Interfacing the QDR™ with Altera APEX20KE

QDR™: An Introduction

The evolution of newer systems has increased demands on speed and performance. As a result of this, faster processors have emerged that have increased the demands on memory performance. Newer memory architectures with higher through-put have been designed that support the current systems and processors. This application note introduces the QDR™ architecture, an SRAM architecture designed to improve the SRAM interface bandwidth by as much as four times the current solutions. This application note also shows how this high performance SRAM can be interfaced to the Altera APEX 20KE FPGA.

QDR is a family of synchronous SRAMs with an innovative architecture designed especially for high-performance-networking systems by the QDR co-development team, comprised of Cypress, IDT, and Micron.

QDR stands for Quad Data Rate, which effectively means that 4 words of data can be transferred through the SRAM in a single clock cycle. The QDR Family (currently 1/01) includes two devices, which differ by the number of data words that will be burst on every access.

Table 1 shows the current and planned devices in the family.

Table 1. Devices in the QDR™ Family

Device	SIze	Description
CY7C1302V25	512K x 18	QDR - Burst of 2
CY7C1304V25	512K x 18	QDR - Burst of 4

What is a QDR?

Most of the existing SRAM's solutions are optimized for PC applications, with interfaces designed to move data efficiently for PC-type common I/O applications. In most networking applications continous movement of data through the SRAM is a neccesity. In such applications, there are continous transitions between read and writes through the memory. Common I/O devices like standard synchrounous pipelined SRAMs do not perform well under these conditions. No Bus Latency™ SRAMs have optimized the synchronous SRAM architecture to allow no latency in the read/write transitions and have 100% utilization of the I/O bus.

In most networking applications, this improvement in through-put is not enough. So applications that need to have the read and write done simultaneously, will benefit tremedously from the QDR SRAM.

Applications such as ATM switches and routers will benefit from the fact that simultaneous reads and write can be done on the SRAM with no latency and the data through the SRAM is guaranteed even for simultaneous access to the same address location.

For more details on the QDR refer to the application note "QDR: The Next Generation High-Performance Networking SRAM."

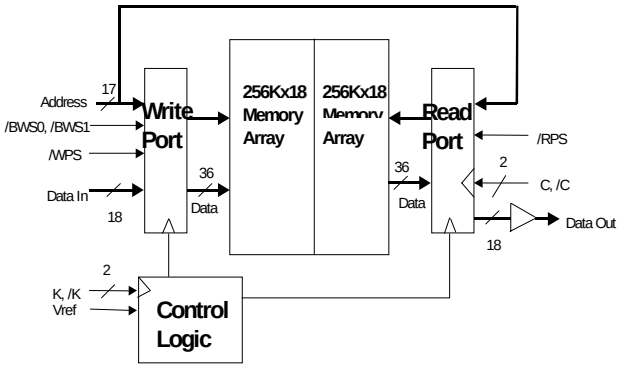


Figure 1. Block Diagram of the CY7C1302V25

CY7C1302V25

Block Diagram of the CY7C1302V25 shows the block diagram of the CY7C1302V25 QDR device.

The QDR SRAM family has two members: the two-word burst device (CY7C1302V25) and the four-word burst device (CY7C1304V25). The difference between the two is the number of words of data that can be obtained from the SRAM on a single read or provided to the SRAM on a write.

The block diagram in *Timing Diagram of the CY7C1302V25* is for the CY7C1302V25. Data ports are seperated for the read and write ports. The address lines are shared for the read and write ports. Data is transferred in a Double Data Rate manner (DDR) on both input and output ports. This allows four words of data to be transferred on every clock cycle.

Description of Inputs to the QDR SRAM

Clock Inputs

QDR SRAMs have four clocks: K, $\overline{K}$, C and $\overline{C}$. The K and $\overline{K}$ clocks are used for sampling the inputs and C and $\overline{C}$ clocks are used to clock out the data from the SRAM. All transactions are initiated on the rising edge of K.

Control Inputs

QDR SRAMs have a very simple control structure. Two control signals: $\overline{RPS}$ (Read Port Select) and $\overline{WPS}$ (Write Port Select) are used to control the read and write operations on the SRAM. These signals are sampled on the rising edge of the K clock. More information on these will be provided in later sections.

Address Inputs

The address inputs for the QDR are common for the read and write ports. On the CY7C1304V25 the addresses are samples on alternate cycles of the clock for a read or a write. On the CY7C1302V25 the address inputs are sampled on the rising edge of K for the read and on the rising edge of $\overline{K}$ for the write.

Data Inputs/Outputs

The data lines on the QDR are unidirectional. Two words can be transferred to and from the QDR in every cycle. More details on the functionality of the device are explained in the section on timings.

Timings

Timing Diagram of the CY7C1302V25 shows the timing diagram for the CY7C1302V25, the two word burst device on the QDR.

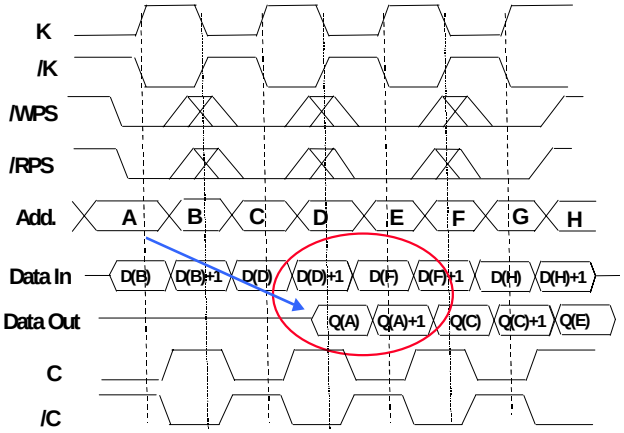


Figure 2. Timing Diagram of the CY7C1302V25

In the first clock cycle, $\overline{RPS}$ and $\overline{WPS}$ are both asserted active LOW. The address for the read (A) is latched on the rising edge of the K clock and the address for the write (B) is latched on the rising edge of the $\overline{K}$ clock.

The data for the write to address B, is also latched on the rising edges of K clock (D(B)) and /K clock (D(B+1). The byte writes are also latched on the same clock as the data.

Read accesses to the CY7C1302V25 are conducted in 2 cycles (a 2-stage pipeline). During the first cycle the address is latched on the rising edge of K clock. The address is then presented to the memory. The next rising edge of C clocks out the first 18-bit data word (Q(A)). The next rising edge of /C clocks out the second 18-bit data word (Q(A+1)).

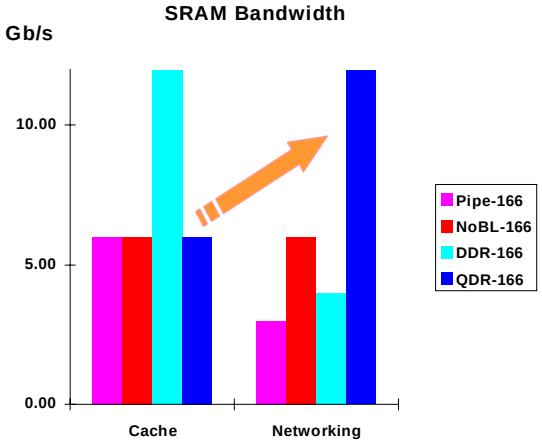
As seen from the timing diagram, one could start QDR read and write to the same address on the same cycle. When such an operation occurs, the QDR forwards the write to the write port and ensures that valid data is driven out on the data bus. By doing so, data coherency is guaranteed.

Performance Comparision

Performance comparison of the QDR 50% / 50% Read/Write shows the performance comparison of the QDR against other SRAMs. The comparison was done assuming that all the interfaces operate at 166 Mhz.

The QDR performance is far ahead of all the other SRAMs and the performance is 4x the closest device in a networking application.

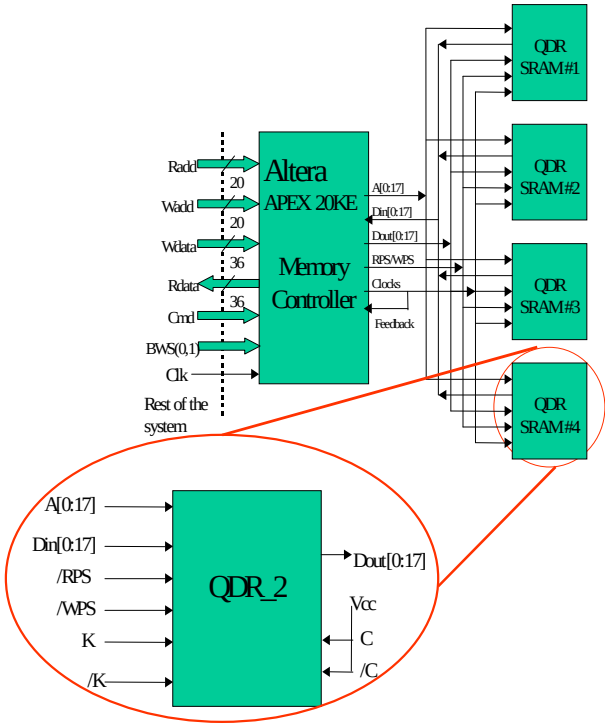
Figure 3. Performance comparison of the QDR 50% / 50% Read/Write



Memory Controller for the QDR SRAMs

To simplify the process of designing an interface to the QDR SRAM, a memory controller was developed using the Altera APEX 20KE FPGA. A block diagram of the memory controller is shown in *Performance comparison of the QDR*.

Figure 4. Performance comparison of the QDR



The memory controller is designed to control four SRAMs in a depth expanded fashion. Each of the QDR SRAMs receive seperate control signals for the read and write ports, while the

address and the data ports are common for all the SRAMs. The SRAMs form a bank of 2M x 18.

The memory controller generates all the control signals for the memory array, The memory controller views the complete SRAM bank like an unified memory array. It supports concurrent double data rate operation on all the inputs and outputs, it also allows byte write operation to the memory bank.

The memory controller assumes that the QDR SRAMs are in the single clock mode. This can simplify the memory interface. It operates at 100 Mhz, allowing a bandwidth of 7.2 Gbits/s. The memory controller has independent read and write state machines. The memory controller is a command based interface with a two-bit command input. This memory controller shows the interface between the APEX and the CY7C1302V25. The memory controller can be modified for an interface to the 4-word burst device, CY7C1304V25.

Table 2. QDR SRAM Interface Signals

Signal Name	Description
K,K_Bar	Clock Inputs to the QDR™ SRAM
C,C_Bar	Output Clocks for the QDR™
RPS_Bar	Read Control Signal
WPS_Bar	Write Control Signal
BWS_Bar	Byte Write Select Signal
A(17..0)	Address Inputs
Dout(0:17)	Read Data output from the QDR™
Din(0:17)	Write Data Input to the QDR™

State Machine

Appendix 11 gives the details of the state machine.

The QDR SRAM can essentialy be viewed as a two-port device, with an input port and an output port for the data.

When using a QDR SRAM in an system, a memory controller generates all the signals needed for the SRAM and serves as the interface to the system.

The memory controller used for the interface below is a Altera APEX20KE. The APEX family is an embedded programmable logic device family which can be used in a broad range of applications. Altera APEX20KE devices, with their speed and configurability are ideal for such an interface.

The Altera APEX architecture features: up to four programmable phase locked loops per device; contains ClockBoost, ClockShift, and ClockLock circuitry for increased performance, flexible clock frequency multiplication and division. The APEX family is a combination of three families into one architecture, which permits designers to integrate a complex system into a single device, eliminating the need for multi-device modules, saving board space and simplifying complex design implementation. Together, these features provide significant improvements in system performance and bandwidth.

The signals needed by the QDR are as specified in *Table 2* and these are generated by the Altera FPGA. Because the interface between the QDR and the APEX device is at a high speed, some timing issues may arise.

Timing

The critical timing parameters are:

a. Set-up/Hold window for QDR on a write.

b. Set-up/Hold window for APEX on a read.

The following sections discuss each of these seperately.

Write Cycle Time

When designing for the proper write cycle timing, meeting the set-up and hold requirements of the SRAM are the primary concern. Set-up and hold specifications for the CY7C1302V25 (100-MHz speed grade) are 1 ns each.

Both the QDR clock and data signals are driven from the controller, so the clock-to-output delay from the APEX device pins is the same for both sets of pins. The same principle applies to board delay, because flight times for clock and data signals are equalized.

The $\overline{K}$ and $\overline{K}$ outputs are clocked out from two registers, clkout and $\overline{clkout}$. If these registers are placed in logic element (LE) registers in the adjacent logic array block (LAB), their clock-to-output time is only slightly greater than the clock-to-output time of the data and address signals. Because the K and $\overline{K}$ are clocked out on the positve edge of the 2X clock while data and address signals are clocked out on the negative edge, there is a cushion of 2.5 ns each way for timing purposes.

The following calculations apply for controller to SRAM data transfers. The calculation allows for up to a 1.2 ns difference between clock t_{CO} and data /address t_{CO} and upto 0.2 ns of board skew.

$[t_{CO}(\text{APEX Clock}) - t_{CO}(\text{APEX Data and Address})] +$

Board Skew(Clock – Data) + t_{SU}(SRAM) < 2.5 ns

[4.5 ns – 3.7 ns] + 0.2 ns +1.0 ns = 2.4 ns < 2.5 ns

$[t_{CO}(\text{APEX Data and Address}) - t_{CO}(\text{ APEX Clock})] +$

Board Skew(Clock – Data) + t_H(SRAM) < 2.5 ns

[3.4 ns -2.2 ns] + 0.2 ns + 1.0 ns = 2.4 ns < 2.5 ns

Read Cycle Timing

When read data is sent from the SRAM to the controller, set-up times on the APEX device must be met. Low set-up times can be achieved be using the programmable delay features available in the APEX device.

Arrival time of the “dout” signal is determined by the clock-to-output specification for the SRAM. For the CY7C1302V25, the maximum t_{CO} value is 3 ns , while for the minimum t_{CO} value (i.e., data output hold time t_{DOH}) is 1.2 ns. Board delay can be ignored, because flight times for the C signal and the “dout” bus are roughly equal. Regardless, the timing calculation allows for some skew between the clock and data lines.

Data is sent out of the SRAM on the rising edge of C and is captured by the controller on the falling edge of C. For a clock speed of 100 MHz , there is a 5-ns window between the rising and falling edges. Subtracting a clock-to-output delay of 3 ns

leaves 2 ns of set-up time on the APEX pins. This timing meets the setup requirement of 0.9 ns, with 0.2 ns margin for any signal skew.

The following calculations apply for data transfer from the SRAM to the controller:

t_{CO}(SRAM) + Board Skew(Clock – Data) + t_{SU}(APEX) < 5 ns

3ns + 0.2 ns + 0.9 ns = 4.1 ns < 5 ns

t_{DOH}(SRAM) + Board Skew(Data – clock) – t_H (APEX) > 0 ns

1.2 ns – 0.2 ns – 0.0 ns = 1.0 ns > 0 ns

Conclusion

The QDR SRAM architecture was designed to greatly increase memory bandwidth for communication systems. APEX devices are suited for the interface to the QDR SRAM because of their speed and programmability.

The VHDL implementation is shown in *Appendix2*. This implementation is available in verilog also. These can be downloaded from the following website.

http://www.cypress.com/design/support/models/sram_models.html

Appendix 1

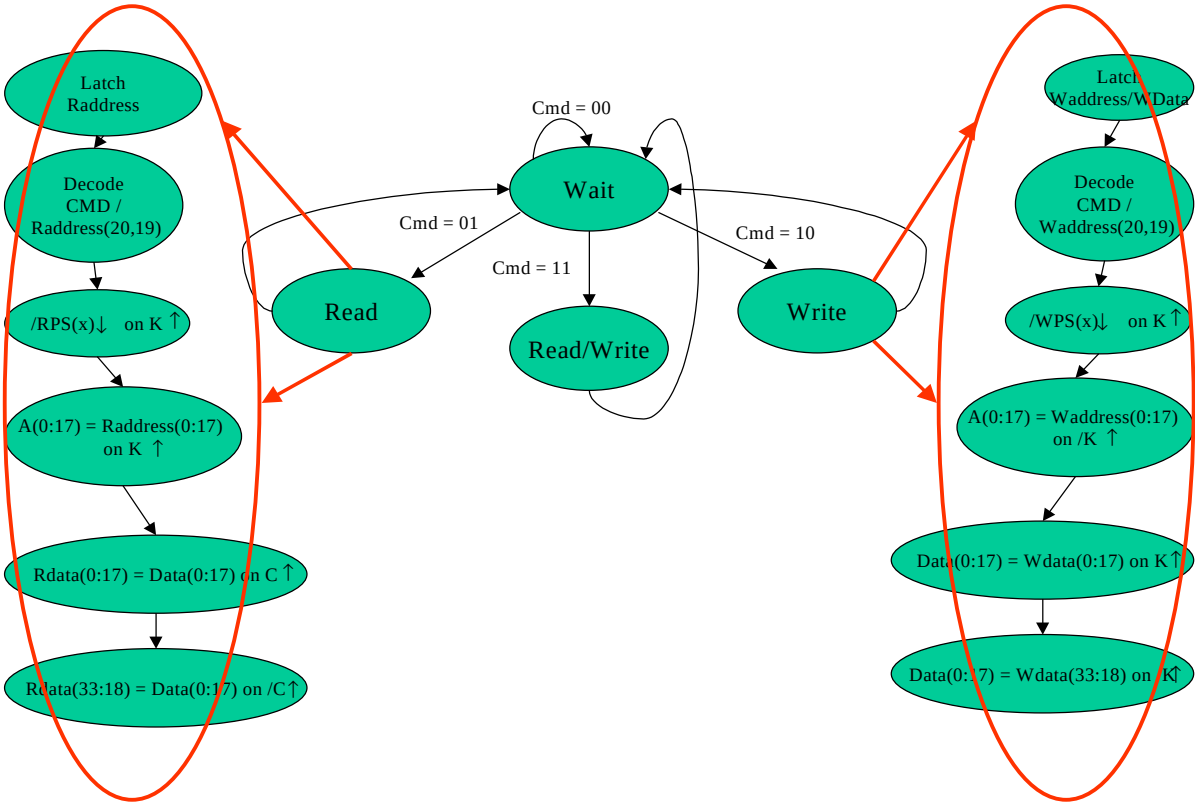


Figure 5. State Machine of the Memory Controller

Appendix2

```
--------------------------------------------------------------------------------
--
-- Quad Data Rate SRAM Controller
-- Altera Corporation
--
--------------------------------------------------------------------------------

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY qdr_ctrl1 IS
PORT (

-- User signals
      Cmd : IN STD_LOGIC_VECTOR(1 DOWNTO 0);-- command input: 00=wait, 01=read,
10=write, 11=read/write
      Raddr : IN STD_LOGIC_VECTOR(17 DOWNTO 0);-- read address input
      Waddr : IN STD_LOGIC_VECTOR(17 DOWNTO 0);-- write address input
      Wdata :   IN STD_LOGIC_VECTOR(35 DOWNTO 0);-- write data input
      Rdata :   OUT STD_LOGIC_VECTOR(35 DOWNTO 0);-- read data output
inclk: INSTD_LOGIC;-- system clock
pll_locked:OUT STD_LOGIC;-- PLL status output

-- QDR interface (all signals HSTL)
      K:   OUTSTD_LOGIC;  -- positive SRAM clock
      K_bar:   OUT STD_LOGIC;  -- negative SRAM clock
C: INSTD_LOGIC;-- positive feedback clock
      A : OUT STD_LOGIC_VECTOR(17 DOWNTO 0);-- address
      Din :   OUT STD_LOGIC_VECTOR(17 DOWNTO 0);-- data in to SRAM
      Dout : IN STD_LOGIC_VECTOR(17 DOWNTO 0);-- data out from SRAM
      WPS_bar :OUT STD_LOGIC;-- write port select
      RPS_bar:   OUT STD_LOGIC;-- read port select
      BWS_bar:   OUT STD_LOGIC_VECTOR(1 DOWNTO 0)-- byte write select

);
END qdr_ctrl1;

ARCHITECTURE behavioral OF qdr_ctrl1 IS
SIGNAL write_address, write_address_reg:  STD_LOGIC_VECTOR(17  DOWNTO  0);--
write address input register and pipeline register
SIGNAL write_data_p,  write_data_n: STD_LOGIC_VECTOR(17 DOWNTO 0);-- positive
and negative edge write data input registers
SIGNAL write_data_p_reg, write_data_n_reg:  STD_LOGIC_VECTOR(17  DOWNTO  0);--
write data pipeline register
SIGNAL temp_write_data, temp_write_data_reg : STD_LOGIC_VECTOR(17 DOWNTO 0);
-- write data output register and pipleine register
SIGNAL read_address, read_address_reg: STD_LOGIC_VECTOR(17 DOWNTO 0);-- read
address input register and pipeline register
SIGNAL read_data, read_data_hold : STD_LOGIC_VECTOR(35 DOWNTO 0);-- read data
input register and pipeline register
SIGNAL read_data_reg, read_data2_reg : STD_LOGIC_VECTOR(35 DOWNTO 0);-- read
data pipeline registers
SIGNAL temp_a, temp_a_reg: STD_LOGIC_VECTOR(17 DOWNTO 0); -- address output
register and pipeline register
SIGNAL cmd1, cmd2: STD_LOGIC_VECTOR(1 DOWNTO 0);-- command registers
SIGNAL clk2x, clk1x  : STD_LOGIC;-- 1x pipeline clock and 2x interface clock from PLL
SIGNAL clk_sel_p, clk_sel_n: STD_LOGIC;-- clocks used as select lines for data and ad-
dress muxes
SIGNAL clk_sel_p_temp, clk_sel_n_temp: STD_LOGIC;-- clocks used as select lines for
data and address muxes
SIGNAL pll_inclocken : STD_LOGIC;-- PLL enable signal
SIGNAL K_temp, K_bar_temp: STD_LOGIC;-- temporary signals for output clocks

COMPONENT pll1
PORT (
inclock: IN STD_LOGIC;
clock0: OUT STD_LOGIC;
clock1: OUT STD_LOGIC;
locked: OUT STD_LOGIC;
inclocken: IN STD_LOGIC
);
END COMPONENT;

COMPONENT clockgen
PORT (
in1x: IN STD_LOGIC;
in2x: IN STD_LOGIC;
clkout: OUT STD_LOGIC;
clkout_bar: OUT STD_LOGIC;
clk_sel_out : OUT STD_LOGIC;
clk_sel_out_bar : OUT STD_LOGIC
);
END COMPONENT;

COMPONENT controlgen
PORT (
clk2x: IN STD_LOGIC;
cmd : IN STD_LOGIC_VECTOR(1 downto 0);
BWS_bar : OUT STD_LOGIC_VECTOR(1 downto 0);
RPS_bar : OUT STD_LOGIC;
WPS_bar : OUT STD_LOGIC
);
END COMPONENT;

BEGIN

-- *** Clock and control generation ***

pll_one: pll1 -- ClockLock instantiation; generate 1x and 2x clocks
PORT MAP (
inclock => inclk,
clock0 => clk1x,
clock1 => clk2x,
locked => pll_locked,
inclocken => pll_inclocken
);

pll_inclocken <= '1'; -- always enable PLL

K_K_bar_generation: clockgen -- SRAM clock generation
PORT MAP (
in1x => clk1x,
in2x => clk2x,
clkout => K_temp,
clkout_bar => K_bar_temp,
clk_sel_out => clk_sel_p_temp,
clk_sel_out_bar => clk_sel_n_temp
);

K <= K_temp;
K_bar <= K_bar_temp;

clk_sel_gen: -- use K clock as select line for address and data muxes
PROCESS(clk2x)
BEGIN
   IF (clk2x'EVENT AND clk2x = '0') THEN
clk_sel_p <= clk_sel_p_temp;
clk_sel_n <= clk_sel_n_temp;
END IF;
END PROCESS clk_sel_gen;

BWS_RPS_WPS_generation: controlgen -- control signal generation
PORT MAP (
clk2x => clk2x,
cmd => cmd2,
bws_bar => bws_bar,
rps_bar => rps_bar,
wps_bar => wps_bar
);

Cmd_pos: -- save previous command signals
PROCESS(clk1x)
BEGIN
IF (clk1x'EVENT AND clk1x = '1') THEN
      cmd1 <= cmd;
      cmd2 <= cmd1;
 END IF;
   END PROCESS Cmd_pos;
```

```
-- *** Address path for reads and writes ***

Addr_inreg: -- clock in addresses from user on rising edge and feed into address pipeline
PROCESS(clk1x)
   BEGIN
IF (clk1x'EVENT AND clk1x = '1') THEN
      write_address <= Waddr;
      read_address <= Raddr;
write_address_reg <= write_address;
read_address_reg <= read_address;
  END IF;
   END PROCESS Addr_inreg;

A_outreg: -- transfer address signals to 2x domain and clock out to SRAM
PROCESS(clk2x) -- address must be valid on posedge K_bar for writes, posedge K for reads
BEGIN
IF (clk2x'EVENT AND clk2x = '0') THEN
IF (clk_sel_n = '0') THEN -- mux with inverse clock to allow for propagation time to SRAM
temp_A_reg <= write_address_reg;
ELSE
temp_A_reg <= read_address_reg;
END IF;
temp_A <= temp_A_reg;
END IF;
END PROCESS A_outreg;

A <= temp_A;
```

```
-- *** Data path for writes ***

Write_data_inreg: -- clock in data from user on rising edge and feed into data pipeline
PROCESS(clk1x)
   BEGIN
  IF (clk1x'EVENT AND clk1x = '1') THEN
      write_data_p <= Wdata(17 DOWNTO 0);
      write_data_n <= Wdata(35 DOWNTO 18);
write_data_p_reg <= write_data_p;
write_data_n_reg <= write_data_n;
  END IF;
   END PROCESS Write_data_inreg;

Din_outreg: -- transfer write data signals to 2x domain and clock out higher and lower bytes
on alternating clock cycles
PROCESS(clk2x)
BEGIN
IF (clk2x'EVENT AND clk2x = '0') THEN
IF (clk_sel_p = '1') THEN
temp_write_data_reg <= write_data_n_reg;
ELSE
temp_write_data_reg <= write_data_p_reg;
END IF;
temp_write_data <= temp_write_data_reg;
END IF;
END PROCESS Din_outreg;

Din <= temp_write_data;
```

```
-- *** Data path for reads ***

Read_data_neg: -- register lower data byte on falling edge of feedback clock
PROCESS(C)
   BEGIN
  IF (C'EVENT AND C = '0') THEN
Read_data_hold(17 DOWNTO 0) <= Dout;
read_data_reg <= read_data_hold;
  END IF;
   END PROCESS Read_data_neg;
```

```
Read_data_pos: -- register higher data byte on rising edge of feedback clock
PROCESS(C)
   BEGIN
  IF (C'EVENT AND C = '1') THEN
      Read_data_hold(35 DOWNTO 18) <= Dout;
  END IF;
   END PROCESS Read_data_pos;

Read_data_reg_pipeline: -- transfer read data back to system clock domain
PROCESS(clk1x)
BEGIN
IF (clk1x'EVENT AND clk1x = '1') THEN
read_data2_reg <= read_data_reg;
Read_data <= read_data2_reg;
END IF;
END PROCESS Read_data_reg_pipeline;

Rdata <= Read_data ;

END behavioral;
```