XAPP865 (v1.0) May 2, 2007

Hardware Accelerator for RAID6 Parity Generation / Data Recovery Controller with ECC and MIG DDR2 Controller

Author: Matt DiPaolo

Summary

A Redundant Array of Independent Disks (RAID) array is a hard-disk drive (HDD) array where part of the physical storage capacity stores redundant information. Data is regenerated from the physical storage if one or more of the disks in the array (including a single failed disk sector) or the access path to a disk in the array fails.

Of the many different RAID levels, the specific level used depends on several factors:

- Overhead of reading and writing data
- Overhead of storing and maintaining parity
- Mean Time to Data Loss (MTDL)

The newest level, RAID6, has two implementations (Reed-Solomon P+Q or Double Parity) and is the first RAID level to allow the simultaneous loss of two disks, resulting in an improved MTDL over RAID5.

Reference Design

This reference design incorporates advantages of immersed IP blocks of the Virtex™-5 architecture, including: distributed memory, FIFO memory, Digital Clock Managers (DCM), 36-Kbit Block Select RAMs (block RAM), and ECC blocks. These advantages, in a hardware acceleration block, support Reed-Solomon RAID6 (allowing for ECC support) and can support other RAID levels, when coupled with the appropriate storage array control firmware.

Introduction

In pre-RAID6 levels, when a disk fails, system firmware uses the remaining disks to regenerate the data lost from the failed disk. If another disk fails before completion of the regeneration, the data is lost forever. At this point, increased MTDL is needed. Until now, the MTDL of RAID5 satisfied the smaller size of HDDs, which due to their size have lower probability of disk failure.

With the rising popularity of inexpensive disks (such as Serial ATA (SATA) and Serial Attached SCSI (SAS)) and larger capacity disks, the Mean Time Between Failures (MTBF) of a disk has increased dramatically.

Here is an example that highlights the increased MTBF (for each disk):

In the case of 50 disks, each with 300 GB capacity, an MTBF of 5×10^5 hours (a 10^{-14} read error rate) results in one array failure in less than eight years for RAID5. RAID6 improves this to one array failure in 80,000 years.

Achieving this large MTDL for the RAID system justifies the increased overhead for:

- disk space for additional parity data
- additional reads and writes to disk drives
- system complexity required for handling multiple disk failures

To understand the Reed-Solomon RAID6, designers must have some familiarity with Galois Field (GF) mathematics. This application note describes the GF equations and refers to GF mathematical definitions. For detailed information on GF mathematics, see [Ref 1], [Ref 2], and [Ref 3].

In GF mathematics, a calculation continues to have the same number of bits as the two operands that generated it (i.e., two 8-bit numbers result in an 8-bit number). Equation 1 and Equation 2 are the definitions of GF multiplication and division. The addition/subtraction in these equations is regular-integer addition/ subtraction, which can be done using the FPGA fabric.

$$0x02 \otimes 0x08 = gfilog[gflog(0x02) + gflog(0x08)] = gfilog[0x01 + 0x03] = gfilog[0x04] = 0x10 \quad \textit{Equation 1}$$

$$0x0d \div 0x11 = gfilog[gflog0x0d - gflog0x11] = gfilog[0x68 - 0x64] = gfilog[0x04] = 0x10 \quad \textit{Equation 2}$$

This reference design implements the *gf*log and *gfi*log values with the polynomial $x^8 + x^4 + x^3 + x^2 = 1$ and generates the following look-up tables (LUTs) (Table 1 and Table 2), which are stored in block RAM.

Table 1: ***GF*LOG LUT, Stored in Block RAM**

GFLOG	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	X(1)	0	1	19	2	32	1A	C6	3	DF	33	EE	1B	68	C7	4B
1	4	64	E0	0E	34	8D	EF	81	1C	C1	69	F8	C8	8	4C	71
2	5	8A	65	2F	E1	24	0F	21	35	93	8E	DA	F0	12	82	45
3	1D	B5	C2	7D	6A	27	F9	B9	C9	9A	9	78	4D	E4	72	A6
4	6	BF	8B	62	66	DD	30	FD	E2	98	25	B3	10	91	22	88
5	36	D0	94	CE	8F	96	DB	BD	F1	D2	13	5C	83	38	46	40
6	1E	42	B6	A3	C3	48	7E	6E	6B	3A	28	54	FA	85	BA	3D
7	CA	5E	9B	9F	0A	15	79	2B	4E	D4	E5	AC	73	F3	A7	57
8	7	70	C0	F7	8C	80	63	0D	67	4A	DE	ED	31	C5	FE	18
9	E3	A5	99	77	26	B8	B4	7C	11	44	92	D9	23	20	89	2E
A	37	3F	D1	5B	95	BC	CF	CD	90	87	97	B2	DC	FC	BE	61
B	F2	56	D3	AB	14	2A	5D	9E	84	3C	39	53	47	6D	41	A2
C	1F	2D	43	D8	B7	7B	A4	76	C4	17	49	EC	7F	0C	6F	F6
D	6C	A1	3B	52	29	9D	55	AA	FB	60	86	B1	BB	CC	3E	5A
E	CB	59	5F	B0	9C	A9	A0	51	0B	F5	16	EB	7A	75	2C	D7
F	4F	AE	D5	E9	E6	E7	AD	E8	74	D6	F4	EA	A8	50	58	AF

Notes:

1. The GFLOG(00) is undefined and requires special treatment in the reference design.

Table 2: **GFILOG LUT, Stored in Block RAM**

GFILOG	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	1	2	4	8	10	20	40	80	1D	3A	74	E8	CD	87	13	26
1	4C	98	2D	5A	B4	75	EA	C9	8F	3	6	0C	18	30	60	C0

Table 2: **GFILOG LUT, Stored in Block RAM *(Continued)***

GFILOG	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
2	9D	27	4E	9C	25	4A	94	35	6A	D4	B5	77	EE	C1	9F	23
3	46	8C	5	0A	14	28	50	A0	5D	BA	69	D2	B9	6F	DE	A1
4	5F	BE	61	C2	99	2F	5E	BC	65	CA	89	0F	1E	3C	78	F0
5	FD	E7	D3	BB	6B	D6	B1	7F	FE	E1	DF	A3	5B	B6	71	E2
6	D9	AF	43	86	11	22	44	88	0D	1A	34	68	D0	BD	67	CE
7	81	1F	3E	7C	F8	ED	C7	93	3B	76	EC	C5	97	33	66	CC
8	85	17	2E	5C	B8	6D	DA	A9	4F	9E	21	42	84	15	2A	54
9	A8	4D	9A	29	52	A4	55	AA	49	92	39	72	E4	D5	B7	73
A	E6	D1	BF	63	C6	91	3F	7E	FC	E5	D7	B3	7B	F6	F1	FF
B	E3	DB	AB	4B	96	31	62	C4	95	37	6E	DC	A5	57	AE	41
C	82	19	32	64	C8	8D	7	0E	1C	38	70	E0	DD	A7	53	A6
D	51	A2	59	B2	79	F2	F9	EF	C3	9B	2B	56	AC	45	8A	9
E	12	24	48	90	3D	7A	F4	F5	F7	F3	FB	EB	CB	8B	0B	16
F	2C	58	B0	7D	FA	E9	CF	83	1B	36	6C	D8	AD	47	8E	X(1)

Notes:

1. The GFILOG(FF) is undefined and requires special treatment in the reference design.

Another differentiator of RAID6 is that data and redundancy information is stored on multiple disks. Figure 1 shows an example of a 7-disk system with five active disks and two spare disks (used as hot spare backups for data recovery). Data and parity information is striped horizontally across the drives in blocks of data. Each block is typically a multiple of 512 bytes, and data is physically stored on 512-byte sectors on the disk drives. To keep the parity drives from being a system bottleneck (which can occur in RAID4), the parity information rotates around the drives in integer increments of a block of data. The five-drive case has a 40% storage overhead for parity, while larger disk arrays can reduce this overhead (e.g., the overhead in a 12-disk system is reduced to 16%).

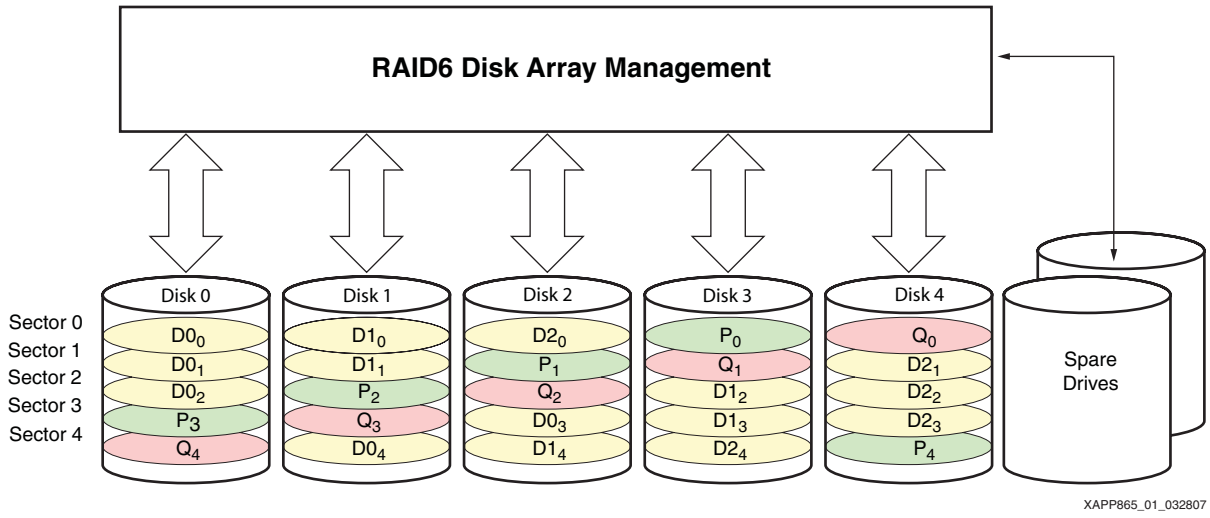


Figure 1: **RAID6 Disk Data Structure**

RAID6 Parity (P and Q) Equations

To recover from two disk failures or two bad sectors on a horizontal stripe across the storage array, RAID6 stores two unique parity values, P and Q. These values, are associated with each horizontal data block stripe on the storage array. The stripes are numbered vertically starting with 0. The data within each stripe is numbered horizontally and vertically to identify data locations within a stripe as well as within a vertical stripe index. Horizontal stripes are made up of an integer number of disk sectors. The P parity block is created by logically XORing data blocks in a horizontal stripe together, as is done in RAID4 and RAID5 systems. The second parity, Q, creates a second equation (Equation 5), which solves for two unknowns (or data failure points). For a detailed discussion on the specific GF mathematics and equations required to implement a RAID6 system, see [Ref 1]. To simplify the discussion, the equations used in this section assume a RAID6 disk array that is composed of three data disks and two parity disks for each block of data. These equations extend up to 255 disks (including the two parity disks), the mathematical limit of the equations. However, most typical applications range from 12 to 16 disks.

P Parity Block

The first RAID 6 equation represents P parity (Equation 3), which is identical to RAID5 and RAID4. A simple XOR function generates the parity block from the data values in the same sector horizontally across the data drives in an array group. P_0 XORs the $D0_0$, $D1_0$, and $D2_0$ (Figure 1).

$$P_N = D0_N \oplus D1_N \oplus D2_N \oplus \ldots \oplus D(M-1)_N \quad \text{Equation 3}$$

N = 0 to maximum number of blocks (sectors) on the disk drive

M = number of data disks in the array group

Equation 4 is the first equation for Sector 0 of a three data drive system. In the event of a single drive failure, any data block can be regenerated using this equation.

$$P_0 = D0_0 \oplus D1_0 \oplus D2_0 \quad \text{Equation 4}$$

Q Parity Block

The RAID6 Q parity assigns a GF multiplier constant associated with each data disk drive (Equation 5). The constant applies only to the data blocks striped horizontally across the array, not to each drive in the array. Each data block is GF multiplied by a constant, before adding to the data elements of the next data drive. The g constants are determined from the GFILOG LUT (Table 2). If another drive is added to the array, then g3 = gfilog(3) = 0x8.

$$Q_N = (g0 \otimes D0_N) \oplus (g1 \otimes D1_N) \oplus (g2 \otimes D2_N) \oplus \ldots \oplus g(M-1) \otimes D(M-1)_N \quad \text{Equation 5}$$

Equation 6 is the equation for the third sector of a three data drive system.

$$Q_2 = (0x01 \otimes D0_2) \oplus (0x02 \otimes D1_2) \oplus (0x04 \otimes D2_2) \quad \text{Equation 6}$$

N = 0 to maximum number of blocks (sectors) on the disk drive

M = number of data disks in the array group

Updating Data, P, and Q Blocks

Whenever a host-write command occurs, the P and Q parities must be updated. For example, the data on Sector 0 of Disk 1 is being written. For this specific sector, the old data block along with the old P and Q parity must be read. Equation 7 through Equation 10 calculate the updated parities. This is commonly referred to as the *write penalty*, encountered in RAID storage systems. Prior to writing new data and new parity blocks to the array, the old data and old parity must first be read from the array and placed in the controller memory, so the new parity can be calculated from the old data and old parity information stored on disk.

$$P_{N\_NEW} = P_{N\_OLD} \oplus D1_{N\_OLD} \oplus D1_{N\_NEW} \quad \text{Equation 7}$$

$$Q_{N\_NEW} = Q_{N\_OLD} \oplus (g1 \otimes (D1_{N\_OLD} \oplus D1_{N\_NEW})) \quad \text{Equation 8}$$

$$P_{0\_NEW} = P_{0\_OLD} \oplus D1_{0\_OLD} \oplus D1_{0\_NEW} \quad \text{Equation 9}$$

$$Q_{0\_NEW} = Q_{0\_OLD} \oplus [0x02 \otimes (D1_{0\_OLD} \otimes D1_{0\_NEW})] \quad \text{Equation 10}$$

Three Data Drive System — Disk Failure Example

With P and Q parity generated and striped across the five-disk (three data) array, as shown in Figure 1, a single or dual-disk failure within the array does not cause loss of data to the host application. Any data block can be regenerated using the P and Q parity information stripped across the array. Double data block loss is more difficult to regenerate than single data block loss. The equations used to regenerate data blocks are discussed in this section.

Typically, arrays either contain hot spares (Figure 1), used when a disk failure occurs, or service personnel are rapidly dispatched to replace the failed disks. Another system implementation that ensures no data is lost is the use of an entire hot spare RAID6 array to facilitate transferring data, reconstructing data, and updating P and Q parity. The system operates in a degraded mode until the disks are replaced or the hot spares are switched. Data regenerated from the remaining information is striped horizontally on the remaining disks. Regenerated data is then sequentially transferred to the hot spare disk until all data (P and Q blocks) has been updated. At that time, the system is back in normal operating mode. Different regeneration algorithms are used depending upon whether a single or dual-disk failure occurs.

In any RAID system, different types of disk failure scenarios occur. Every data disk added to the Bunch of Disks (BOD) adds more variables to the parity generation or data regeneration equations that are discussed later in this section. A system with three data disks is used to facilitate understanding of data recovery equations that are used in the reference design. Array management firmware is responsible for managing the data reconstruction process. The algorithms to regenerate data depends on the number of data disks in the array. Assuming that the regeneration firmware starts at Sector 0 and works its way to the maximum sector on the disk, the equations used to regenerate data are repetitive.

- If a single disk failure occurs when the P block sector is ***not*** lost, data is regenerated from the remaining data and parity block.
- If the P block sector is lost, data does not have to be regenerated because the data is valid and stored on one of the remaining disks in the array.
- If a hot spare has been activated and the array management software is rebuilding a replaced drive, then the P and Q values are regenerated and copied to the new drive.

For this example, assume that Disk 0 fails. Equation 11, Equation 12, and Equation 13 are used to regenerate data, sector by sector vertically through a disk (Figure 1), read, and returned to a host system.

$$D0_0 = D1_0 \oplus D2_0 \oplus P_0 \quad \text{Equation 11}$$

$$D0_1 = D1_1 \oplus P_1 \oplus D2_1 \quad \text{Equation 12}$$

$$D0_2 = P_2 \oplus D1_2 \oplus D2_2 \quad \text{Equation 13}$$

P_3 does not need to be regenerated for host read commands. Q_4 does not need to be regenerated for host read commands.

The system ***cannot*** accept new write data because there is no space to store the information unless hot spare drives are present or the failed disks have been replaced. The following discussion assumes that the hot spare drives are enabled and ready to accept regenerated data and parity blocks from the RAID controller.

In this example, if a dual disk fails, one of the eight equations (double data, P and Q, P and D_0, P and D1, P and D2, Q and Do, Q and D1, and Q and D2) must be used to regenerate data and parity blocks in a RAID6 system. Determining which equation to use depends upon the set of disks that fail and the sector of the disk that is currently being regenerated. Disk array management firmware is typically responsible for managing the disk interface and regeneration algorithms. Array management firmware is not part of this reference design. See Equation 16, Equation 17, Equation 20, Equation 21, Equation 26, Equation 27, Equation 34, and Equation 35.

P Parity Generation or Regeneration

P Parity Generation and P Parity Regeneration are the simplest regeneration possibilities described in this application note; in either case, the data is all that is needed. The simple XOR, Equation 14, finds the P value. In this case, the third drive has failed and the parity is being regenerated. Regenerated parity is written to one of the hot spare drives under control of the array management firmware.

$$P_1 = D0_1 \oplus D1_1 \oplus D2_1 \qquad \textit{Equation 14}$$

Q Parity Generation or Regeneration

Q Parity Generation or Regeneration requires GF mathematics, using the GFLOG and GFILOG LUTs (hard-coded into block RAM) (see Table 1 and Table 2) and the fabric for integer addition. Equation 15 regenerates Sector 2 of Disk 2 shown in Figure 1. Q parity is written to one of the hot spare drives under control of the array management firmware.

$$Q_2 = (0x01 \otimes D0_2) \oplus (0x02 \otimes D1_2) \oplus (0x04 \otimes D2_2) \qquad \textit{Equation 15}$$

P and Q Regeneration

P and Q Regeneration is a superset of the previous two cases ("P Parity Generation or Regeneration" and "Q Parity Generation or Regeneration") because all of the data disks are still available. To save calculation time, Equation 16 and Equation 17 are run at the same time, using multiple datapaths. In this case, Sector 4 of Disk 0 and Disk 4 have failed. Regenerated P and Q parity blocks are written to the hot spare drives under control of the array management firmware.

$$P_4 = D0_4 \oplus D1_4 \oplus D2_4 \qquad \textit{Equation 16}$$

$$Q_4 = (0x01 \otimes D0_4) \oplus (0x02 \otimes D1_4) \oplus (0x04 \otimes D2_4) \qquad \textit{Equation 17}$$

Q and Data Regeneration

Q and Data Regeneration is the next level of complexity. Since the P parity is intact, the data can be recovered with the simple XOR equation. The Q parity regeneration is possible using the recovered data. The D1 and D2 GF multiplication of the Q parity is calculated in parallel to the D0 parity to reduce latency. Equation 18 through Equation 21 show the scenario of Disk 0 and Disk 1 failing and regenerating Sector 4, as shown in Figure 1. When a host read command is in progress, the regenerated data block is returned to the host. Q parity and regenerated data blocks are written to the hot spare drives under control of the array management firmware.

$$D0_N = D1_N \oplus D2_N \oplus \ldots \oplus D(M-1)_N \oplus P_N \quad \text{Equation 18}$$

$$Q_N = (0x01 \otimes D0_N) \oplus (0x02 \otimes D1_N) \oplus (0x04 \otimes D2_N) \oplus \ldots \oplus (g(M-1) \otimes D(M-1)_N) \quad \text{Equation 19}$$

$$D0_4 = D1_4 \oplus D2_4 \oplus P_4 \quad \text{Equation 20}$$

$$Q_4 = (0x01 \otimes D0_4) \oplus (0x02 \otimes D1_4) \oplus (0x04 \otimes D2_4) \quad \text{Equation 21}$$

P and Data Regeneration

With P and data regeneration, lost data must be generated with the Q parity equations, creating an intermediate Q' value. The P parity then uses the newly generated data to complete the XOR equation. The general case equations, Equation 22 through Equation 24 (assuming $D0_N$ is lost), are followed by the specific equations (Equation 25 through Equation 27) for the case where Sector 1 is lost for both Disk 1 and Disk 2, based on the example shown on Figure 1. When a host read command is in process, the regenerated data block is returned to the host. Regenerated data and parity blocks are written to the hot spare drives under control of the array management firmware.

$$Q'_N = (0x02 \otimes D1_N) \oplus (0x04 \otimes D2_N) \oplus \ldots \oplus (g(M-1) \otimes D(M-1)_N) \quad \text{Equation 22}$$

$$D0_N = 0x01 \otimes (Q_N \oplus Q'_N) \quad \text{Equation 23}$$

$$P_N = D0_N \oplus D1_N \oplus D2_N \oplus \ldots \oplus D(M-1)_N \quad \text{Equation 24}$$

$$Q'_1 = (0x02 \otimes D1_1) \oplus (0x04 \otimes D2_1) \quad \text{Equation 25}$$

$$D0_1 = 0x01 \otimes (Q_1 \oplus Q'_1) \quad \text{Equation 26}$$

$$P_1 = D0_1 \oplus D1_1 \oplus D2_1 \quad \text{Equation 27}$$

Double Data Regeneration

Double data regeneration is the most complicated case in RAID6 and in the reference design. Two intermediate calculations (P' and Q') are required. The general equations (Equation 28 through Equation 31) are lengthy. The assumption is that $D0_N$ and $D1_N$ are lost in the example shown in Figure 1. Since there are only three data disks in this example, and two disks are missing Sector 0 of Disk 0 and Disk 1, the intermediate equations (Equation 32 through Equation 35) are defined by the remaining data in Disk 2. When a host read command is in progress, the regenerated data blocks are returned to the host. Regenerated data blocks are written to the hot spare drives under control of the array management firmware.

$$Q'_N = (0x04 \otimes D2_N) \oplus (0x08 \otimes D3_N) \oplus \ldots \oplus (g(M-1) \otimes D(M-1)_N) \quad \text{Equation 28}$$

$$P'_N = D2_N \oplus \ldots \oplus D(M-1)_N \quad \text{Equation 29}$$

$$D0_N = (0x01 \oplus 0x02)^{-1} \otimes ((0x02 \otimes (P_N \oplus P'_N)) \oplus Q_N \oplus Q'_N) \quad \text{Equation 30}$$

$$D1_N = D0_N \oplus (P_N \oplus P'_N) \quad \text{Equation 31}$$

$$Q'_0 = (0x04 \otimes D2_0) \quad \text{Equation 32}$$

$$P'_0 = D2_0 \quad \text{Equation 33}$$

$$D0_0 = (0x01 \oplus 0x02)^{-1} \otimes ((0x02 \otimes (P_0 \oplus P'_0)) \oplus Q_0 \oplus Q'_0) \quad \text{Equation 34}$$

$$D1_0 = D0_0 \oplus (P_0 \oplus P'_0) \quad \text{Equation 35}$$

Reference Design

System Architecture

This application note assumes the system architecture shown in Figure 2.

The system contains a RAID with ECC host controller on an ML555 demonstration board. This board contains a Virtex-5 LX50T FPGA (in green) along with a DDR2 SODIMM memory and a Serial ATA (SATA) connection that allows connection to a port multiplier. The port multiplier connects to multiple SATA HDDs. A SATA protocol controller that interfaces with the Xilinx Memory Interface Generator (MIG) memory controller and the system controller can be implemented in the FPGA. Replacing the SATA protocol controller with Serial Attached SCSI (SAS), Fibre Channel (FC), or any other disk interface protocol is possible depending on the overall system requirements.

This application note concentrates on the hardware acceleration portion of a RAID6 with ECC system, as shown in red in Figure 2, and includes:

- Simplified system controller (testblock module)
- DDR2 memory controller that contains 8 additional data for the ECC parity created in the ECC block discussed later in this section (the memory controller treats these 8 bits as extra data)
- RAID with ECC IP block

The other portions of Figure 2 are only there to show a possible system-level implementation using the RAID with ECC-IP hardware connected in a SATA system.

The simplified system controller block:

- Controls the RAID6 with ECC hardware
- Sets up pointers to the data and parity blocks of DDR2 memory.
- Sets up the hardware:
  - Creates values for D0, D1, and D2
  - Loads the values into the DDR2 memory
  - Creates the requests for the different regeneration cases (discussed later in this section)
  - Checks the data in the DDR2 memory against the expected values
  - Sets the LEDs to indicate the pass/fail of the test (details in the `README` file)

The reference design does not include the disk array management or the Serial ATA interface to the disk drives. The simplified system controller included in the reference design generates data and parity blocks that come from an HDD connection and is not required with this reference design. The testblock generates data placed in the DDR2 memory that serves as a cache for the RAID with ECC hardware accelerator. A MIG memory controller controls the DDR2 memory. The memory controller provides access to the DDR2 memory. The memory controller connects to both the testblock and the RAID with ECC hardware accelerator via a multiplexer.

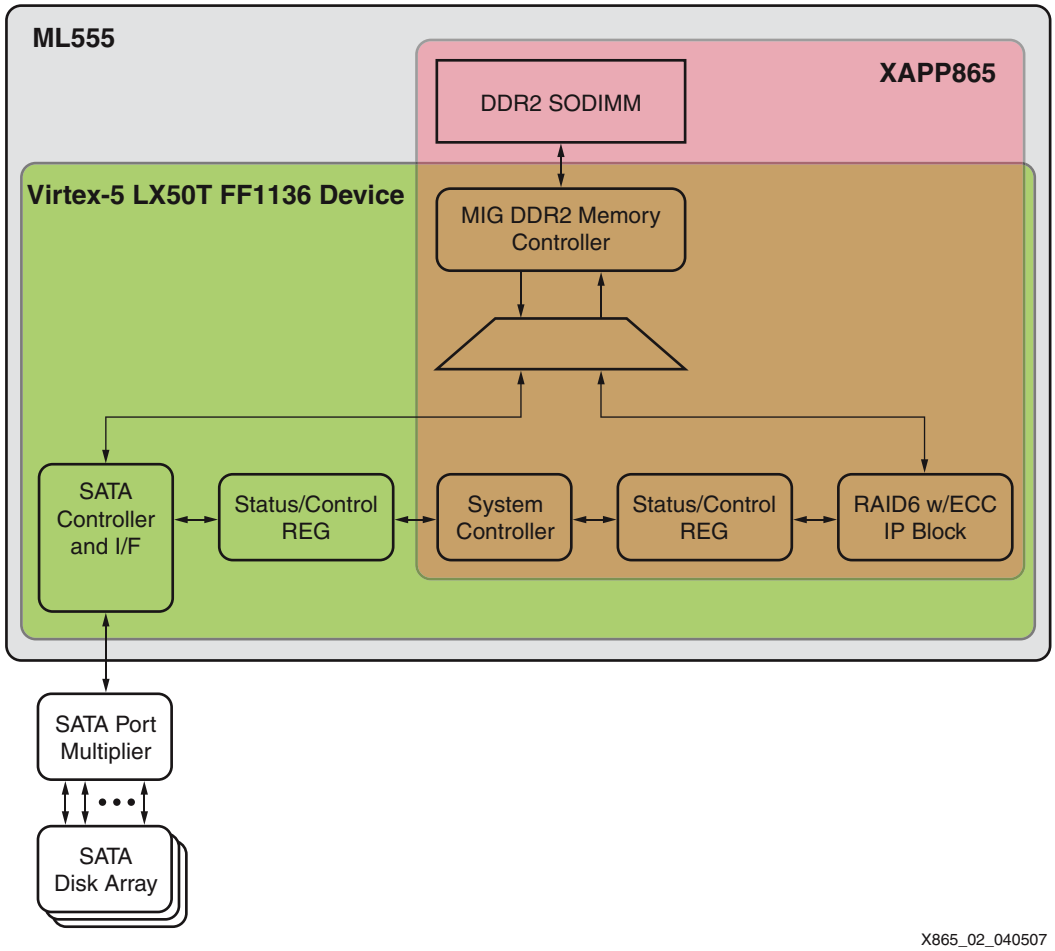


Figure 2: **Possible RAID6 System Implementation**

Hardware Accelerator

A RAID6 hardware accelerator is a mathematically intensive RAID level. In previous RAID levels, the only computation is a simple XOR of the data. However, the RAID6 calculations require each block of data to be stored in a memory buffer, and then read into a temporary data buffer while being XORed or added (using Galois mathematics) to other data elements.

A large amount of data manipulation is required. Data manipulation is time intensive, but hardware implementations are faster than processor-only register calculations because hardware provides parallel manipulation of data blocks, clock-rate LUT access, clock rate integer addition of two 8-bit values, and multipliers much faster than a processor. The RAID6 calculations are a small portion of the overall time period used (including disk seek, disk access, data transfer (HDD to/from cache memory), and the RAID6 acceleration).

The RAID hardware accelerator memory interface is based on the topography of the ML555 board (which is the verification platform). This determines the data flow block discussed in "Data Flow for Different Regeneration Cases," page 11.

The hardware accelerator has four main blocks (Figure 3).

- Data Manipulation Block (DMB)
- RAID Finite State Machine (FSM)
- Device Control Register FSM (DCR FSM)
- MIG_MEM_IF

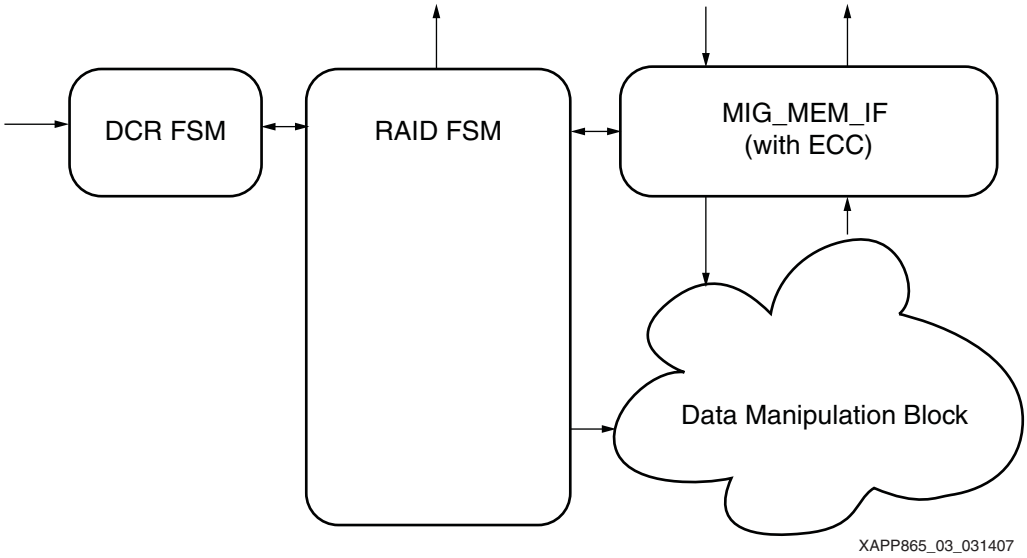


Figure 3: **RAID6 Hardware Accelerator Block Diagram**

Data Manipulation Block

The Data Manipulation Block (DMB) shown in Figure 4 is the modular block of logic that actually performs the mathematical operations on one byte of the data. (Depending on the system data width, more DMB logic blocks can be added to support larger data widths.) This reference design is set up for an 8-byte implementation for the LX50T and a 16-byte implementation for the LX110T (8 and 16 instantiations of the DMB, respectively). It can create parity and regenerate data of any case discussed in the "Three Data Drive System — Disk Failure Example" section. Since all of the equations involve GF addition (XOR) or GF multiplication (GFILOG, GFLOG, and integer addition), there are several main building blocks of the DMB:

- XOR_BRAM (blue blocks in Figure 4)
- MUX_BRAM (green blocks in Figure 4)
- RAID_MULT_4 (yellow blocks in Figure 4)

The XOR_BRAM block completes the simple XOR function for calculating the P parity. The XOR_BRAM block contains a 32-bit register and a 2:1 MUX for use in several other regeneration scenarios.

The MUX_BRAM block(s) hold calculated data blocks until other data blocks are either retrieved from the MIG memory controller or calculated in other blocks of the DMB. The 2:1 MUXs are used for many regeneration possibilities that the DMB covers.

The RAID_MULT_4 block completes the GF multiplication portion of the equations (see Equation 1). This is done in three steps.

1. The GFLOG is calculated. This is implemented with a block RAM used as a LUT (one dual port block RAM is used for two LUTs). The GFLOG table (Table 1) is hard-coded into the block RAM and is addressed by the data coming into the RAID_MULT_4 block.
2. The result of the GFLOG table (Table 1) is added to the GFLOG of the Gn constant (the GFLOG result is hard-coded for the small number of possible outcomes with the reference design using a 3 data disk system. For systems with a large number of disk drives in an array, the Gn constant multiplexer can be implemented as a LUT, with the input controlled through a DCR register to select the appropriate Gn constant) by a portion of one of the 8-bit adders.

3. The sum enters the GFILOG LUT implemented in another block RAM (similar to the GFLOG implementation). If the input to the RAID_MULT_4 block is zero, the result of this special GF multiplication case is zero.

Several other XOR and MUXs tie these building blocks together, as shown in Figure 4. Detailed analysis of the data flow through the DMBs, covering all of the logic equations implemented in the reference design, is covered in the "Data Flow for Different Regeneration Cases" section that follows.

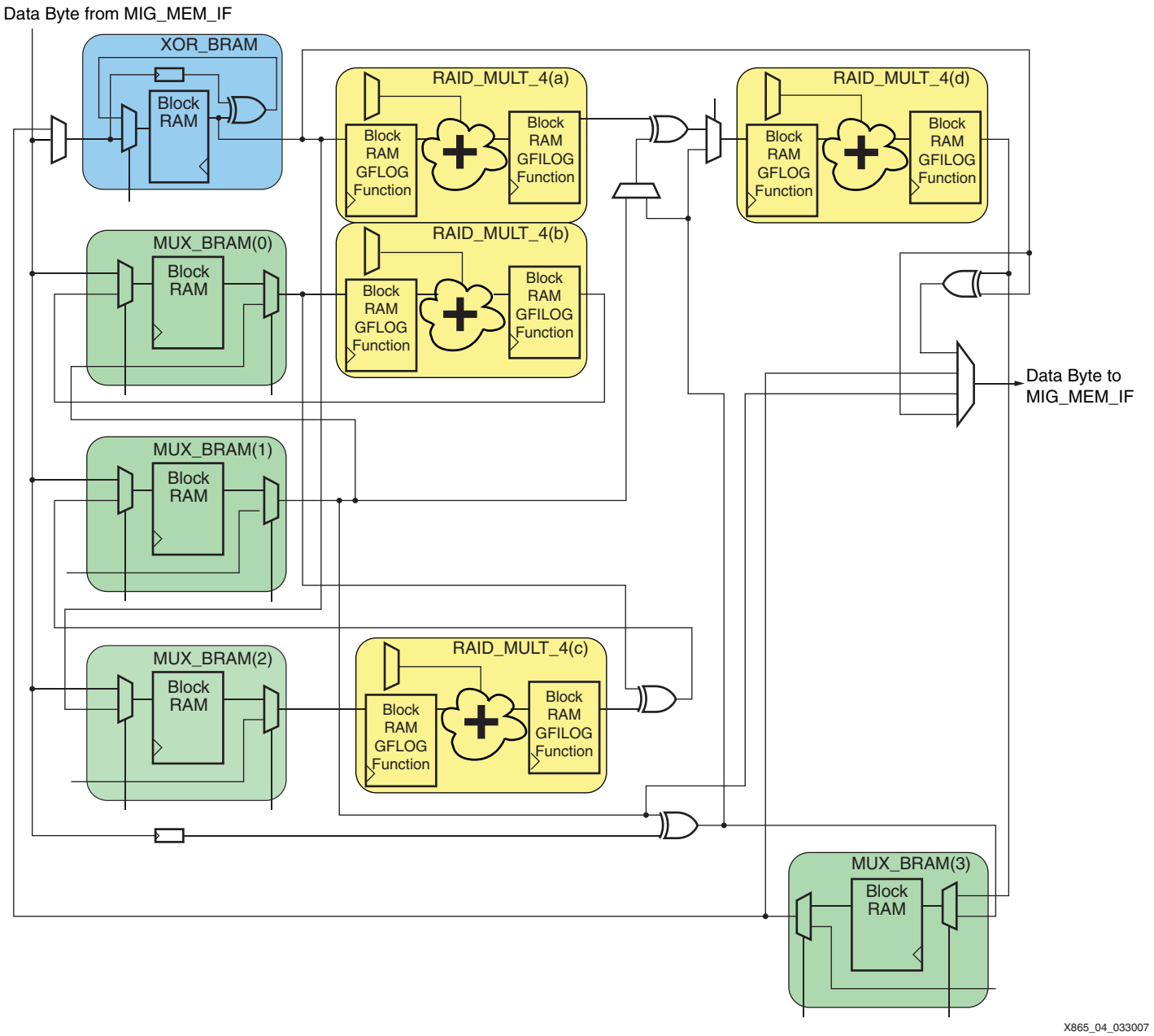


Figure 4: **Data Manipulation Block Logic for One Byte**

Data Flow for Different Regeneration Cases

This section summarizes data flow through DMBs for the different regeneration scenarios for the five disk drive example. Detailed analysis of the data flow through the DMB blocks, covering all of the logic equations implemented in the reference design, is covered in the "Data Flow for

Different Regeneration Cases" section. The actual hardware has many cycle-to-cycle dependencies that are not described here (to simplify readability and understanding).

Generate/Regenerate P Parity

1. Write the first data block from the MIG memory controller into XOR_BRAM.
2. XOR the second data block from the MIG memory controller with output of XOR_BRAM and write it back into XOR_BRAM.
3. XOR the third data block from the MIG memory controller with output of XOR_BRAM and write it back into XOR_BRAM.
4. When the MIG memory controller is ready, read P parity out of the XOR_BRAM through the 4:1 MUX to the MIG memory controller.

Generate/Regenerate Q Parity

1. Write the first data block from the MIG memory controller into MUX_BRAM_0.
2. Read the data block out of MUX_BRAM_0 and run through RAID_MULT_4(b), then write the result back into MUX_BRAM_0.
3. Write the second data block from the MIG memory controller into MUX_BRAM_2, XOR the read data out of MUX_BRAM_0 with the data read from MUX_BRAM_2 passed through RAID_MULT_4(c), and then write the result into MUX_BRAM_1.
4. Write the third data block from the MIG memory controller into MUX_BRAM_2, XOR the read data out of MUX_BRAM_2 passed through RAID_MULT_4(c) with read data out of MUX_BRAM_1 through the MUX_BRAM_0, and then write the result into MUX_BRAM_1.
5. When the MIG memory controller is ready, read the Q parity block out of the MUX_BRAM_1 through the 4:1 MUX to the MIG memory controller.

Regenerate Data

1. Write the P parity block from the MIG memory controller into XOR_BRAM.
2. XOR a data block from the MIG memory controller with output of XOR_BRAM and write it back into XOR_BRAM.
3. XOR another data block from the MIG memory controller with output of XOR_BRAM and write it back into XOR_BRAM.
4. When the MIG memory controller is ready, read reconstructed data out of the XOR_BRAM through the 4:1 MUX to the MIG memory controller.

Regenerate Q (a) and P (b) Parity

Letters ***(a)*** and ***(b)*** are used to distinguish *simultaneous parallel calculations*.

1. Write the first data block from the MIG memory controller into MUX_BRAM_0 and XOR_BRAM memory. This step stores identical data into two block RAM elements in preparation for a parallel calculation in step 2.
2. *(a)* Write the second data block from the MIG memory controller into MUX_BRAM_2, XOR read data out of MUX_BRAM_0 with the data read from MUX_BRAM_2 passed through RAID_MULT_4(c), and then write the result into MUX_BRAM_1.

   (b) XOR the second data block from MIG memory controller with output of XOR_BRAM and write it back into XOR_BRAM.
3. *(a)* Write the third data block from the MIG memory controller into MUX_BRAM_2, XOR read data of MUX_BRAM_2 passed through RAID_MULT_4(c) with read data out of MUX_BRAM_1 passed through the MUX_BRAM_0, and then write the result into MUX_BRAM_1.

   (b) XOR the third data block from the MIG memory controller with output of XOR_BRAM and write it back into XOR_BRAM.

4. *(b)* When the MIG memory controller is ready, read the P parity block out of the XOR_BRAM through the 4:1 MUX to the MIG memory controller.
5. *(a)* When the MIG memory controller is ready, read the Q parity block out of the MUX_BRAM_1 through the 4:1 MUX to the MIG memory controller. Writes to the MIG memory controller must be single threaded as the destination addresses are different for the P and Q parity block information.

Regenerate Q (a) Parity and Data (b)

Letters ***(a)*** and ***(b)*** are used to distinguish *simultaneous parallel calculations*.

1. *(b)* Write the P parity block from the MIG memory controller into XOR_BRAM.
2. *(a)* Write the first data block from the MIG memory controller into MUX_BRAM_0, read data out of MUX_BRAM_0 and pass through RAID_MULT_4(b), and then write back into MUX_BRAM_0.

   (b) XOR first data block from the MIG memory controller with the parity block output of XOR_BRAM and write it back into XOR_BRAM.
3. *(a)* Write another data block from the MIG memory controller into MUX_BRAM_2, XOR the read data out of MUX_BRAM_2 passed through RAID_MULT_4(c) with read data out of MUX_BRAM_0, and then write into MUX_BRAM_1.

   (b) XOR the second data block from the MIG memory controller with output of XOR_BRAM and write it back into XOR_BRAM, which now contains the regenerated data block.
4. *(b)* When the MIG memory controller is ready, read the lost data block out of XOR_BRAM through the 4:1 MUX to the MIG memory controller, plus write the lost data block into MUX_BRAM_2.
5. *(a)* Read data block from MUX_BRAM_2 and run through RAID_MULT_4(c), read data out of MUX_BRAM_0 and XOR these data blocks, and write the result into MUX_BRAM_1.
6. *(a)* When the MIG memory controller is ready, read the Q parity block out of the MUX_BRAM_1 through the 4:1 MUX to the MIG memory controller.

Regenerate P (a) Parity and Data (b)

1. *(a)* Write the first data block from the MIG memory controller into XOR_BRAM.

   (b) Write the first data block from the MIG memory controller into MUX_BRAM_0, read data out of MUX_BRAM_0 and pass through RAID_MULT_4(b), then write back into MUX_BRAM_0.
2. *(a)* XOR the second data block from the MIG memory controller with output of XOR_BRAM and write it back into XOR_BRAM.

   (b) Write the second data block from the MIG memory controller into MUX_BRAM_2, XOR read data out of MUX_BRAM_2 passed through RAID_MULT_4(c) with read data out of MUX_BRAM_0, and then write into MUX_BRAM_1.
3. *(b)* XOR the read Q' value from MUX_BRAM_1 with registered Q parity, pass that result through RAID_MULT_4(d), and then write lost data into MUX_BRAM_3.
4. *(b)* When the MIG memory controller is ready, read the lost data block from MUX_BRAM_3 through the 4:1 MUX to the MIG memory controller. If a host read command is in process, and this data block is the target of the read command, the array management firmware can return the regenerated data to the host after the data has been written to the DDR2 memory.

   (a) XOR the data block from *(b)* with the output of the XOR_BRAM, and write the P parity block into the XOR_BRAM.
5. *(a)* When the MIG memory controller is ready, read the P parity block from the XOR_BRAM through the 4:1 MUX to the MIG memory controller.

Regenerate Double Data

1. Write only known good data blocks from the MIG memory controller into XOR_BRAM and MUX_BRAM_0.
2. Read the data block out of MUX_BRAM_0, pass it through RAID_MULT_4(b), and then write the value back into MUX_BRAM_0.
3. XOR the P parity block from the MIG memory controller with the output of the XOR_BRAM, and write it back into the XOR_BRAM.
4. Write the Q parity block from the MIG memory controller into MUX_BRAM_2, and XOR the read data block out of MUX_BRAM_2 passed through RAID_MULT_4(c) with read data out of MUX_BRAM_0, and then write the result into MUX_BRAM_1. MUX_BRAM_0 contains the only known good data block at this point of the calculation for this data stripe.
5. XOR the read value out of XOR_BRAM, which is passed through RAID_MULT_4(a) with the read value output of MUX_BRAM_1, then pass this result through RAID_MULT_4(d), and finally write the lost data value into MUX_BRAM_3.
6. When MIG memory controller is ready, read the first data block from MUX_BRAM_3 through the 4:1 MUX to the MIG memory controller, XOR this lost data block with the output of the XOR_BRAM, and write the result back into the XOR_BRAM.
7. When the MIG memory controller is ready, read the second data block out of the XOR_BRAM through the 4:1 MUX to the MIG memory controller. Two single-threaded writes to the MIG memory controller are required because the two regenerated data blocks must be written into two different memory locations in DDR2 memory.

Update P and Q Parity

These calculations are only required when a host write command is issued to the array and new parity blocks must be generated to replace old parity blocks. Array management firmware is responsible for retrieving data and parity information from the disk array and placing the contents in the DDR2 memory. Writing to an array that is being constructed can be permitted, depending on the system specific implementations (which are beyond the scope of this application note).

1. Write the old data block from the MIG memory controller into XOR_BRAM and MUX_BRAM_0.
2. Write the new data block from the MIG memory controller into MUX_BRAM_2, XOR the new data block from the MIG memory controller with output of XOR_BRAM, and then write the result back into XOR_BRAM.
3. XOR the values out of MUX_BRAM_0 with the value read out of MUX_BRAM_2, which are passed through (g=1 so data is not modified) RAID_MULT_4(c), and then write the value into MUX_BRAM_1.
4. XOR the old P parity block from the MIG memory controller with the output of the XOR_BRAM, and write the result back into XOR_BRAM.
5. XOR the registered old Q parity block with the output of the MUX_BRAM_1, then write the result in MUX_BRAM_3.
6. When the MIG memory controller is ready, read the new P parity out of the XOR_BRAM through the 4:1 MUX to the MIG memory controller.
7. When the MIG memory controller is ready, read the new Q parity out of the MUX_BRAM_3 through the 4:1 MUX to the MIG memory controller. At this point in the calculation, array management firmware can write to the disk array and update the new data, P parity, and Q parity blocks from information contained in the DDR2 memory.

RAID FSM

The RAID Finite State Machine (FSM) is the main control logic of the reference design. It controls the MIG_IF and ECC block along with all the DMB signals, including the block RAM

address, read, write control, and MUX select lines. This FSM maintains all of the data pipelining and efficiently passes the data through the DMB.

Status/Control Registers and DCR FSM

The hardware accelerator depends on firmware, which in this reference design is emulated with the testblock, to place disk data into the DDR2 memory and to manage the memory buffer. The testblock controls the hardware accelerator with a set of registers. Hardware status is provided to the testblock on status registers. Control registers are provided to point to data and parity block starting addresses in the external DDR2 memory as well as control registers to define the type of regeneration calculation to be performed. Another register sets the size of the data block calculated; the default is 512 bytes. Status registers indicate when the calculation, on a block basis (determined by the RAID_SIZE register), is finished.

Ten registers are provided to interface to the accelerator hardware. Table 3 contains the address and a description of the register function. The testblock reads and writes to all of these registers. All registers are readable by the hardware accelerator and only RAID_RECON is writable.

Table 3: **Status/Control Register Descriptions**

Register Name	Description
RAID_RECON	Bit[0] indicates to the RAID IP a regenerate request when `1`. All other DCR registers should be configured for the RAID6 calculation to be performed prior to asserting this bit.
	Bit[0] indicates to the processor a regenerate is complete when `0`. The processor polls this register to determine when the RAID6 hardware has completed the requested operation.
RAID_LOST	Indicates to the RAID IP which type of disks has failed: `0x01` Q regenerate `0x02` P regenerate `0x03` P &Q regenerate `0x04` D0 regenerate(1) `0x05` D0 & Q regenerate `0x06` D0 & P regenerate `0x08` D1 regenerate(1) `0x09` D1 & Q regenerate `0x0A` D1 & P regenerate `0x0C` D0 & D1 regenerate `0x10` D2 regenerate(1) `0x11` D2 & Q regenerate `0x12` D2 & P regenerate `0x14` D2 & D0 regenerate `0x18` D1 & D2 regenerate
RAID_1	Memory base address pointer to first data block stored in DDR2 memory(2).
RAID_2	Memory base address pointer to second data block stored in DDR2 memory(3).
RAID_3	Memory base address pointer to third data block stored in DDR2 memory.
RAID_4	Memory base address pointer reserved for a fourth data block stored in DDR2 memory.
RAID_P	Memory base address pointer to the P parity block stored in DDR2 memory.
RAID_Q	Memory base address pointer to Q parity block stored in DDR2 memory.

Table 3: **Status/Control Register Descriptions *(Continued)***

Register Name	Description
RAID_M	Bits [31:29] indicates to the RAID IP which type of regeneration is requested: 000 indicates a single D regeneration 001 indicates a single P regeneration 010 indicates a single Q regeneration 011 indicates updating P & Q for a Data write 100 indicates a double D & D regeneration 101 indicates a double D & Q regeneration 110 indicates a double D & P regeneration 111 indicates a double Q & P regeneration
RAID_LED	Indicates to the system or user if RAID test is in progress or completed. Controls diode DS11 on the ML405 hardware platform.
RAID_SIZE	Bits [27:31] indicate to the RAID IP the horizontal size of the data and parity blocks: 00000 512 byte (default) 00001 1 Kbyte 00010 2 Kbyte 00100 4 Kbyte 01000 8 Kbyte 10000 16 Kbyte
SBERR	Indicates when at least one single bit error occurred but has been corrected. A write to address x09 clears this register.
DBERR	Indicates when at least one double bit error has occurred and data is now invalid. A write to address x0A clears this register.
INJECT	Writing the following values to register x0B allows either a single or double bit error to be injected. [00] no injected errors [01] or [10] cause a single bit errors, [11] double bit errors

Notes:

1. Also indicates data to be written when the RAID_MODE = 011
2. Indicates old data when RAID_MODE = 011
3. Indicates new data when RAID_MODE = 011

MIG Memory Controller and MIG_MEM_IF (with ECC)

The memory controller in this reference design was created by the MIG tool for the memory on the ML555. This usage of the MIG memory controller allows the other types and sizes of memories to be connected to the RAID with ECC IP block based on system requirements.

The MIG_MEM_IF block uses several FIFOs to stage data write and reads in the large burst blocks that the RAID with ECC processes. Also this module utilizes the ECC blocks of the FIFO_36_72 to generate ECC parity and check and correct the data based on the ECC parity bits. Figure 5 shows the enabled paths for the write and read FIFO and the ECC blocks that are enabled. The FIFO that puts the RAID data into the DDR2 memory generates the ECC parity to write into the DDR2 memory. The FIFO that takes the read data of the DDR2 memory uses the ECC correction block to correct bit errors on the DDR2 memory. For a 40-bit DDR2 memory interface, only one FIFO_36_72 is required. For a 72-bit DDR2 memory interface, two FIFO_36_72 instantiations are required.

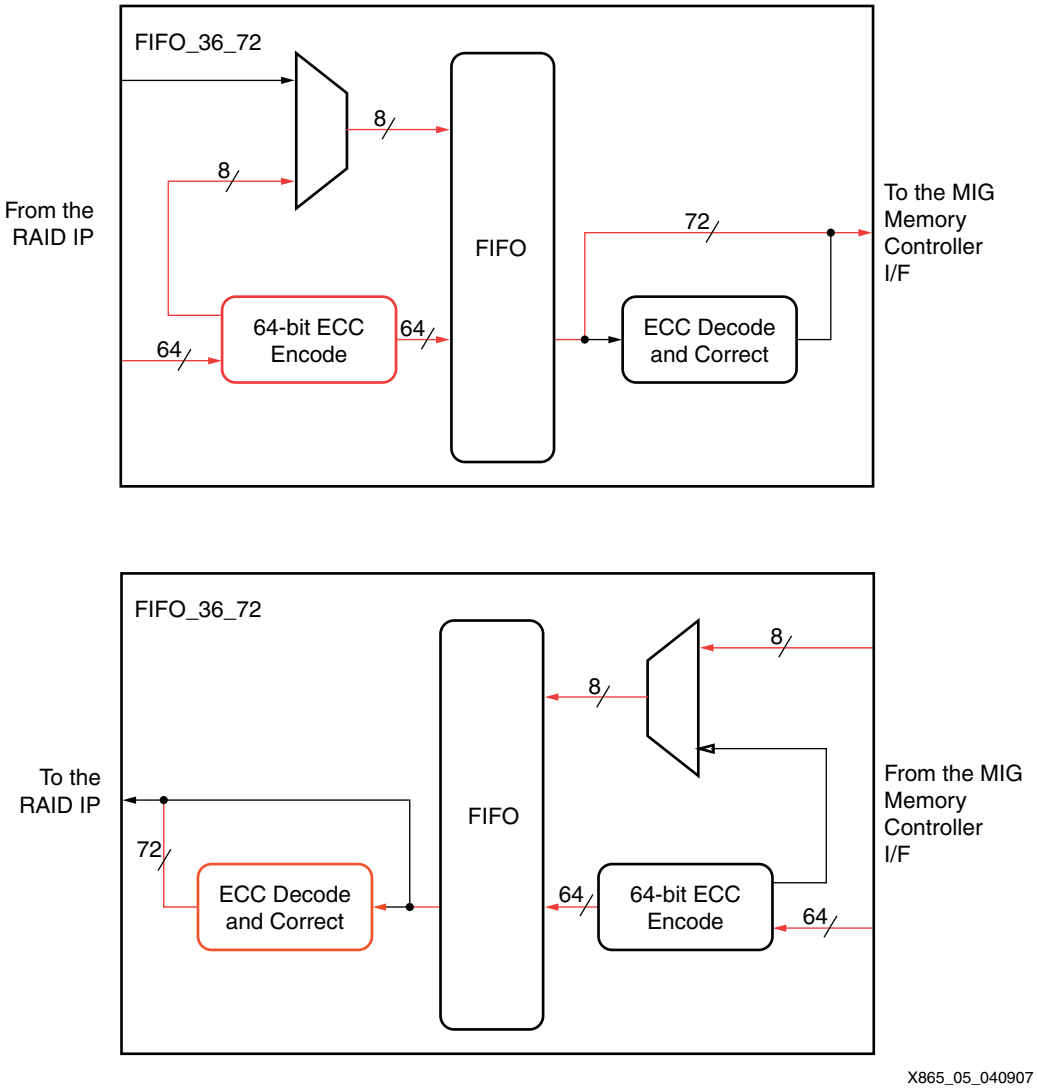


Figure 5: **ECC Implementation**

Figure 6 shows data from the memory where a single bit error was injected (x0A) is replaced with (x0B). The parity bits allow the ECC logic to recognize the bit error and corrects it in Figure 7. The data reads out as the correct (x0A), and indicates that there was a single bit error that was corrected.

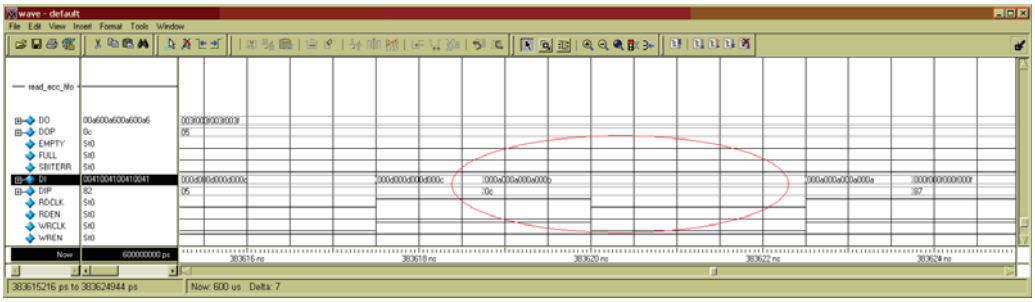
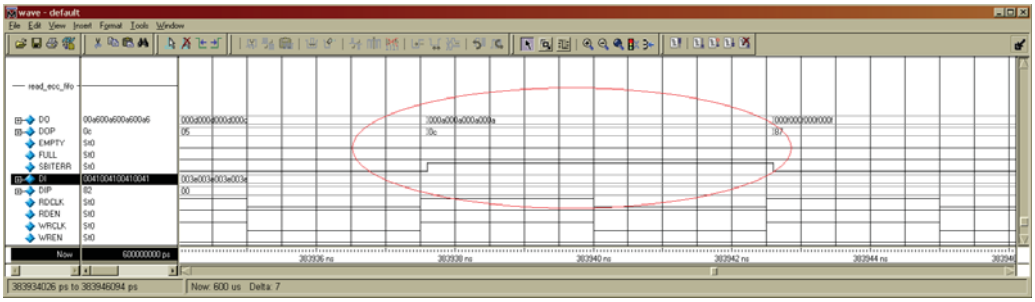


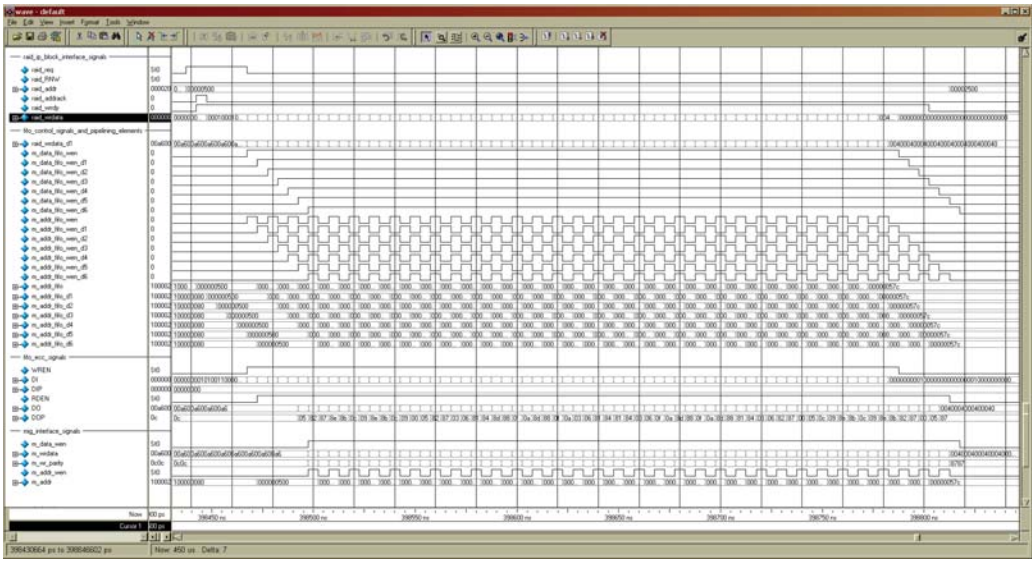
Figure 6: **Data from Memory with a Single Bit Error**



XAPP865_07_032807

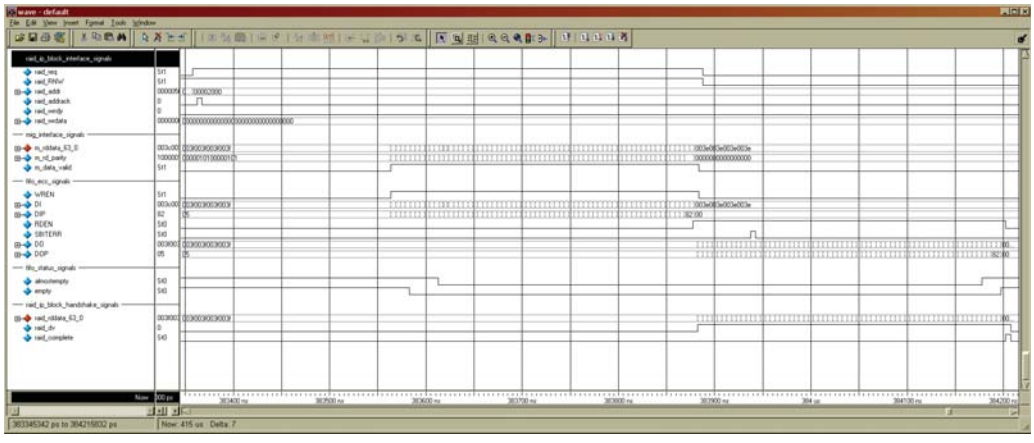
Figure 7: **RAID6 Data Input with the Corrected Data**

Figure 8 and Figure 9 show the logic used to write and read data from FIFO_36_72 with a limited latency.



XAPP865_08_032807

Figure 8: **Writing Data from RAID into the MIG Interface and Generating the ECC**



XAPP865_09_032807

Figure 9: **Reading Data from the MIG Interface into the RAID Block and Correcting Corrupt Data**

Performance and Utilization

Time to Complete Regeneration Calculations

Table 4 shows the regeneration time for the RAID hardware to complete different regeneration calculations. The time for the testblock to setup the registers is *not* included in these regeneration times. Time is measured from the point the testblock processor sets the RAID_RECON control register bit to the time the hardware resets the RAID_RECON status bit. All measurements are for the default 512-byte blocks for a configuration of 64/32 RAID/Memory data width in the reference design files. The 128/72 bit version takes half the time of the 64/40 bit version for a 512-byte block due to the double datapath width. Because of the three data drive topography, the update P and Q process takes a longer processing time than double P and Q generations; this longer processing time is due to the requirement of the equation to write three blocks to the DDR2 memory in the update P and Q mode versus only two blocks in the double P and Q mode. In a structure with more data drives, the update improves its latency over the double P and Q generation.

Because the RAID6 uses a large amount of block RAM, only the 64/40-bit version fits in the LX50T FPGA on the ML555. Table 5 compares the 128/72-bit version to the 64/40-bit version and how the 128/72-bit version needs an LX110T to be implemented.

Table 4: **Block Regeneration Time for a Three Data Drive Structure Comparison for 128/72 and 64/40 versions of the Reference Design**

Type of Request	Time for 512 Bytes		Number of DDR2 Memory Reads	Number of DDR2 Memory Writes
	64/40(1)	128/72		
single P	6.08 µs	3.04 µs	3	1
single Q	8.40 µs	4.20 µs	3	1
single Data	6.16 µs	3.08 µs	3	1
double P & Q	8.92 µs	4.46 µs	3	2
double Q & D	8.70 µs	4.35 µs	3	2
double P & D	8.40 µs	4.20 µs	3	2
double D & D	9.38 µs	4.69 µs	3	2
update P & Q	9.98 µs(2)	4.99 µs(2)	3	3

Notes:

1. The 64/40 design is supported on the ML555, while the 128/72 is not supported based on the block RAM requirement shown in Table 5.
2. Improved vs. double P and Q when more data drives are added.

The RAID6 IP, including the ECC support, the MIG memory controller, and testblock, showcase a *real world* design.The performance and utilization of this *real world* system is shown in Table 5.

Table 5: **System Performance and Utilization**

Device	RAID/Memory Width[1]	Slices	Block RAM	DCM	I/O	BUFG	Frequency[2]
XC5VLX50T FF1136	64/40	5991	36[3]	1	129[4]	5	200 MHz
		20%	60%	8%	26%	15%	
XC5VLX110T FF1136	128/72	9188	72	1	141[5]	6	200 MHz
		13%	48%	8%	29%	18%	

Notes:

1. Includes the 8 bits of ECC.
2. Frequency in a -3 speed grade.
3. When targeting a 72-bit DDR2 with ECC interface, the 72 block RAM requirement exceeds the 60 block RAM maximum of the LX50T. This forces the design into the LX110T, with the resource utilization values shown.
4. For a memory controller configured for 64 bits, but only 40 bits are utilized.
5. For a memory controller configured for 72 bits.

Ports

Most ports of the hardware accelerator interface directly to the MIG memory controller and several others hook up to the status/control registers. Additional I/O pins can be allocated for the two clocks and the system reset. These ports are described in Table 6.

Table 6: **List of Hardware Accelerator Ports**

Port	I/O	Signal Width	Interface	Description
clk	I		CLK	RAID logic clock
clk_mem	I		CLK	Memory interface clock[1]
rst	I		RST	RAID logic reset
start	I		REG	Start the regeneration process when High
DCR_sel	O	[1:0]	REG	Select which DCR registers to read
DCR_reg0	I	[31:0]	REG	First DCR register input
DCR_reg1	I	[31:0]	REG	Second DCR register input
RAID_SIZE	I	[31:0]	REG	Determines burst size to process: 0000001 512 bytes 0000010 1024 bytes 0000100 2048 bytes 0001000 4096 bytes 0010000 8192 bytes 0100000 16384 bytes Bits 31:7 should always be zero
done	O		REG	Indicates the regeneration process is completed when High
SINGLE_ERR	O		REG	Indicates a single bit error occurred
DOUBLE_ERR	O		REG	Indicates a double bit error occurred
mem_wrdata	O	[63:0][2]	MIG	Write data for MIG interface
mem_wr_parity	O	[15:0]	MIG	Write ECC data from MIG interface
mem_rddata	I	[63:0][2]	MIG	Read data for MIG interface
mem_rd_parity	I	[15:0]	MIG	Read ECC data from MIG interface
mem_addr	O	[35:0]	MIG	Address for MIG interface

Table 6: **List of Hardware Accelerator Ports *(Continued)***

Port	I/O	Signal Width	Interface	Description
mem_data_wen	O		MIG	Data write enable for MIG interface
mem_addr_wen	O		MIG	Address write enable for MIG interface
mem_data_valid	I		MIG	Data valid status from MIG interface

Notes:

1. The Memory interface clock = the RAID clock (additional clock domain crossing must be added for other ratios).
2. This interface depends on the memory width used; it is twice the external memory width.

The status/control register implementation for hardware accelerator and testblock allow clock domain crossing communication. Table 7 lists the ports that can connect to a system controller (testblock).

Table 7: **Status/Control Register Ports**

Port	I/O	Signal Width	Interface	Description
rst	I		RST	Register reset.
clk	I		CLK	Register clock.
addr	I	[4:0]	SYS CNTRL	Address for system controller.
datain	I	[31:0]		Data in from the system controller.
ren	I			Read signal from the system controller.
wen	I			Write signal from read signal from the system controller.
dataout	O	[31:0]		Data out from the system controller.
RAID_clear	I		RAID	Register output for RAID Hardware Accelerator.
RAID_lost	O	[31:0]		
RAID_reconstruct	O			
RAID_D1	O	[31:0]		
RAID_D2	O	[31:0]		
RAID_D3	O	[31:0]		
RAID_D4	O	[31:0]		
RAID_P	O	[31:0]		
RAID_Q	O	[31:0]		
RAID_M	O	[31:0]		
RAID_SIZE	I	[31:0]		
SBERR	O		SYS CNTRL	Indicates that a single bit error occurred. Only a reset or a write to x09 can clear this bit.
DBERR	O			Indicates that a double bit error occurred. Only a reset or a write to x0A can clear this bit.
INJECT	I	[1:0]		Signals that either a single or double bit error should be injected.

Table 7: **Status/Control Register Ports *(Continued)***

Port	I/O	Signal Width	Interface	Description
RAID_S_BIT_ERROR	I		RAID	Signal that sets the SBERR register when High.
RAID_D_BIT_ERROR	I		RAID	Signal that sets the DBERR register when High.

Expanding Design for Larger Arrays

This reference design is designed for a three data drive system (five total drives). Because of the redundancy of the calculations, the datapath has the ability to support larger arrays. The state machines need modifications to loop multiple times in certain states to calculate values for larger arrays. The status/control registers must be expanded to provide additional pointers to data blocks as well as control functions for data recovery operations. Also, the system controller (testblock) must manage more base memory address pointers.

Other Uses of the Hardware Accelerator

As mentioned in "Reference Design," page 8, firmware plays a major role in all RAID systems. Because the hardware changes little for the different RAID levels, the hardware accelerator can remain the same, and the firmware changes to incorporate the different ways the data is organized on the HDD and how the parity is generated.

The hardware accelerator can support other RAID levels beyond RAID6 with minimal (if any) modifications. RAID Double Parity (DP), RAID5, RAID4, and RAID3 are among the supportable levels.

A brief discussion of these RAID levels and how the hardware accelerator can support them is described in "RAID DP," "RAID5," and "RAID3 and RAID4" sections.

RAID DP

RAID DP can support two simultaneous disk failures and has the advantage of generating both parities with simple XOR function. RAID DP performs a horizontal parity calculation as used in RAID3 and RAID4 systems. In addition, RAID DP performs a diagonal parity calculation. The parity information is not rotated across the drives as done in RAID5. See [Ref 5] for a detailed discussion on RAID Double Parity. While the diagonal parity calculation simplifies the hardware parity calculation, the disadvantage is the need for more disk accesses to read additional disk sectors for the diagonal parity calculation. For example, for RAID DP in Figure 10, seven different blocks are accessed to regenerate the loss of two blocks.

The reference design has the ability to cover all of these levels with the appropriate firmware (not included in the reference design). The firmware passes the data to the hardware accelerator and always assumes either a P generation/regeneration, one data regeneration, or in the case of two data regenerations, two subsequent single data regenerations with the data to be XORed. In the example shown in Figure 10, if $D0_0$ and $D1_0$ are lost, the firmware must first request a single data regeneration for $D0_0$ using $D1_1$, $D2_2$, P_3, and $P2_0$ (highlighted in red). After regenerating $D0_0$, $D1_0$ is regenerated by using $D0_0$, $D2_0$, $P0_0$, and $P2_0$.

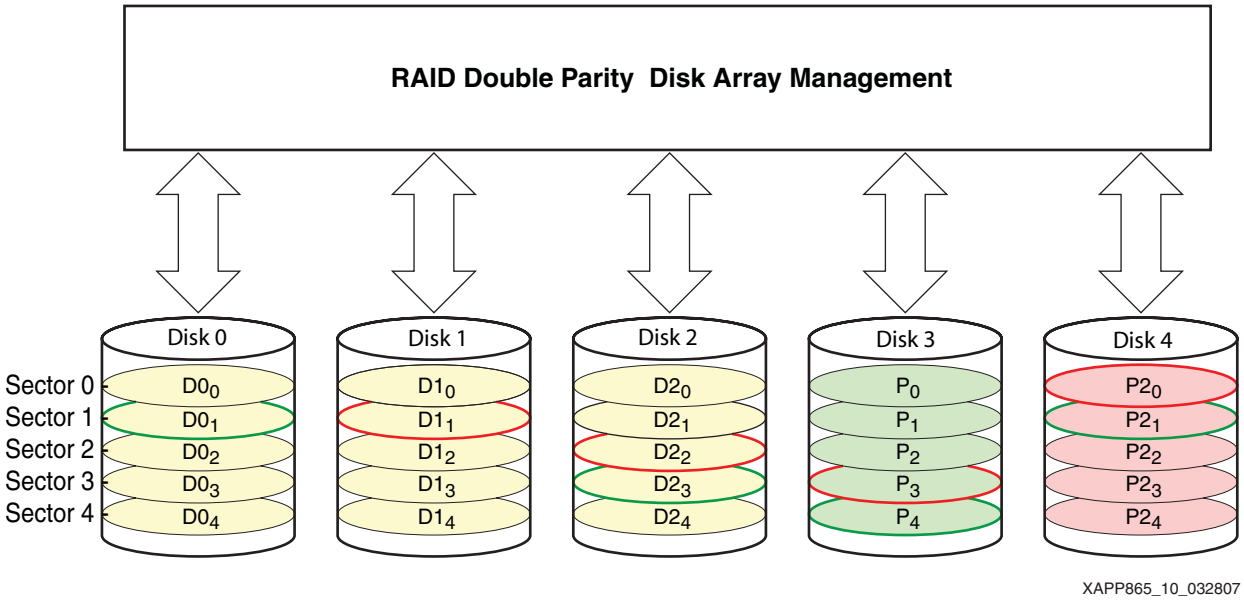


Figure 10: **RAID DP Data Structure**

RAID5

RAID5, shown in Figure 11, handles just one disk failure at a time. It rotates the parity information on multiple drives to improve the read/write latency associated with accessing the HDDs in the same way as RAID6. However, in this case, there is only one set of parities. Regenerating single data blocks is identical to RAID6, except one more data block needs to XORed for the same five-disk system. There is only 20 percent storage overhead for parity and 20 percent more storage capacity in the RAID5 five-disk array than in an equivalent RAID6 five-disk array system.

RAID5 systems only utilize the XOR_BRAM block shown in the DMB datapath logic (see Figure 4).

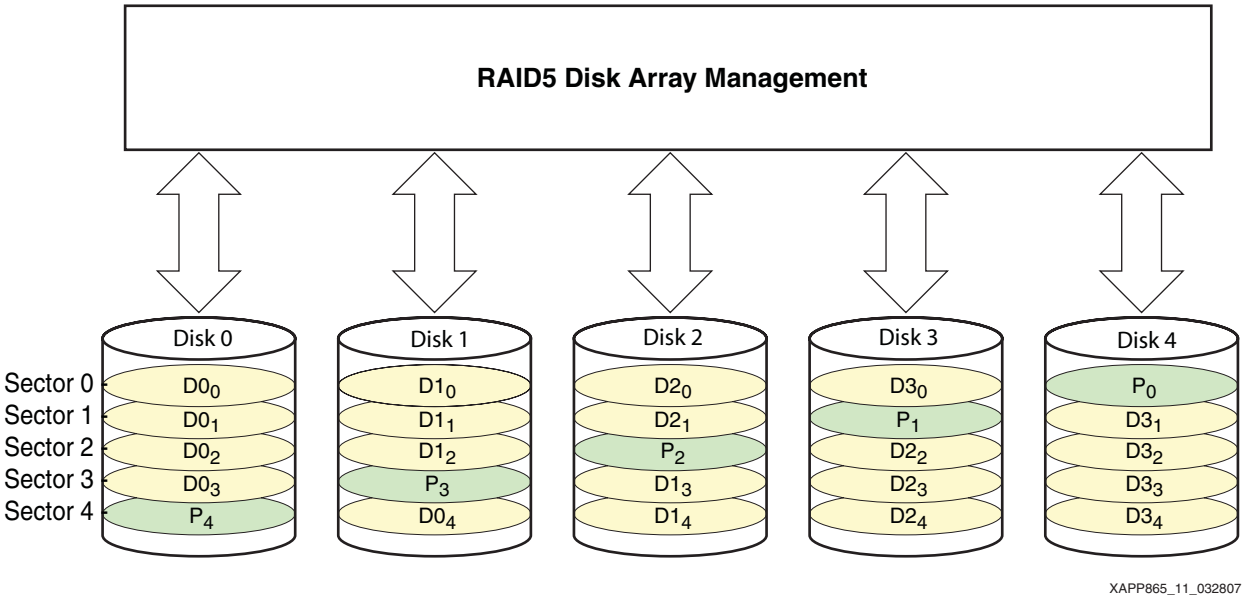


Figure 11: **RAID 5 Data Structure**

RAID3 and RAID4

RAID3 and RAID4 are identical from a parity generation and data regeneration perspective. The only difference is the organization of the congruent data. RAID3 and RAID4 have a fixed parity disk and RAID5 and RAID6 both use rotating parity disks. Systems with fixed parity disks can experience bottlenecks, if there is a large write-to-read ratio, because the parity disk must be accessed twice for each block-write access. A RAID4 configuration is shown in Figure 12.

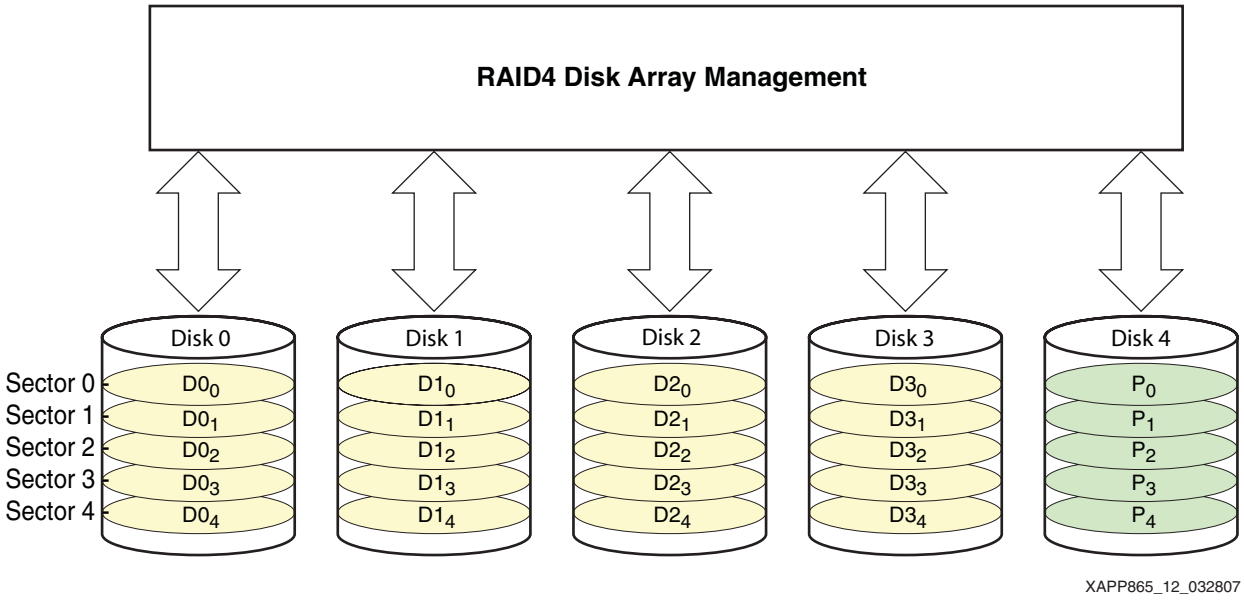


Figure 12: **RAID4 Data Structure**

Reference Design Simulation

The reference design supplied in the ZIP file (http://www.xilinx.com/bvdocs/appnotes/xapp865.zip) is the design that works on the ML555 demonstration board. The design contains the testblock, status/control registers, RAID6 with ECC IP, a MIG-based DDR2 memory controller, and a DDR2 model for the SODIMM of the ML555. This design was verified in ISE™ 9.1 SP1. Use the latest MIG tool to generate the memory controller design portion of this reference design. The directory structure is shown in Figure 13.

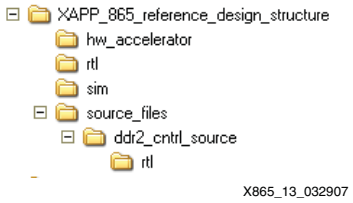
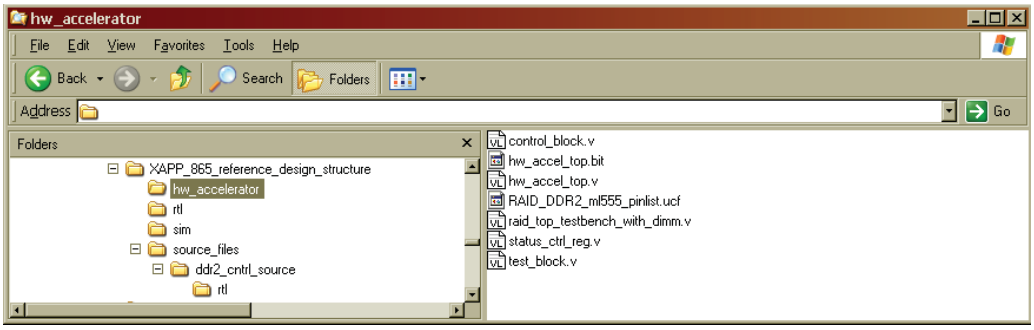


Figure 13: **Reference Design Structure**

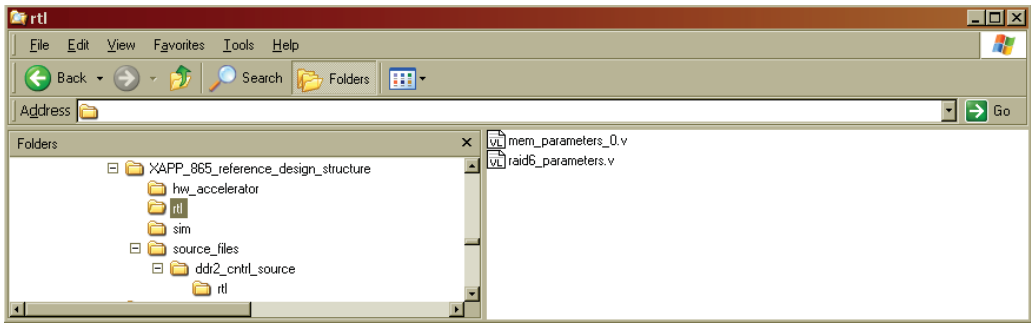
The hw_accelerator directory contains several files for implementation, including a UCF for the ML555 and a bit file of this design for the ML555. Also included are the top-level modules and the system controller modules. These files are shown in Figure 14.



XAPP865_14_032807

Figure 14: **Hardware Accelerator Contents**

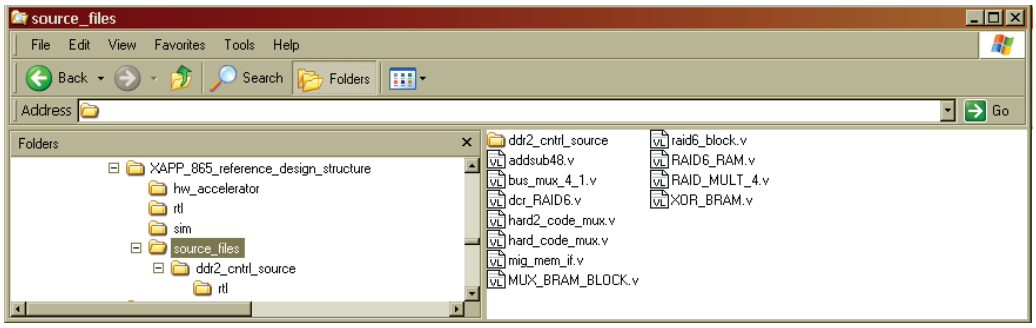
The `rtl` directory contains the two parameter files referenced by the `raid` and `ddr2 memory controller` files to define the data width. These are shown in Figure 15.



XAPP865_15_032807

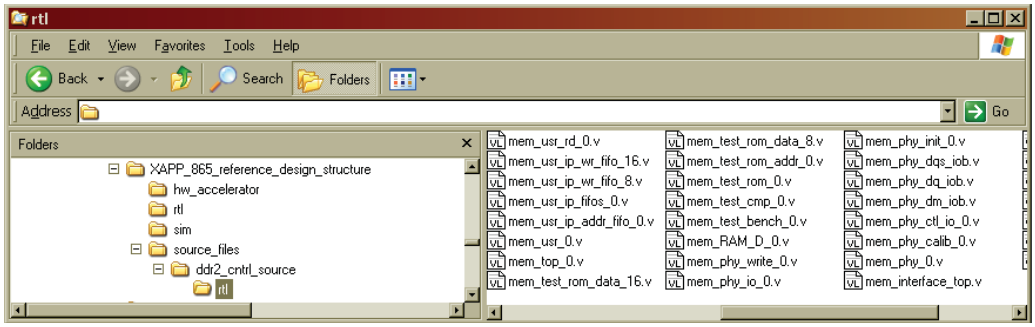
Figure 15: **Parameter Files**

The `source_files` directory contains the `RAID with ECC IP` source files and the directory `ddr2_cntrl_source/rtl`, which contains the source files generated by the MIG tool (see Figure 16). If creating a new design, it is recommended that the memory controller be generated with the latest version of MIG tool.



XAPP865_16_032807

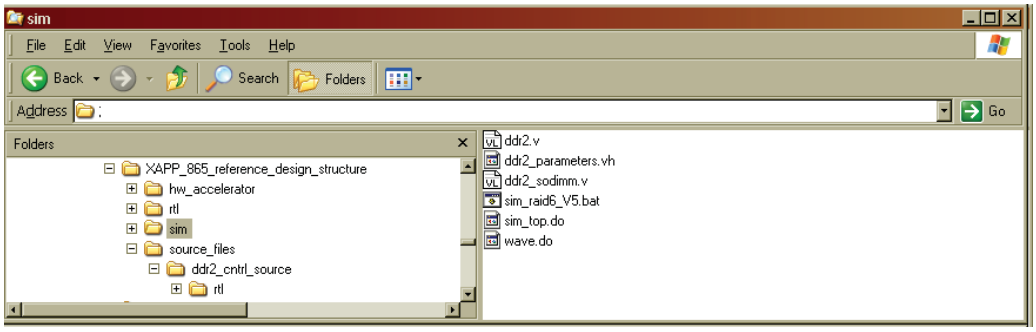
Figure 16: **RAID6 with ECC Source Files**



XAPP865_17_032807

Figure 17: **MIG Generated DDR2 Memory Controller Source Files**

Finally, the `sim` directory contains the DDR2 memory simulation files along with simulation scripts. The `sim_raid6_v5.bat` calls the `sim_top.do` that compiles all the source files along with linking the appropriate simulation libraries. Double click on this batch file and the simulation runs until the RAID has reconstructed all of the cases discussed in Table 5.



XAPP865_18_032807

Figure 18: **Simulation Files**

Conclusions

This reference design supports calculating Reed-Solomon RAID6 parity generation and data regeneration on 512- to 16,384-byte blocks of data from a DDR2 memory. This design takes advantage of multiple immersed IP blocks of the Virtex-5 FPGA to improve performance and decrease fabric utilization. The block RAMs are used for the GF mathematical LUTs, and the ECC blocks are used with the FIFO_32_72 primitives to support ECC-enabled memories, plus the potential to handle the RAID-level data structure in the HDDs. Other Xilinx solutions also increase the performance of calculating RAID6 in a Virtex-5 FPGA, among them is using the MIG memory controller to allow shared memory bandwidth and the RocketIO™ transceivers to allow serial interfaces to many different available HDDs.

This generation/regeneration is time intensive, but it still takes less time than the equivalent firmware application. As in all systems, hardware and firmware tradeoffs are evaluated. This application note covers all of the equations for a small three data drive array and shows how one type of system can be implemented on a Xilinx ML555 demonstration board.

References

1. *Intelligent RAID6 Theory Overview and Implementation*
   http://www.intel.com/design/storage/papers/308122.htm
2. A tutorial on Reed-Solomon Coding for Fault-Tolerance in RAID-Like Systems
   http://www.cs.utk.edu/~plank/plank/papers/CS-96-332.html
3. The Mathematics of RAID6
   http://kernel.org/pub/linux/kernel/people/hpa/raid6.pdf
4. XAPP657, *Virtex-II Pro RAID-5 Parity and Data Regeneration Controller*
   http://www.xilinx.com/bvdocs/appnotes/xapp657.pdf
5. *NetApp® Data Protection: Double Parity RAID for Enhanced Data Protection with RAID-DP™* www.netapp.com/tech_library/3298.html

Additional Resources

1. XAPP731, Hardware Accelerator for RAID6 Parity Generation / Data Recovery Controller
   http://www.xilinx.com/bvdocs/appnotes/xapp731.pdf

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
05/02/07	1.0	Initial Xilinx release.