Application Note

AN1846/D
Rev. 1.2, 9/2003

Designing an MPC107 Local-Bus Slave Interface

Gary Milliorn
CPD Applications

This application note describes the steps for designing an interface device that provides access to I/O and memory peripherals using the local-bus slave features of the MPC107 PCI Bridge/Memory controllers. The MPC107 and local-bus slave (LBS) feature may be used with any Freescale microprocessor that implements the PowerPC™ 60x bus protocol. These devices include the following products:

- MPC603, MPC603e, MPC603ev
- MPC740, MPC745, MPC750, MPC755
- MPC7400, MPC7410
- MPC7441, MPC7445, MPC7450, MPC7451, MPC7455

This application note covers the following topics:

To see published errata or updates to this document, refer to the web site at http://www.freescale.com/semiconductors.

1 Introduction

The MPC107 (and the MPC106) provide support for an interface called the *local-bus slave* (LBS) that allows devices to be attached more easily to the high-speed 60x bus. The LBS can respond to read and write cycles while relying upon the MPC107 to provide the majority of the interface controls. An LBS device monitors a subset of the 60x bus and asserts $\overline{\text{TA}}$ when the read or write operation is completed. A device that provides local-bus slave I/O and memory access is outlined in the remaining sections of this application note, and is called the Applications Engineering Input/Output Unit (AEIOU). For the rest of this document, the MPC107 refers to either the MPC107 or the MPC106; the information contained herein applies to both devices unless otherwise stated.

NOTE

The VHDL in this application note was compiled and verified using a software test bench, but was not verified in hardware. The AEIOU can contain significant errors, and the 60x bus test bench might not have revealed latent errors in the VHDL code that is presented in this document. Treat this application note as general design information for creating an LBS I/O controller, rather than as a drop-in component.

NOTE

All the software contained in this application note is copyrighted 2003 by Freescale, and Freescale customers may use it freely as long as the copyright notice remains present in each literal or derived module or source file. The code may be freely modified to suit customized applications.

2 Conventions

This application note refers to both hardware and software signals (pins and nets) and components (physical and logical). Table 1 shows typographic conventions that are used.

Table 1. Typographic Conventions

Class	Example	Typography	Description
Hardware	TT(0:4)	Uppercase, no overbar	Hardware pin that is asserted when active high
	$\overline{\text{TA}}$	Uppercase, with overbar	Hardware pin that is asserted when active low
Software	done	Lowercase	Internal signal that is asserted when logic '1' or 'H' (high)
	go_L	Lowercase, '_L' appended	Internal signal that is asserted when logic '0' or 'L' (low)
	TT0	Uppercase	Internal signal that is asserted when logic '1' or 'H' and drives an external hardware pin of the same name
	TA_L	Uppercase, '_L' appended	Internal signal that is asserted when logic '0' or 'L' and drives an external hardware pin of the same name
	BYTEW	Uppercase, bold	Name of a VHDL entity or module

In VHDL, a signal may be high when set to the value 'H' or '1'; the former is a 'weak' high-level that other signals can override, while the latter is a ‘strong’ high level that cannot be overridden (see Table 2).

Table 2. IEEE 1164 Logic Conventions

Example	Description	Usage
‘0’	Logic low, forcing	Internally and on outputs that are not shared (for example, $\overline{\text{LBCLAIM}}$)
‘1’	Logic high, forcing	Internally and on outputs that are not shared
‘L’	Logic low, weak	On outputs that are shared (for example, $\overline{\text{TA}}$)
‘H’	Logic high, weak	On outputs that are shared

3 Local-Bus Slave Architecture

The MPC107 provides 60x-bus arbitration for itself, a 60x-bus microprocessor, and optionally an additional processor. Typically, the MPC107 claims all non-snoop cycles for itself and forwards them to the PCI bus, the memory controller, or asserts an error signal. The MPC107 provides an input signal called $\overline{\text{LBCLAIM}}$ so that another 60x-bus device can claim a bus cycle. If asserted during the address phase of a bus transaction, the MPC107 handles the termination of the address phase, but not the data phase of the transaction. Instead, the MPC107 disconnects from the data phase and waits until the LBS device completes that portion.

The LBS is called a *slave* because it cannot initiate bus transactions on its own (bus mastery); instead, it relies upon an external master (principally the processor) to initiate a transaction specifically to it. This reliance can limit the architectures in which an LBS is suitable, but for many applications bus mastering may not be a concern. Furthermore, using software to initiate bus transactions (for example, using interrupts/exceptions to trigger bus operations from program- or device-initiated loads and stores to which the LBS can respond) is still possible. Figure 1. shows the general architecture of an LBS system.

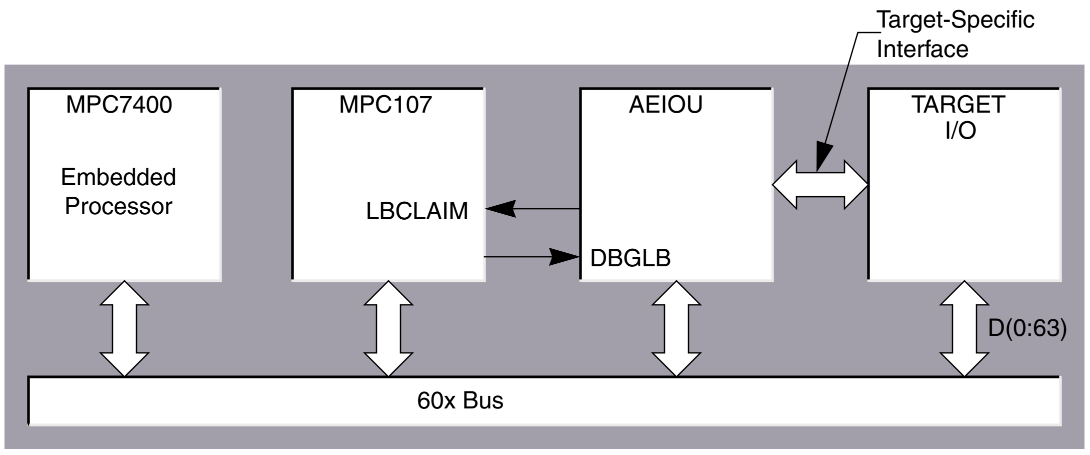

Figure 1. Local Bus Slave Architecture.

The LBS communicates with the 60x bus using the same signals as any other device (A(0:31), D(0:63), TS, TT, TSIZ, AACK, and so on), though not all are required for the subset an LBS may use. Two additional side-band signals between the LBS and the MPC107 ($\overline{\text{LBCLAIM}}$ and $\overline{\text{DBGLB}}$) can be used respectively to claim and be granted a cycle.

Figure 2 shows the general sequence of a LBS cycle.

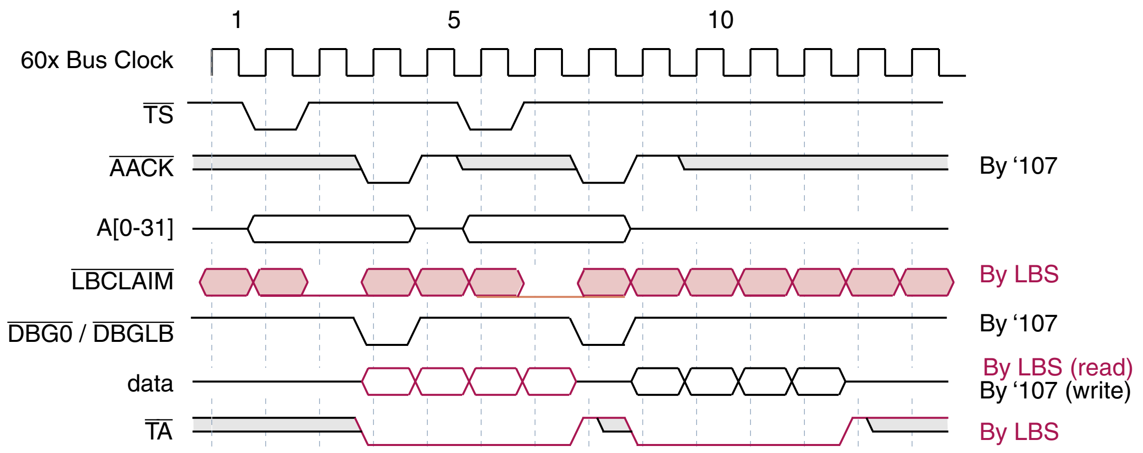

Figure 2. LBS Transaction.

One complication in the design of an LBS controller is that the 60x-bus implements separate address and data tenures. The address bus is not tightly coupled to the data bus, and while data is being transferred, the address of that data may no longer be present on the address bus. Figure 2 shows an example where in cycle #6 $\overline{\text{TS}}$ is asserted for a new transaction while data is still transferred. The effect of split tenures is that the LBS controller usually requires the capability of storing the current address, size, and transfer type information of the current cycle. Because the MPC107 does not allow the 60x-bus protocol to pipeline more than one address tenure, the information storage requirements are modest.

NOTE

Although the 60x-bus does allow the control of the address and data tenure to overlap (by delaying the assertion of $\overline{\text{AACK}}$), the MPC107 does not have this facility and cannot be used with an LBS interface. When the LBS claims a cycle, the MPC107 asserts $\overline{\text{AACK}}$ as soon as possible, either immediately (if the system bus is idle) or after the preceding data tenure is completed (if pipelining has already occurred (shown in cycle #8 of Figure 2). Consequently, LBS controllers must accept the possibility of pipelined addresses. This problem has two solutions:

- Implement address tenure data storage. In this solution, as each local-bus cycle is claimed, all needed information is stored in a register. This approach is relatively easy and inexpensive (in an ASIC or FPGA), and the 60x bus interface guarantees that no more than one address cycle is pending.
- The second solution is more simple and less expensive, but moves the complexity from the hardware into the software. The system and software expect that any cycle after an LBS I/O cycle may be missed (unclaimable). To guarantee that LBS-targeted cycles are not performed back-to-back, software must either allow other instructions to run or perform a write to a non-LBS address. A read cycle is not effective because the PowerPC instruction-set architecture allows loads to bypass stores under certain circumstances. A 'sync' instruction is not effective either because it causes the MPC107 to flush its internal buffers, which could trigger a PCI-to-local-bus snoop transaction. A dummy write to an unused memory location usually suffices.

3.1 Coherency

To keep the LBS design simple for I/O-controller purposes, the design assumes that accesses to the LBS-controlled addresses are coherent. The system does not expect the LBS to snoop the system bus to supply cached data to external devices, nor that it invalidates internally cached data. This assumption does not imply that LBS-controlled devices must be non-cacheable. (This restriction on the 60x-bus would imply that burst transfers are not allowed.) It means only that if the LBS-controlled devices are cacheable, the I/O software must enforce coherency if required. In this design, the AEIOU includes a pipelined burst SRAM controller that requires the ability to accept burst transfers.

The *MPC106 PCI Bridge/Memory Controller User’s Manual* (see the section about 60x local bus slave support) and the *MPC107 PCI Bridge/Memory Controller User’s Manual* present a complex state machine for tracking the 60x-bus state. The state machine logic is necessary only when the LBS must maintain cache coherency with external bus masters, or with the external L2 cache controller for the MPC106. For the purposes of this application note, coherency may be disregarded. (I/O controllers are often required to be non-cacheable), obviating the need for coherency.)

4 Interactions between the LBS and Memory

Some interactions occur between the LBS interface of the MPC107 and the memory controller. When the MPC107 is programmed to trap on illegal memory operations known as *memory select errors* (see Chapter 13 of the *MPC107 PCI Bridge/Memory Controller User’s Manual* for details on error handling), the memory controller interferes with LBS operations. To avoid this interference, observe the following restrictions:

- If memory select errors were enabled by setting the ErrEnR1[MSE] bit, the address chosen for the LBS must be in the range 0-1 Gbytes (0-0x3FFF_FFFF). Furthermore, the selected range for LBS accesses must be stored into an unused memory boundary register (one of eight bit fields in the MSAR/EMSAR/MEAR/EMEAR registers). This restriction implies that the memory cannot use eight physical banks of memory because one must be reserved for the LBS.
- If memory select errors are not enabled, the address chosen for the LBS may be anywhere from 0-4 Gbytes (0-0xFFFF_FFFF), including the ROM and extended ROM areas, except for the PCI configuration address and the interrupt acknowledge address.

When the above restrictions are followed, the MPC107 operates properly with an activated memory controller. The MPC106 cannot place an LBS anywhere above 2Gbytes.

The second issue is that the MPC106 (and only the MPC106) multiplexes the SDRAM clock enable signal (CKE) with the $\overline{\text{DBGLB}}$ signal. Therefore, SDRAM-based systems that use local bus slaves must provide a data bus grant signal to the local bus slave by an alternate means. In a uniprocessor system, the $\overline{\text{DBG0}}$ signal can be used for $\overline{\text{DBGLB}}$. In a multiprocessor system, $\overline{\text{DBG}}$[0–3] can be logically ANDed to create a suitable $\overline{\text{DBGLB}}$ signal. Note that using these methods to provide a data bus grant signal for the local bus slave is incompatible with the external L2 interface of the MPC106. Therefore, SDRAM-based systems that use local bus slaves cannot use an external L2 cache. Note that this restriction refers only to the external 60x-bus-based MPC106-controlled L2 cache, and not the MPC75x/MPC74xx “backside” L2/L3 cache interfaces.

An example of a two-processor connection is shown in Figure 3.

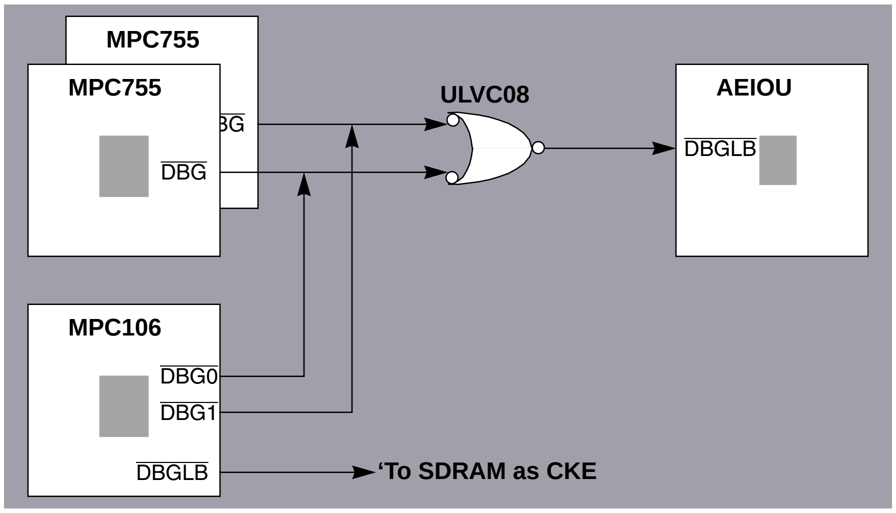

Figure 3. DBGLB Recovery Logic for the MPC106.

Note that $\overline{\text{DBGLB}}$ recovery logic requires that the LBS interface does not drive the data bus unless it has also decoded an LBS transaction. This logic is implemented in the design of the AEIOU. The MPC107 does not require this logic.

5 AEIOU Architecture

This section defines the architecture of the AEIOU. Most real-world applications of the AEIOU should be highly customized for the target system; but here a common set of features is provided as follows:

- Address tenure storage (hardware overlap control)
- General purpose I/O port (8 inputs, 8 outputs)
- 8-bit register file (7 read/write registers, 1 read-only ID register)
- External pipelined burst SRAM interface (chip-select, write strobe and output enable)

The AEIOU implementation for this application note provides an interface to these I/O devices to demonstrate the flexibility of the LBS I/O interface.

Figure 4 shows the general block diagram with connections to the 60x bus on the left and the connections to the I/O devices on the right.

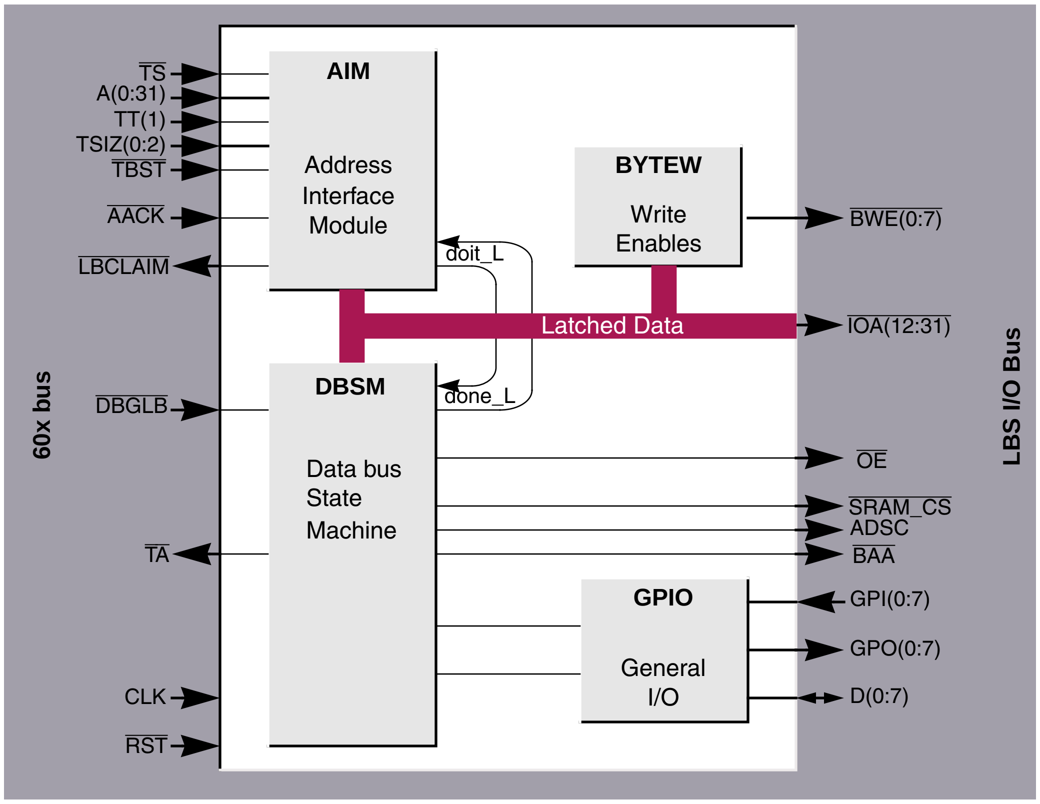

Figure 4. AEIOU Architecture.

6 Address Bus Interface

The AEIOU design has a module that captures address transactions into a holding register. The Address Interface Module (AIM) captures all important address tenure information (whether or not it is an LBS cycle) at every assertion of the $\overline{\text{TS}}$ signal.

Give careful attention to the performance of AIM, because delays there affect the rest of the system. Consider how the MPC107 implements transactions when an LBS is enabled. On each transaction, the MPC107 waits a programmable number of bus clocks (that PICR1[CF_LBCLAIM_DELAY] sets) in case an LBS claims the cycle .If there is no $\overline{\text{LBCLAIM}}$, the cycle proceeds to the PCI or memory bus. If the CF_LBCLAIM_DELAY setting must be set to '3' to accommodate a slow LBS address decoder so that every cycle that the system runs incurs a three-clock delay. For example, an SDRAM memory system configured to run at 3-1-1-1 would slow down to 6-1-1-1. It is advantageous to eliminate dead clock cycles from the address phase decoder of the LBS.

The general block diagram of the AIM is shown in Figure 5.

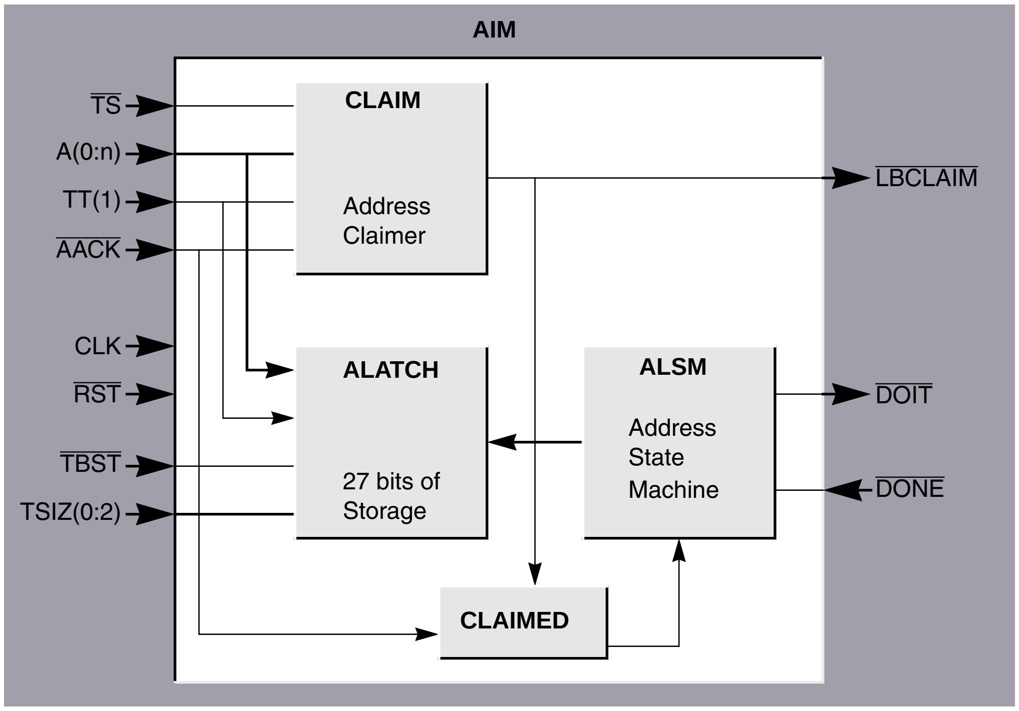

Figure 5. Address Bus Interface of AEIOU.

The address information module (AIM) is composed of several modules that decode addresses and capture the information from the address phase of the bus cycle into a holding register. The CLAIM module generates the required $\overline{\text{LBCLAIM}}$ signal on any LBS-targeted transactions. Note that CLAIM is the only module to which careful placement and timing controls must be observed for the reasons stated above (fast overall system speed). In parallel with CLAIM, the ALATCH module latches the preserved address and address attributes of the cycle.

All other modules in the AEIOU can proceed at a somewhat more leisurely pace, because the bus transactions, when claimed, can proceed at the natural speed of the I/O device without greatly interfering with the performance of other bus transactions, including those to SDRAM or flash. The ALSM module tracks the state of the latch for use by the remainder of the AEIOU. Furthermore, ALSM communicates with the data interface module DBSM (see Section 8, “Data Bus Interface” on page 19) to begin and end LBS transactions.

7 Address Decoder

The first step of any LBS interface is to decode the address and transfer attributes (for example, A[0:n], TSIZ, $\overline{\text{TBST}}$ (optionally), TT and so on) presented at the start of each 60x bus transaction when the bus master asserts transfer start ($\overline{\text{TS}}$). The address decoder must assert $\overline{\text{LBCLAIM}}$ when any transaction hits within the space claimed by the LBS. To keep the decoder simple, the AEIOU claims all transactions within the range 0x2000_0000 to 0x3FFF_FFFF. This address is compatible with the address maps that MPC107 provides, and also meets the restrictions for SDRAM with an LBS, (see Section 4, “Interactions between the LBS and Memory” on page 5).

The LBS address space occupies 512 Mbytes of the 4-Gbyte available space. Although this size can be quite reduced if space is required for other purposes, many systems do not need the additional logic to decode a smaller space completely. Because many systems do not use all the SDRAM memory banks, and because only one LBS is allowed per system, it is not necessary to decode much more than the upper three or four bits of the address.

The VHDL entity that implements the CLAIM module follows:

```
------------------------------------------------------------------------
------
-- VHDL Entity AEIOU.CLAIM.symbol
--
-- Copyright 1999, by
-- All rights reserved. No warranty, expressed or implied, is made as to
the
-- accuracy of this code.
--
-- Revision: 990406 - 1.0 - Created.

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY CLAIM IS
   PORT(
      a          : IN     std_logic_vector (0 to 2) ;
      aack_L     : IN     std_logic   ;
      clk        : IN     std_logic   ;
      rst_L      : IN     std_logic   ;
      ts_L       : IN     std_logic   ;
      lbclaim_L : OUT    std_logic
   );

END CLAIM ;
------------------------------------------------------------------------
------
ARCHITECTURE BEHAVIOR OF CLAIM is
        SIGNAL lbc_L                       : std_logic; -- local LBClaim*
```

```
BEGIN

        monitor : PROCESS( clk, rst_L )

        BEGIN

                                                    IF (rst_L = '0') THEN

                                                            lbc_L <= '1';


                                   ELSIF (clk = '1'  AND  clk'event) THEN

                  IF ((ts_L = '0' AND a = "001")-- TS* and address is LBS

        or  (lbc_L = '0' AND aack_L = 'H')) THEN-- claimed, but not AACK'd

                                                            lbc_L <= '0';

                                                     ELSE-- not LBS cycle

                                                            lbc_L <= '1';

                                                                  END IF;

                                                                  END IF;

        END PROCESS;


        lbclaim_L <= lbc_L;                                 -- copy BUFFER to OUT

END BEHAVIOR;

------------------------------------------------------------------------
-----
```

CLAIM asserts $\overline{\text{LBCLAIM}}$ on any LBS-related transaction and keeps $\overline{\text{LBCLAIM}}$ asserted until the MPC107 asserts $\overline{\text{AACK}}$, acknowledging that the $\overline{\text{LBCLAIM}}$ has been accepted. It is not required for the AEIOU to hold $\overline{\text{LBCLAIM}}$ asserted until $\overline{\text{AACK}}$; however, it must be asserted at least during the interval programmed into the MPC107s PICR1[CF_L2_HITDELAY] register. Alternatives include:

- Assert $\overline{\text{LBCLAIM}}$ for only the one clock cycle in which the MPC107 samples it.
- Assert $\overline{\text{LBCLAIM}}$ for three clock cycles (the maximum sample width).

In general, preserving $\overline{\text{LBCLAIM}}$ until $\overline{\text{AACK}}$ is asserted is usually the easiest method.

7.1 Address Latch State Machine (ALSM)

The second portion of the address interface is the implementation of a simple state machine, ALSM, that tracks the presence of a pending transaction in the holding register. ALSM provides a signal to the DBSM on any claimed transaction and waits for an acknowledgement from the DBSM. Figure 6 shows the state machine diagram.

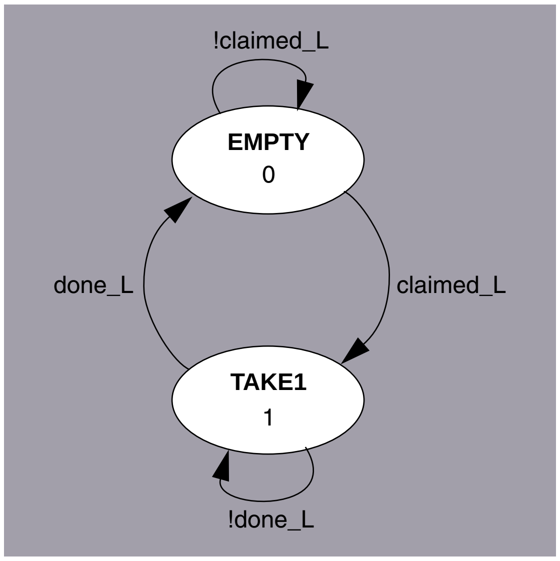

Figure 6. Address FIFO State Machine.

The state encoding (below the state name) directly controls the (active-low) latch enables for the ALATCH module. When idling in state EMPTY (0), the latch enables are asserted and data flows into the latch. When the state machine transitions to state TAKE1 (1), the latch is closed and the address information is captured. The state machine uses the following two input signals:

- claimed_L is asserted for one clock when the address phase ends. It is similar to $\overline{\text{AACK}}$ but is asserted only on LBS cycles.
- done_L is asserted for one clock when a previous LBS I/O cycle ends.

Thereafter, transitions from TAKE1 to EMPTY re-open all the latches. This extremely simple state machine can directly control the latches. The following VHDL entity implements the ALSM state machine:

```
-------------------------------------------------------------------------
-- VHDL Entity AEIOU.ALSM.symbol

--

-- Copyright 1999,

-- All rights reserved. No warranty, expressed or implied, is made as to
the

-- accuracy of this code.

--

-- Revision: 990406 - 1.0 - Created.


LIBRARY ieee ;

USE ieee.std_logic_1164.all;
```

```
ENTITY ALSM IS
   PORT(
      claimed_l : IN     std_logic  ;
      clk       : IN     std_logic  ;
      done_L    : IN     std_logic  ;
      rst_L     : IN     std_logic  ;
      doit_L    : OUT    std_logic  ;
      lg        : OUT    std_logic
   );

END ALSM ;
------------------------------------------------------------------------
----
ARCHITECTURE BEHAVIOR OF ALSM IS

   -- Architecture Declarations
   CONSTANT EMPTY : std_logic := '0';    -- Don't change!
   CONSTANT TAKE1 : std_logic := '1';    -- "

   SUBTYPE state_type IS std_logic;

   -- State vector declaration
   ATTRIBUTE state_vector : string;
   ATTRIBUTE state_vector OF BEHAVIOR : architecture IS "fsm" ;

   -- Declare current and next state signals
   SIGNAL fsm, next_fsm : state_type ;

BEGIN

clocked : PROCESS ( clk, rst_L )
BEGIN
        IF (rst_L = '0') THEN
```

```
                                                fsm <= EMPTY;-- Reset Values
             ELSIF (clk'EVENT AND clk = '1') THEN
                            fsm <= next_fsm;-- Default Assignment To Internals
             END IF;
     END PROCESS clocked;


     nextstate : PROCESS ( claimed_l, done_L, fsm )
     BEGIN
           CASE fsm IS
           WHEN EMPTY =>
              IF ((claimed_L = '0')) THEN
                 next_fsm <= TAKE1;
              ELSE
                 next_fsm <= EMPTY;
              END IF;
           WHEN TAKE1 =>
              IF ((done_L = '0')) THEN
                 next_fsm <= EMPTY;
              ELSE
                 next_fsm <= TAKE1;
              END IF;
           WHEN OTHERS =>
              next_fsm <= EMPTY;
           END CASE;

        END PROCESS nextstate;


     -- Concurrent Statements
     -- Now the outputs. This is a simple Moore machine, and the outputs are
     -- only state-dependant. In fact, the actual output is the encoded state,
     which
     -- is even simpler.
```

```
   lg <= fsm;                            -- Copy SIGNALs (buffers) to OUTs


-- Drive 'doit_L' active when there is anything in the FIFO, which is true
when

-- we are not idling.


   doit_L <= '0' WHEN (fsm /= EMPTY) ELSE '1';


END BEHAVIOR;
```

7.2 Address Latch

The next portion of the address interface is the address latch (ALATCH). Only one level of buffering is needed for the single-overlap 60x bus. The number of bits needed to store a complete address transaction ($\overline{\text{TBST}}$, TSIZ, TT, and A[0:n]) determines the width of ALATCH. To save silicon space, only the required address transaction signals are saved (see Table 3).

Table 3. Address Transaction Signals Preserved

Signal	Defined Bits	Preserved Bits	Notes
$\overline{\text{TBST}}$	1	1	Can be reduced to none if only non-cacheable/non-burst I/O will be controlled.
TSIZ(0:2)	3	3	All bits are needed.
TT(0:4)	5	1	TT(1) is sufficient to show read/write selection for valid LBS transactions.
A(0:31)	32	22	Upper 3 not needed; low 3 required for byte lane selection; the rest are determined by the size of the I/O needed. This example is sufficient to support a 256Kx64 SRAM space plus bits to select SRAM or I/O.
Total	**40**	**27**	Total needed for storage

TT(0:4) may be reduced to TT1 because address-only cycles are forbidden to the LBS I/O space; the remaining cycles reduce to simple single-beat or burst reads or writes, which TT(1) can detect. Consequently, the latch needs to preserve only 27 bits of information. (Note that this reduced amount is application-dependent).

The following VHDL entity describes the latch:

```
-----------------------------------------------------------------------
------

-- VHDL Entity AEIOU.ALATCH.symbol

--

-- Copyright 1999,
-- All rights reserved. No warranty, expressed or implied, is made as to
the
-- accuracy of this code.
--
-- Revision: 990406 - 1.0 - Created.


LIBRARY ieee ;
USE ieee.std_logic_1164.all;


ENTITY ALATCH IS
   PORT(
      a_low     : IN     std_logic_vector (10 TO 31) ;
      lg        : IN     std_logic  ;
      rst_L     : IN     std_logic  ;
      tbst_L    : IN     std_logic  ;
      tsiz      : IN     std_logic_vector (0 to 2) ;
      tt1       : IN     std_logic  ;
      ff_a_low  : OUT    std_logic_vector (10 to 31) ;
      ff_tbst_L : OUT    std_logic  ;
      ff_tsiz   : OUT    std_logic_vector (0 to 2) ;
      ff_tt1    : OUT    std_logic
   );


END ALATCH ;
-----------------------------------------------------------------------
------
ARCHITECTURE BEHAVIOR OF ALATCH is


BEGIN
L0: PROCESS( lg, rst_L, tbst_L, tsiz, tt1, a_low )
BEGIN
        IF (rst_L = '0') THEN
                                                        ff_tbst_L <= '0';
```

```
                                                    ff_tsiz   <= (OTHERS => '0');
                                                              ff_tt1    <= '0';
                                                    ff_a_low  <= (OTHERS => '0');
        ELSIF (lg = '0') THEN
                                                             ff_tbst_L <= tbst_L;
                                                               ff_tsiz   <= tsiz;
                                                                ff_tt1    <= tt1;
                                                              ff_a_low  <= a_low;
        END IF;
END PROCESS;
END BEHAVIOR;
------------------------------------------------------------------------
----
```

7.3 Address Interface Module

The AIM module integrates the other address decoding modules. Because the ALSM module can directly control the address latch module (ALATCH), the AIM module connects only the other modules and creates the $\overline{\text{CLAIMED}}$ signal. The $\overline{\text{CLAIMED}}$ signal must be asserted for one clock cycle for all LBS I/O cycles claimed; neither $\overline{\text{AACK}}$ nor $\overline{\text{LBCLAIM}}$ alone is sufficient. The logical NOR of the two signals (asserted when both are low) ensures that only claimed LBS cycles trigger the state machine.

The following VHDL entity describes the top-level address interface:

```
------------------------------------------------------------------------
------
-- VHDL Entity AEIOU.AIM.symbol
--
-- Copyright 1999,
-- All rights reserved. No warranty, expressed or implied, is made as to
the
-- accuracy of this code.
--
-- Revision: 990406 - 1.0 - Created.

LIBRARY ieee ;
USE ieee.std_logic_1164.all;
```

```
ARCHITECTURE BEHAVIOR OF AIM IS
-- Internal signal declarations
SIGNAL claimed_L : std_logic;
SIGNAL iclaim_L  : std_logic;
SIGNAL lg        : std_logic;

-- Component Declarations
COMPONENT CLAIM
   PORT (
      a         : IN     std_logic_vector (0 to 2);
      aack_L    : IN     std_logic ;
      clk       : IN     std_logic ;
      rst_L     : IN     std_logic ;
      ts_L      : IN     std_logic ;
      lbclaim_L : OUT    std_logic
   );
END COMPONENT;
COMPONENT ALATCH
   PORT (
      a_low     : IN     std_logic_vector (10 TO 31);
      lg        : IN     std_logic ;
      rst_L     : IN     std_logic ;
      tbst_L    : IN     std_logic ;
      tsiz      : IN     std_logic_vector (0 to 2);
      tt1       : IN     std_logic ;
      ff_a_low  : OUT    std_logic_vector (10 to 31);
      ff_tbst_L : OUT    std_logic ;
      ff_tsiz   : OUT    std_logic_vector (0 to 2);
      ff_tt1    : OUT    std_logic
   );
END COMPONENT;
COMPONENT ALSM
```

```
    PORT (
       claimed_l : IN      std_logic ;
       clk       : IN      std_logic ;
       done_L    : IN      std_logic ;
       rst_L     : IN      std_logic ;
       doit_L    : OUT     std_logic ;
       lg        : OUT     std_logic
    );
 END COMPONENT;

 BEGIN

 -- Drive claimed_L low for one clock cycle.
         claimed_L <= '0' WHEN (iclaim_L = '0' AND aack_L = 'L') ELSE '1';



 -- Copy from buffer to output.
         lbclaim_L <= iclaim_L;



 -- Instance port mappings.
    CLz : CLAIM
       PORT MAP (
          a         => a_high,
          aack_L    => aack_L,
          clk       => clk,
          rst_L     => rst_L,
          ts_L      => ts_L,
          lbclaim_L => iclaim_L
       );
    Foz : ALATCH
       PORT MAP (
          a_low      => a_low,
          lg         => lg,
          rst_L      => rst_L,
          tbst_L     => tbst_L,
          tsiz       => tsiz,
          tt1        => tt1,
          ff_a_low   => ff_a_low,
          ff_tbst_L  => ff_tbst_L,
          ff_tsiz    => ff_tsiz,
          ff_tt1     => ff_tt1
       );
    SMz : ALSM
       PORT MAP (
          claimed_l  => claimed_L,
          clk        => clk,
          done_L     => done_L,
          rst_L      => rst_L,
          doit_L     => doit_L,
          lg         => lg
       );
 END BEHAVIOR;
```

8 Data Bus Interface

After the address phase has been handled, the AEIOU waits for doit_L (from the AIM module) to be signaled and $\overline{\text{DBGLB}}$ (from the MPC107) to be asserted, indicating that the AEIOU has control of the data bus. Because $\overline{\text{DBGLB}}$ acts as a gating factor in deciding whether to proceed, the MPC107 must not have parked the bus. PICR1[DPARK] must be cleared.

Depending on the complexity of the addressed device, the LBS interface might immediately assert $\overline{\text{TA}}$ for one cycle and do nothing more. This interface would be appropriate and minimal for devices such as high-speed register files, SRAMs, or FIFOs that can capture single-beat cycles at the full bus rate (usually 15 ns or faster).

Delaying the assertion of $\overline{\text{TA}}$ for a fixed number of cycles to allow for access to slower devices, such as Flash, ROM, and device I/O (UARTs and so forth) is another frequently required action. These devices are usually accessed with single-beat transfers, but have access times on the order of 90-200 ns. For such devices, the data bus interface logic must wait a specified number of cycles after $\overline{\text{DBGLB}}$ before asserting $\overline{\text{TA}}$ but is otherwise similar.

To support the highest transfer rates, the data bus interface can respond to burst transfers and supply data in beats of four. The AEIOU supports all of these types of cycles to demonstrate the flexibility of the LBS interface. Table 4 lists the characteristics of the I/O devices.

Table 4. AEIOU I/O Device Characteristics

Interface Type	Address Range	Read/Write Size	Bus Clocks	Speed (66 MHz)	Cache/Burst Support?
Register	0x2X00_0000 ... 0x2x3F_FFFF	1, 2, 4, 8 bytes	1	15 ns	No
I/O	0x2x40_0000 ... 0x2x7F_FFFF	1, 2, 4, 8 bytes	6	90 ns	No
SRAM	0x2x80_0000 ... 0x2xFF_FFFF	1, 2, 4, 8 bytes	3-1-1-1	90 ns	Yes

To implement all of these cycles, a simple state machine asserts $\overline{\text{TA}}$ at the proper interval: after one clock for register accesses, after five clocks for I/O accesses, and in a 3-1-1-1 sequence for bursts to SRAM. Negating $\overline{\text{TA}}$ temporarily can insert wait states in burst transfers, but it is not necessary at the speed of the local bus interface. Because the AEIOU is not programmable (though it *could* be, but that is another application note), wait states must be added to support slower devices when the bus speed is increased to 83, 100, or 133 MHz.

Figure 7 shows the state machine for the DBSM.

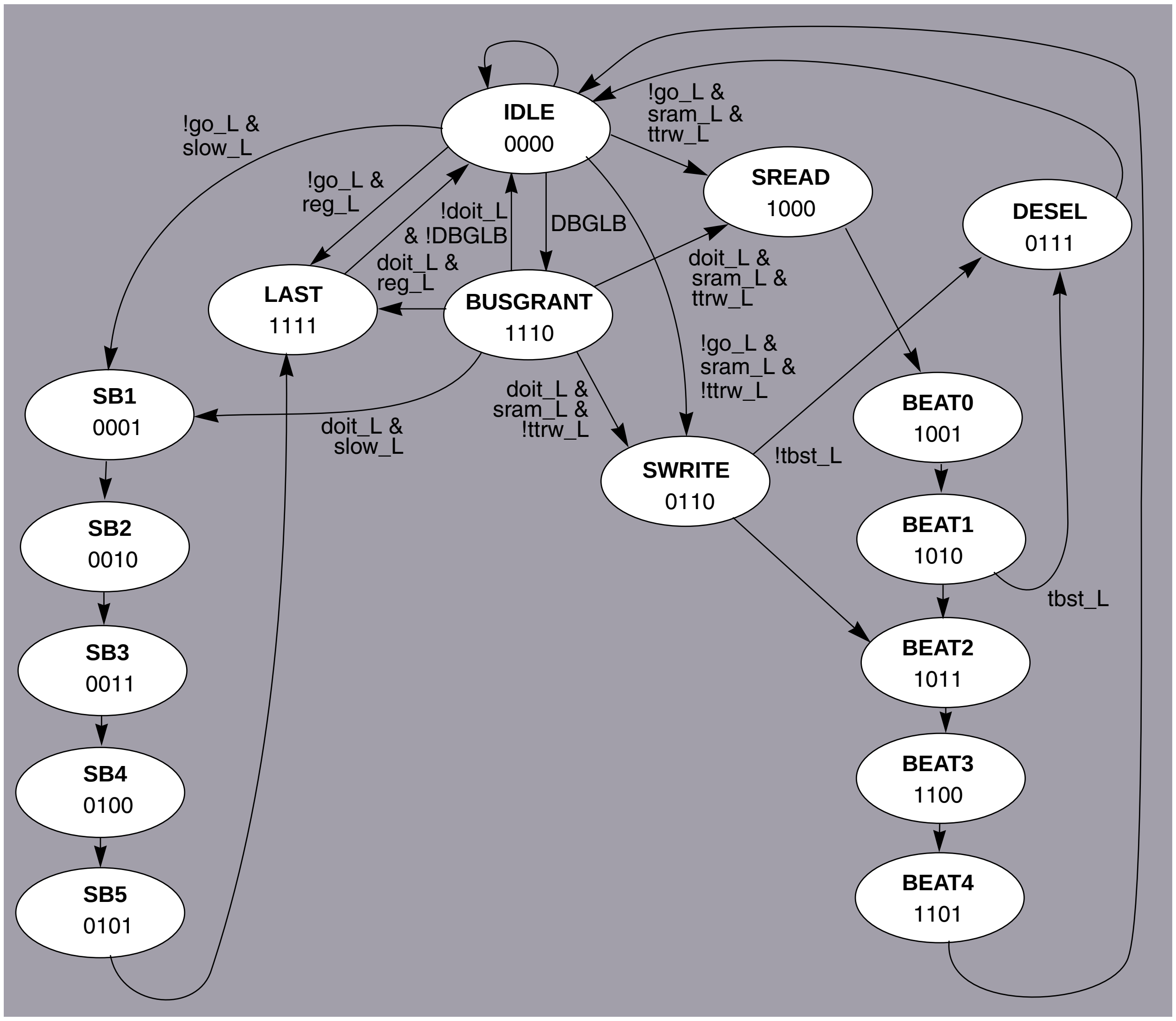

Figure 7. DBSM State Machine Flow.

The main flow of the DBSM state machine is the transition from IDLE to SREAD, SWRITE, or SB1. The first two transitions are for the pipelined-burst SRAM interface (single-beat or burst) and comprise the right-hand side of the diagram. $\overline{\text{TA}}$ is asserted on each state from BEAT1 to BEAT4.

The left-hand side of the state machine shows accesses to non-burst, slow I/O. $\overline{\text{TA}}$ is asserted only at LAST, with the preceding states SB1 to SB5 simply marking time. A counter that would add flexibility (particularly at variable bus speeds) can replace these delay states, but it requires the addition of more complex timer logic.

The state BUSGRANT tracks when $\overline{\text{DBGLB}}$ has asserted. As the *MPC107 PCI Bridge/Memory Controller User's Manual* notes, the local bus slave needs to sample $\overline{\text{DBGLB}}$ continuously. If the local bus slave claims the transaction (by asserting $\overline{\text{LBCLAIM}}$) and $\overline{\text{DBGLB}}$ was asserted for that address tenure, the local bus slave can drive $\overline{\text{TA}}$. If $\overline{\text{DBGLB}}$ was not asserted when the local bus slave claims a transaction, it must wait for the MPC107 to grant the data bus to the processor before the local bus slave can drive $\overline{\text{TA}}$. This way, the MPC107 can maintain the pipeline and the previous data tenure is allowed to complete before the MPC107 relinquishes the data bus to the processor and the local bus slave. The DBSM handles this event, switching to the BUSGRANT state when $\overline{\text{DBGLB}}$ is asserted.

The following VHDL describes the implementation of the $\overline{\text{DBSM}}$:

```
------------------------------------------------------------------------
------

-- VHDL Entity AEIOU.DBSM.symbol

--

-- Copyright 1999

-- All rights reserved. No warranty, expressed or implied, is made as to
the

-- accuracy of this code.

--

-- Revision: 990406 - 1.0 - Created.


LIBRARY ieee ;

USE ieee.std_logic_1164.all;


ARCHITECTURE BEHAVIOR OF DBSM IS


-- Architecture Declarations

CONSTANT IDLE                          : std_logic_vector(0 to 3) := "0000";

CONSTANT SB1                           : std_logic_vector(0 to 3) := "0001";

CONSTANT SB2                           : std_logic_vector(0 to 3) := "0010";

CONSTANT SB3                           : std_logic_vector(0 to 3) := "0011";

CONSTANT SB4                           : std_logic_vector(0 to 3) := "0100";

CONSTANT SB5                           : std_logic_vector(0 to 3) := "0101";

CONSTANT DESEL                         : std_logic_vector(0 to 3) := "0111";

CONSTANT SWRITE                        : std_logic_vector(0 to 3) := "0110";

CONSTANT SREAD                         : std_logic_vector(0 to 3) := "1000";

CONSTANT BEAT0                         : std_logic_vector(0 to 3) := "1001";

CONSTANT BEAT1                         : std_logic_vector(0 to 3) := "1010";

CONSTANT BEAT2                         : std_logic_vector(0 to 3) := "1011";

CONSTANT BEAT3                         : std_logic_vector(0 to 3) := "1100";

CONSTANT BEAT4                         : std_logic_vector(0 to 3) := "1101";
CONSTANT BUSGRANT                    : std_logic_vector(0 to 3) := "1110";
CONSTANT LAST                        : std_logic_vector(0 to 3) := "1111";

SIGNAL slow_L                    : std_logic;-- Set if address to slow I/O
SIGNAL reg_L                 : std_logic;-- Set if address to register I/O
SIGNAL sram_L             : std_logic;-- Set if address may be to SRAM I/O
SIGNAL go_L                     : std_logic;-- Triggered on LBS bus grant.

SUBTYPE state_type IS std_logic_vector(0 to 3);

-- State vector declaration
ATTRIBUTE state_vector : string;
ATTRIBUTE state_vector OF BEHAVIOR : architecture IS "dbsm" ;

-- Declare current and next state signals
SIGNAL dbsm, next_dbsm : state_type ;

BEGIN
        clocked : PROCESS( clk, rst_L )

        BEGIN
                                                        IF (rst_L = '0') THEN
                                                   dbsm <= IDLE;-- Reset Values
                                             ELSIF (clk'EVENT AND clk = '1') THEN
                             dbsm <= next_dbsm;-- Default Assignment To Internals
                                                                          END IF;
        END PROCESS clocked;

        nextstate : PROCESS ( dbglb_L, dbsm, doit_L, go_L, reg_L, slow_L,
sram_L,
                                                            tbst_L, tt_rw_L )
        BEGIN
                                                                   CASE dbsm IS
                                             WHEN IDLE =>
              IF ((sram_L = '0'  AND  go_L = '0'  AND  tt_rw_L = '1')) THEN
                                                next_dbsm <= SREAD;
           ELSIF ((sram_L = '0'  AND  go_L = '0'  AND  tt_rw_L = '0')) THEN
                                               next_dbsm <= SWRITE;
                                ELSIF ((reg_L = '0'  AND  go_L = '0')) THEN
                                                 next_dbsm <= LAST;
                               ELSIF ((slow_L = '0'  AND  go_L = '0')) THEN
                                                  next_dbsm <= SB1;
                                             ELSIF ((dbglb_L = '0')) THEN
                                             next_dbsm <= BUSGRANT;
                                                               ELSE
                                                 next_dbsm <= IDLE;
                                                            END IF;
                                                     WHEN BEAT0 =>
                                                next_dbsm <= BEAT1;
                                                     WHEN BEAT1 =>
                                              IF ((tbst_L = '1')) THEN
                                                next_dbsm <= DESEL;
                                                               ELSE
                                                next_dbsm <= BEAT2;
                                                            END IF;
                                                     WHEN BEAT2 =>
                                                next_dbsm <= BEAT3;
                                                     WHEN BEAT3 =>
                                                next_dbsm <= BEAT4;
                                                     WHEN SREAD =>
                                                next_dbsm <= BEAT0;
                                                     WHEN BEAT4 =>
                                                 next_dbsm <= IDLE;
                                                      WHEN LAST =>
                                                 next_dbsm <= IDLE;
```

```
WHEN SB1 =>
next_dbsm <= SB2;
WHEN SB2 =>
next_dbsm <= SB3;
WHEN SB3 =>
next_dbsm <= SB4;
WHEN SB4 =>
next_dbsm <= SB5;
WHEN SB5 =>
next_dbsm <= LAST;
WHEN BUSGRANT =>
IF ((sram_L  = '0'  AND  doit_L = '0'  AND  tt_rw_L = '1')) THEN
next_dbsm <= SREAD;
ELSIF ((sram_L  = '0'  AND  doit_L = '0'  AND  tt_rw_L = '0')) THEN
next_dbsm <= SWRITE;
ELSIF ((reg_L  = '0' AND  doit_L = '0')) THEN
next_dbsm <= LAST;
ELSIF ((slow_L  = '0'  AND  doit_L = '0')) THEN
next_dbsm <= SB1;
ELSIF ((doit_L = '1'  AND  dbglb_L = '1')) THEN
next_dbsm <= IDLE;
ELSE
next_dbsm <= BUSGRANT;
END IF;
WHEN DESEL =>
next_dbsm <= IDLE;
WHEN SWRITE =>
IF ((tbst_L = '1')) THEN
next_dbsm <= DESEL;
ELSE
next_dbsm <= BEAT2;
END IF;
```

```
                                                                    WHEN OTHERS =>
                                                                next_dbsm <= IDLE;
                                                                         END CASE;
            END PROCESS nextstate;


    -- Concurrent Statements
    -- Do chip selects here, since they're so easy.

            reg_L        <= '0'WHEN( a(10) = '0'  AND  a(11) = '0' )ELSE '1';
            slow_L       <= '0'WHEN( a(10) = '0'  AND  a(11) = '1' )ELSE '1';
            sram_L                         <= '0'WHEN( a(10) = '1')ELSE '1';

    -- Implement the state machine transition triggers.

        go_L          <= '0'WHEN (dbglb_L = '0'  AND  doit_L = '0')ELSE '1';

    ------------------------------------------------------------------------
    -----
    -- Now the outputs of the state machine.
    -- Assert TA* (the most important LBS signal).

        ta_L                                        <= 'L'WHEN ( dbsm = SWRITE
                                                 ORdbsm = BEAT1ORdbsm = BEAT2
                                                OR dbsm = BEAT3ORdbsm = BEAT4
                                                                ORdbsm = LAST
                                                                            )
                                                                     ELSE 'H';

    -- Drive 'done_L' when a cycle completes.

        done_L                      <= '0'WHEN (dbsm = LAST   OR  dbsm = BEAT4
                                                               ORdbsm = DESEL
                                                                            )
                                                                    ELSE '1';


    -- Drive we_L low while running any kind of write cycle.  Drive oe_L low
    -- when running any sort of read cycle.

        we_L                                            <= '0'WHEN (tt_rw_L = '0'
                                                 ANDdbsm /= IDLE AND dbsm /= BUSGRANT)
                                                                    ELSE '1';
        oe_L                                            <= '0'WHEN (tt_rw_L = '1'
                                               AND  dbsm /= IDLE AND dbsm /= BUSGRANT)
                                                                    ELSE '1';


    -- Drive chip selects with copies of internal logic.

        iocs_L     <= slow_LWHEN (dbsm = SB1  OR  dbsm = SB2  OR  dbsm = SB3
                               ORdbsm = SB4  OR  dbsm = SB5  OR  dbsm = LAST)
                                                                    ELSE '1';
        fcs_L                                          <= reg_LWHEN (dbsm = LAST)
                                                                   ELSE '1' ;
        scs_L                    <= sram_LWHEN (dbsm = SREAD  OR  dbsm = SWRITE)
                                                                    ELSE '1';


    -- Special signals for burst-mode accesses.

        adsc_L  <= '0'WHEN (dbsm = SREAD OR dbsm = SWRITE OR dbsm = DESEL)
                                                                    ELSE '1';
        baa_L    <= '0'WHEN (dbsm = BEAT0  OR  dbsm = BEAT1  OR  dbsm = BEAT2
                                            ORdbsm = BEAT3  OR  dbsm = BEAT4)
                                                                    ELSE '1';


    END BEHAVIOR;
    ------------------------------------------------------------------------
    -----
```

9 Cycle Completion

Another design issue for the AEIOU is that the $\overline{TA}$ signal must be actively negated at the end of the LBS data cycle (see Figure 8 to see this event at the end of the assertion of $\overline{TA}$ by the AEIOU).

There are two methods to achieve this requirement. The first is to use a half-phase (or inverted) clock signal to delay the negation of $\overline{TA}$ by one half-clock. While the AEIOU drives the $\overline{TA}$ signal high (internally) on completion of the transaction, the $\overline{TA}$ output enable is removed half-way into the cycle, allowing the signal to tri-state in preparation for the next device to assert $\overline{TA}$ (which may or may not be the AEIOU). This extension method is shown in the last three waveforms of Figure 8.

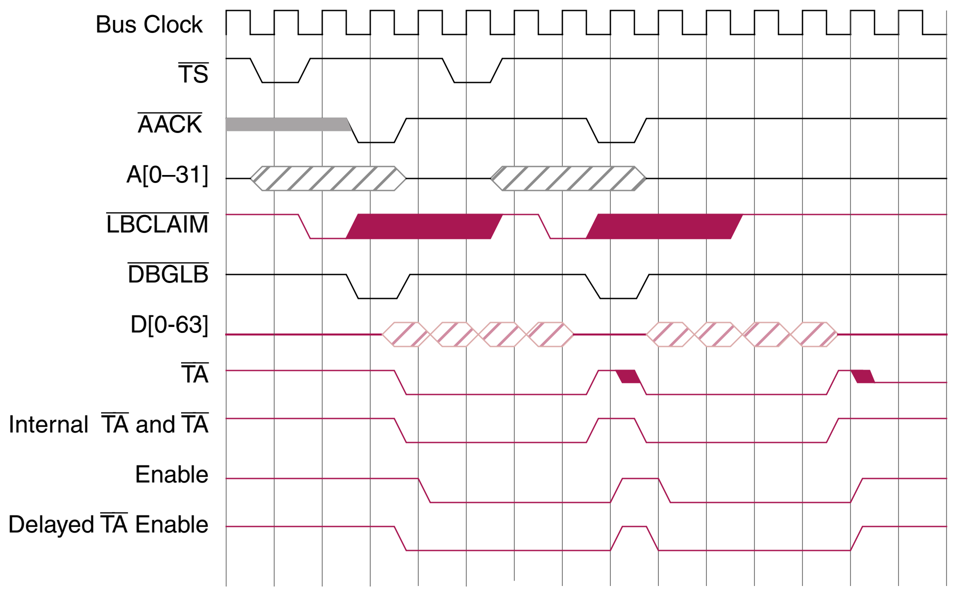

Figure 8. LBS Transaction with TA Enabling.

```
------------------------------------------------------------------------
-----

-- VHDL Entity AEIOU.TADRIVE

--

-- Copyright 1999,

-- All rights reserved. No warranty, expressed or implied, is made as to
the

-- accuracy of this code.

--

-- Revision: 990406 - 1.0 - Created.


LIBRARY ieee ;

USE ieee.std_logic_1164.all;
```

```
ARCHITECTURE behavior OF TADRIVE IS
SIGNAL ta_delay_L : std_logic;
SIGNAL ta_oen_L  : std_logic;
BEGIN

        PROCESS ( clk, rst_L )
        BEGIN
                                             IF (rst_L = '0') THEN
                                                ta_delay_L <= 'H';
                                  ELSIF (falling_edge( clk )) THEN
                                      ta_delay_L <= ta_internal_L;
                                                           END IF;
        END PROCESS;

        ta_oen_L <= '0'   WHEN (ta_delay_L = 'L'  OR ta_internal_L = 'L')
                                                          ELSE '1';

        ta_L                        <= ta_internal_LWHEN (ta_oen_L = '0')
                                                          ELSE 'Z';

END behavior;
-----------------------------------------------------------------------
-----
```

An alternate method is to use a strong pull-up in conjunction with accurate models of all devices that attach to the $\overline{\text{TA}}$ signal. If the pull-up is strong enough to achieve the timing requirements for $\overline{\text{TA}}$ precharge without violating the output current ratings of all the devices, the pull-up may be used instead. The only way to compute the proper pull-up value is to use SPICE modeling; no single specific resistance value guarantees that the system will work perfectly.

10 Byte Write Enable

An additional set of signals is needed for those devices that span multiple byte lanes (for example, DH(0-7), DH(8-15)) on the system bus. In most cases, Freescale does not recommend requiring that a 64-bit-wide SRAM, for example, could be written to in 64-bit quantities only while disallowing byte writes or smaller sizes. For such devices, it is necessary to use a write enable that is conditional on the size and address of the transfer, instead of a global write ($\overline{\text{WE}}$) as provided by the DBSM logic.

As the 60x bus ignores any data placed on bytes lanes that are not needed on a read operation, the BYTEW logic is specific to write operations only. Note that this entire logic block is not needed if all the devices attached to the AEIOU are 8 bits, or if they are only written to in their natural sizes (defined as the number of data bits connected to the 60x bus). For example, a 16-bit FIFO does not need the BYTEW module, because FIFOs are read or written only as 16-bit quantities. For those devices that require byte lane enables, the logic shown in the following VHDL entity is needed.

```
------------------------------------------------------------------------
------

-- VHDL Entity AEIOU.BYTEW

--

-- Copyright 1999

-- All rights reserved. No warranty, expressed or implied, is made as to
the

-- accuracy of this code.

--

-- Revision: 990406 - 1.0 - Created.


LIBRARY ieee ;

USE ieee.std_logic_1164.all;


ARCHITECTURE BEHAVIOR OF BYTEW is

BEGIN


   -- Copy 'we_L' to 'bwe_L(x)' as indicated by transfer size and address.


   bwe_L(0) <= we_L WHEN (  (tsiz = "001"  and  a = "000")    -- byte

                         or (tsiz = "010"  and  a = "000")    -- half-word

                         or (tsiz = "100"  and  a = "000")    -- word

                         or (tsiz = "000"  and  a = "000")    -- double-word

                         or (tsiz = "011"  and  a = "000")    -- three-byte

                         or (tbst_L = '0')                     -- burst

                         )

                     ELSE '1';


   bwe_L(1) <= we_L WHEN (  (tsiz = "001"  and  a = "001")    -- byte
                         or (tsiz = "010"  and  a = "000")    -- half-word
                         or (tsiz = "100"  and  a = "000")    -- word
                         or (tsiz = "000"  and  a = "000")    -- double-word
                         or (tsiz = "011"  and  a = "000")    -- three-byte
                         or (tsiz = "011"  and  a = "001")    -- three-byte
                         or (tbst_L = '0')                    -- burst
                         )
                     ELSE '1';

   bwe_L(2) <= we_L WHEN (  (tsiz = "001"  and  a = "010")    -- byte
                         or (tsiz = "010"  and  a = "010")    -- half-word
                         or (tsiz = "100"  and  a = "000")    -- word
                         or (tsiz = "000"  and  a = "000")    -- double-word
                         or (tsiz = "011"  and  a = "000")    -- three-byte
                         or (tsiz = "011"  and  a = "001")    -- three-byte
                         or (tbst_L = '0')                    -- burst
                         )
                     ELSE '1';

   bwe_L(3) <= we_L WHEN (  (tsiz = "001"  and  a = "011")    -- byte
                         or (tsiz = "010"  and  a = "010")    -- half-word
                         or (tsiz = "100"  and  a = "000")    -- word
                         or (tsiz = "000"  and  a = "000")    -- double-word
                         or (tsiz = "011"  and  a = "001")    -- three-byte
                         or (tbst_L = '0')                    -- burst
                         )
                     ELSE '1';

   bwe_L(4) <= we_L WHEN (  (tsiz = "001"  and  a = "100")    -- byte
                         or (tsiz = "010"  and  a = "100")    -- half-word
                         or (tsiz = "100"  and  a = "100")    -- word
                         or (tsiz = "000"  and  a = "000")    -- double-word
```

```
                       or (tsiz = "011"  and  a = "100")    -- three-byte
                        or (tbst_L = '0')                    -- burst
                        )
                 ELSE '1';

bwe_L(5) <= we_L WHEN (  (tsiz = "001"  and  a = "101")    -- byte
                        or (tsiz = "010"  and  a = "100")    -- half-word
                        or (tsiz = "100"  and  a = "100")    -- word
                       or (tsiz = "000"  and  a = "000")    -- double-word
                       or (tsiz = "011"  and  a = "100")    -- three-byte
                       or (tsiz = "011"  and  a = "101")    -- three-byte
                        or (tbst_L = '0')                    -- burst
                        )
                 ELSE '1';

bwe_L(6) <= we_L WHEN (  (tsiz = "001"  and  a = "110")    -- byte
                        or (tsiz = "010"  and  a = "110")    -- half-word
                        or (tsiz = "100"  and  a = "100")    -- word
                       or (tsiz = "000"  and  a = "000")    -- double-word
                       or (tsiz = "011"  and  a = "100")    -- three-byte
                       or (tsiz = "011"  and  a = "101")    -- three-byte
                        or (tbst_L = '0')                    -- burst
                        )
                 ELSE '1';

bwe_L(7) <= we_L WHEN (  (tsiz = "001"  and  a = "111")    -- byte
                        or (tsiz = "010"  and  a = "110")    -- half-word
                        or (tsiz = "100"  and  a = "100")    -- word
                       or (tsiz = "000"  and  a = "000")    -- double-word
                       or (tsiz = "011"  and  a = "101")    -- three-byte
                        or (tbst_L = '0')                    -- burst
                        )
```

```
                    ELSE '1';

END BEHAVIOR;

------------------------------------------------------------------------
------
```

The values for the VHDL code for the BYTEW module are directly derived from the data alignment tables in the processor user's manuals, for example, Table 8-3 and Table 8-4 of the *MPC750 RISC Microprocessor User's Manual*. Burst transfers enable all byte lanes, while all other transfers enable only the byte lanes based on the address and transfer size.

The three-byte cycles arise from the requirement that 60x bus masters handle misaligned transfers by breaking them into two separate cycles. See the *MPC750 RISC Microprocessor User's Manual* for details on this process. These cycles do not occur unless the program generates misaligned transfers; therefore, the three-byte logic elements could conceivably be eliminated. Note, though, that because I/O spaces are usually designated as non-cacheable, the L1 cache of the processor does not filter these misaligned accesses. If they occur, the program fails. However, Freescale recommends retaining the three-byte cases if possible.

11 Internal Peripherals

To show the capabilities of the non-burst capabilities, an additional module is included to implement some general-purpose I/O and a register file. The GPIO module contains an 8-bit output port, an 8-bit input port, and eight 8-bit registers. The register file implements an array of 8 locations (all upper-byte aligned); the first location is read-only and contains a version ID; the remainder locations are read/write.

Although a UART or other complex function might be more desirable, it is beyond the scope of this application note to examine the internals of a UART. Implementing such devices is often device- or vendor-specific.

```
-- VHDL Entity AEIOU.GPIO

--

-- Copyright 1999,

-- All rights reserved. No warranty, expressed or implied, is made as to
the

-- accuracy of this code.

--

-- Revision: 990406 - 1.0 - Created.


LIBRARY ieee ;

USE ieee.std_logic_1164.all;


ARCHITECTURE BEHAVIOR OF GPIO is


SIGNAL                          gout_L: std_logic; -- GPIO Write strobe.
SIGNAL                          rout_L: std_logic; -- Reg Write strobe.


TYPE                                          regfileIS ARRAY (0 to 7)
                           OF std_logic_vector(0 to 7); -- Regfile array.
SIGNAL                                                  regs    : regfile;
BEGIN


-------------------------------------------------------------------------
--------
-- GPIO Ports:
-- The output latch stores data whenever writes occur to GPIO space at
address
-- 'xx_xxx0'.  We do not check the transfer size, so any size write can be
used
-- (though byte is more typical).


gout_L <= '0'     WHEN (we_L = '0'  AND  gpiocs_L = '0'  AND  a = "000")
                                                                ELSE '1';


gr: PROCESS ( gout_L, rst_L, d_in )
BEGIN
        IF (rst_L = '0') THEN
                                             gpio_out <= (OTHERS => '1');
        ELSIF (gout_L = '0') THEN
                                                        gpio_out <= d_in;
        END IF;
END PROCESS;


-- Input devices are handled a little differently; we have to share the
data bus
-- at the top level, so here we provide the data as-is and supply an
-- output enable strobe that does most of the work.


gpiorden_L <= '0'  WHEN (oe_L = '0'  AND  gpiocs_L = '0'  AND  a = "000")
```

```
                                                                    ELSE '1';

gpiord_out <= gpio_in;


----------------------------------------------------------------------
------

-- Register File


rout_L <= '0'      WHEN (we_L = '0'  AND  regcs_L = '0'  AND  a /= "000")
                                                                    ELSE '1';


rw: PROCESS ( rout_L, rst_L, d_in, a, regs )
BEGIN
        IF (rst_L = '0') THEN
        regs(0) <= CONV_STD_LOGIC_VECTOR( 16#41#, 8 ); -- Register 0 : "A".
        regs(1) <= CONV_STD_LOGIC_VECTOR( 16#45#, 8 ); -- Register 1 : "E".
        regs(2) <= CONV_STD_LOGIC_VECTOR( 16#49#, 8 ); -- Register 2 : "I".
        regs(3) <= CONV_STD_LOGIC_VECTOR( 16#4F#, 8 ); -- Register 3 : "O".
        regs(4) <= CONV_STD_LOGIC_VECTOR( 16#55#, 8 ); -- Register 4 : "U".
        regs(5) <= CONV_STD_LOGIC_VECTOR( 16#5F#, 8 ); -- Register 5 : "_".
        regs(6) <= CONV_STD_LOGIC_VECTOR( 16#30#, 8 ); -- Register 6 : "0".
        regs(7) <= CONV_STD_LOGIC_VECTOR( 16#31#, 8 ); -- Register 7 : "1".

        ELSIF (rout_L = '0') THEN
                                                                    CASE a IS
                                       WHEN "000"  =>   regs(0)  <= d_in;
                                       WHEN "001"  =>   regs(1)  <= d_in;
                                       WHEN "010"  =>   regs(2)  <= d_in;
                                       WHEN "011"  =>   regs(3)  <= d_in;
                                       WHEN "100"  =>   regs(4)  <= d_in;
                                       WHEN "101"  =>   regs(5)  <= d_in;
                                       WHEN "110"  =>   regs(6)  <= d_in;
                                       WHEN "111"  =>   regs(7)  <= d_in;
                         WHEN OTHERS=>   NULL;-- Shouldn't be possible..
```

```
                                                            END CASE;
             END IF;
        END PROCESS;

        -- Reading is similar to GPIO case (except there's lots more).

        regrden_L <= '0'                  WHEN (oe_L = '0'  AND  regcs_L = '0')
                                                            ELSE '1';

        rr: PROCESS ( rout_L, rst_L, d_in, a )
        BEGIN
             CASE a IS
             WHEN "000"  =>    regrd_out  <= regs(0);
             WHEN "001"  =>    regrd_out  <= regs(1);
             WHEN "010"  =>    regrd_out  <= regs(2);
             WHEN "011"  =>    regrd_out  <= regs(3);
             WHEN "100"  =>    regrd_out  <= regs(4);
             WHEN "101"  =>    regrd_out  <= regs(5);
             WHEN "110"  =>    regrd_out  <= regs(6);
             WHEN "111"  =>    regrd_out  <= regs(7);
             WHEN OTHERS                  =>  NULL; -- Shouldn't be possible...
             END CASE;
          END PROCESS;

        END BEHAVIOR;
        --------------------------------------------------------------------------
        ----
```

In addition, the following code snippet merges the GPIO data bus at the top-most level of the design to avoid the use of tri-state devices inside the FPGA/ASIC, which some manufacturers shun for causing test difficulties.

```
        --------------------------------------------------------------------------
        ----
        -- Create the bidirectional data bus. The following way makes it
        -- easier to analyze (no timing loops) but makes the wiring a little more
```

```
-- difficult.


D <= gpiord_out              WHEN (gpiorden_L = '0')ELSE (OTHERS => 'Z');
D <= regrd_out                WHEN (regrden_L = '0')ELSE (OTHERS => 'Z');
d_in <= D;
------------------------------------------------------------------------
----
```

12 The AEIOU

Finally, the AEIOU entity can be created from the previously created modules. The AEIOU block has no logic functions; it only connects instances of the AIM, BYTWE, DBSM, GPIO and TADRIVE modules to the I/O pins.

```
------------------------------------------------------------------------
------
-- VHDL Entity AEIOU.AEIOU
--
-- Copyright 1999,
-- All rights reserved. No warranty, expressed or implied, is made as to
the
-- accuracy of this code.
--
-- Revision: 990406 - 1.0 - Created.


LIBRARY ieee ;
USE ieee.std_logic_1164.all;


ENTITY AEIOU IS
   PORT(
      AACK_L    : IN     std_logic  ;
      A_HIGH    : IN     std_logic_vector (0 to 2) ;
      A_LOW     : IN     std_logic_vector (10 to 31) ;
      BUSY_L    : IN     std_logic  ;
      CLK       : IN     std_logic  ;
      DBGLB_L   : IN     std_logic  ;
      GPIO_IN   : IN     std_logic_vector (0 to 7) ;
      RST_L     : IN     std_logic  ;
      TBST_L    : IN     std_logic  ;
      TSIZ      : IN     std_logic_vector (0 to 2) ;
      TS_L      : IN     std_logic  ;
      TT1       : IN     std_logic  ;
      ADSC_L    : OUT    std_logic  ;
      BAA_L     : OUT    std_logic  ;
      BWE_L     : OUT    std_logic_vector (0 to 7) ;
      GPIO_OUT  : OUT    std_logic_vector (0 to 7) ;
      IOA       : OUT    std_logic_vector (12 TO 31) ;
      LBCLAIM_L : OUT    std_logic  ;
      OE_L      : OUT    std_logic  ;
      SRAM_CS_L : OUT    std_logic  ;
      TA_L      : OUT    std_logic  ;
      D         : INOUT  std_logic_vector (0 to 7)
   );

END AEIOU ;

LIBRARY AEIOU;
------------------------------------------------------------------------
----
ARCHITECTURE BEHAVIOR OF AEIOU IS

-- Architecture declarations
SIGNAL ta_oen_L : std_logic;



-- Internal signal declarations
SIGNAL d_in          : std_logic_vector(0 to 7);
SIGNAL doit_L        : std_logic;
SIGNAL done_L        : std_logic;
SIGNAL fastcs_L      : std_logic;
SIGNAL ff_IOA        : std_logic_vector(10 TO 31);
SIGNAL ff_tbst_L     : std_logic;
SIGNAL ff_tsiz       : std_logic_vector(0 to 2);
SIGNAL gpiord_out    : std_logic_vector(0 to 7);
SIGNAL gpiorden_L    : std_logic;
SIGNAL iocs_L        : std_logic;
SIGNAL regrd_out     : std_logic_vector(0 to 7);
SIGNAL regrden_L     : std_logic;
SIGNAL ta_internal_L : std_logic;
SIGNAL tt_rw_L       : std_logic;
SIGNAL we_L          : std_logic;

-- Implicit buffer signal declarations
SIGNAL OE_L_internal : std_logic ;

-- Component Declarations
COMPONENT AIM
   PORT (
      a_high    : IN     std_logic_vector (0 to 2);
      a_low     : IN     std_logic_vector (10 to 31);
      aack_L    : IN     std_logic ;
      clk       : IN     std_logic ;
      done_L    : IN     std_logic ;
      rst_L     : IN     std_logic ;
      tbst_L    : IN     std_logic ;
      ts_L      : IN     std_logic ;
      tsiz      : IN     std_logic_vector (0 to 2);
      tt1       : IN     std_logic ;
      doit_L    : OUT    std_logic ;
      ff_a_low  : OUT    std_logic_vector (10 TO 31);
      ff_tbst_L : OUT    std_logic ;
      ff_tsiz   : OUT    std_logic_vector (0 to 2);
      ff_tt1    : OUT    std_logic ;
      lbclaim_L : OUT    std_logic
   );
END COMPONENT;
COMPONENT BYTEW
   PORT (
      a      : IN     std_logic_vector (29 to 31);
      tbst_L : IN     std_logic ;
      tsiz   : IN     std_logic_vector (0 to 2);
      we_L   : IN     std_logic ;
      bwe_L  : OUT    std_logic_vector (0 to 7)
   );
END COMPONENT;
COMPONENT DBSM
   PORT (
      a       : IN     std_logic_vector (10 to 31);
      busy_L  : IN     std_logic ;
      clk     : IN     std_logic ;
      dbglb_L : IN     std_logic ;
      doit_L  : IN     std_logic ;
      rst_L   : IN     std_logic ;
      tbst_L  : IN     std_logic ;
      tt_rw_L : IN     std_logic ;
      adsc_L  : OUT    std_logic ;
      baa_L   : OUT    std_logic ;
      done_L  : OUT    std_logic ;
      fcs_L   : OUT    std_logic ;
      iocs_L  : OUT    std_logic ;
      oe_L    : OUT    std_logic ;
      scs_L   : OUT    std_logic ;
      ta_L     : OUT    std_logic ;
      we_L     : OUT    std_logic
   );
END COMPONENT;
COMPONENT GPIO
   PORT (
      a           : IN     std_logic_vector (26 to 28);
      d_in        : IN     std_logic_vector (0 to 7);
      gpio_in     : IN     std_logic_vector (0 to 7);
      gpiocs_L    : IN     std_logic ;
      oe_L        : IN     std_logic ;
      regcs_L     : IN     std_logic ;
      rst_L       : IN     std_logic ;
      we_L        : IN     std_logic ;
      gpio_out    : OUT    std_logic_vector (0 to 7);
      gpiord_out  : OUT    std_logic_vector (0 to 7);
      gpiorden_L  : OUT    std_logic ;
      regrd_out   : OUT    std_logic_vector (0 to 7);
      regrden_L   : OUT    std_logic
   );
END COMPONENT;

COMPONENT TADRIVE
   PORT (
      clk            : IN     std_logic ;
      rst_L          : IN     std_logic ;
      ta_internal_L  : IN     std_logic ;
      ta_L           : OUT    std_logic
   );
END COMPONENT;

-- Optional embedded configurations
--synopsys translate_off
```

```
FOR ALL : AIM USE ENTITY AEIOU.AIM;

FOR ALL : BYTEW USE ENTITY AEIOU.BYTEW;

FOR ALL : DBSM USE ENTITY AEIOU.DBSM;

FOR ALL : GPIO USE ENTITY AEIOU.GPIO;

FOR ALL : TADRIVE USE ENTITY AEIOU.TADRIVE;

--synopsys translate_on


BEGIN

-- Architecture concurrent statements

-- HDL Embedded Text Block 1 eb1

-- Create the bidirectional data bus.  The following way makes it easier
to analyze

-- (no timing loops) but makes the wiring a little more difficult.


D <= gpiord_out              WHEN (gpiorden_L = '0')ELSE (OTHERS => 'Z');

D <= regrd_out                WHEN (regrden_L = '0')ELSE (OTHERS => 'Z');

d_in <= D;




-- HDL Embedded Text Block 2 buscp1

IOA(12 TO 31) <=

   ff_IOA(12 TO 31);



-- Instance port mappings.

   AIz : AIM

      PORT MAP (

         a_high    => A_HIGH,

         a_low     => A_LOW,

         aack_L    => AACK_L,

         clk       => CLK,

         done_L    => done_L,

         rst_L     => RST_L,
         tbst_L    => TBST_L,
         ts_L      => TS_L,
         tsiz      => TSIZ,
         tt1       => TT1,
         doit_L    => doit_L,
         ff_a_low  => ff_IOA(10 TO 31),
         ff_tbst_L => ff_tbst_L,
         ff_tsiz   => ff_tsiz,
         ff_tt1    => tt_rw_L,
         lbclaim_L => LBCLAIM_L
      );
   BEz : BYTEW
      PORT MAP (
         a       => ff_IOA(29 TO 31),
         tbst_L => ff_tbst_L,
         tsiz   => ff_tsiz,
         we_L   => tt_rw_L,
         bwe_L  => BWE_L
      );
   DBz : DBSM
      PORT MAP (
         a        => ff_IOA(10 TO 31),
         busy_L   => BUSY_L,
         clk      => CLK,
         dbglb_L => DBGLB_L,
         doit_L   => doit_L,
         rst_L    => RST_L,
         tbst_L   => ff_tbst_L,
         tt_rw_L => tt_rw_L,
         adsc_L   => ADSC_L,
         baa_L    => BAA_L,
         done_L   => done_L,
         fcs_L    => fastcs_L,
         iocs_L   => iocs_L,
         oe_L     => OE_L_internal,
         scs_L    => SRAM_CS_L,
         ta_L     => ta_internal_L,
         we_L     => we_L
      );
   GPz : GPIO
      PORT MAP (
         a           => ff_IOA(26 TO 28),
         d_in        => d_in,
         gpio_in     => GPIO_IN,
         gpiocs_L    => iocs_L,
         oe_L        => OE_L_internal,
         regcs_L     => fastcs_L,
         rst_L       => RST_L,
         we_L        => we_L,
         gpio_out    => GPIO_OUT,
         gpiord_out  => gpiord_out,
         gpiorden_L  => gpiorden_L,
         regrd_out   => regrd_out,
         regrden_L   => regrden_L
      );
   TDz : TADRIVE
      PORT MAP (
         clk            => CLK,
         rst_L          => RST_L,
         ta_internal_L  => ta_internal_L,
         ta_L           => TA_L
      );

   -- Implicit buffered output assignments
```

```
   OE_L <= OE_L_internal;

END BEHAVIOR;

------------------------------------------------------------------------
------
```

13 Conclusion

The LBS interface represents an easy means of connecting high-speed peripherals to the 60x bus By using the facilities of the MPC107, a high-performance interface can be created in an ASIC or FPGA without the need to design a 60x bus master.

14 Revision History

Table 5 provides a revision history of this document.

Table 5. Document History

Revision Number	Changes
0	Original release of document
1.1	• Clarified addressing for MPC107 • Corrected VHDL code formatting issues • Corrected labels for Figure 2 and annotated with color
1.2	Nontechnical reformatting

THIS PAGE INTENTIONALLY LEFT BLANK

THIS PAGE INTENTIONALLY LEFT BLANK

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

freescale™
semiconductor

AN1846/D