Implementing a 128Kx32 Dual-Port RAM Using the FLASH370™

Introduction

More and more communication systems require the use of very deep, high-speed dual-port memories to provide a common storage area for use between processors. System designers are looking for dual-port memories of 128 KByte and larger in size. These same systems are using 32-bit buses. These larger dual-port memories are not readily available as monolithic devices. As a result, the designer is left with the task of implementing these devices using discrete components. A full-featured implementation would include some static RAM combined with external support logic, arbitration, and control functions. This application note describes how to implement a 128K x 32-bit-wide dual-port memory or larger, using high-speed 1M SRAMs and a Cypress CPLD, the CY7C371. The CPLD, or Complex Programmable Logic Device, will be used to implement the memory control functions of the dual-port system and will be coded using VHDL.

Dual-Port Block Diagram

A good reference for the function and operation of a dual-port memory can be found in the application note in the Cypress *Applications Handbook* titled "Understanding Dual-Port RAMs." To reiterate, the block diagram of a standard dual-port memory is shown in *Figure 1*. This block diagram indicates the various blocks associated with a dual-port. There are four major blocks: the memory array, the

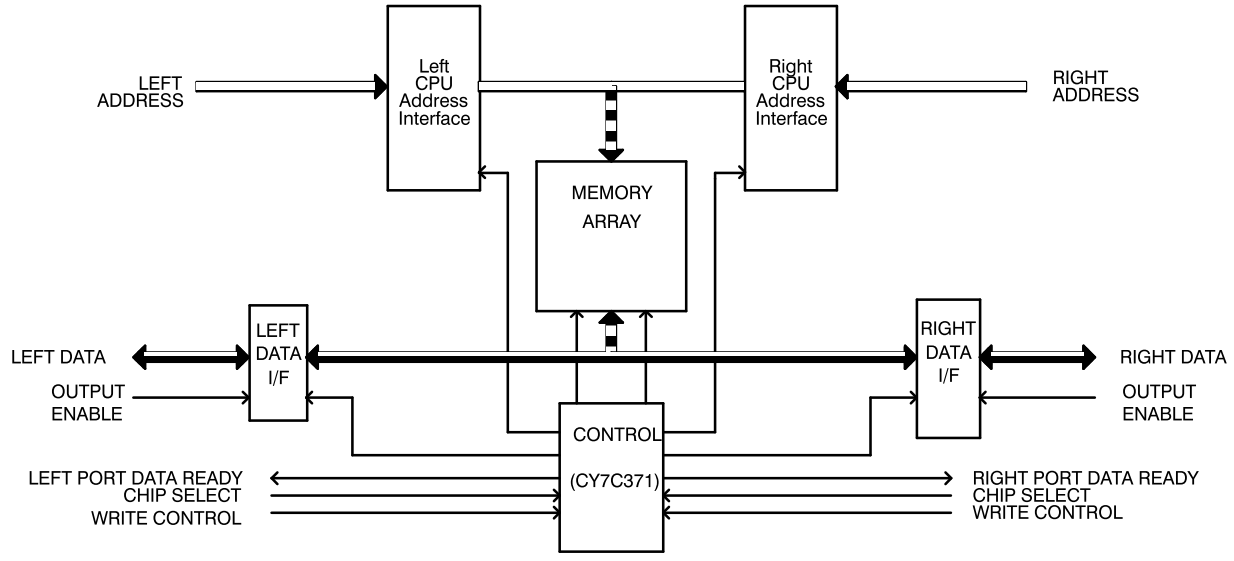


Figure 1. Dual-Port Memory Array Block Diagram

arbitration/control function, the right port or interface, and the left port or interface.

As can be seen from the block diagram in *Figure 1*, there are a series of signals that are required both internal and external to this system. The external signals are the normal signals that a monolithic dual-port chip would have. These are the signals that are labeled in the block diagram. The other signals are the internal signals that are used to allow the pieces of this dual-port system to communicate with one another. These are the address output enables for the address interface logic, the data output enable and the latch enable for the data interface logic, and the RAM output enable and write enable. These will be discussed in detail later.

The memory array consists of a single, standard SRAM or group of SRAMs to make up the overall array size. This array can be expanded in depth and width as needed. The arbitration/control logic accepts asynchronous read or write requests from each port or interface and sequences through a series of internal states that perform the read or write operation on the memory array. A CPLD is used in this example to implement this logic. The control logic must arbitrate between requests as well as synchronize the inputs to the internal clock frequency of the control function. The address buffers are used to isolate the address bus of the memory array from the left and right address ports. This allows the control-logic CPLD to select the correct address at the proper time. The bidirectional, latched data path allows data to be written to or read from the memory array. The data is also held in the latch during the remainder of the access.

Use of SRAM for Dual-Port

A 128Kx8 SRAM (like the Cypress CY7C109, 25-ns SRAM, as used in this note) was chosen here to implement a 128K x 32 sized array. Appendix A shows the schematic representation of the design. The array can be any size; this note shows this configuration because it depicts how to expand in the width direction. Cascading devices to expand the depth of the array is just as easily implemented. In either case, the contents of the control logic CPLD remain the same. The array could also be implemented with a single SRAM device if the array size warrants it.

A Brief Description of the CY7C371

The CY7C371 is a complex PLD with 32 macrocells, 32 I/O pins and 6 dedicated input pins (including 2 clock pins). The macrocells are grouped into two Logic Blocks of 16 macrocells each. There is a programmable interconnect matrix or PIM that connects the two logic blocks to the inputs and to each other. The macrocells themselves contain a register that can be configured as a T flip-flop, a D flip-flop, a level-triggered latch, or can be bypassed for combinatorial product terms. Each macrocell can support up to 16 product terms. For more detailed information on the CY7C371 and the whole FLASH370 family of CPLDs, please consult the application note "The FLASH370™ Family Of CPLDs and Designing with *Warp2*™" in the *Cypress Applications Handbook*.

The CY7C371 is well suited to this application. The dedicated inputs can be configured with a double registering mechanism to synchronize asynchronous signals so that they can be used synchronously inside the CPLD. The double registering will also dramatically reduce the chance of a metastable condition. The CPLD architecture is optimal for state machine designs and this arbiter requires three state machines to define it. The double-registered input configuration will be used in this example to resync the asynchronous chip select and write control inputs from both ports.

State Machine Design

The finite state machine that controls the dual-port memory array is really comprised of three "dependent" state machines operating concurrently as shown in *Figure 2*. Dependent state machines monitor or depend on the state of another state machine in order to change state. The first two machines, called "leftside" and "rightside," are identical. Their primary task is to monitor the interface of both ports. When the chip select input ($\overline{R\_CS}$ or $\overline{L\_CS}$) goes active (logic LOW), the appropriate machine advances from the Ready state to the

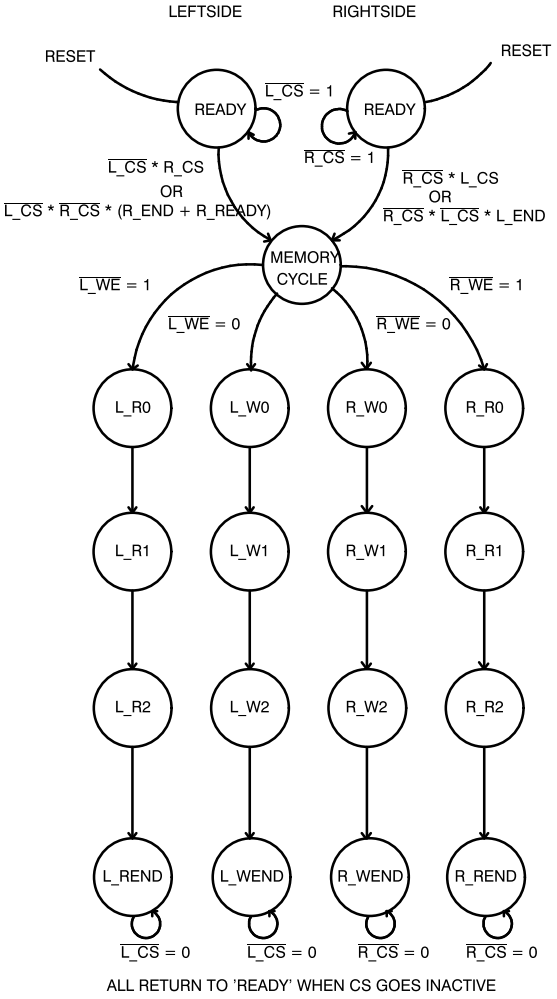


Figure 2. Memory Control Function State Machine

Memory Cycle state. The Memory Cycle state will start one of the memory access sequences. The length of each memory sequence (i.e., the number of state machine cycles) can be "tuned" to the access time of the SRAMs in the memory array. The memory cycle state machine will cycle back to the Ready state at the same time the memory access sequence ends and the select input goes inactive. It will either wait for a new request or start another memory access depending on the state of the other state machine ("leftside" or "rightside"). In the case where two requests are pending or appear at the same time, the left port gets priority. This means that the memory access for the left port is performed first. A READY signal (L_READY and R_READY) indicates when data is available on either port, it can only be active when either select input is active.

State Machine Implementation

The actual implementation of the state machines in the CY7C371 is done using VHDL. The structure of VHDL allows for simplification in coding these dependent state machines; the use of multiple processes and the CASE statement prove to be very powerful and efficient ways to perform this task.

Upon reset, both rightside and leftside state machines enter the Ready state and wait for a memory access. The leftside state machine will be used as an example. Both sides are identical at this point. Once a request is detected [for example $\overline{\text{L\_CS}}$ goes active (=0)], the leftside state machine transitions into a memory cycle. A priority scheme favoring the left port is encoded into the process for both state machines. If two accesses occur simultaneously, the left one is performed first. If one port request is detected before the other, it is completed while the other is held off. This extends the overall access time of the memory, but allows for "fair" operation. Each memory access sequence, Left Read, Left Write, Right Read, and Right Write, is comprised of four states. The four states (R0, R1, R2, REND or W0, W1, W2, WEND) run sequentially, one per clock cycle. They are there to allow the proper timing for the generation of control signals to the various components in the dual-port system. The REND or WEND state indicates the end of a memory cycle and is also a hold state if the CS is still active for that particular port. Once the REND or WEND state is reached and the CS is inactive, the state machine returns to the READY state and another access can be initiated.

CY7C371 Signals

A total of ten outputs are required to control the memory array and both the left and right ports. Refer to Appendix A for the 128K x 8 dual-port memory array schematic. The SRAM in the array is controlled by $\overline{\text{RAM\_OE}}$ and $\overline{\text{RAM\_WE}}$. The $\overline{\text{RAM\_OE}}$ signal is created when either port executes a read successfully. Therefore, the $\overline{\text{RAM\_OE}}$

signal is enabled during either read sequence only during the R0 through R2 cycles. Writes to the SRAM are controlled by the write state machine for either port. The $\overline{\text{RAM\_WE}}$ is generated for either port during the W1 and W2 cycles of a write access only. The port address inputs are isolated from the memory array by a set of 74FCT244Ts. The left port is controlled by $\overline{\text{L\_ADD\_OE}}$ and is generated during the left memory access sequence states 0 through 2 for either a read or a write to the left port. The right port address is controlled in the same manner, by using the right memory access sequence states 0 through 2. The data buffer functions are implemented using 74FCT543Ts with the "B" (HIGH current) side interfaced to the outside and the "A" side interfaced to the memory array. During reads, the latch enables ($\overline{\text{L\_LAT\_EN}}$, $\overline{\text{R\_LAT\_EN}}$) are used to hold the data read from the array in the latches. The output enables ($\overline{\text{L\_OE}}$, $\overline{\text{R\_OE}}$) are then driven directly to access the read data. During writes, the output enables ($\overline{\text{L\_DAT\_OE}}$, $\overline{\text{R\_DAT\_OE}}$) are used to allow the data to pass from the outside into the memory array. These output enables and latch enables are controlled by the OR of the appropriate memory access sequence states. Mealy outputs are used for the L_READY and R_READY signals. These outputs are active whenever the respective state machine is in state 2 and the CS is active. Using Mealy outputs here allows the ready signal to go inactive as soon as the CS input ($\overline{\text{L\_CS}}$ or $\overline{\text{R\_CS}}$) goes inactive instead of waiting for the state machine to transition back to the READY state.

VHDL Code for Controller in 371

Appendix B contains the VHDL code used for the CY7C371 in this design. This code was compiled with the Cypress *Warp2* tool and targeted for the CY7C371 to generate the programming (JEDEC) and simulation file(s). The Nova simulator in the *Warp2* tool was used to verify the design. For details on these tools please refer to the *Warp2 User's Guide*. Furthermore, a thorough explanation of VHDL constructs can be found in the *Warp2 Reference Manual*.

The code in Appendix B starts out by defining the inputs and outputs and the internal signals required. The first process is for the Chip Select and Write Enable resync. This is where the double registering occurs, as mentioned in the description of the CY7C371 earlier in this application note. The next process is where the state machine definitions start. It begins by defining the rightside state machine and uses a separate process to define the leftside state machine. Buried within each of these processes is the Memory Cycle state machine for the READ and WRITE cycles of each port. The next process is used to define the RAM_OE and RAM_WE for the memory array control. This is a simple IF-THEN-ELSE clause. The last process is used to generate the signal which gets used in the Mealy equations for the leading edge of the L_READY and R_READY signals. Lastly, the L_READY and R_READY signals are defined outside of a process by gating state2 with the CS input.

Performance Evaluation

To evaluate the performance of this dual-port system, three different timing scenarios were looked at. The first scenario is for an unarbitrated access from either port. This assumes that both port state machines are in the Ready state and only one access occurs. The second scenario involves the right port being granted access shortly before the left port, forcing the left port to wait. The third involves simultaneous accesses from each port. In this case the left side has priority (by design) and the right side is held off. These cases are shown in the following three timing diagrams (*Figures 3*, *4*, and *5*). From these it is possible to determine the timing of each access by counting the number of clock cycles for each scenario.

Table 1 lists the number of clock cycles for each of the three cases of *Figures 3*, *4*, and *5*. These numbers reflect the worst case situations for Case #2 and #3 where the maximum possible delay is assumed.

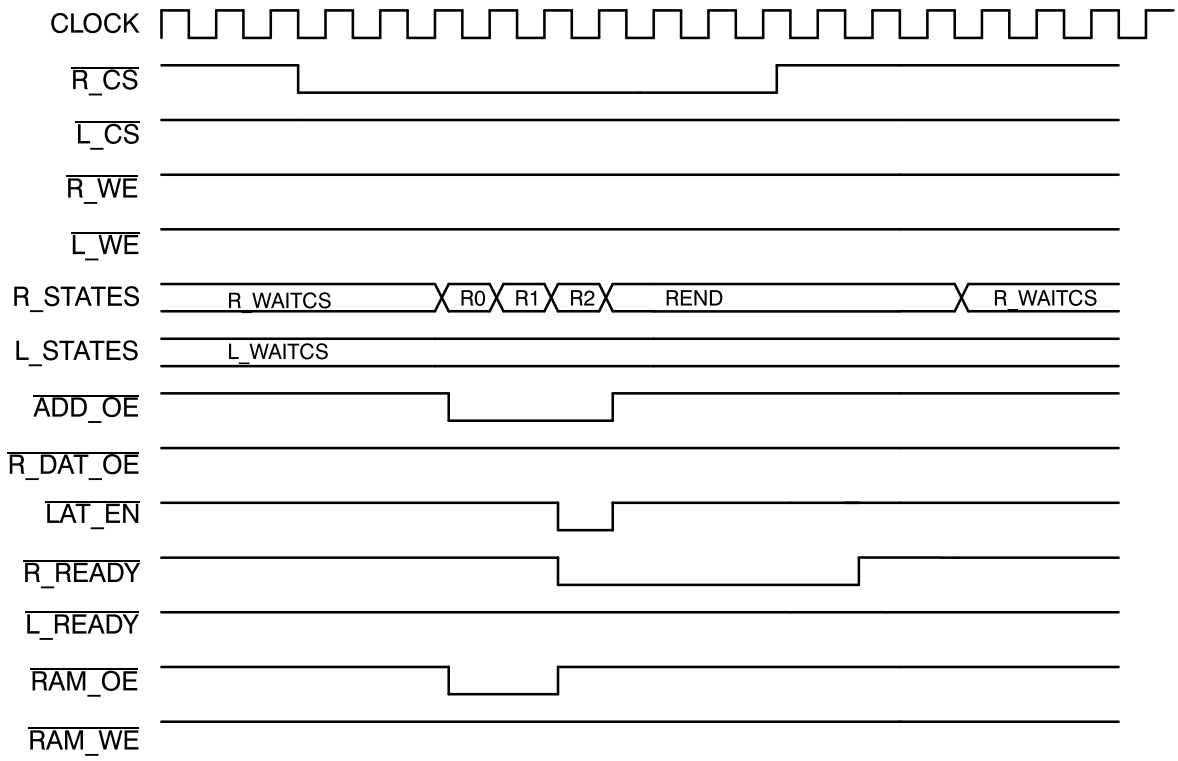


Figure 3. Timing Diagram—Unarbitrated Access From Right Port

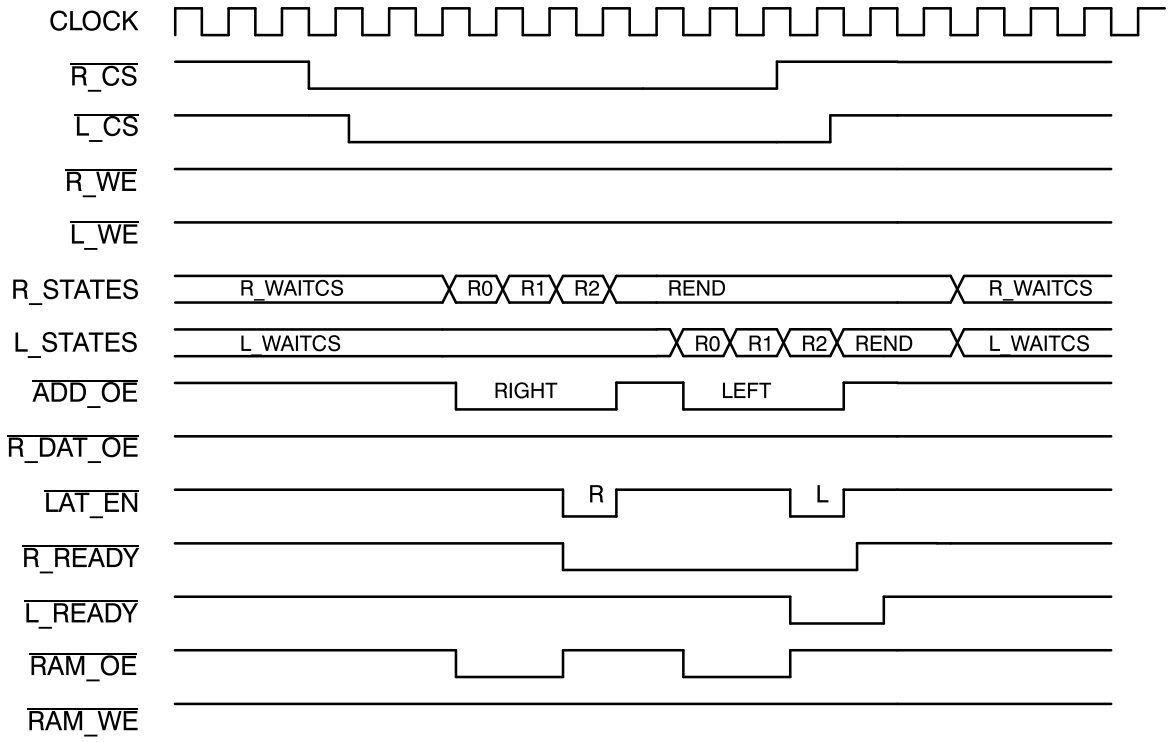


Figure 4. Timing Diagram—Right Port Access Before Left Port

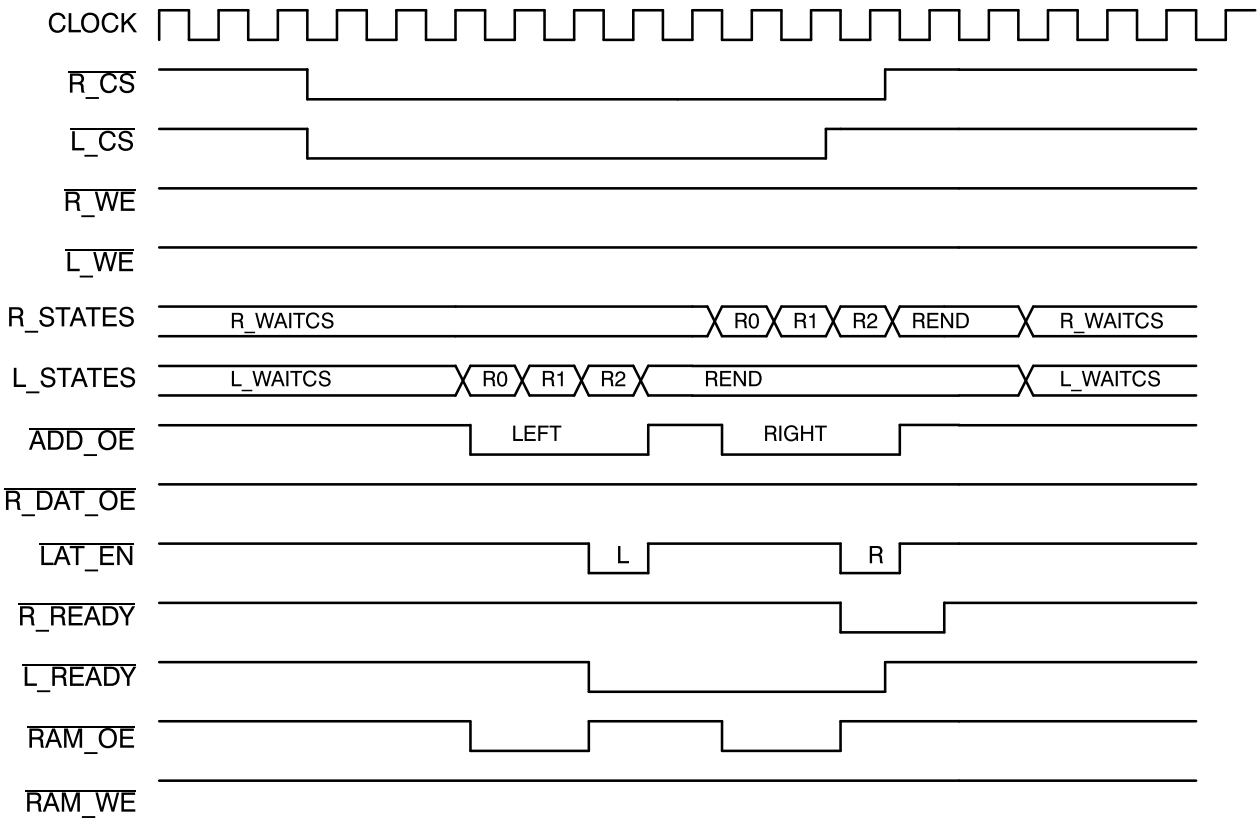


Figure 5. Timing Diagram—Simultaneous Access

Table 1. Access Time in Clock Cycles

Timing Parameter	Case #1	Case #2	Case #3
LEFT			
Input Set-Up Timing	2 clocks	Note 1	2 clocks
Arbitration Cycle	1 clock	Note 1	1 clock
Memory Access	3 clocks	3 clocks	3 clocks
Latch Hold Cycle	1 clock	1 clock	1 clock
Total Number of Clock Cycles	7 clocks	11 clocks	7 clocks
RIGHT			
Input Set-Up Timing	N/A[2]	2 clocks	Note 1
Arbitration Cycle	N/A	1 clock	Note 1
Memory Access	N/A	3 clocks	3 clocks
Latch Hold Cycle	N/A	1 clock	1 clock
Total Number of Clock Cycles	N/A	7 clocks	11 clocks

Notes:

1. Worst case input set-up timing and arbitration cycle assumes 7 clock access delay on opposite port.
2. N\A means No Activity on this port.

To calculate the access time in nanoseconds, the following formula is applied:

$t_{ACC} = t_{IS371} + [t_{CYC371} \times \#clocks] + t_{PD543}$

Where:

t_{ACC} = total access time

t_{IS371} = CY7C371 input register set-up time = 2 ns

t_{CYC371} = clock cycle of CY7C371 = 7 ns

#clocks = number of clocks from *Table 1*

t_{PD543} = 74FCT543CT transparent to latched propagation delay = 7 ns

Since the CY7C371 inputs are double registered, two clock cycles are required to resync the Chip Select and Write Enable inputs. If the input set-up timing can be guaranteed, this internal delay of two cycles can be eliminated by using single- or non-registered inputs.

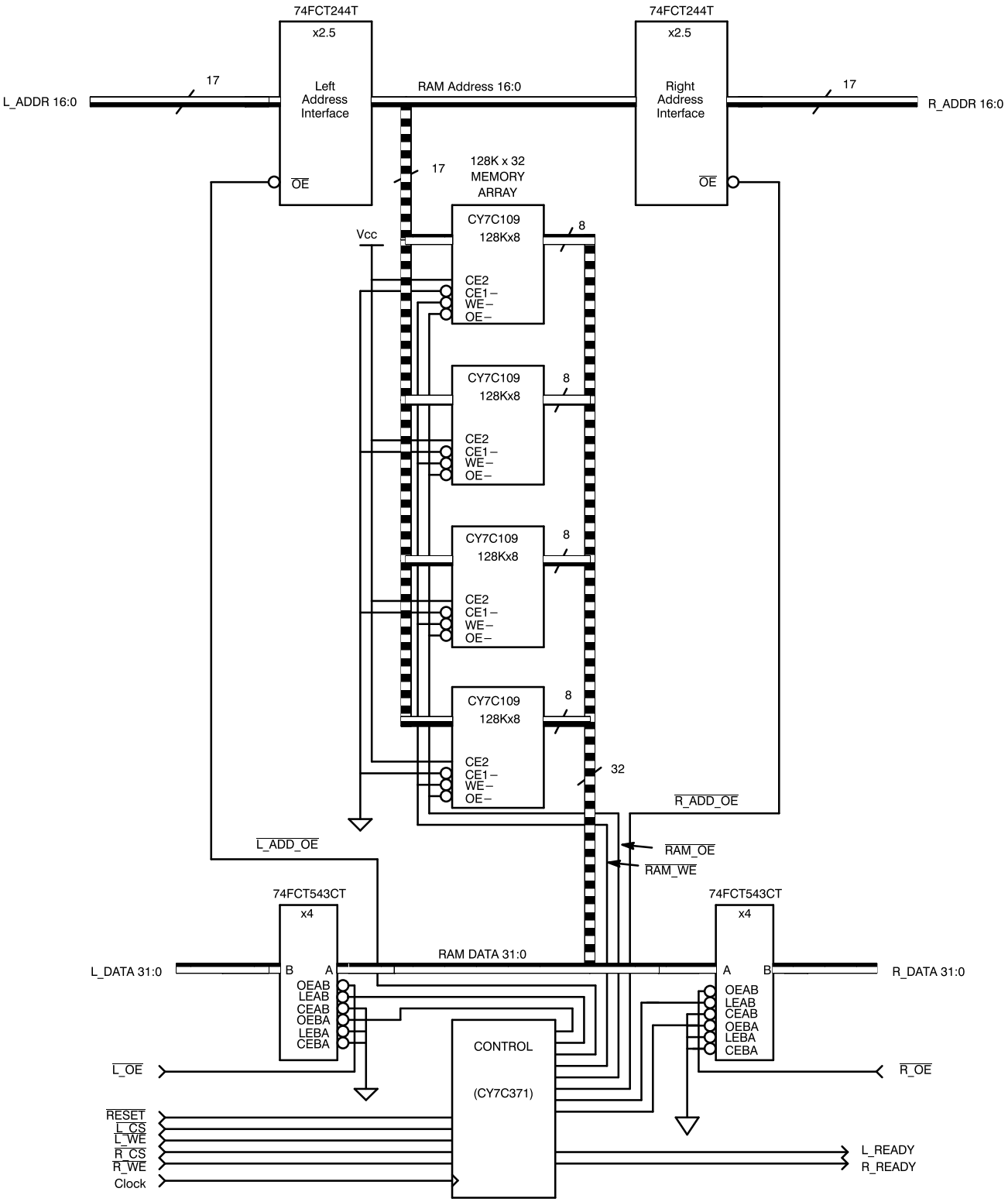
Memory Expansion

The example used here shows that an array of any size can be easily implemented. The addition of memories and associated address buffers makes depth expansion easy. The width may also be increased by cascading memories and adding additional buffers. Both techniques would be utilized to expand in depth and width. These enhancements are possible without making any changes to the CY7C371 Control Function PLD design. Likewise this design could implement a smaller array than shown here, again without revising the CY7C371.

Summary

This application note has demonstrated the implementation of a large asynchronous dual-port memory array by utilizing standard memory and logic devices and the CY7C371. The performance of this design is limited by various factors. The access time of the SRAM and the clock speed of the CY7C371 used are two factors that could improve performance without changing the VHDL code for the CY7C371. Another option would require some design changes, though minor. Making one or both ports synchronous with respect to the CPU would eliminate the two-clock delay associated with the resync function of the CY7C371. The implementation of these improvements offers the designer a few options to tailor the design to fit specific system requirements and achieve the desired level of performance.

Appendix A. Schematic



Appendix B. VHDL Code for Controller

```
-- Dual-port memory controller

ENTITY dpram IS
        PORT (clock, r_we_n, r_cs_n, l_we_n, l_cs_n, reset_n: IN BIT;    --INPUTS
              ram_oe_n, ram_we_n : OUT BIT;                              --OUTPUTS
              r_ready, r_add_oe, r_dat_oe, r_lat_en : OUT BIT;
              l_ready, l_add_oe, l_dat_oe, l_lat_en : OUT BIT
              );
END dpram;

-------------------------------------------------------------------------
USE work.rtlpkg.all;

ARCHITECTURE ARCHdpram OF dpram IS
  TYPE ctrl_states IS (waitcs, r0, r1, r2, rend, w0, w1, w2, wend);      --Internal signal declaration
  SIGNAL rightside, leftside : ctrl_states;
  SIGNAL r_we_ndd, r_we_nd, l_we_ndd, l_we_nd : BIT;
  SIGNAL r_cs_ndd, r_cs_nd, l_cs_ndd, l_cs_nd : BIT;
  SIGNAL r_ready_int, l_ready_int : BIT;
BEGIN

--Double register the input we and cs signals for sync & metastability hardening
  PROCESS BEGIN
    WAIT UNTIL clock = '1';
      r_we_ndd <= r_we_nd; r_we_nd <= r_we_n;
      l_we_ndd <= l_we_nd; l_we_nd <= l_we_n;
      r_cs_ndd <= r_cs_nd; r_cs_nd <= r_cs_n;
      l_cs_ndd <= l_cs_nd; l_cs_nd <= l_cs_n;
  END PROCESS;

--RIGHTSIDE STATE MACHINE
  PROCESS BEGIN
    WAIT UNTIL clock = '1';
      CASE rightside IS
        WHEN waitcs =>
          r_add_oe <= '1'; r_dat_oe <= '1'; r_lat_en <= '1';
--goto state 0 if : r_cs is active + L_cs  inactive or r_cs active + (l_cs active but at end)
          IF (((r_cs_ndd = '0') AND (l_cs_ndd = '1')) OR
              ((r_cs_ndd = '0') AND (l_cs_ndd = '0') AND
                ((leftside = wend) OR (leftside = rend)))) THEN
--start write state machine if WE active
            IF r_we_ndd = '0' THEN
              rightside <= w0;
              r_add_oe <= '0'; r_dat_oe <= '0'; r_lat_en <= '1';
            ELSE
--start read state machine if WE inactive
              rightside <= r0;
              r_add_oe <= '0'; r_dat_oe <= '1'; r_lat_en <= '1';
            END IF;
          ELSE
            rightside <= waitcs;
          END IF;

--RIGHTSIDE READ STATE MACHINE
        WHEN r0 =>
          rightside <= r1;
          r_add_oe <= '0'; r_dat_oe <= '1'; r_lat_en <= '1';
        WHEN r1 =>
          rightside <= r2;
          r_add_oe <= '0'; r_dat_oe <= '1'; r_lat_en <= '0';
        WHEN r2 =>
          rightside <= rend;
          r_add_oe <= '1'; r_dat_oe <= '1'; r_lat_en <= '1';
```

Appendix B. VHDL Code for Controller (continued)

```
        WHEN rend =>
          r_add_oe <= '1'; r_dat_oe <= '1'; r_lat_en <= '1';
          IF r_cs_ndd = '1' THEN
            rightside <= waitcs;
          ELSE
            rightside <= rend;
          END IF;

--RIGHTSIDE WRITE STATE MACHINE
        WHEN w0 =>
          rightside <= w1;
          r_add_oe <= '0'; r_dat_oe <= '0'; r_lat_en <= '1';
        WHEN w1 =>
          rightside <= w2;
          r_add_oe <= '0'; r_dat_oe <= '0'; r_lat_en <= '1';
        WHEN w2 =>
          rightside <= wend;
          r_add_oe <= '1'; r_dat_oe <= '1'; r_lat_en <= '1';
        WHEN wend =>
          r_add_oe <= '1'; r_dat_oe <= '1'; r_lat_en <= '1';
          IF r_cs_ndd = '1' THEN
            rightside <= waitcs;
          ELSE
            rightside <= wend;
          END IF;
        WHEN others =>
          rightside <= waitcs;
          r_add_oe <= '1'; r_dat_oe <= '1'; r_lat_en <= '1';
      END CASE;
  END PROCESS;

--LEFTSIDE STATE MACHINE
  PROCESS BEGIN
    WAIT UNTIL clock = '1';
      CASE leftside IS
        WHEN waitcs =>
          l_add_oe <= '1'; l_dat_oe <= '1'; l_lat_en <= '1';
--goto state 0 if l_cs is active + r_cs is inactive or l_cs active + (r_cs active but at end or in waitcs
 state)
          IF (((l_cs_ndd = '0') AND (r_cs_ndd = '1')) OR
              ((l_cs_ndd = '0') AND (r_cs_ndd = '0') AND
                ((rightside = wend) OR (rightside = rend) OR (rightside = waitcs)))) THEN
--start write state machine if WE active
            IF l_we_ndd = '0' THEN
              leftside <= w0;
              l_add_oe <= '0'; l_dat_oe <= '0'; l_lat_en <= '1';
            ELSE
--start read state machine if WE inactive
              leftside <= r0;
              l_add_oe <= '0'; l_dat_oe <= '1'; l_lat_en <= '1';
            END IF;
          ELSE
            leftside <= waitcs;
          END IF;

--LEFTSIDE READ STATE MACHINE
        WHEN r0 =>
          leftside <= r1;
          l_add_oe <= '0'; l_dat_oe <= '1'; l_lat_en <= '1';
        WHEN r1 =>
          leftside <= r2;
          l_add_oe <= '0'; l_dat_oe <= '1'; l_lat_en <= '0';
        WHEN r2 =>
          leftside <= rend;
          l_add_oe <= '1'; l_dat_oe <= '1'; l_lat_en <= '1';
```

Appendix B. VHDL Code for Controller (continued)

```
        WHEN rend =>
          l_add_oe <= '1'; l_dat_oe <= '1'; l_lat_en <= '1';
          IF l_cs_ndd = '1' THEN
            leftside <= waitcs;
          ELSE
            leftside <= rend;
          END IF;

--LEFTSIDE WRITE STATE MACHINE
        WHEN w0 =>
          leftside <= w1;
          l_add_oe <= '0'; l_dat_oe <= '0'; l_lat_en <= '1';
        WHEN w1 =>
          leftside <= w2;
          l_add_oe <= '0'; l_dat_oe <= '0'; l_lat_en <= '1';
        WHEN w2 =>
          leftside <= wend;
          l_add_oe <= '1'; l_dat_oe <= '1'; l_lat_en <= '1';
        WHEN wend =>
          l_add_oe <= '1'; l_dat_oe <= '1'; l_lat_en <= '1';
          IF l_cs_ndd = '1' THEN
            leftside <= waitcs;
          ELSE
            leftside <= wend;
          END IF;
        WHEN others =>
          leftside <= waitcs;
          l_add_oe <= '1'; l_dat_oe <= '1'; l_lat_en <= '1';
      END CASE;
  END PROCESS;

--RAM_OE and RAM_WE control signal logic
  PROCESS BEGIN
    WAIT UNTIL clock = '1';
          IF (((rightside = waitcs) AND ((((r_cs_ndd = '0') AND (l_cs_ndd = '1') AND (r_we_ndd = '1')) OR
              ((r_cs_ndd = '0') AND (l_cs_ndd = '0') AND (r_we_ndd = '1') AND
                ((leftside = wend) OR (leftside = rend))))))
             OR
             ((leftside = waitcs) AND (((l_cs_ndd = '0') AND (r_cs_ndd = '1') AND (l_we_ndd = '1')) OR
              ((l_cs_ndd = '0') AND (r_cs_ndd = '0') AND (l_we_ndd = '1') AND
                ((rightside = wend) OR (rightside = rend) OR (rightside = waitcs)))))
             OR
             (rightside = r0) OR (rightside = r1)
             OR
             (leftside = r0) OR (leftside = r1))
          THEN
            ram_oe_n <= '0';
          ELSE
            ram_oe_n <= '1';
          END IF;

          IF ((leftside = w0) OR (leftside = w1) OR
              (rightside = w0) OR (rightside = w1))
          THEN
            ram_we_n <= '0';
          ELSE
            ram_we_n <= '1';
          END IF;
  END PROCESS;
```

Appendix B. VHDL Code for Controller (continued)

```
--READY signal logic for leading edge of signal
  PROCESS BEGIN
    WAIT UNTIL clock = '1';
      IF ((rightside = r1) OR (rightside = w1)) THEN
        r_ready_int <= '0';
      END IF;
      IF ((r_cs_nd = '1') OR (reset_n = '0')) THEN
        r_ready_int <= '1';
      END IF;

      IF ((leftside = r1) OR (leftside = w1)) THEN
        l_ready_int <= '0';
      END IF;
      IF ((l_cs_nd = '1') OR (reset_n = '0')) THEN
        l_ready_int <= '1';
      END IF;
  END PROCESS;

--MEALY outputs for READY signal to turn off as soon as CS goes inactive
  l_ready <= '0' WHEN ((l_ready_int = '0') AND (l_cs_nd = '0')) ELSE '1';
  r_ready <= '0' WHEN ((r_ready_int = '0') AND (r_cs_nd = '0')) ELSE '1';

END ARCHdpram;
```