-- ******************************************

library ieee;
use ieee.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity reg_fifo is

  generic (width    : integer:=8;
           depth    : integer:=8;
           addr     : integer:=3);

  port (Data   : in std_logic_vector(width-1
```

```
downto 0);
        Q      : out std_logic_vector(width-1
downto 0);
        Aclr   : in std_logic;
        Clock : in std_logic;
        WE     : in std_logic;
        RE     : in std_logic;
        FF     : out std_logic;
        EF     : out std_logic);

end reg_fifo;

library ieee;
use ieee.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

architecture behavioral of reg_fifo is

  type MEM is array(0 to depth-1) of
std_logic_vector(width-1 downto 0);
  signal ramTmp : MEM;
  signal WAddress : std_logic_vector(addr-1
downto 0);
  signal RAddress : std_logic_vector(addr-1
downto 0);
  signal words : std_logic_vector(addr-1 downto
0);

begin

--  words <= conv_std_logic_vector (depth-1,ad-
dr);

  -- ########################################
  -- # Write Functional Section
  -- ########################################

  WRITE_POINTER : process (Aclr,Clock)
  begin
    if (Aclr = '0') then
      WAddress <= (others => '0');
    elsif (Clock'event and Clock = '1') then
      if (WE = '1') then
        if (WAddress = words) then
          WAddress <= (others => '0');
        else
          WAddress <= WAddress + '1';
        end if;
      end if;
    end if;
  end process;

  WRITE_RAM : process (Clock)
  begin
    if (Clock'event and Clock = '1') then
      if (WE = '1') then
       ramTmp(conv_integer (WAddress)) <= Data;
      end if;
    end if;
  end process;

  -- ########################################
  -- # Read Functional Section
  -- ########################################

  READ_POINTER : process (Aclr,Clock)
  begin
    if (Aclr = '0') then
      RAddress <= (others => '0');
    elsif (Clock'event and Clock = '1') then
      if (RE = '1') then
        if (RAddress = words) then
          RAddress <= (others => '0');
        else
          RAddress <= RAddress + '1';
        end if;
      end if;
    end if;
  end process;

  READ_RAM : process (Clock)
  begin
    if (Clock'event and Clock = '1') then
      if (RE = '1') then
        Q <= ramTmp(conv_integer(RAddress));
      end if;
    end if;
  end process;

  -- ########################################
  -- # Full Flag Functional Section : Active high
  -- ########################################

  FFLAG : process (Aclr,Clock)
  begin
    if (Aclr = '0') then
      FF <= '0';
    elsif (Clock'event and Clock = '1') then
      if (WE = '1' and RE = '0') then
        if ((WAddress = RAddress-1) or
            ((WAddress = depth-1) and (RAddress
= 0))) then
          FF <= '1';
        end if;
      else
   FF <= '0';
      end if;
    end if;
  end process;

  -- ########################################
  -- # Empty Flag Functional Section : Active low
  -- ########################################

  EFLAG : process (Aclr,Clock)
  begin
    if (Aclr = '0') then
      EF <= '0';
    elsif (Clock'event and Clock = '1') then
      if (RE = '1' and WE = '0') then
        if ((WAddress = RAddress+1) or
            ((RAddress = depth-1) and (WAddress
= 0))) then
          EF <= '0';
        end if;
      else
   EF <= '1';
      end if;
    end if;
  end process;

end behavioral;
```

RTL Synthesis Results:

The above RTL synthesized to 86 registers with a total of 155 logic modules, which utilized 78% of a 1415A. It uses 23 IOs. It runs at 45MHz in -2 speed grade. The performance could be enhanced if Actgen counter were instanciated instead of being synthesized.

RTL Simulation:

Below is the verilog self-checking testbench. This testbench was used to verify the verilog RTL code and gate level results from the VHDL and the verilog synthesis.

```
`timescale 1 ns/100 ps
module test;

parameter numvecs = 25; // actual number of vec-
tors
parameter width = 8;     // data bit width
parameter Clockper = 1000; // 100ns period

reg [width-1:0] Data;
reg Aclr, Clock, WE, RE;
reg [width-1:0] data_in  [0:numvecs-1];    // in
vector matrix
reg [width-1:0] data_out [0:numvecs-1];   // out
vector matrix

wire [width-1:0] Q;
wire FF, EF;

reg_fifo u0(.Data(Data), .Q(Q), .Aclr(Aclr),
.Clock(Clock), .WE(WE),
      .RE(RE), .FF(FF), .EF(EF));

integer i;
integer numerrors;

initial
begin
        // sequential test patterns entered at
neg edge Clock

      data_in[0] =8'hff; data_out[0] = 8'hxx;
      data_in[1] =8'h00; data_out[1] = 8'hff;
      data_in[2] =8'h00; data_out[2] = 8'hff;
   data_in[3] =8'h01; data_out[3] = 8'hff;
   data_in[4] =8'h02; data_out[4] = 8'hff;
   data_in[5] =8'h03; data_out[5] = 8'hff;
   data_in[6] =8'h04; data_out[6] = 8'hff;
   data_in[7] =8'h05; data_out[7] = 8'hff;
   data_in[8] =8'h06; data_out[8] = 8'hff;
   data_in[9] =8'h07; data_out[9] = 8'hff;
   data_in[10] =8'h08; data_out[10] = 8'h00;
   data_in[11] =8'h09; data_out[11] = 8'h01;
   data_in[12] =8'h10; data_out[12] = 8'h02;
   data_in[13] =8'h11; data_out[13] = 8'h03;
   data_in[14] =8'h12; data_out[14] = 8'h04;
   data_in[15] =8'h13; data_out[15] = 8'h05;
   data_in[16] =8'h14; data_out[16] = 8'h06;
   data_in[17] =8'hff; data_out[17] = 8'h07;
   data_in[18] =8'hff; data_out[18] = 8'h07;
   data_in[19] =8'haa; data_out[19] = 8'hff;
   data_in[20] =8'h55; data_out[20] = 8'haa;
   data_in[21] =8'haa; data_out[21] = 8'h55;
   data_in[22] =8'h00; data_out[22] = 8'haa;
   data_in[23] =8'hff; data_out[23] = 8'h00;
   data_in[24] =8'haa; data_out[24] = 8'hff;

end

initial
begin
   Aclr = 0;
   Clock = 0;
   WE = 0;
   RE = 0;
   Data = 0;
end

always
begin
   #(Clockper / 2)  Clock = ~Clock;
end

initial #3450 Aclr = 1;

initial
begin
   #1450 WE = 1;
   #1000 WE = 0;
   #1000 WE = 1;
   #8000 WE = 0;
   #8000 WE = 1;
   #6000 WE = 0;
end

initial
begin
   #2450 RE = 1;
   #1000 RE = 0;
   #8000 RE = 1;
   #8000 RE = 0;
   #1000 RE = 1;
end

initial
begin
   numerrors = 0;
   $display("\nBeginning Simulation...");

   //skip first rising edge
   for (i = 0; i <= numvecs-1; i = i + 1)
   begin
      @(negedge Clock);
      // apply test pattern at neg edge
      Data = data_in[i];

      @(posedge Clock)
      #450; //45 ns later
      // check result at posedge + 45 ns
      $display("Pattern#%d time%d: Aclr=%b;
WE=%b; RE=%b; Data=%h; FF=%b; EF=%b; Expected
Q=%h; Actual Q=%h", i, $stime, Aclr, WE, RE, Da-
ta, FF, EF, data_out[i], Q);

      if ( Q !== data_out[i] )
         begin
         $display("  ** Error");
         numerrors = numerrors + 1;
         end
   end
   if (numerrors == 0)
```

```
      $display("Good!  End of Good Simula-
tion.");
   else
      if (numerrors > 1)

         $display(
           "%0d ERRORS!  End of Faulty Simula-
tion.",numerrors);
      else
         $display(
          "1 ERROR!  End of Faulty Simulation.");

   #1000 $finish; // after 100 ns later
end

endmodule
```

Actel Corporation
955 East Arques Avenue
Sunnyvale, CA 94086
Tel: (408) 739-1010
Fax: (408) 739-1540

Actel Europe Ltd.
Daneshill House, Lutyens Close
Basingstoke, Hampshire RG24 8AG
United Kingdom
Tel: (+44) (1256) 305600
Fax: (+44) (1256) 355420

5192633-0