XAPP731(v1.1) March 20, 2007

Hardware Accelerator for RAID6 Parity Generation / Data Recovery 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 it fails.

There are many different levels of RAID. The RAID 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 of RAID is RAID6, which has two implementations (Reed-Solomon P+Q or Double Parity). RAID6 is the first RAID level that allows 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™-4 architecture including distributed memory, FIFO memory, Digital Clock Managers (DCM), 18-Kbit Block Select RAMs (block RAM), PowerPC™ processor (PowerPC 405), and DSP48 blocks in a hardware acceleration block that supports Reed-Solomon RAID6 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 x 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

NOTICE OF DISCLAIMER: Xilinx is providing this design, code, or information "as is." By providing the design, code, or information as one possible implementation of this feature, application, or standard, Xilinx makes no representation that this implementation is free from any claims of infringement. You are responsible for obtaining any rights you may require for your implementation. Xilinx expressly disclaims any warranty whatsoever with respect to the adequacy of the implementation, including but not limited to any warranties or representations that this implementation is free from claims of infringement and any implied warranties of merchantability or fitness for a particular purpose.

To understand the Reed-Solomon RAID6, designers must have some familiarity with Galois Field (GF) mathematics. This application note only covers the GF equations and refers to the 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 immersed DSP48 blocks in Virtex-4 FPGAs.

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

$$0x0d \div 0x11 = gfilog[gflog0x0d - gflog0x11] = gfilog[0x68 - 0x64] = gfilog[0x04] = 0x10$$ *Equation 2*

This reference design implements the `gflog` and `gfilog` 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: **GFLOG 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
2	9D	27	4E	9C	25	4A	94	35	6A	D4	B5	77	EE	C1	9F	23

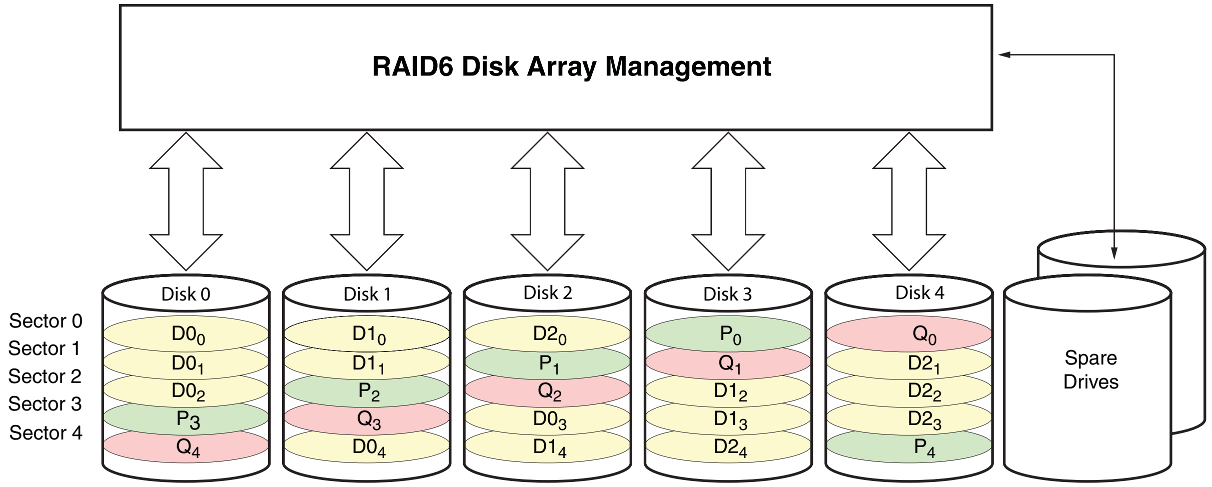
Table 2: **GFILOG LUT, Stored in Block RAM *(Continued)***

GFILOG	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
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 the method 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 percent* 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 percent*).



XAPP731_01_030607

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).

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

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.

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

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.

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

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

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

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.

$$P_{N\_NEW} = P_{N\_OLD} \oplus D1_{N\_OLD} \oplus D1_{N\_NEW} \qquad \textit{Equation 7}$$

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

$$P_{0\_NEW} = P_{0\_OLD} \oplus D1_{0\_OLD} \oplus D1_{0\_NEW} \qquad \textit{Equation 9}$$

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

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, there are different types of disk failure scenarios. 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.

$$D0_0 = D1_0 \oplus D2_0 \oplus P_0 \qquad \textit{Equation 11}$$

$$D0_1 = D1_1 \oplus P_1 \oplus D2_1 \qquad \textit{Equation 12}$$

$$D0_2 = P_2 \oplus D1_2 \oplus D2_2 \qquad \textit{Equation 13}$$

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, 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 app note; in either case, the data is all that is needed. The simple XOR Equation 10 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 DSP48 block 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 are first followed by 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.

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

$$\begin{aligned}Q_N = {} & (0x01 \otimes D0_N) \oplus (0x02 \otimes D1_N) \oplus (0x04 \otimes D2_N) \oplus \ldots \\ & \oplus (g(M-1) \otimes D(M-1)_N)\end{aligned} \tag*{Equation 19}$$

$$D0_4 = D1_4 \oplus D2_4 \oplus P_4 \tag*{Equation 20}$$

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

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.

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

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

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

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

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

$$P_1 = D0_1 \oplus D1_1 \oplus D2_1 \tag*{Equation 27}$$

Double Data Regeneration

Double data regeneration is the most complicated case in RAID6 and in the reference design as well. 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) represent the remaining data 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.

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

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

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

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

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

$$P'_0 = D2_0 \tag*{Equation 33}$$

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

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

Reference Design

System Architecture

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

The system contains a RAID host controller on an ML405 demonstration board. This board contains a Virtex-4 FPGA along with a DDR memory and Serial ATA (SATA) connectors attached to a port multiplier. The port multiplier connects to five SATA HDDs. A SATA protocol controller that interfaces with the memory controller and the PowerPC 405 processor 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 system as shown in Figure 2: PowerPC 405 embedded controller, DDR memory controller, and RAID IP block. The shaded portions of Figure 2 are only there to show a possible system level implementation using the RAID-IP hardware connected in a Serial ATA system.

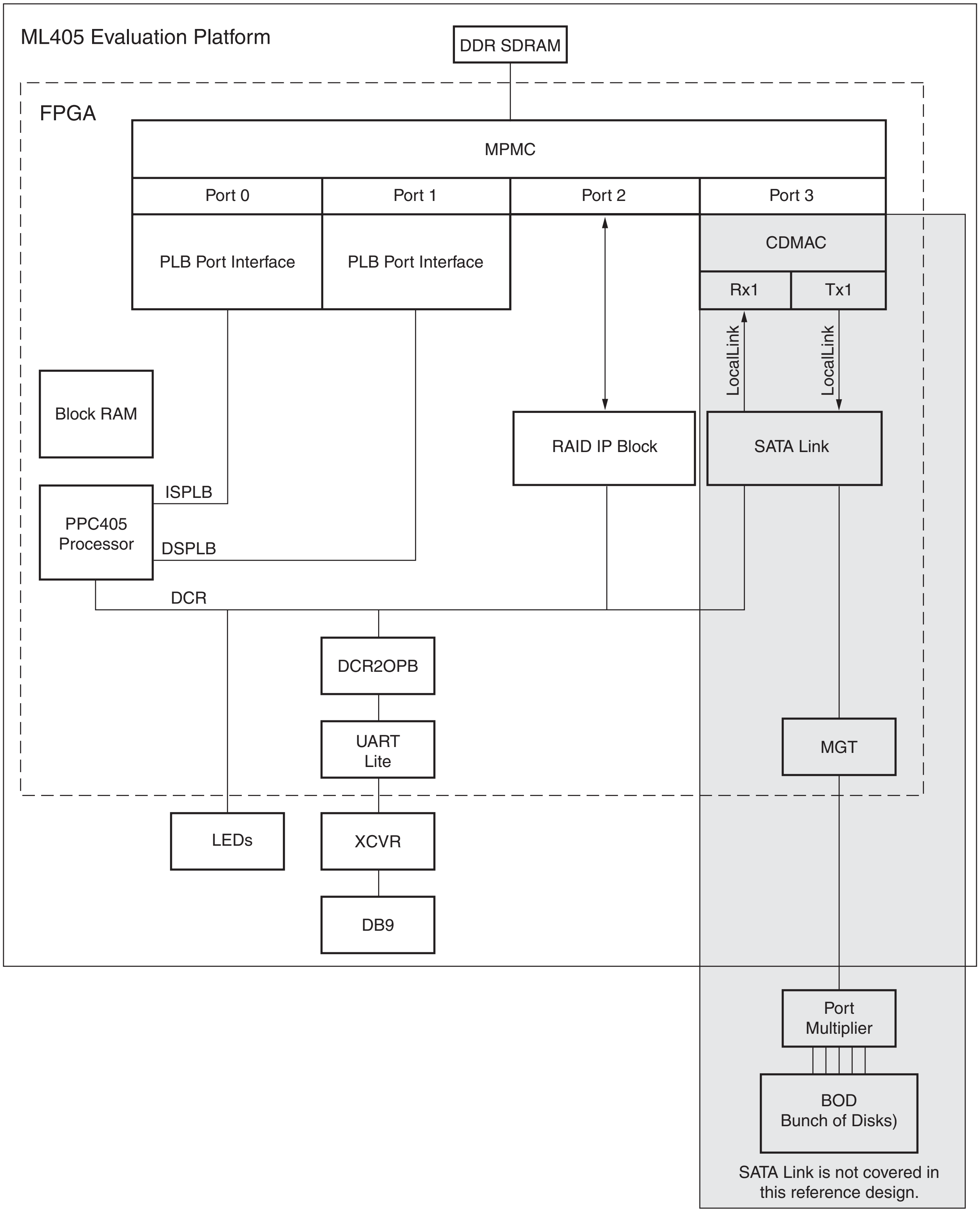
An embedded PowerPC 405 block:

- Controls the RAID6 hardware.
- Sets up pointers to the data and parity blocks of memory.
- Sets up the hardware.

The reference design does ***not*** include the disk array management firmware or the Serial ATA interface to the disk drives.

A microprocessor runs the RAID firmware and configures the RAID hardware accelerator block. Demonstration firmware included in the reference design generates data and parity blocks to emulate the data that would come from an HDD connection in a "real" system. This emulation allows the reference design to run without an HDD connection. The PowerPC 405 firmware generates data placed in the DDR memory that serves as a cache for the RAID hardware accelerator.

A Xilinx Multiple Port Memory Controller (MPMC) controls the DDR memory. The MPMC provides multiple memory masters to access a shared memory. The MPMC connects to each of the PowerPC 405 instruction and data side processor local bus interfaces. Two other ports provide system specific implementations for this reference design; one is used for the RAID hardware accelerator. For additional information on the MPMC, see [Ref 6].



XAPP731_02_030807

Figure 2: **Possible RAID6 System Implementation**

Hardware Accelerator

A RAID6 hardware accelerator is mathematically intensive. 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 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 eight 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 ML405 board (which is the verification platform). This determines the data flow block discussed in "Data Flow for Different Regeneration Cases," page 12.

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

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

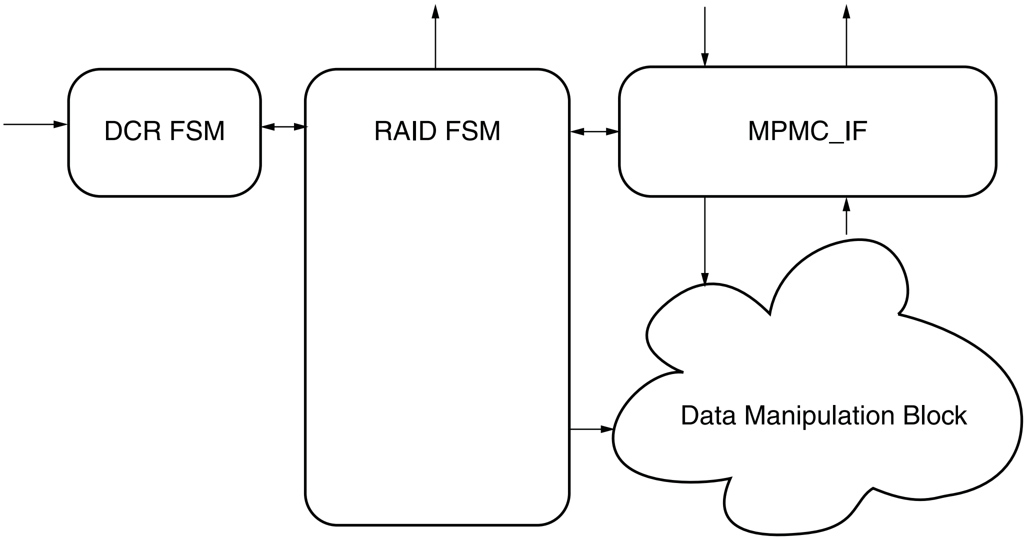


Figure 3: **RAID6 Hardware Accelerator Block Diagram**

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 a four-byte implementation (four instantiations of the DMB). It can create parity and regenerate data of any case discussed in the RAID6 equations 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)
- MUX_BRAM (GREEN)
- RAID_MULT_4 (YELLOW)

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) holds calculated data blocks until other data blocks are either retrieved from the MPMC 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 g 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, this could be implemented as a LUT with the input controlled through a processor DCR register to select the Gn constant) by a portion of one DSP48 block. Only *one* DSP48 block is used for the four GF multiplication functions in the RAID_MULT_4 block of a one-byte data width system.
3. The sum enters the GFILOG LUT implemented in another block RAM, implemented similarly to the GFLOG. 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 DMB blocks, covering all of the logic equations implemented in the reference design, is covered in the "Data Flow for Different Regeneration Cases" section that follows.

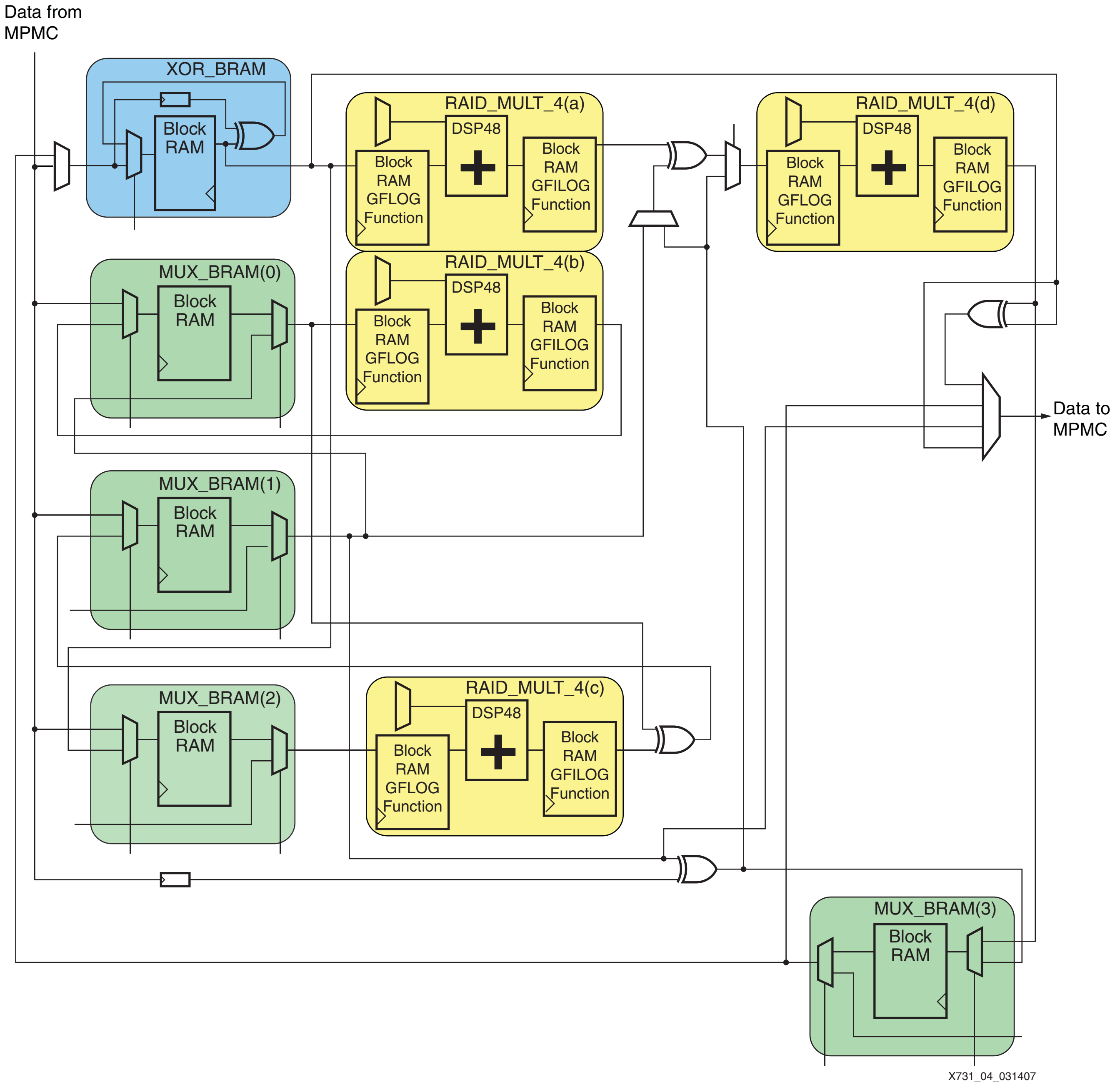


Figure 4: **Data Manipulation Block Logic**

Data Flow for Different Regeneration Cases

This section summarizes data flow through Data Manipulation Blocks 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 that follows. 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 MPMC into XOR_BRAM.
2. XOR the second data block from the MPMC with output of XOR_BRAM and write it back into XOR_BRAM.

3. XOR the third data block from the MPMC with output of XOR_BRAM and write it back into XOR_BRAM.
4. When the MPMC is ready, read P parity out of the XOR_BRAM through the 4:1 MUX to the MPMC.

Generate/Regenerate Q Parity

1. Write the first data block from the MPMC 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 second data block from the MPMC 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 MPMC 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 MPMC is ready, read the Q parity block out of the MUX_BRAM_1 through the 4:1 MUX to the MPMC.

Regenerate Data

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

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 MPMC 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 MPMC 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 MPMC with output of XOR_BRAM and write it back into XOR_BRAM.
3. *(a)* Write the third data block from the MPMC 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 MPMC with output of XOR_BRAM and write it back into XOR_BRAM.
4. *(b)* When the MPMC is ready, read the P parity block out of the XOR_BRAM through the 4:1 MUX to the MPMC.
5. *(a)* When MPMC is ready, read Q parity block out of the MUX_BRAM_1 through the 4:1 MUX to the MPMC. Writes to the MPMC 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 MPMC into XOR_BRAM.
2. *(a)* Write the first data block from the MPMC 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 MPMC with the parity block output of XOR_BRAM and write it back into XOR_BRAM.
3. *(a)* Write another data block from the MPMC 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 MPMC with output of XOR_BRAM and write it back into XOR_BRAM, which now contains the regenerated data block.
4. *(b)* When MPMC is ready, read lost data block out of XOR_BRAM through the 4:1 MUX to the MPMC, plus write 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 MPMC is ready, read the Q parity block out of the MUX_BRAM_1 through the 4:1 MUX to the MPMC.

Regenerate P (a) Parity and Data (b)

1. *(a)* Write the first data block from the MPMC into XOR_BRAM.

   (b) Write the first data block from the MPMC 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 MPMC with output of XOR_BRAM and write it back into XOR_BRAM.

   (b) Write the second data block from the MPMC 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 MPMC is ready, read the lost data block from MUX_BRAM_3 through the 4:1 MUX to the MPMC. 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 once the data has been written to the DDR 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 MPMC is ready, read the P parity block from the XOR_BRAM through the 4:1 MUX to the MPMC.

Regenerate Double Data

1. Write only known good data blocks from the MPMC 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 MPMC with the output of the XOR_BRAM, and write it back into the XOR_BRAM.
4. Write the Q parity block from the MPMC 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 MPMC is ready, read the first data block from MUX_BRAM_3 through the 4:1 MUX to the MPMC, XOR this lost data block with the output of the XOR_BRAM, and write the result back into the XOR_BRAM.
7. When the MPMC is ready, read the second data block out of the XOR_BRAM through the 4:1 MUX to the MPMC. Two single-threaded writes to the MPMC are required because the two regenerated data blocks must be written into two different memory locations in DDR 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 DDR memory. Writing to an array that is being constructed can be permitted, depending on the system specific implementations (that are beyond the scope of this application note).

1. Write the old data block from the MPMC into XOR_BRAM and MUX_BRAM_0.
2. Write the new data block from the MPMC into MUX_BRAM_2, XOR the new data block from the MPMC 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 MPMC 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 MPMC is ready, read the new P parity out of the XOR_BRAM through the 4:1 MUX to the MPMC.
7. When MPMC is ready, read the new Q parity out of the MUX_BRAM_3 through the 4:1 MUX to the MPMC. 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 DDR memory.

RAID FSM

The RAID Finite State Machine (FSM) is the main control logic of the reference design. It controls the MPMC_IF 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.

DCR and DCR FSM

The hardware accelerator depends on firmware to place disk data into the DDR memory and to manage the memory buffer. PowerPC 405 firmware controls the hardware accelerator with a set of registers implemented on the PowerPC 405 DCR bus. Hardware status is provided to the PowerPC 405 processor on DCR status registers. DCR control registers are provided to point to data and parity block starting addresses in the external DDR 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, is finished.

Twelve DCR registers are provided to interface to the accelerator hardware. Table 3 contains the DCR address and a description of the register function. The PP405 processor reads and writes to all of these registers. All registers are readable by the hardware accelerator except the RAID_LED, and only RAID_RECON is writable. Also in Table 3 are the registers that control the CDMAC interface and the SATA controller.

Table 3: **DCR Register Descriptions**

DCR Address	Register Name	Description
`0x180`	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.
`0x181`	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
`0x182`	RAID_1	Memory base address pointer to first data block stored in DDR memory[2].
`0x183`	RAID_2	Memory base address pointer to second data block stored in DDR memory[3].
`0x184`	RAID_3	Memory base address pointer to third data block stored in DDR memory.
`0x185`	RAID_4	Memory base address pointer reserved for a fourth data block stored in DDR memory.
`0x186`	RAID_P	Memory base address pointer to the P parity block stored in DDR memory.
`0x187`	RAID_Q	Memory base address pointer to Q parity block stored in DDR memory.

Table 3: **DCR Register Descriptions *(Continued)***

DCR Address	Register Name	Description
0x188	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
0x189		Reserved.
0x18A	RAID_LED	Indicates to the system or user if RAID test is in progress or completed. Controls diode DS11 on the ML405 hardware platform.
0x18B	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
0X140-0x16F	CDMAC	Initiates transactions through the CDMAC. See [Ref 6] for details. *Not used in this reference design.*
0x200-0x2FF	SATA Controller	Not used in this reference design.

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'

Table 4 shows other DCR registers that are associated with the PowerPC 405 processor. There are the internal registers for the embedded Tri-mode EMACs and the DCR registers that control the DSOCM and ISOCM controllers.

Table 4: **Other Internal DCR Register Descriptions**

DCR Address	Register Name	Description
0x412–0x416 (1)	EMAC0	Internal registers that are unused in this design.
0x412–0x412 (1)	EMAC1	Internal registers that are unused in this design.
0x406–0x407	DSOCM(2)	Settings for the Data Side OCM.
0x000–0x403	ISOCM(2)	Settings for the Instruction Side OCM.
0x404–0x405	APU	Settings for the Auxiliary Processor Unit (APU) (not used).

Notes:

1. Both EMACs share a host interface of an internal bus.
2. Application firmware for the reference design is stored in the instruction OCM block RAM memory (data side OCM block RAM is also available if the designer chooses to use this resource). The DDR memory is used exclusively by the RAID-IP hardware.

MPMC and MPMC_IF

The DDR memory controlled by the MPMC is memory mapped into two locations, as shown in Table 5. Also in the memory map are the two OCM memories, where the processor runs the software code instructions.

Table 5: **PowerPC 405 Memory Addressing**

Memory Name	Origin	Size	Description
plb_mpmc_if_0	0x00000000	0x0FFFFFFF	Used for RAID data and parity block storage space
iocm_cntlr	0xFFFC000	0x00003FFF	Used for SW instructions
plb_mpmc_if_1	0x00000000	0x0FFFFFFF	Used for cache memory space
docm_cntlr	0xFE004000	0x000007FF	Used for SW instructions and data

The MPMC [Ref 6] is a 4-port DDR memory controller. These four ports are time multiplexed to allow shared access to the memory device by four different bus masters:

- Two of the four ports are allocated to the PowerPC 405 instruction and data cache processor local bus (PLB) interfaces. In the reference design, these ports are active only when the processor is generating blocks of data and parity to emulate data stored on a disk array, or when checking that regenerated data is correct.
- The third port is for the RAID6 hardware engine.
- The fourth port is reserved for a disk interface controller.

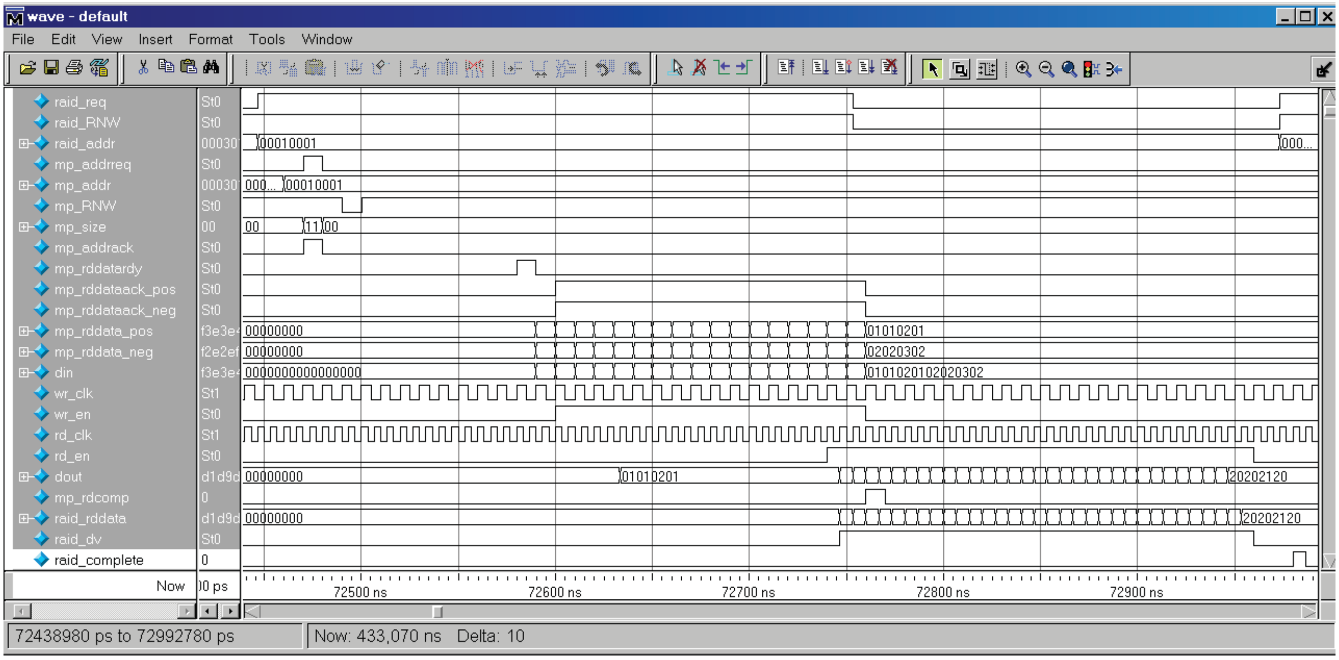
The MPMC contains an internal arbiter that dictates how the memory bandwidth is prioritized. Time-sharing of the memory bus adds some uncertainty as to how quickly the hardware accelerator can receive data from the cache memory; however, after the memory bus is granted to the RAID6 engine, the memory controller transmits or receives bursts of 128 bytes of data. Since default data blocks are multiples of 512 bytes, the RAID6 hardware requests an additional three memory cycles to process a complete 512-byte sector (if other ports of the MPMC have requested access, these other requests are fulfilled between the multiple RAID requests).

The CDMAC and Local Link interface block connections to the RAID-IP memory port are intentionally not used for one main reason (shown in Figure 2, note that the SATA link does contain these blocks and connections). The CDMAC uses DCR writes from the PowerPC 405 processor to set up the DMA transactions to the MPMC. The CDMAC and Local Link interface adds unnecessary overhead for the software to oversee. The software can handle this extra housekeeping for a handful of regenerations. However, when 400 GB of HDD space needs to be regenerated, keeping track of how, where, and when each cache memory is accessed becomes a burden that software needs to pass onto hardware. Now the software can continue to focus on the data structure on the HDDs and the continuous read and write requests from the host.

The MPMC_IF is the substitute for the CDMAC and Local Link. This block handles several functions:

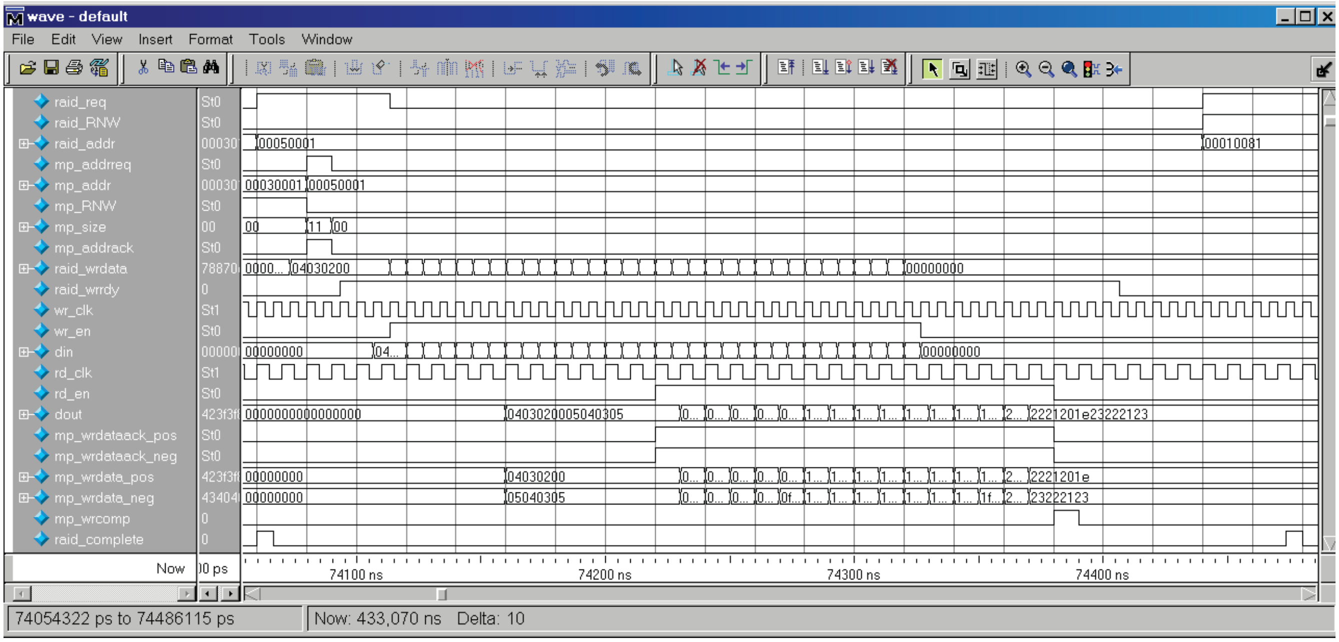
- First, it handles several functions, such as staged memory requests, that would not be efficient in the software based structure of the CDMAC.
- Second, it contains logic that allows for a clock domain crossing of the MPMC and the RAID circuitry. This allows the RAID/MPMC clock ratio to range from 3:2 (the MPMC running 64 bits at 100 MHz while the RAID hardware accelerator manipulates 32 bits at 150 MHz).

The waveform that shows how the MPMC_IF requests a read is shown in Figure 5. The waveform that shows how the MPMC_IF requests a write is shown in Figure 6. The signals with the *raid* prefix interface with other portions of the RAID hardware accelerator logic; the *mp* prefixed signals interface with the MPMC block; and all other signals are signals for the two CORE Generator™ asynchronous FIFO/data width converters. Settings for the CORE Generator FIFOs are covered in "CORE Generator FIFO Settings," page 28.


wave - default
File Edit View Insert Format Tools Window
raid_req St0
raid_RNW St0
raid_addr 00030 00010001 000...
mp_addrreq St0
mp_addr 00030 000... 00010001
mp_RNW St0
mp_size 00 00 11 00
mp_addrack St0
mp_rddatardy St0
mp_rddataack_pos St0
mp_rddataack_neg St0
mp_rddata_pos f3e3e 00000000 01010201
mp_rddata_neg f2e2e 00000000 02020302
din f3e3e 0000000000000000 0101020102020302
wr_clk St1
wr_en St0
rd_clk St1
rd_en St0
dout d1d9d 00000000 01010201 20202120
mp_rdcomp 0
raid_rddata d1d9d 00000000 20202120
raid_dv St0
raid_complete 0
Now 0 ps
72500 ns
72600 ns
72700 ns
72800 ns
72900 ns
72438980 ps to 72992780 ps
Now: 433,070 ns Delta: 10

XAPP731_05_021406

Figure 5: **MPMC Read Waveform**


wave - default
File Edit View Insert Format Tools Window
raid_req St0
raid_RNW St0
raid_addr 00030 00050001 00010081
mp_addrreq St0
mp_addr 00030 00030001 00050001
mp_RNW St0
mp_size 00 00 11 00
mp_addrack St0
raid_wrdata 78870 0000... 04030200 00000000
raid_wrrdy 0
wr_clk St1
wr_en St0
din 00000 00000000 04... 00000000
rd_clk St1
rd_en St0
dout 423f3f 0000000000000000 0403020005040305 2221201e23222123
mp_wrdataack_pos St0
mp_wrdataack_neg St0
mp_wrdata_pos 423f3f 00000000 04030200 2221201e
mp_wrdata_neg 43404 00000000 05040305 23222123
mp_wrcomp 0
raid_complete 0
Now 0 ps
74100 ns
74200 ns
74300 ns
74400 ns
74054322 ps to 74486115 ps
Now: 433,070 ns Delta: 10

XAPP731_06_021406

Figure 6: **MPMC Write Waveform**

PowerPC 405 Firmware

In any RAID system, there is a large amount of firmware. This firmware controls where specific data is stored and how it is stored. This is true for the HDD and the DDR memory. This reference design has minimal firmware to indicate to the RAID hardware accelerator where the different data, P parity, and Q parity blocks are located in the DDR memory.

The hardware accelerator does not need to know the location or arrangement of data on the HDD; therefore, this type of firmware is not part of this reference design.

Below is a code snippet from `gf_gen.c` that shows a P and Q regeneration case:

```
int request_reconstruct_p_q(void) {
//setting up the DCR registers for the RAID system
//for regenerating P and Q

//set up memory base address pointer for DATA1 data block
mtdcr(RAID_1, 0x00010001);
//set up memory base address pointer for DATA2 data block
mtdcr(RAID_2, 0x00020001);
//set up memory base address pointer for DATA3 data block
mtdcr(RAID_3, 0x00030001);
//set up memory base address pointer for DATA4 data block (reserved)
mtdcr(RAID_4, 0x00040001);
//set up memory base address pointer for Parity block result
//to be stored by RAID6 hardware
mtdcr(RAID_P, 0x00050001);
//set up memory base address pointer for Q parity block result
//to be stored by RAID6 hardware
mtdcr(RAID_Q, 0x00060001);
//indicate which mode (which disks are missing)
mtdcr(RAID_M, 0x00000007);//find P&Q
//indicate what is lost
mtdcr(RAID_LOST, 0x00000003);// P and Q is lost
//indicate that a reconstruction is needed
mtdcr(RAID_RECON, 0x00000001);

return 0;
}
```

Four software files used to communicate with the hardware accelerator:

- `raid_sim.c` – is the main program that request different regeneration cases
- `gf_gen.c` – defines the subroutines for the different data losses like the one shown above for P and Q regeneration
- `ddr_mpmc_access.c` – defines subroutines to access the cache memory
- `raid_sim.h` – defines variables for different DCR registers

Performance and Utilization

Time to Complete Regeneration Calculations

Table 6 shows the regeneration time for the RAID hardware to complete different regeneration calculations. The time for the PowerPC 405 processor to setup the DCR registers is *not* included in these regeneration times. Time is measured from the point the PowerPC 405 processor sets the RAID_RECON DCR control register bit to the time the hardware resets the RAID_RECON status bit. All measurements are for the default 512-byte blocks. 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 read four blocks from the DDR memory in the update P and Q process versus only three 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.

Table 6: **Block Regeneration Time for a Three Data Drive Structure**

Type of Request	Time for 512 bytes	Number of DDR Memory Reads	Number of DDR Memory Writes
single P	8.6 µs	3	1
single Q	11.4 µs	3	1
single Data	8.7 µs	3	1
double P & Q	14.0 µs	3	2
double Q & D	13.1 µs	3	2
double P & D	13.1 µs	3	2
double D & D	14.2 µs	3	2
update P & Q	15.0[1]	3	3

Notes:

1. Improved vs. double P and Q because more data drives are added.

The hardware accelerator is added to a modified Gigabit System Reference Design (GSRD) reference design that uses the MPMC of Xilinx application note XAPP535: *High Performance Multi-Port Memory Controlle*r. This configuration contains the PowerPC 405 processor, DCR, PLB bus, MPMC, OCM and the 32-bit data bus version of the RAID hardware accelerator that includes four instantiations of the DMB logic block. The performance and utilization of this *real world* system is shown in Table 7. The reference design can be implemented on the slowest speed grade device.

Table 7: **System Performance and Utilization in XC4VFX20-FF672 FPGAs**

Slices	Block RAM	I/O	DCM	DSP48 Block	PowerPC 405 Processor	RAID	MPMC
3900	57	76	3	4	300 MHz	150 MHz	100 MHz
45%	83%	10%	75%	12%	–	–	–

The hardware accelerator-only metrics are shown in Table 8.

Table 8: **RAID Performance and Utilization[1]**

Slices	Block RAM	DCM	DSP48 Block
1252	36	1[2]	4
14%	52%	25%	12%

Notes:

1. 4-byte data manipulation.
2. The design shares a system DCM, so an additional one is not needed for this block.

Ports

Most ports of the hardware accelerator interface directly to the MPMC and several others hook up to DCR raid block. Additional I/O pins can be allocated for the two clocks and the system reset. These ports are described in Table 9.

Table 9: **List of Hardware Accelerator Ports**

Port	I/O	Signal Width	Interface	Description
clk	I		CLK	RAID logic clock
clk_half	I			MPMC interface clock
rst	I		RST	RAID logic reset

Table 9: **List of Hardware Accelerator Ports *(Continued)***

Port	I/O	Signal Width	Interface	Description
start	I		DCR	Start the regeneration process when High
DCR_sel	O	[1:0]		Selects which DCR registers to read
DCR_reg0	I	[31:0]		First DCR register input
DCR_reg1	I	[31:0]		Second DCR register input
mem_dataout	O	[7:0]	MPMC	Write data to the MPMC
done	O		DCR	Indicates regeneration complete when High
gfc_done	O			Reserved
mp_addr	O	[31:0]	MPMC	Cache memory address
mp_addrreq	O			Request access to cache memory
mp_RNW	O			Read or write to cache memory
mp_size	O	[1:0]		Size of cache memory burst (set to 32)
mp_addrack	I			MPMC acknowledge cache memory request
mp_rddataack_pos	O			Unused in burst mode
mp_rddataack_neg	O			
mp_rdcomp	O			Indicates the cache memory read is complete
mp_rd_rst	O			Reset the read of cache memory
mp_rddata_pos	I	[31:0]		Cache memory read data positive edge
mp_rddata_neg	I	[31:0]		Cache memory read data negative edge
mp_rdwdaddr_pos	I	[4:0]		Unused in burst mode
mp_rdwdaddr_neg	I	[4:0]		
mp_rddatardy	I			Indicates the cache memory read data is ready
mp_rdfifoempty	I			Indicates the MPMC read FIFO is empty
mp_wrdataack_pos	O			Unused in burst mode
mp_wrdataack_neg	O			
mp_wrdata_pos	O	[31:0]		Cache memory write data positive edge
mp_wrdata_neg	O	[31:0]		Cache memory write data negative edge
mp_wrcomp	O			Indicates the cache memory write is complete
mp_wr_rst	O			Resets the write of cache memory
mp_wrfifobusy	I			Indicates the MPMC write to the cache memory
mp_wrfifofull_pos	I			Indicates the MPMC write FIFO is full for positive edge clock
mp_wrfifofull_neg	I			Indicates the MPMC write FIFO is full for negative edge clock

The DCR raid block is a DCR register implementation for hardware accelerator. Table 10 lists the ports that can connect to a PowerPC 405 DCR system.

Table 10: **List of DCR RAID Block Ports**

Port	I/O	Data Width	Interface	Description
RST_I	I		RST	DCR reset
CLK_I	I		CLK	DCR clock
DCR_Abus	I	[9:0]	PPC	DCR address bus from the PowerPC 405 processor
DCR_DBusIn	I	[31:0]		DCR data bus from the PowerPC 405 processor or other DCR register blocks
DCR_Read	I			Read signal from the PowerPC 405 processor
DCR_Write	I			Write signal from PowerPC 405
DCR_Ack	O			DCR register acknowledge to the PowerPC 405 processor or other DCR register blocks
DCR_DBusOut	O	[31:0]		DCR data bus from registers to the PowerPC 405 processor or other DCR register blocks
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]		
LED	O	[3:0]		
GFC_DONE	I		RAIDGEN	Reserved

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 DCR registers must be expanded to provide additional pointers to data blocks as well as control functions for data recovery operations. Also, the software must manage more base memory address pointers.

Other Uses of the Hardware Accelerator

As mentioned earlier 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.

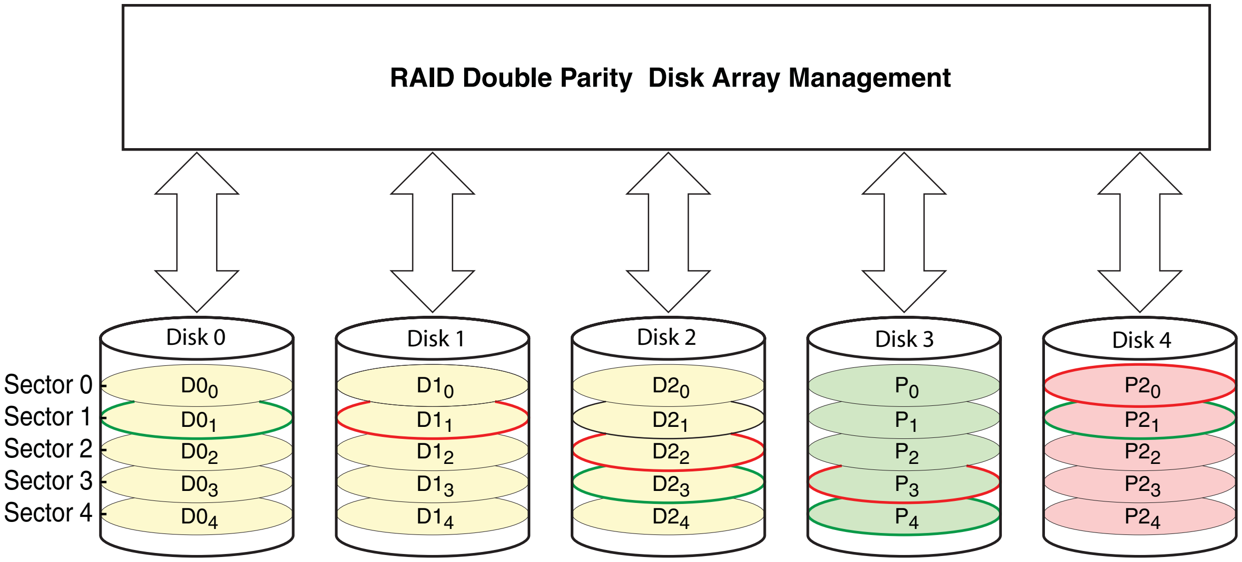
The hardware accelerator can support other RAID levels beyond RAID6 with minimal (if any) modifications. RAID Double Parity (DP), RAID5, RAID4, and RAID3 are among the supportable levels.

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

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 7, 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 7, 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 $P2_0$.



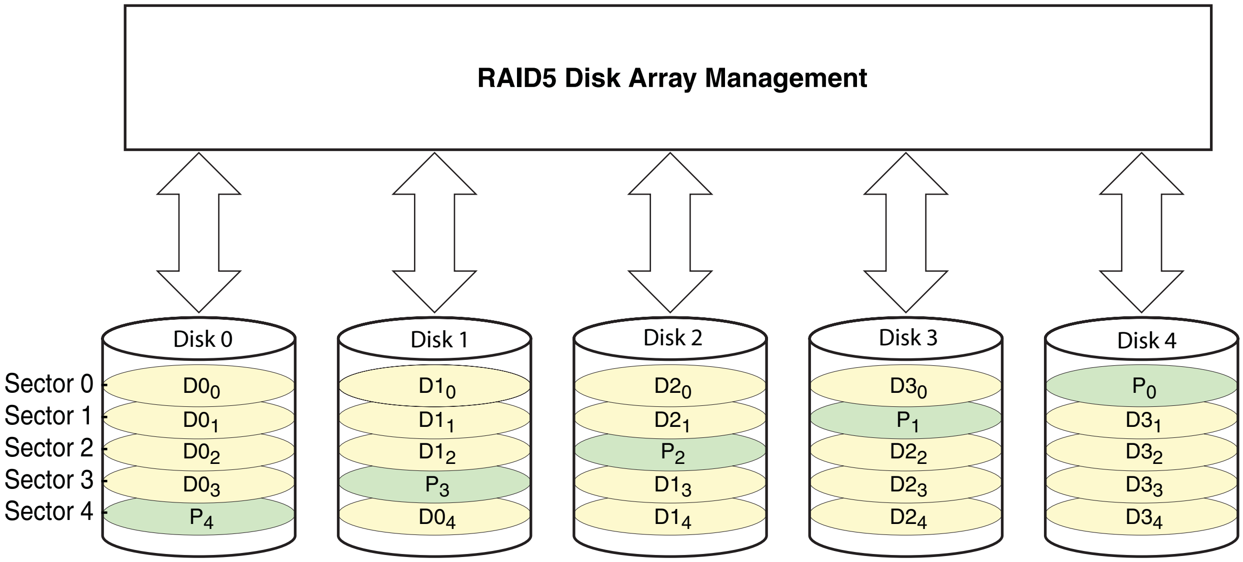
XAPP731_07_062006

Figure 7: **RAID DP Data Structure**

RAID5

RAID5, shown in Figure 8, 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).

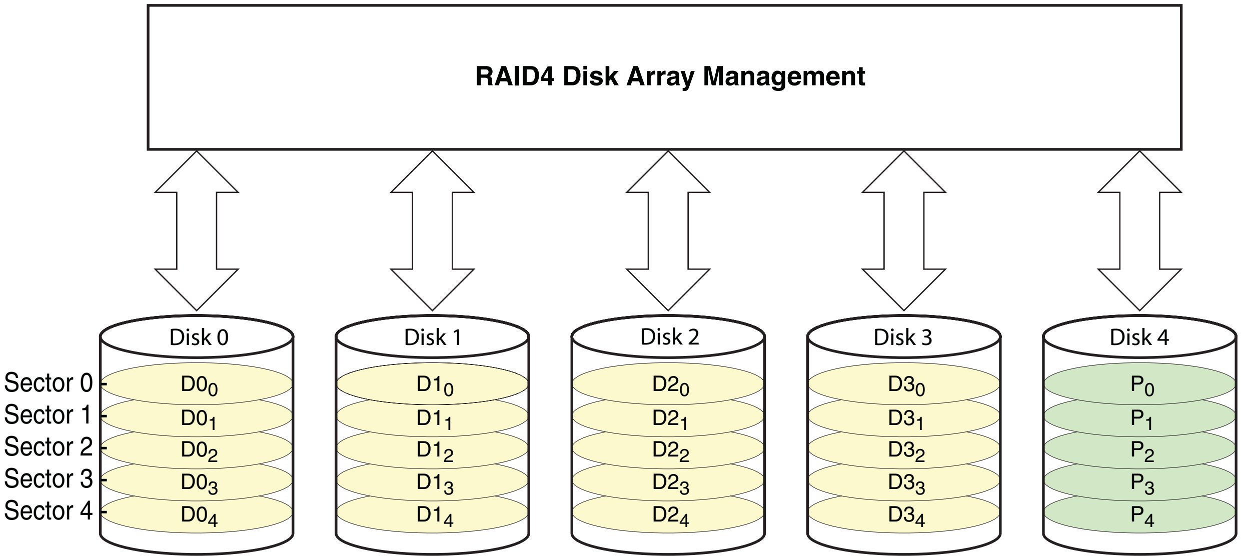


XAPP731_08_062006

Figure 8: **RAID 5 Data Structure**

RAID4 and RAID3

RAID4 and RAID3 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 9.



XAPP732_09_062006

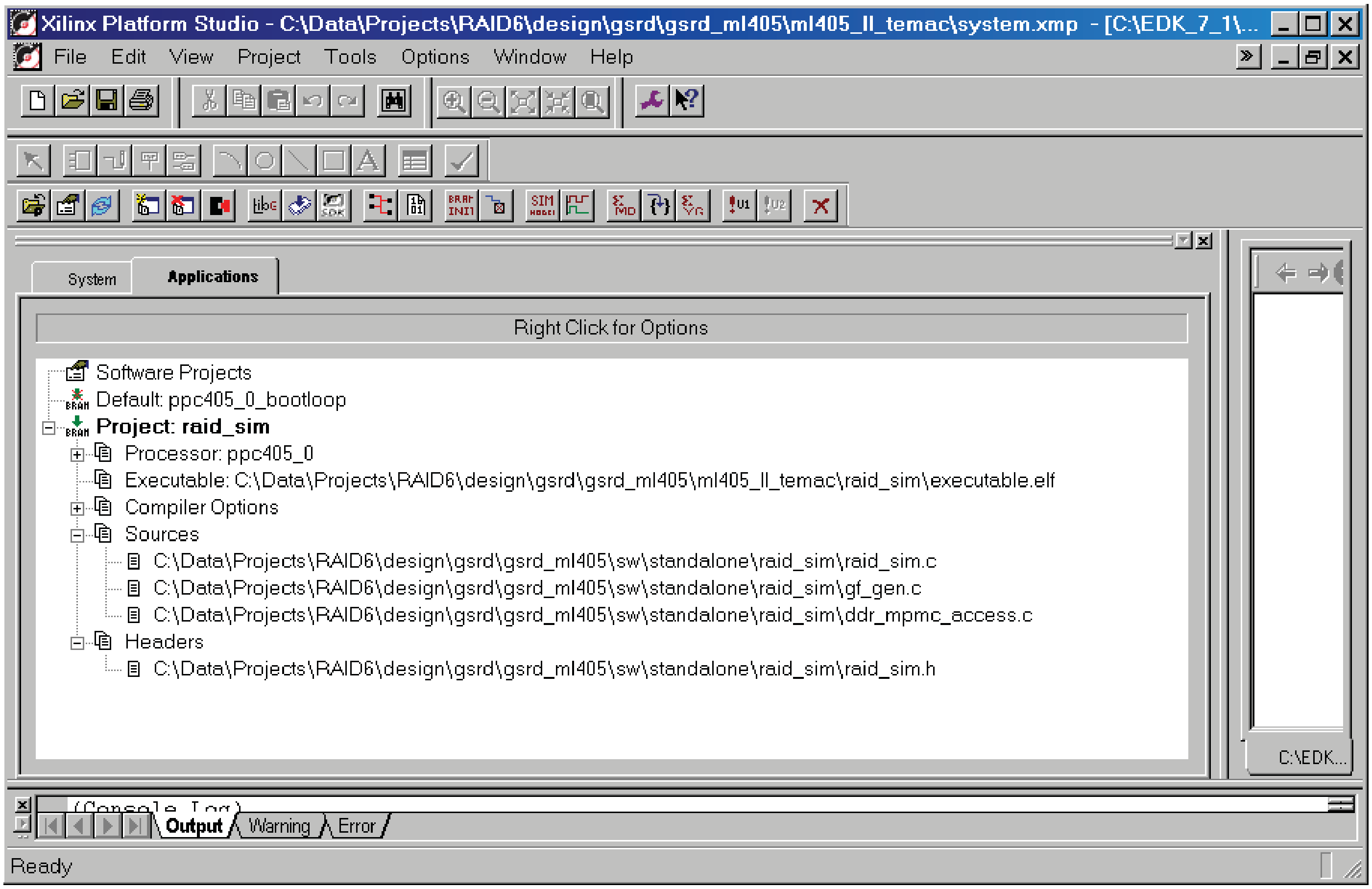
Figure 9: **RAID4 Data Structure**

Reference Design Simulation

The reference design is based on a Platform Studio (or EDK) 7.1i system. To simulate this design, downloading the GSRD reference design is required. This design can be downloaded from http://www.xilinx.com/esp/wired/optical/xlnx_net/gsrd_download

The GSRD design does require registration. Once registered, the useer can download ZIP files for EKD 7.1 SP2 (requires ISE 7.1 SP4). After the GSRD design has been successfully simulated to verify that the EDK simulation libraries work properly, modification of the design can begin. The design file from can be downloaded from: http://www.xilinx.com/bvdocs/appnotes/xapp731.zip

1. Replace the `system.mhs` in the top directory (same level as the Platform Studio Project is at) with the one in the ZIP file.
2. Load the `RAID6_block_ip` and `DCR_raid6` folders from the ZIP file into the `pcore` directory.
3. Create a `/sw/standalone/raid_sim` directory. Copy the three `.c` files and the `.h` files from the ZIP file into this directory.
4. Setup a new software project in Platform Studio and add the source and header files as shown in Figure 10. See the EDK documentation for more details on step-by-step procedures.



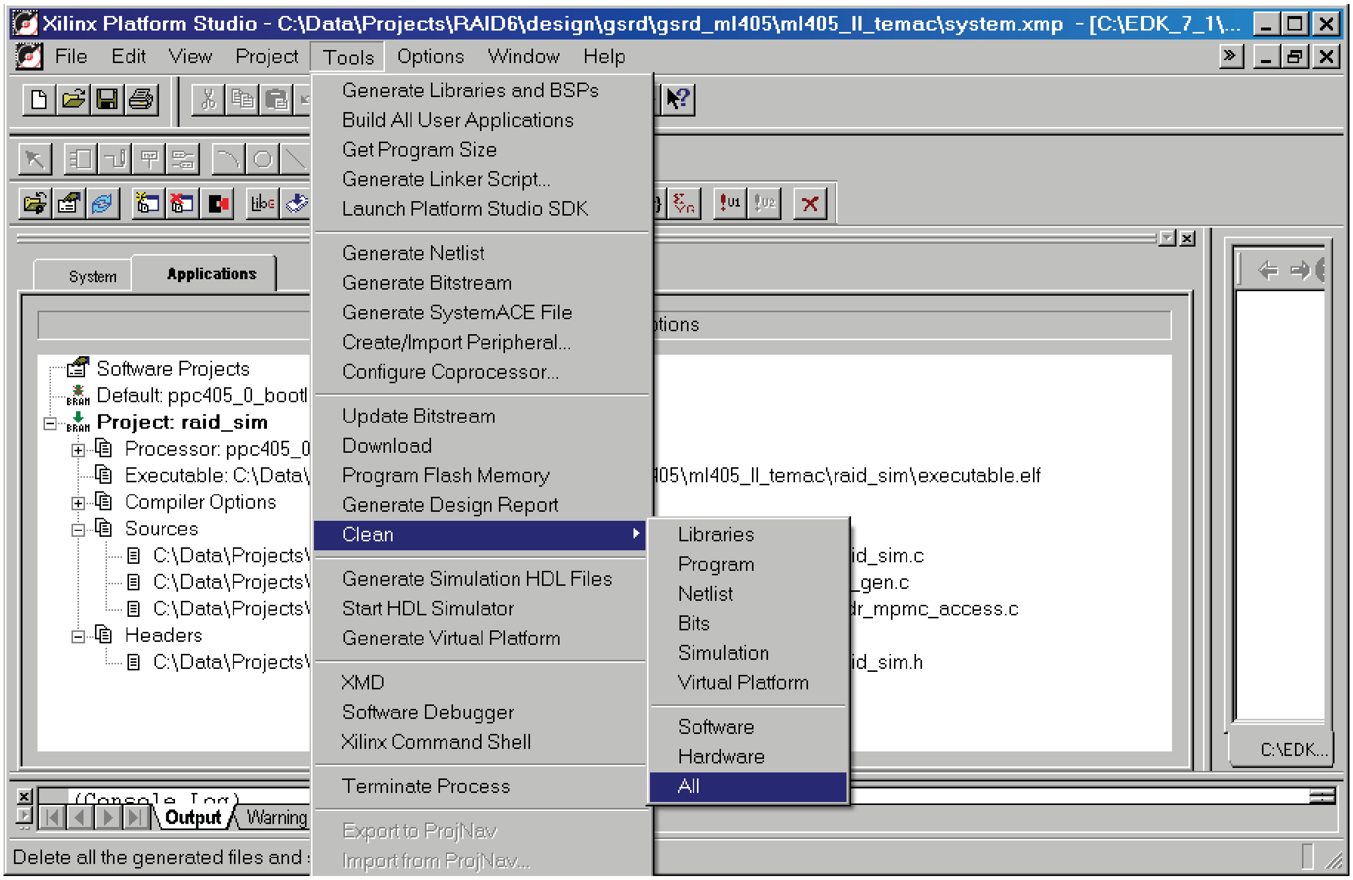
Xilinx Platform Studio - C:\Data\Projects\RAID6\design\gsrd\gsrd_ml405\ml405_ll_temac\system.xmp - [C:\EDK_7_1\...

Software Projects
Default: ppc405_0_bootloop
Project: raid_sim
Processor: ppc405_0
Executable: C:\Data\Projects\RAID6\design\gsrd\gsrd_ml405\ml405_ll_temac\raid_sim\executable.elf
Compiler Options
Sources
C:\Data\Projects\RAID6\design\gsrd\gsrd_ml405\sw\standalone\raid_sim\raid_sim.c
C:\Data\Projects\RAID6\design\gsrd\gsrd_ml405\sw\standalone\raid_sim\gf_gen.c
C:\Data\Projects\RAID6\design\gsrd\gsrd_ml405\sw\standalone\raid_sim\ddr_mpmc_access.c
Headers
C:\Data\Projects\RAID6\design\gsrd\gsrd_ml405\sw\standalone\raid_sim\raid_sim.h

XAPP731_10_030807

Figure 10: Structure of the RAID Software Project

5. Clean all generated files (Figure 11).

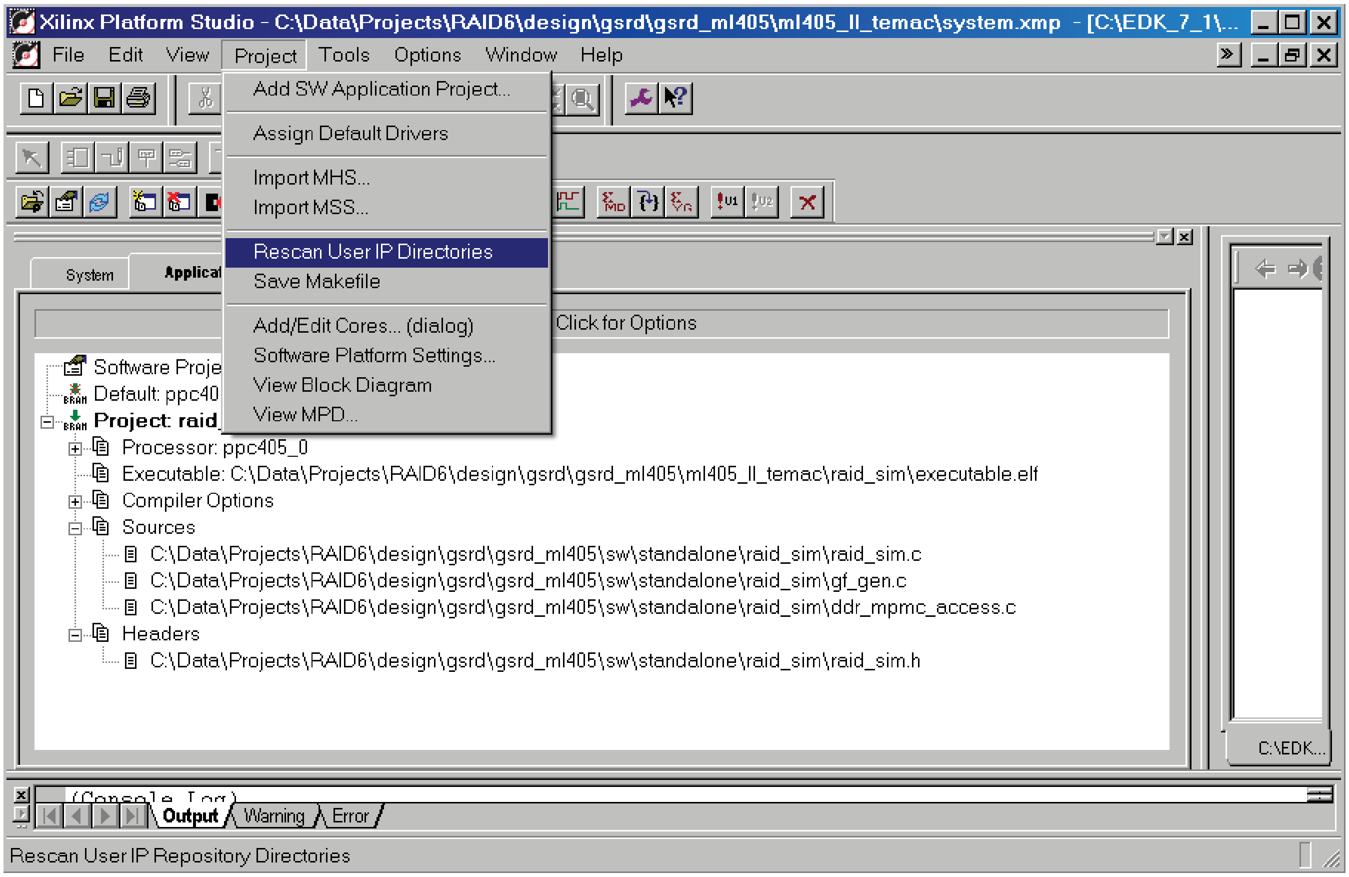


Delete all the generated files and

XAPP731_11_021706

Figure 11: Cleaning All EDK Generated Files

6. Re-scan the IP (Figure 12).


Xilinx Platform Studio - C:\Data\Projects\RAID6\design\gsrd\gsrd_ml405\ml405_ll_temac\system.xmp - [C:\EDK_7_1\...
File Edit View Project Tools Options Window Help
Add SW Application Project...
Assign Default Drivers
Import MHS...
Import MSS...
Rescan User IP Directories
Save Makefile
Add/Edit Cores... (dialog)
Software Platform Settings...
View Block Diagram
View MPD...
System
Applicat
Click for Options
Software Proje
Default: ppc40
Project: raid
Processor: ppc405_0
Executable: C:\Data\Projects\RAID6\design\gsrd\gsrd_ml405\ml405_ll_temac\raid_sim\executable.elf
Compiler Options
Sources
C:\Data\Projects\RAID6\design\gsrd\gsrd_ml405\sw\standalone\raid_sim\raid_sim.c
C:\Data\Projects\RAID6\design\gsrd\gsrd_ml405\sw\standalone\raid_sim\gf_gen.c
C:\Data\Projects\RAID6\design\gsrd\gsrd_ml405\sw\standalone\raid_sim\ddr_mpmc_access.c
Headers
C:\Data\Projects\RAID6\design\gsrd\gsrd_ml405\sw\standalone\raid_sim\raid_sim.h
C:\EDK...
(Console Log)
Output
Warning
Error
Rescan User IP Repository Directories

XAPP731_12_021706

Figure 12: Rescanning all IP Directories

7. Generate the simulation files.
8. Run the simulation.
9. Copy files from the `sim` directory of the ZIP file into behavioral directory of the EDK design.
10. Modify `system_init.v` by replacing every instance of `system.` with `tb.system.`
11. Type `do run_me.do` in MTI window (Modelsim SE 6.0C was used for simulation).

This testbench has the software request regeneration types for data of `0x00–0xFF`.

A UCF file for the ML405 demo board is also included (in the reference design ZIP file) for running the design through the implementation tools. Replace the original `system.ucf` with the file from the reference design ZIP file. A `download.bit` file is also included in the ZIP file for implementing the design on the ML405 demo board.

CORE Generator FIFO Settings

Two CORE Generator FIFOs are part of the MPMC_IF block, providing clock domain crossing between the 100 MHz MPMC memory controller and the 150 MHz RAID clock domains. It is possible that they will need regeneration in the future releases of the Xilinx implementation tools. To regenerate these FIFOs, use the process shown under "`async_read_fifo`," "`async_write_fifo`," and "`handshake_fifo`" by opening the CORE Generator project in `pcores/raid6_block/hdl/verilog/fifo_coregen` directory, which is supplied in the ZIP file.

`async_read_fifo`

- Select Independent Clocks – Block RAM
- Select Register Outputs
- Write width and Depth is 64
- Read width is 32
- Select almost full and almost empty flags
- No programmable full or empty thresholds

`async_write_fifo`

- Select Independent Clocks - Block Ram
- Select Register Outputs
- Write width is 32
- Write Depth is 64
- Read width is 64
- Select almost full and almost empty flags
- No programmable full or empty thresholds

`handshake_fifo`

- Select Independent Clocks - Distributed RAM
- Select Register Outputs
- Write width is 36
- Write depth is 16
- Select almost full and almost empty flags
- No programmable full or empty thresholds

Then place the wrapper files (`async_read_fifo.v`, `async_write_fifo.v, and handshake_fifo.v`) and `ngc` files (`async_read_fifo.ngc`, `async_write_fifo.ngc, and handshake_fifo.ngc`) in the `pcores/raid6_block/hdl/verilog` directory.

Reimplementation of the Design

If reimplementation of this design is required, please see `readme.txt` in the ZIP file: http://www.xilinx.com/bvdocs/appnotes/xapp731.zip.

Conclusions

This reference design supports calculating Reed-Solomon RAID6 parity generation and data regeneration on 512- to 4096-byte blocks of data from a DDR memory. This design takes advantage of multiple immersed IP blocks of the Virtex-4 FPGA to improve performance and decrease fabric utilization. The block RAMs are used for the GF mathematical LUTs, the DSP48 blocks perform fast integer addition, and the PowerPC 405 processor handles the memory/address management, 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-4 FPGA, among them is using the MPMC 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 trade-offs 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 ML405 demonstration board.

References and Additional Information

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
6. XAPP535, *High-Performance Multi-Port Memory Controller*
   http://www.xilinx.com/bvdocs/appnotes/xapp535.pdf
7. XAPP536, *Gigabit System Reference Design*
   http://www.xilinx.com/bvdocs/appnotes/xapp536.pdf
8. UG073, *XtremeDSP for Virtex-4 FPGAs User Guide*
   http://www.xilinx.com/bvdocs/appnotes/ug073.pdf

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
04/19/06	1.0	Initial Xilinx release.
03/20/07	1.1	Fixed hex notation in Equations, added Sector[4:0] and Disk[4:0] notation to RAID Disk Array figures, changed Equation cross-references, and updated Figure 4.