

Revision: July 18, 2008

## Overview

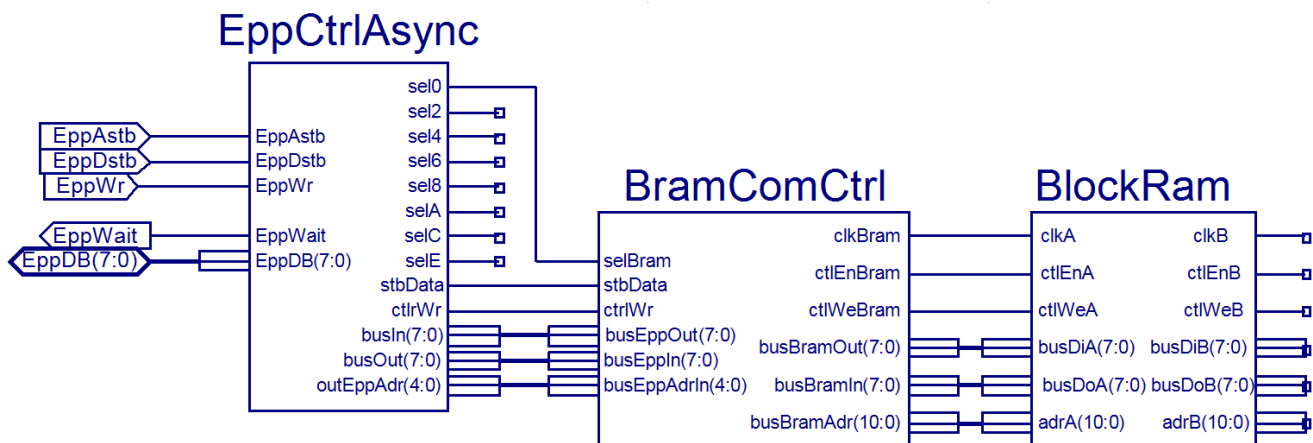
The BramCfg reference project contains the design for a Xilinx Spartan 3E FPGA Block RAM (BRAM) controller. This configuration, in conjunction with a PC running software and a communication module (Diligent OnBoard USB circuitry), allows the user to read or write the BRAM. The PC application can be a Diligent utility (TransPort, etc.) or a user-generated application. Diligent Adept Suite software provides both utility programs and DLLs to be used with user applications for interfacing with Diligent communication modules.

## Functional Description

The BramCfg reference project implements an EPP interface (EppCtrlAsync) that is able to communicate with the EPP port emulated by the Diligent OnBoard USB circuitry. The EPP interface controls the EPP data registers implemented in the BRAM communication controller (BramComCtrl). The BRAM communication controller generates the signal sequence needed to read or write the BRAM. The BRAM module instantiates a 2K X 8-bit BRAM Xilinx primitive. Port A is used to communicate via BramComCtrl and EppCtrlAsync, while Port A is available for user extensions.

The project's BRAM memory can be expanded by increasing the size of the regBramAdr in the BramComCtrl component and correspondingly increasing the size of the BRAM component.

For detailed descriptions of each component, see the *Diligent Component Library* and the VHDL source file header and comments.



**Figure 1 The BramCfg Reference Project**

## Port Definitions

### Epp Bus Signals

EppAstb	in, Address strobe
EppDstb	in, Data strobe
EppWr	in, write signal
EppDB	inout, 8-bit data bus
EppWait	out, wait signal

## The EppCtrl Interface

This file contains the design for an EPP Asynchronous Interface Module.

All data transfers are synchronized only with the external EppAstb and EppDstb strobes.

The EPP controller serves as the interface between an EPP compatible port (emulated by the Digilent OnBoard USB circuitry and firmware) and client components in the user HDL project. It is used in conjunction with a program running on a PC (a Digilent utility such as TransPort or a user-generated application) which in turn uses the APIs provided by Digilent Adept Suite.

### Functional Description

The controller performs the following functions:

- it manages the EPP standard handshake (asynchronously)
- it implements the standard EPP Address Register
- it provides the signals needed to read/write EPP Data Registers
- It provides selection signals to enable up to eight clients, assigning to each a contiguous range of 32 EPP addresses.

No handshake is provided to client components. They should be fast enough to use EppDstb as a clock signal. The client components are responsible for implementing the specific required data registers. They must declare the data read and write registers and assign an EPP address for each.

A couple of read-respective write-registers can be assigned to the same EPP address. A unique 5-bit address must be assigned to each register (couple) of a client component.

If less than 32 (couples of) registers are required, "mirror" or "alias" addresses can be used (incomplete regEppAdrOut(4:0) decoding) as follows:

- use stbData as clock signal for all data registers (rising edge)
- connect the inputs of all write registers to busOut(7:0)
- decode regEppAdrOut(4:0) to generate the Enable signal for each internal register
- use ctrlWr as WE signal for all the write registers
- connect the output data busses of all client components to busIn(7:0) through an OR circuit.

The EPP controller implements the EPP address register (regEppAdr), which can be written and read with EppAstb active. regEppAdr is written on the falling edge of EppAstb, when EppWr is active (LOW).

The upper 3-bits are decoded to generate SELECT signals for clients. The lower 5-bits are made available for clients.

EppDstb and EppWr are passed to clients.

EppWait is asserted inactive (HIGH) asynchronously when either EppAstb or EppDstb are active (LOW). No Wait states are requested.

The bidirectional EppDB is split in busIn and busOut for clients.

busOut always repeats EppDB.

EppDB is set HiZ when EppWr is inactive (HIGH). Otherwise, EppDB sends either regEppAdr (when EppAstb is active) or the client busIn (when EppAstb is inactive).

## BramComCtrl

This component, in conjunction with a communication module (Nexys OnBoard USB controller), a PC application ( a Diligent utility or a user-generated application) and the EppCtrlAsync Diligent Library component allows the user to read or write a Xilinx Spartan 3E FPGA Block RAM (BRAM).

### Functional Description

BramComCtrl acts as a "client" for EppCtrlAsync.

All data transfers are synchronized only with the external EppAstb and EppDstb strobes.

The BRAM controller implements a BRAM address register (regBramAdr), pointing the BRAM address for the next byte to be written or read. regBramAdr can be written or read on the falling edge of EppDstb, when regEppAdr = "xxx00001" (lower byte) or "xxx00010" (higher byte).

The BRAM data byte at the address shown by regBramAdr can be written or read on the falling edge of EppDstb, when regEppAdr is "xxx00000".

The rising edge of EppDstb increments regBramAdr by one. This way successive BRAM data read or write operations will access successive BRAM addresses (auto address post increment).

Then the BRAM controller:

- implements the regBramAdr register/counter for the BRAM address bus
- transparently sends the data bus from the EPP controller to the BRAM
- sends to the EppCtrlAsync either BRAM data or regBramAdr content
- provides the inverted EppDstb to be used as the BRAM clock
- provides the Enable signal to the BRAM, when regEppAdr is "xxx00000"
- provides the Write Enable signal to the BRAM, when regEppAdr = "xxx00000" and EPP ctrlWr is active (LOW).

## BlockRam

The BlockRam component simply instantiates a 2K X 8-bit BRAM, disabling the Set/Reset inputs and Parity bits. The BRAM content is initialized to all zeros.

## BramCfg Reference Project User's Manual

The generic BRAM configuration reference project, BramCfg, allows read/write operations on a RAM Spartan 3E FPGA BRAM. This component is used in conjunction with the Digilent Adept Suite software and the Digilent OnBoard USB circuitry to exchange data with an application running on a host PC and the BRAM. The PC application can be a Digilent program (such as ExPort) or a user-generated application that uses Digilent .dll's to communicate via the USB port.

## Using Digilent TransPort Software to Configure the BRAM

The TransPort software allows the user to access the BRAM at byte- or file-level commands. TransPort is useful for debugging purposes and for checking the command sequences before implementing them in a user-generated software application.

The software functions are as follows.

**Properties.** The user sets:

- Connection. In the pull-down menu, the desired connection is selected. Multiple communication modules (Ethernet, USB, Serial, OnBoard USB) can be connected at the same time to the host PC. A MemUtil instance uses a single such connection.
- Configure. The selected connection can be configured. For more information, see the *Adept User Manual*.

**Load File.** Sends a file fragment to a specified EPP Data Register. The user specifies:

- the source file
- the File Start Location
- the Destination Register
- the Load Entire File option
- the length of transferred data block (only if the Load Entire File option is not checked).

The load button launches the transfer.

**Store File.** Reads a specified EPP Data Register and saves data to a file (fragment). The user specifies:

- the destination file
- the File Output Mode (Append, Replace, Overwrite)
- the File Start Location (only for Overwrite)
- the Source Register
- the length of transferred data block.

The store button launches the transfer.

**Register I/O.** Reads or writes a byte or group of bytes from/to a specified EPP Data Register. The user specifies:

- the EPP Register Address (for read and write operations)
- the data (for write operations)
- the read Display Format.

There are eight lines, each able to define a register address and data. Their transfer can be performed line by line (by pressing the Read or Write button of the desired line) or as a full sequence, pressing the Read All or Write All buttons.

Multiple instances of TransPort can run at the same time, on the same PC, sharing the same connection and system board. The system board needs to be configured with the BramCfg project. Other Digilent applications (such as Digilent MemUtil and Digilent ExPort) can also share the same connection.

Memory modules are identified by the Register Address range. The BramCfg project assigns the Register Address range 0x00 ...0x1F to the BRAM module. However, only registers 0, 1, 2 are implemented. Register 3 cannot be written and returns always 0x00 when read. Registers 0...3 show 8 mirror images in the address range 0x00 ...0x1F.

Successive operations can be launched through any active TransPort instance. The user should allow a command to be fully executed before launching another one for the same communication module.

Multiple TransPort instances can be used to control different board sets. For example, different USB channels can be connected each to its own set (USB module + system board). Each system board needs to be configured with the OnBoardMemCfg project. In this case, TransPort instances are set to use different connections.

As described above, the TransPort software can be used to control any EPP interface implemented in the FPGA circuit. The following section shows how to access the specific BRAM features. The information below is also useful as a guide for software developers, showing the low-level command sequence to be implemented using Adept APIs to access BRAM via the BramCfg project.

## Reading/Writing BRAM Data with Register I/O Operations

1. Load the Memory Address Bus Register (MemAdr(10 downto 0)):
  - Write the lower address byte to EPP Register regBramAdr(7 downto 0) at EPP address adrBramAdrL = 1
  - Write the higher address byte to EPP Register regBramAdr (10 downto 8) at EPP address adrBramAdrL = 2
  - The above steps can be skipped for bytes that keep the previous value.
2. Read or write a BRAM data byte at address adrBramDB = 0. BramCfg automatically increments the regBramAdr register.
3. Repeat step 2 to read or write successive RAM locations.

## Storing the BRAM Content to a File Using the Store File Function

1. Load the Memory Address Bus Register (MemAdr(10 downto 0)):
  - Write the lower address byte to EPP Register regBramAdr(7 downto 0) at EPP address adrBramAdrL = 1
  - Write the higher address byte to EPP Register regBramAdr (10 downto 8) at EPP address adrBramAdrL = 2
  - The above steps can be skipped for bytes that keep the previous value.
2. Select the Store File tab in the TransPort software. Fill in the required fields. Select Source Register 0 (adrBramDB). Press the Store button.

## Loading a File to the BRAM Using the Load File Function

1. Load the Memory Address Bus Register (MemAdr(10 downto 0)):
  - Write the lower address byte to EPP Register regBramAdr(7 downto 0) at EPP address adrBramAdrL = 1
  - Write the higher address byte to EPP Register regBramAdr (10 downto 8) at EPP address adrBramAdrL = 2
  - The above steps can be skipped for bytes that keep the previous value.
2. Select the Load File tab in the TransPort software. Fill in the required fields. Select Source Register 0 (adrBramDB). Press the Load button.

## Using Digilent Adept SDK API Functions to Configure the Memory Module

Digilent Adept SDK provides API functions to exchange data with a project implemented in the gate-array. The TransPort application calls these API functions to perform the EPP data write and read at different addresses. These APIs are enumerated below for both versions of Adept. A user program can use these API functions to perform the steps described previously.

### Adept SDK

Download the Digilent Adept SDK and read the *Digilent Port Communications Programmers Reference Manual*.

Before performing any EPP transfer, open the device with *DpcOpenData* API and after finishing the transfer(s) close the device with *DpcCloseData*.

To replicate the actions described previously, use the following APIs.

- To write a value to the EPP register at the EPP address:  
use *DpcPutReg* API with the address in the bAddr argument and the register value in the bData argument.
- To read the value from an EPP register at the EPP address:  
use *DpcGetReg* API with the address in the bAddr argument and the pointer to the register value in the pbData argument.
- To load a file:  
use *DpcPutRegRepeat* with the address in the bAddr argument, the number of data bytes to write in the cbData argument, and the buffer of bytes in the rgbData argument.

- To store a file:  
use *DpcGutRegRepeat* with the address in the *bAddr* argument, the number of data bytes to read in the *cbData* argument, and the buffer for the bytes to return in the *rgbData* argument.

## Adept 2 SDK

Download the Digilent Adept 2 SDK and read the *Digilent Adept 2 Programmers Reference Manual*.

Before performing any EPP transfer, open the device with *DmgrOpen* or *DmgrOpenEx* and the EPP protocol enabled with the *DeppEnable* APIs. After finishing the transfer(s) disable the EPP protocol with *DeppDisable* and close the device with *DmgrClose* APIs.

To replicate the actions described previously, use the following APIs.

- To write a value to the EPP register at the EPP address:  
use *DmgrPutReg* API with the address in the *bAddr* argument and register the value in the *bData* argument.
- To read the value from an EPP register at the EPP address:  
use *DmgrGutReg* API with the address in the *bAddr* argument and pointer to register the value in the *pbData* argument.
- To load a file:  
use *DmgrPutRegRepeat* with the address in the *bAddr* argument, the number of data bytes to write in the *cbData* argument, and the buffer of bytes in the *rgbData* argument.
- To store a file:  
use *DmgrGutRegRepeat* with the address in the *bAddr* argument, the number of data bytes to read in the *cbData* argument, and the buffer for the bytes to return in the *rgbData* argument.