WP143 (v1.0) May 8, 2001

Xilinx Generic Flash Memory Interface Solutions

Introduction

This white paper shows how a generic flash memory interface can be combined with Xilinx IP interface cores to add flash memory to Xilinx Spartan device designs. The flexible Xilinx programmable solutions presented here address three issues of primary importance in a flash memory interface design:

- **Design Reuse** — A solution must accommodate design changes made to increase performance and meet different memory density requirements.
- **Performance** — A solution should maximize system-level performance by providing automatic status polling, write buffering, and other system-friendly features.
- **Device Availability** — Flash memory devices are popular choices for product design, and device demand often outpaces supply. The flash memory interface solution should allow designers to choose among different flash memory devices late in the product development cycle.....

Figure 1 shows a block diagram overview of the architecture of a generic flash memory interface. Application-specific logic connects to flash memory through a generic flash interface. Note that application logic, the generic flash interface, and flash interface cores are implemented on a single Xilinx Spartan-II device. Since the generic interface supports multiple flash memory devices, the designer can place footprints for more than one flash family device on the printed-circuit board.

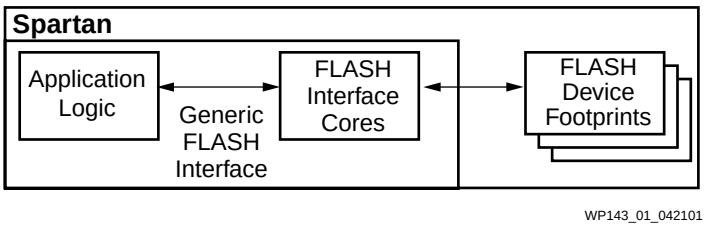

Figure 1: **Generic Flash Interface Architecture**

The following sections provide an overview of the flash memory market and flash technologies, describe a generic flash interface, then illustrate the implemenation of the interface for the two prevailing flash memory technologies.

The Flash Memory Market

Demand for flash memory has grown rapidly, driven by the manufacture of a wide range of portable and embedded products with increased storage requirements. The most notable of the application areas that employ flash memory storage are:

- **Bulk Data Storage** — Flash memory replaces bulk storage media, such as hard disks. Typical applications include PCMCIA flash-storage cards, MP3 players, and digital voice recorders. Key application requirements are high density (>1 MB), low cost per byte, and tolerance for storage errors
- **Embedded Code Storage** — Flash memory stores program code for an embedded processor, such as a high-end 32-bit, RISC CPU used in a router or a DSP used in a cellular phone. Key application requirements are modest (<1 MB) memory density, high performance, and error-free storage

Flash Technology Background

There are two primary flash technologies — NOR and NAND technologies — whose names are derived from the type of semiconductor logic used to implement flash memory structures. Figure 2 illustrates the structures used in each flash technology type.

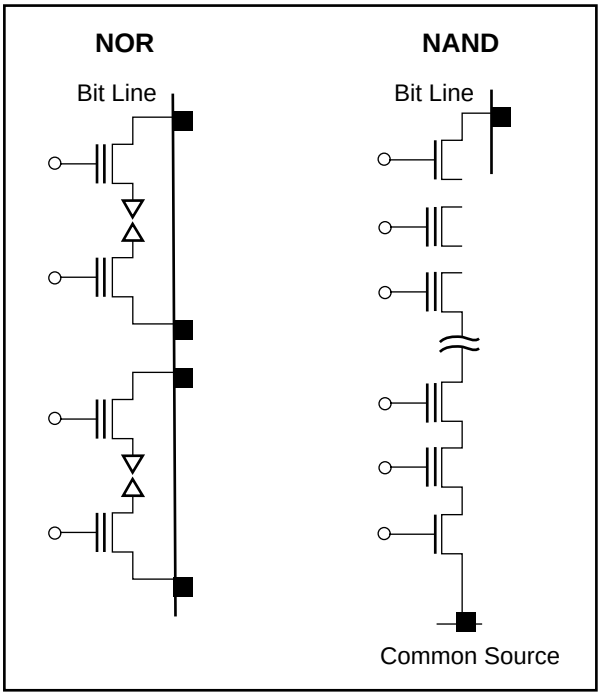

Figure 2: **NOR and NAND Flash Memory Structures**

NOR vs. NAND Technology

Each flash memory technology is designed to match the basic requirements of the two primary flash application areas: NOR technology finds frequent use in embedded core applications while NAND flash memory is commonly used to replace bulk storage.

- NOR technology is featured in Intel and AMD flash memory devices. It features high-speed, random access to bit storage and low error rates, but offers less density than NAND flash technology. It has high endurance—that is, it can reliably support up to one million program/erase cycles without requiring error correction
- NAND technology ties a string of 16 or 32 single-bit-storage transistors together. This structure reduces the number of memory array contacts and allows greater density. The trade-offs involved with implementing this structure are 1) data cells must accessed sequentially, and 2) read and write errors are more prevalent. As a result, NAND devices require error correction

Flash Device Families

Manufacturers produce several flash device families, each targeted at different application areas. *Table 1* lists several popular flash device families and their characteristics.

TABLE 1 – FLASH DEVICE FAMILIES (NEED INPUT FOR THIS TABLE--WAS NOT ABLE TO CONVERT--NOT IN WORD FILE

AMD Am29Fxxx

This NOR device family led the migration path to flash memory by featuring EPROM pin compatibility. Limitations include lack of support for 3.3V (or lower) operation, 32 Mb maximum density, and a relatively high cost per bit. The Am29BDS643D is a recent AMD family member that targets cell phones and other low-power, embedded applications. It includes the latest

flash industry features, including dual banks for simultaneous red/write support and burst-mode operation

Intel StrataFlash

The NOR devices in the Intel StrataFlash family employ four levels of charge storage, which allows each transistor to store two data bits. This design gives the StrataFlash family the lowest cost per bit among NOR flash families

Micron SyncFlash

This recently-introduced, high-performance NOR device family is designed to plug into existing synchronous DRAM interfaces. It does not require a flash interface

Samsung/Toshiba NAND

These NAND devices feature a low cost per bit, making them ideal for use in hard disk replacement and Internet audio player applications. They lack random read/write capability, and like other NAND devices, require system-level error checking and correction code (ECC)

AMD UltraNAND

The UltraNAND family represents AMD's first entry in the NAND flash market. UltraNAND devices are pin and command-set compatible with standard NAND devices, and specify a 100,000 program/erase cycle capability without error correction

Generic Flash Interface

The goal of a generic flash interface design is to provide a 32-bit, synchronous, memory-mapped interface that is seen by application-specific logic as a standard RAM module. To implement this functionality, the interface provides data and address lines, select lines that choose memory or control register operation, and, ideally, an interrupt line to signal the completion of flash program and erase cycles.. Providing an interrupt output eliminates the need for application polling to determine program/erase cycle status. Table 1 provides a list of generic flash interface signals, and Figure 3 shows read/write timing for the generic interface.

Table 1: **Generic Flash Interface Signals**

Signal	Direction	Description
CLK	To Core	System Clock—data, address, and status signal transfers across the interface are relative to the rising edge of this clock
FL_SEL_N	To Core	Flash Select—an active-low signal that selects flash memory for the current operation
CR_SEL_N	To Core	Control Register Select—an active-low signal that selects one of the control/status registers for the current operation
ADDR[N:0]	To Core	Address Vector—an address vector whose width varies according to the application
DI[31:0]	To Core	Data Input—a 32-bit data input port for write (program/erase) operations
DO[31:0]	From Core	Data Output—a 32-bit data output port for read operations
WR_N	To Core	Write—an active-low signal that indicates the presence of valid data input and address data for a flash write operation

Table 1: **Generic Flash Interface Signals *(Continued)***

Signal	Direction	Description
RD_N	To Core	Read—an active-low signal that indicates the presence of valid address data for a flash read operation.
INT_N	From Core	Interrupt—an active-low signal that indicates the completion of a flash operation
RDY_N	From Core	Ready—an active-low signal that indicates the flash core has accepted write data, or has presented valid read data on the DO port

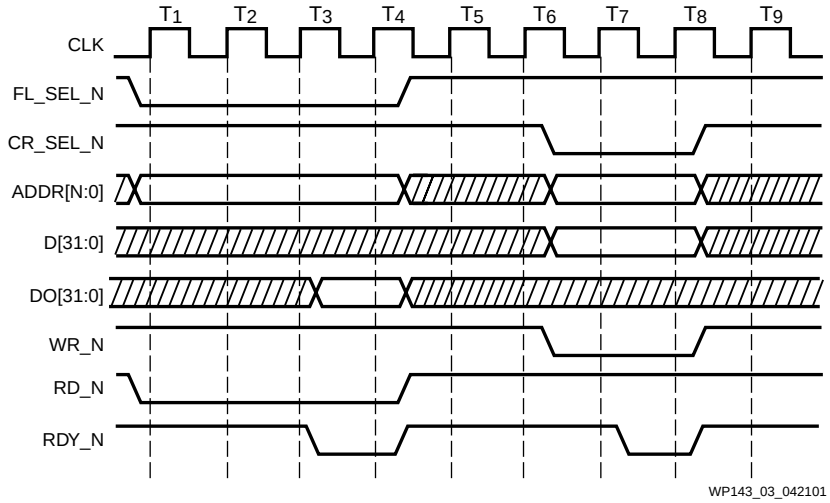

Figure 3: **Generic Flash Interface Read/Write Timing**

System-Level Interface Considerations

The primary design goal of a generic flash interface is to simplify the interoperability of different flash families with common application-specific logic. Specifically, a generic interface design should mask flash device differences from the application, such as:

- Data path width
- Transfer timing
- Transfer handshaking
- Polling algorithms

Due to the wide range of sector sizes supported by different flash families, complete device masking is normally not possible. (Also, a flash device may support more than one sector size.) Therefore, system-level logic may need to accommodate different flash device sector sizes.

System-Level Software Interfaces

Software interfaces have been developed to help integrate flash memory at the system level. Two common flash software protocols are:

- **Flash Translation Layer (FTL)** — FTL is designed for flash disk-replacement applications. It specifies an interface that makes linear flash memory appear as a disk drive to the system application. FTL was defined as part of the PCMCIA-promulgated PC Card Standard
- **Common Flash Interface (CFI)** — CFI is a specification that describes how system software and device programmers can query flash memory devices—for example, to determine flash device characteristics

Also, flash device manufacturers provide subroutine libraries of abstract functions to perform flash device operations, such as memory block erasure and device status polling.

Flash Core Interface Examples

This section provides examples of generic flash interface designs for both NOR and NAND flash memory cores.

NOR Flash Interface Example

This example summarizes the logic necessary to implement a generic flash interface for a NOR flash device. The device family selected for this example is the AMD Am29Fxxx family (see *Table 1*). This flash family includes flash devices with densities from 1 to 32 Mb, access times as low as 45 ns, and a guaranteed erase-cycles-per-sector specification of 1 million cycles. Figure 4 is a block diagram of the largest member of the family, the Am29F032B (for a data sheet, see the AMD website at **http://www.amd.com/**).

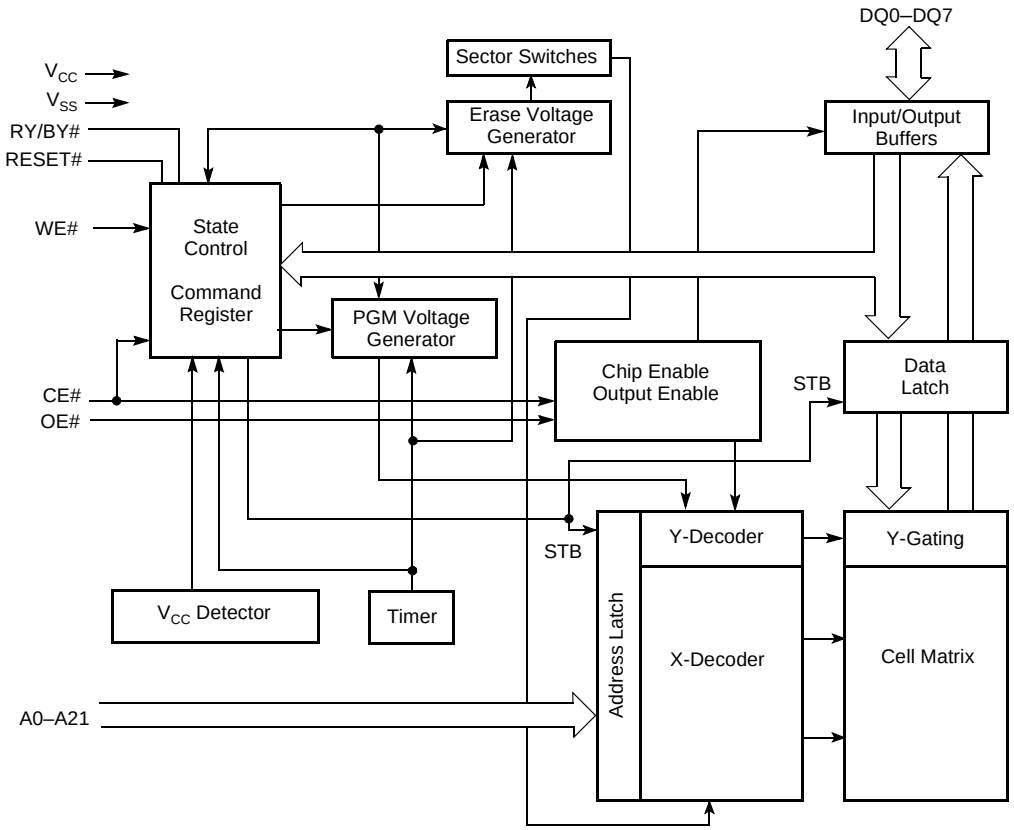

Figure 4: **Am29F032B Block Diagram**
(Figure Courtesy Advanced Micro Devices)

The Am29Fxxx flash family features a straightforward asynchronous, non-multiplexed interface, originally designed to allow flash device use in existing EPROM designs. Table 2 lists the signals, and Figure 5 the read protocol and timing, for this device family.

Table 2: **Am29Fxxx Device Signal Summary**

Signal	Direction	Description
A[N:0]	To Flash	Address bus
DQ[7:0]	To/From Flash	Data bus—used to transfer write and read data to and from the flash
CE_N	To Flash	Chip Enable—an active-low signal that enables the device
WE_N	To Flash	Write Enable—write data is latched on the rising edge of this strobe
OE_N	To Flash	Output enable—an active-low enable for device output buffers
RESET_N	To Flash	Reset—an active-low signal that resets device state machines.
RY/BY_N	From Flash	Ready/Busy—an open-drain output that is pulled low when a program, erase or read operation is in progress.

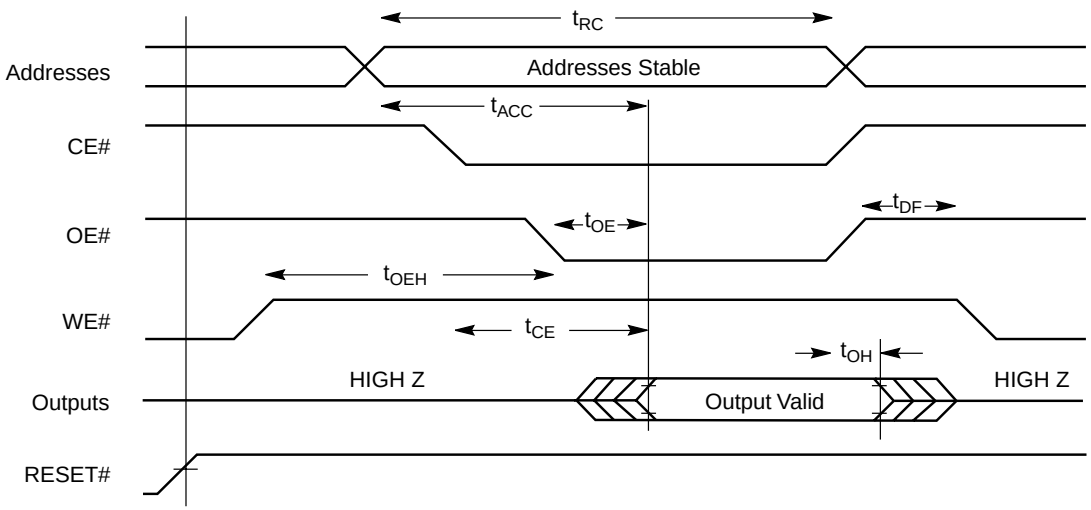

Figure 5: **Am29Fxxx Read Protocol and Timing**
(Figure Courtesy Advanced Micro Devices)

NOR Flash Interface Implementation

Implementing an interface for a Am29Fxxx NOR device is relatively simple. It requires a non-multiplexed, memory-mapped interface to perform the following functions:

- Generation of read/write timing
- Byte/word multiplexing and demultiplexing
- Status polling for devices without a RY/BY_N signal

Figure 6 illustrates the blocks for implementing a NOR flash interface. Application signals are on the left, and flash device signals are on the right. Most of the complexity is in the control logic. The control logic block accesses the address and read data paths to support automatic polling (to sense the completion of erase and program operations). Data path functions are

performed by the multiplexer and registers block, which convert the data bus width from 8 to 32 bits.

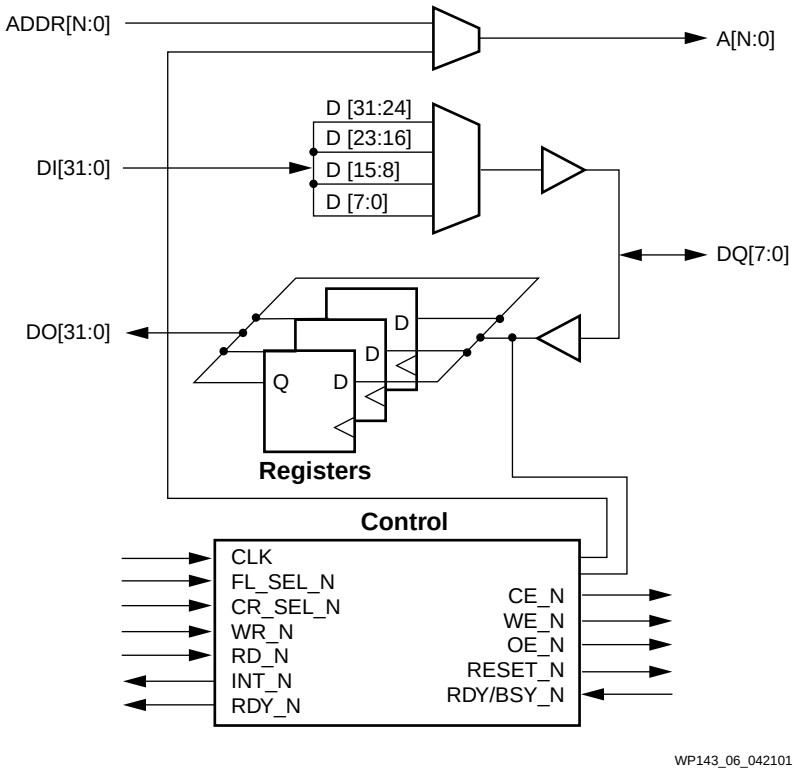

Figure 6: **NOR Flash Interface Block Diagram**

NAND Flash Interface Example

This example describes the logic necessary to implement a generic flash interface for the popular Samsung K9F6408U0M (8M x 8-bit) NAND device (a pin-for-pin compatible device is

available from Toshiba). Figure 7 shows the block diagram for this device (for a data sheet, see the Samsung website at **http://www.samsung.com/**).

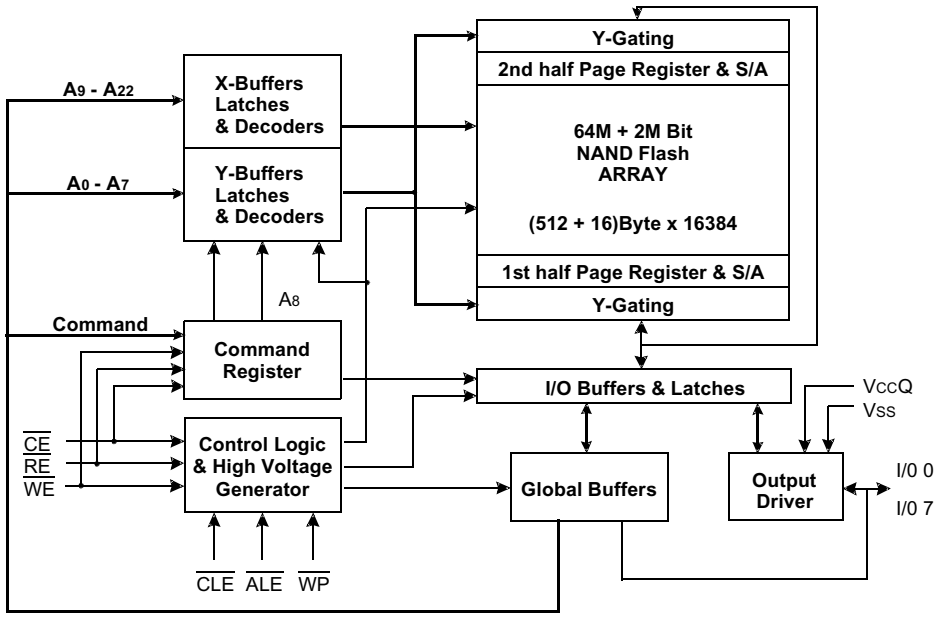

Figure 7: **K9F6408U0M Block Diagram**
(Figure Courtesy Samsung Semiconductor)

There are several key NAND flash technology characteristics that affect interface design. These include:

- Block-transfer architecture
- Access latency
- Memory errors

Block Transfer Architecture

NAND devices only support block-read and block-write operations. This block-transfer orientation is reflected in a narrow, highly-multiplexed device architecture. Specifically, the K9F6408U0M employs an eight-bit port, which serves both as an address and data bus. This block-transfer architecture complicates the task of interfacing NAND devices to the generic flash interface, which must provide the host processor with random access to flash memory. Table 3 lists the signals, and Figure 8 the read protocol and timing, for the K9F6408U0M device.

Table 3: **K9F6408U0M Device Signal Summary**

Signal	Direction	Description
CLE	To Flash	Command Latch Enable—an active-high command latch enable signal
ALE	To Flash	Address Latch Enable—active-high address latch enable signal
IO[7:0]	To/From Flash	Input/Output bus—used to transfer commands, addresses, and write data to the flash, and to transfer read data from the flash
WE_N	To Flash	Write Enable—write data is latched on the rising edge of this strobe
RE_N	To Flash	Read enable—an active-low enable for the device data output buffers. Assertion of RE_N also increments the column address counter

Table 3: **K9F6408U0M Device Signal Summary *(Continued)***

Signal	Direction	Description
SE_N	To Flash	Spare area Enable—an active-low signal that enables 16 bytes of spare memory on each page
WP_N	To Flash	Write Protect—an active-low signal that disables device writes when asserted
R/B_N	From Flash	Ready/Busy—an open-drain output that is pulled low when a program, erase, or read operation is in progress

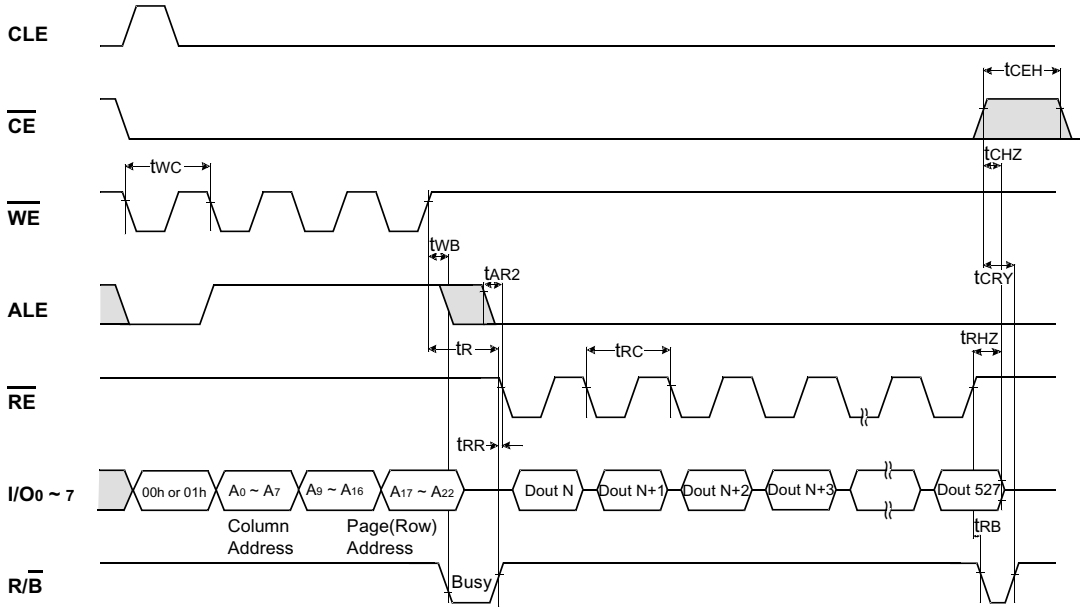

Figure 8: **K9F6408U0M Read Protocoland Timing**
(Figure Courtesy Samsung Semiconductor)

Access Latency

NAND flash devices exhibit significant read-access latency. NAND read-access latency is depicted as t_R in Figure 8 (maximum worst-case t_R for the K9F6408U0M is 7 us). Although AMD UltraNAND devices have a gapless read feature that significantly reduces access time to sequential blocks, random access exhibits latency. Due to these latency characteristics, NAND devices normally are not used to directly execute program code (for example, in embedded code storage applications). As a work-around to latency constraints, program code can be copied from flash to RAM at boot time, then executed from RAM. However, since the initial boot code necessary to initialize system RAM and perform the flash-to-RAM copy operation must execute directly from the flash device, this is only a partial solution.

Memory Errors

Even though a narrow block-transfer oriented NAND interface requires additional logic glue, a more challenging system-level interface issue is data integrity. There are two NAND data-integrity considerations: 1) devices may ship with invalid memory blocks, and 2) additional memory blocks may fail during device operation. The next sections describes software invalid-block mapping and error-correction-code (ECC) strategies that help ensure NAND data integrity.

Bad-Block Mapping

The number of valid blocks that a flash device contains when shipped is specified by its N_{VB} data sheet parameter. This value varies from device to device—the K9F6408U0M ships with standard minimum, maximum and typical values specified at 1,014, 1,024, and 1,020, respectively. The first block of a flash memory array is guaranteed be valid , but the remaining memory blocks may contain errors. Invalid blocks are marked at the factory with a “0” value stored at location 0 of the first or second page of the bad block (8K blocks are organized into 16 pages, with each page holding 512 bytes—each page also provides a spare 16 bytes). To avoid writing to and reading from bad memory blocks, system software must create a map of invalid memory blocks. If the application code executes from RAM rather than flash memory, system software bad-block mapping is only necessary at boot time and during flash storage updates. Further, since the first memory block is guaranteed to be valid, the first 8 kB of flash memory can safely be used for system bootstrapping functions.

Error Checking and Correction (ECC)

NAND devices are subject to data failures that occur during device operation. To ensure data read/write integrity, system error-checking and correction (ECC) algorithms must be implemented. Samsung suggests using a Hamming code to implement these functions. NAND devices provide 16 bytes of spare memory in each 512-byte memory page, which can be used to store ECC codes.

> **Note**: Invalid block-mapping, as described above, is used together with ECC techniques. Since operational failures often involve just one invalid bit per block, which can be corrected with ECC algorithms, it is normally not necessary to map out an entire block of memory to correct memory bit-failures (see the description of ECC logic, below, for a website link to ECC algorithm information).

NAND Flash Interface Implementation

Figure9 illustrates the functional blocks necessary to implement a NAND flash controller interface. Application signals are on the left, and flash device signals are on the right. This design can be used for both Samsung and Toshiba NAND devices as well as AMD UltraNAND devices. This NAND flash interface implements the following functions (the same as those listed for the NOR flash interface, above):

- Generation of read/write timing
- Byte/word multiplexing and demultiplexing
- Status polling for devices without a RY/BY_N signal

It also implements the following NAND-specific interface functions:

- ECC functions
- A two-page cache

The cache is necessary to buffer pages to perform ECC functions. It also improves system performance by reducing multiple-read operations and by decoupling the timing between application logic and the NAND device.

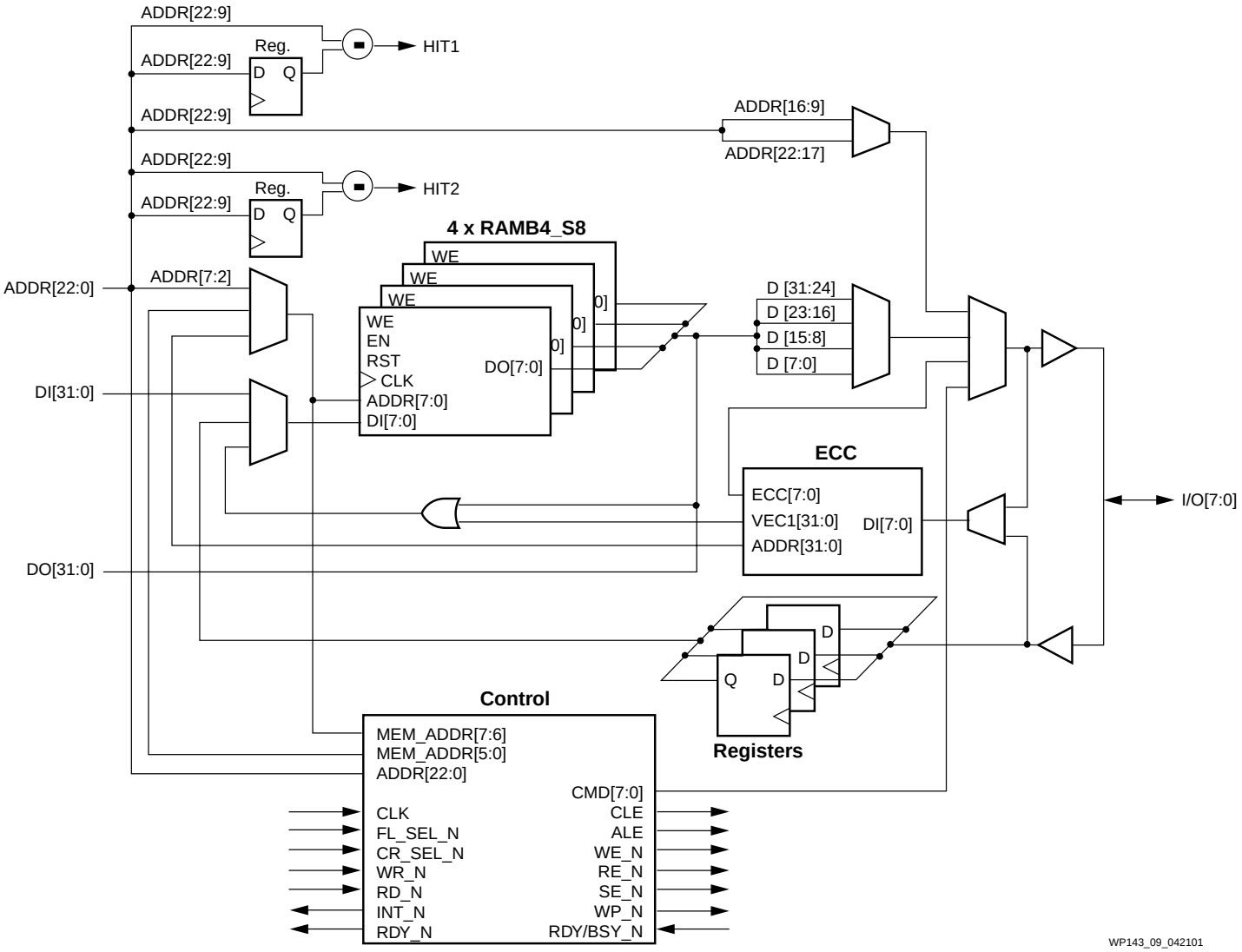

Figure 9: **NAND FLASH INTERFACE Block Diagram**

The three key NAND flash interface blocks are 1) block RAM memory used for storage, 2) ECC logic, and 3) control logic.

Page Caching Memory

The interface implements a two-entry page cache with storage implemented by four RAMB4_S8 Block RAM macros. The memory is connected in a 32-bit wide, 256-word deep configuration, which provides the bandwidth necessary for data storage access during read and write cycles. Two address registers with comparators are used to check for cache hits during interface read/write cycles.

For read operations, the cache is transparent to system software. However, due to the coarse, 8 KB erase granularity of the NAND device, write transparency is not achievable (read-modify-write techniques do not work). This means that write operations must be managed by system software. The software must write to a control logic register to initiate a block-write operation, then use page memory as a write buffer for the block write operation.

ECC Logic

The ECC logic implements double-bit error detection and single-bit error correction using a Hamming code that generates a three-byte check field for each 256-byte data segment (six code bytes per page). This coding scheme is described by Samsung in their application literature, available at **http://www.samsung.com/**. The ECC block compares stored ECC write-data codes to ECC read-data codes. If a correctable bit error is found, the ECC generates the corrected data.

The ECC block has the following data ports:

- D**ata In** — A byte-wide input port, which receives incoming and outgoing memory data
- **ECC Data** — A byte-wide output port, which produces the ECC code at the end of a page transfer. ECC values are appended to the transfer during write cycles
- **Correction Vector** — A 32-bit vector that flags an invalid bit. This vector is XORed with the invalid data word during memory-scrub operations
- **Correction Address** — A six-bit vector that identifies the address of the cache word with the bit error

Control Logic

The Control Logic consists of several small state machines. It is responsible for sequencing interface signals and managing the cache and ECC logic.

Conclusion

The interface implementations presented in this application note meet the design criteria required for generic flash memory interfaces. They adapt to design reuse requirements, off-load flash memory interface chores from application logic and software, and allow device targeting late in the manufacturing cycle. They provide additional system-level benefits as well: the NOR interface implementation handles erase and program-completion polling, while the NAND interface implementation provides ECC and cache capabilities.

References

1. *Xilinx Spartan-II FPGA data sheet*, January 2000, Xilinx
2. *Am29F032B 32 Megabit Uniform Sector Flash Memory* data sheet, November 1999, Advanced Micro Devices
3. *K9F6408U0M 8M x 8 Bit NAND Flash Memory* data sheet, September 1999, Samsung Semiconductor

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
05/08/01	1.0	Initial Xilinx release.