Application Note
Designing a Local-Bus-Slave Interface

by Gary Milliorn
PCSD
risc10@email.sps.mot.com

This document describes the steps for designing an interface device that provides access to I/O and memory peripherals on a high-performance PowerPC™ 60x bus, using the local-bus slave features of the MPC106 or MPC107 PCI Bridge/Memory controllers. This document contains the following topics:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Section 1.2, “Conventions”</td>
<td>2</td>
</tr>
<tr>
<td>Section 1.3, “Local-Bus Slave Architecture”</td>
<td>3</td>
</tr>
<tr>
<td>Section 1.4, “Interactions between the LBS and Memory”</td>
<td>5</td>
</tr>
<tr>
<td>Section 1.5, “AEIOU Architecture”</td>
<td>6</td>
</tr>
<tr>
<td>Section 1.6, “Address Bus Interface”</td>
<td>7</td>
</tr>
<tr>
<td>Section 1.7, “Address Decoder”</td>
<td>8</td>
</tr>
<tr>
<td>Section 1.8, “Data Bus Interface”</td>
<td>17</td>
</tr>
<tr>
<td>Section 1.9, “Cycle Completion”</td>
<td>24</td>
</tr>
<tr>
<td>Section 1.10, “Byte Write Enable”</td>
<td>25</td>
</tr>
<tr>
<td>Section 1.11, “Internal Peripherals”</td>
<td>28</td>
</tr>
<tr>
<td>Section 1.12, “The AEIOU”</td>
<td>31</td>
</tr>
</tbody>
</table>

For published errata or updates to this document, refer to the web site at http://www.motorola.com/semiconductors.
Introduction

1.1 Introduction

The MPC106 and the MPC107 provide support for an interface method called local-bus slave (LBS) that allows devices to be more easily attached 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 assert TA when the read or write operation has been 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 referred to as the AEIOU (Applications Engineering Input/Output Unit). For the remainder of this document, the MPC107 is used to refer 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 has been compiled and verified using a software test bench but has not been verified in hardware. It is possible that there are significant errors in the AEIOU, or that the 60x bus test bench has not revealed latent errors in the VHDL code presented herein. In summary, this application note should be treated as general design information for creating an LBS I/O controller and not as a drop-in component.

NOTE:
All the software contained in this application note is copyrighted 1999 by Motorola, Inc. and may be used freely by Motorola customers as long as the copyright notice remains present in each module or source file. The code may be freely modified to suit customized applications.

1.2 Conventions

This application note refers to both hardware and software signals (pins and nets) and components (physical and logical). To avoid confusion, the typographic conventions shown in Table 1 are used:

Table 1. Typographic Conventions

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Hardware</strong></td>
<td></td>
</tr>
<tr>
<td>TT(0:4) (uppercase, no overbar)</td>
<td>A pin that is asserted when active high.</td>
</tr>
<tr>
<td>TA (Uppercase, overbar)</td>
<td>A pin that is asserted when active low.</td>
</tr>
<tr>
<td><strong>Software</strong></td>
<td></td>
</tr>
<tr>
<td>done (lowercase)</td>
<td>An internal signal that is asserted when logic ‘1’ or high.</td>
</tr>
<tr>
<td>go_L (lowercase with _L appended)</td>
<td>An internal signal that is asserted when logic ‘0’ or low.</td>
</tr>
<tr>
<td>TTO (uppercase)</td>
<td>An internal signal that is asserted when logic ‘1’ or high and drives an external hardware pin of the same name.</td>
</tr>
<tr>
<td>TA_L (uppercase with _L appended)</td>
<td>An internal signal that is asserted when logic ‘0’ or low and drives an external hardware pin of the same name.</td>
</tr>
<tr>
<td><strong>BYTEW</strong> (uppercase, bold)</td>
<td>The name of a VHDL entity or module.</td>
</tr>
</tbody>
</table>
In addition, when writing VHDL, IEEE 1164 standard logic levels are used as described in Table 2.

### Table 2. IEEE 1164 Logic Conventions

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>'0'</td>
<td>Logic low, forcing</td>
<td>Internally and on outputs that are not shared (for example, LBCLAIM).</td>
</tr>
<tr>
<td>'1'</td>
<td>Logic high, forcing</td>
<td>Internally and on outputs which are not shared.</td>
</tr>
<tr>
<td>'L'</td>
<td>Logic low, weak</td>
<td>On outputs which are shared (for example, TA).</td>
</tr>
<tr>
<td>'H'</td>
<td>Logic high, weak</td>
<td>On outputs which are shared.</td>
</tr>
</tbody>
</table>

### 1.3 Local-Bus Slave Architecture

The MPC107 provides the system bus arbitration for itself, the PowerPC processor, 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. In order for another device to claim a bus cycle for itself, the MPC107 provides an input signal called LBCLAIM. 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 has completed that portion.

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

The LBS communicates with the 60x bus using the same signals as any other device. There are two additional side-band signals (LBCLAIM, DBGLB) that are used to claim and be granted a cycle, respectively. The general sequence of a LBS cycle is shown in Figure 2.
One complication to the design of an LBS controller is that the 60x bus implements separate address and data tenures. That is, 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. An example of this is shown in Figure 2, where in cycle #6, TS is asserted for a new transaction while data is still being transferred. The effect of split tenures is that the LBS controller must have the capability of storing the current address, size and transfer type information of the current cycle. The 60x bus protocol does not pipeline more than one address tenure, so the information storage requirements this causes are modest.

Note that while the 60x bus does allow the control of the address and data tenure overlap (by delaying the assertion of AACK), the MPC107 does not have this facility; therefore, it cannot be used with an LBS interface. When the LBS claims a cycle, the MPC107 asserts 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 interfaces must accept the possibility of pipelined addresses. There are two solutions to this problem.

The first solution is to 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 ever pending.

The second solution is simpler and less expensive but moves the complexity from the hardware into the software. Basically, the system allows any cycle after an LBS I/O cycle to be missed (unclaimed). Software must guarantee that LBS-targeted cycles are not performed back-to-back, by allowing other instructions to run, or by performing a write to a non-LBS address; (Note that a a read cycle is not effective because the PowerPC 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, possibly triggering a PCI-to-local-bus snoop transaction). A dummy write to an unused memory location would suffice.
1.3.1 Coherency

To keep the design simple for I/O-controller purposes, the design presumes that accesses to the LBS-controlled addresses are coherent. That is, the system does not expect the LBS to snoop the system bus to supply cached data to external devices or to invalidate internally cached data. This 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 only means 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 which requires the ability to accept burst transfers.

The MPC106 PCI Bridge/Memory Controller User’s Manual (see Section 4.4.5, “60x Local Bus Slave Support”) (and see also the MPC107 PCI Bridge/Memory Controller User’s Manual when it becomes available) presents a complex state machine for tracking the 60x bus state. The state machine logic is only necessary when the LBS must maintain cache coherency with external bus masters, or also with the 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.

1.4 Interactions between the LBS and Memory

There are some interactions 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 9 of the MPC107 PCI Bridge/Memory Controller User’s Manual for details on error handling), then the memory controller interferes with LBS operations. This interference can be avoided by following these restrictions:

- If memory select errors have been enabled (by setting the ErrEnR1[MSE] bit), then 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 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, then the address chosen for the LBS must be in the range 0-2 Gbytes (0-0x7FFF_FFFF).
- Under no circumstances can the local bus slave reside in the 2-4-Gbyte address range.

When the above restrictions are followed, the MPC107 operates properly with the memory controller activated.

The second issue is that the MPC106 (and only the MPC106) multiplexes the SDRAM clock enable signal (CKE) with the 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 DBG0 signal can be used for DBGLB. In a multiprocessor system, DBG[0–3] can be logically ANDed to create a suitable 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.
AEIOU Architecture

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

Figure 3. DBGLB Recovery Logic for the MPC106

Note that DBGLB recovery logic requires that the LBS interface 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.

1.5 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; here, instead, 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. The general block diagram is shown in Figure 4 with connections to the 60x bus on the left, and the connections to the I/O devices on the right.
1.6 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 TS signal.

The performance of AIM must be given careful attention because delays here 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 (as set by PICR1[CF_LBCLAIM_DELAY]) in case an LBS claims the cycle; if there is no LBCLAIM, then 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, then every cycle 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 well worth any effort made to eliminate dead clock cycles from the address phase decoder of the LBS.
Address Decoder

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

![Figure 5. Address Bus Interface of AEIOU](image)

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 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, since the bus transactions, once claimed, can proceed at the natural speed of the I/O device without interfering with the performance of SDRAM/EDO, Flash, etc. as previously described. The ALSM module tracks the state of the latch, for use by the remainder of the AEIOU. In addition, ALSM also communicates with the data interface module DBSM (see Section 1.8, “Data Bus Interface” on page 17) to begin and end LBS transactions.

1.7 Address Decoder

The first step of any LBS interface is to decode the address and transfer attributes (for example, A[0:n], TSIZ, TBST (optionally), TT and so on) presented at the start of each 60x bus transaction when the bus master asserts transfer start (TS). The address decoder must assert 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 provided by the MPC107 and also meets the restrictions for SDRAM with an LBS, (see Section 1.4, “Interactions between the LBS and Memory” on page 5).

The LBS address space occupies 512 Mbytes of the 4-Gbyte available space. While this can be reduced to a much smaller size if the space is needed for other purposes, many systems do not need the additional logic to completely decode a smaller space. Because many systems do not use all the SDRAM memory banks, and since 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 Motorola Inc.
-- 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;     --
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;
END;

-- local LBClaim*

ARCHITECTURE BEHAVIOR OF CLAIM is
BEGIN
  IF (rst_L = '0') THEN
    lbc_L <= '1';
  ELSIF (clk = '1' AND clk'event) THEN
    IF ((ts_L = '0' AND a = "001") -- TS*
      OR (lbc_L = '0' AND aack_L = 'H')) THEN-- claimed,
      lbc_L <= '0';
    ELSE -- not
      lbc_L <= '1';
    END IF;
  END IF;
END;
CLAIM asserts \texttt{LBCLAIM} on any LBS-related transaction and keeps \texttt{LBCLAIM} asserted until the MPC107 asserts \texttt{AA\_ACK} acknowledging that the \texttt{LBCLAIM} has been accepted. It is not required for the \texttt{AEIOU} to hold \texttt{LBCLAIM} asserted until \texttt{AA\_ACK}; however, it must be asserted at least during the interval programmed into the MPC107s PICR1[CF\_L2\_HITDELAY] register. Alternatives include:

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

In general, preserving \texttt{LBCLAIM} until \texttt{AA\_ACK} is asserted is usually the easiest method.

### 1.7.1 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. The state machine diagram is shown in Figure 6.

![Figure 6. Address FIFO State Machine](image)

The state encoding (below the state name) directly controls the (active-low) latch enables for the ALATCH module. When idling in state \texttt{EMPTY} (0), the latch enables are asserted and data flows into the latch. When the state machine transitions to state \texttt{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 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 works out to be an extremely simple state machine that can directly control the latches. The following VHDL entity implements the ALSM state machine:

------------------------------------
-- VHDL Entity AEIOU.ALSM.symbol --
-- Copyright 1999, by Motorola Inc. --
-- 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" ;
Address Decoder

-- 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;
### 1.7.2 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 width of ALATCH is determined by the number of bits needed to store a complete address transaction (TBST, TSIZ, TT, and A[0:n]). To save silicon space, only the required address transaction signals are saved as described in Table 3.

**Table 3. Address Transaction Signals Preserved**

<table>
<thead>
<tr>
<th>Signal</th>
<th>Defined Bits</th>
<th>Preserved Bits</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBST</td>
<td>1</td>
<td>1</td>
<td>Can be reduced to zero if only non-cacheable/non-burst I/O is controlled.</td>
</tr>
<tr>
<td>TSIZ(0:2)</td>
<td>2</td>
<td>2</td>
<td>All bits needed</td>
</tr>
<tr>
<td>TT(0:4)</td>
<td>5</td>
<td>1</td>
<td>TT(1) shows R/W status for valid LBS transactions.</td>
</tr>
<tr>
<td>A(0:31)</td>
<td>32</td>
<td>23</td>
<td>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.</td>
</tr>
</tbody>
</table>

TT(0:4) may be reduced since 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 can be detected by TT(1). 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
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 ;
   );
```

---

-- VHDL Entity AEIOU.ALSM.symbol
--
-- Copyright 1999, by Motorola Inc.
-- 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 ;
   );
```
Address Decoder

```vhdl
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;

1.7.3 Address Interface Module

The AIM module integrates the other address decoding modules. Since the ALSM module can directly control the address latch module (ALATCH), the AIM module only connects the other modules and creates the CLAIMED signal. The CLAIMED signal must be asserted for one clock cycle for all LBS I/O cycles claimed; neither AAACK nor LBCLAIM alone are 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
-- VHDL Entity AEIOU.AIM.symbol
--
-- Copyright 1999, by Motorola Inc.
-- 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);
    ttl          : 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_ttl       : 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;
Address Decoder

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,
        ttl       => ttl,
        ff_a_low  => ff_a_low,
        ff_tbst_L => ff_tbst_L,
        ff_tsiz   => ff_tsiz,
        ff_ttl    => ff_ttl
    );
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;
1.8 Data Bus Interface

After the address phase has been handled, the AEIOU waits for doit_L (from the AIM module) to be signaled and DBGLB (from the MPC107) to be asserted, indicating that the AEIOU has control of the data bus. Depending on the complexity of the addressed device, the LBS interface might immediately assert TA for one cycle and do nothing more. This would be an appropriate, minimal interface 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).

Another action frequently required is to delay the assertion of TA for a fixed number of cycles to allow for access to slower devices, such as Flash, ROM, and device I/O (UARTs). 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 DBGLB before asserting 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 in order to demonstrate the flexibility of the LBS interface. The characteristics of the I/O devices are listed in Table 4.

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

To implement all of these cycles, a simple state machine asserts TA at the proper interval: after 1 clock for register accesses, after 5 clocks for I/O accesses, and in a 3-1-1-1 sequence for bursts to SRAM. It is possible to insert wait states in burst transfers by negating TA temporarily, but it is not necessary at the speed of the local bus interface. Because the AEIOU is not programmable, wait states must be added to support slower devices when the bus speed is increased to 83 MHz or 100 MHz. Figure 7 shows the state machine for the DBSM.
Data Bus Interface

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. 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. TA is asserted only at LAST, with the preceding states SB1 to SB5 simply marking time. These delay states can be replaced by using a counter, that would add flexibility (particularly at variable bus speeds), but it requires the addition of more complex timer logic.

The state BUSGRANT is used to track when DBGLB has asserted. As noted in the MPC107 PCI Bridge/Memory Controller User’s Manual, the local bus slave needs to sample DBGLB continuously. If the local bus slave claims the transaction (by asserting LBCLAIM) and DBGLB has been asserted for that address tenure, then the local bus slave can drive TA. If DBGLB has not been asserted when the local bus slave claims a transaction, then it must wait for the MPC107 to grant the data bus to the processor before the local bus slave can drive 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. This is handled by the DBSM by switching to the BUSGRANT state when DBGLB is asserted.
The following VHDL describes the implementation of the DBSM:

-----------------------------------------------
-- VHDL Entity AEIOU.DBSM.symbol
-- Copyright 1999, by Motorola Inc.
-- 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
Designing a Local-Bus-Slave Interface

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
      next_dbsm <= IDLE; -- Default Assignment To Internals
    ELSIF (clk'EVENT AND clk = '1') THEN
      dbsm <= next_dbsm;
    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;
    END CASE;
  END PROCESS nextstate;

END Designing a Local-Bus-Slave Interface;
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

Data Bus Interface

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.

ELSE '1' reg_L <= '0' WHEN( a(10) = '0' AND a(11) = '0' )
ELSE '1'; slow_L <= '0' WHEN( a(10) = '0' AND a(11) = '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
= BEAT2 OR dbsm = BEAT1 OR dbsm
= BEAT4 OR dbsm = BEAT3 OR dbsm
= BEAT1 OR dbsm = LAST OR dbsm
)
ELSE 'H';

-- Drive 'done_L' when a cycle completes.

done_L <= '0' WHEN (dbsm = LAST OR dbsm
= BEAT4 OR dbsm = 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'
    AND dbsm /= 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_L WHEN ( dbsm = SB1 OR dbsm = SB2 OR dbsm = SB3
    OR dbsm = SB4 OR dbsm = SB5 OR dbsm = LAST)
    ELSE '1';

fcs_L <= reg_L WHEN (dbsm = LAST)
    ELSE '1';

scs_L <= sram_L WHEN (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 = BEAT1 OR dbsm = BEAT2
    OR dbsm = BEAT3 OR dbsm = BEAT4)
    ELSE '1';
Cycle Completion

1.9 Cycle Completion

Another design issue for the AEIOU is that the \( \overline{T_A} \) signal must be actively negated at the end of the LBS data cycle. This is shown in Figure 8 at the end of the assertion of \( \overline{T_A} \) 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 \( \overline{T_A} \) by one half-clock. While the AEIOU drives the \( \overline{T_A} \) signal high (internally) on completion of the transaction, the \( \overline{T_A} \) output enable is removed half-way into the cycle, allowing the signal to tri-state in preparation for the next device to assert \( \overline{T_A} \) (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](image)

The VHDL for the delay method is in the code for the AEIOU top level because it is relatively trivial and not entity-worthy. The following VHDL describes the implementation:

```vhdl
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;
```

---

Figure 8. LBS Transaction with TA Enabling
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_L WHEN (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 TA signal. If the pull-up is strong enough to achieve the timing requirements for TA precharge without
violating the output current ratings of all the devices, then the pull-up may be used instead. The only way
to compute the proper pull-up value is to use SPICE modeling; there is not one specific resistance value that
to guarantee that the system will work perfectly.

1.10 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, it is unacceptable to require that a 64-bit-wide SRAM, for
example, could be written to only in 64-bit quantities with byte writes or smaller sizes disallowed. 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 (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, as
FIFOs are only read or written 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, by Motorola Inc.
-- 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 = "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 = "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';

END;


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 = "101")    -- 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.
Internal Peripherals

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 misaligned transfers are generated by the program; therefore, the three-byte logic elements can be eliminated. Note, though, that since I/O spaces are usually designated as non-cacheable, the L1 cache of the processor does not filter these misaligned accesses; therefore, if they occur, the program fails. However, it is recommended that the three-byte cases be retained if possible.

1.11 Internal Peripherals

To show the capabilities of the non-burst capabilities, an additional module is included which implements 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.

While 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; such devices are often device- or vendor-specific in implementation.

```vhdl
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 regfile IS ARRAY (0 to 7) OF std_logic_vector(0 to 7); -- Regfile array.
SIGNAL regs : regfile;

BEGIN
```
Internal Peripherals

-- 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).

```
"000") 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 which 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 ); --
```
Internal Peripherals

Register 4 : "U".

Register 5 : "_".

Register 6 : "0".

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. This avoids the use of tri-state devices inside the FPGA/ASIC.

```vhdl
-- 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;
```

### 1.12 The AEIOU

Finally, the AEIOU entity can be created from the previously created modules. The AEIOU block connects instances of the DBSM and AIM modules to the I/O pins and also includes the special `TA` control logic.
The AEIOU

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);
  ttl       : 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_ttl    : 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 ;
);
The AEIOU

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);
The AEIOU

-- 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,
  ttl     => TT1,
  doit_L  => doit_L,
  ff_a_low => ff_IOA(10 TO 31),
  ff_tbst_L => ff_tbst_L,
  ff_tsiz => ff_tsiz,
  ff_ttl  => 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,
  dbgblb_L => DBGBLB_L,
  doit_L => doit_L,
  rst_L  => RST_L,
  tbst_L => ff_tbst_L,
  tt_rw_L => tt_rw_L,
  adsc_L => ADSC_L,
Designing a Local-Bus-Slave Interface

The AEIOU

begin
  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;

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

MOTOROLA

Designing a Local-Bus-Slave Interface 37
1.13 Conclusion

The LBS interface represents an easy means of connecting high-speed peripherals to the 60x bus of a PowerPC system. 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.
Conclusion
Information in this document is provided solely to enable system and software implementers to use PowerPC microprocessors. There are no express or implied copyright licenses granted hereunder to design or fabricate PowerPC integrated circuits or integrated circuits based on the information in this document.

Motorola reserves the right to make changes without further notice to any products herein. Motorola makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Motorola assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. "Typical" parameters can and do vary in different applications. All operating parameters, including "Typicals" must be validated for each customer application by customer's technical experts. Motorola does not convey any license under its patent rights nor the rights of others. Motorola products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Motorola product could create a situation where personal injury or death may occur. Should Buyer purchase or use Motorola products for any such unintended or unauthorized application, Buyer shall indemnify and hold Motorola and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Motorola was negligent regarding the design or manufacture of the part.

Motorola and are registered trademarks of Motorola, Inc. Motorola, Inc. is an Equal Opportunity/Affirmative Action Employer.

Motorola Literature Distribution Centers:
USA/EUROPE: Motorola Literature Distribution; P.O. Box 5405; Denver, Colorado 80217; Tel.: 1-800-441-2447 or 1-303-675-2140;
World Wide Web Address: http://ldc.nmd.com/
JAPAN: Nippon Motorola Ltd SPD, Strategic Planning Office 4-32-1, Nishi-Gotanda Shinagawa-ku, Tokyo 141, Japan Tel.: 81-3-5487-8488
ASIA/PACIFIC: Motorola Semiconductors H.K. Ltd Silicon Harbour Centre 2, Dai King Street Tai Po Industrial Estate Tai Po, New Territories, Hong Kong

Mfax™: RMFAX0@email.sps.mot.com; TOUCHTONE 1-602-244-6609; US & Canada ONLY (800) 774-1848;
World Wide Web Address: http://spsp.motorola.com/mfax
INTERNET: http://motorola.com/sps

Technical Information: Motorola Inc. SPS Customer Support Center 1-800-521-6274; electronic mail address: crc@wmkmail.sps.mot.com.
World Wide Web Addresses: http://www.motorola.com/PowerPC
http://www.motorola.com/netcomm
http://www.motorola.com/Coldfire

Mfax is a trademark of Motorola, Inc.
The PowerPC name, the PowerPC logotype, and PowerPC 603e are trademarks of International Business Machines Corporation used by Motorola under license from International Business Machines Corporation.