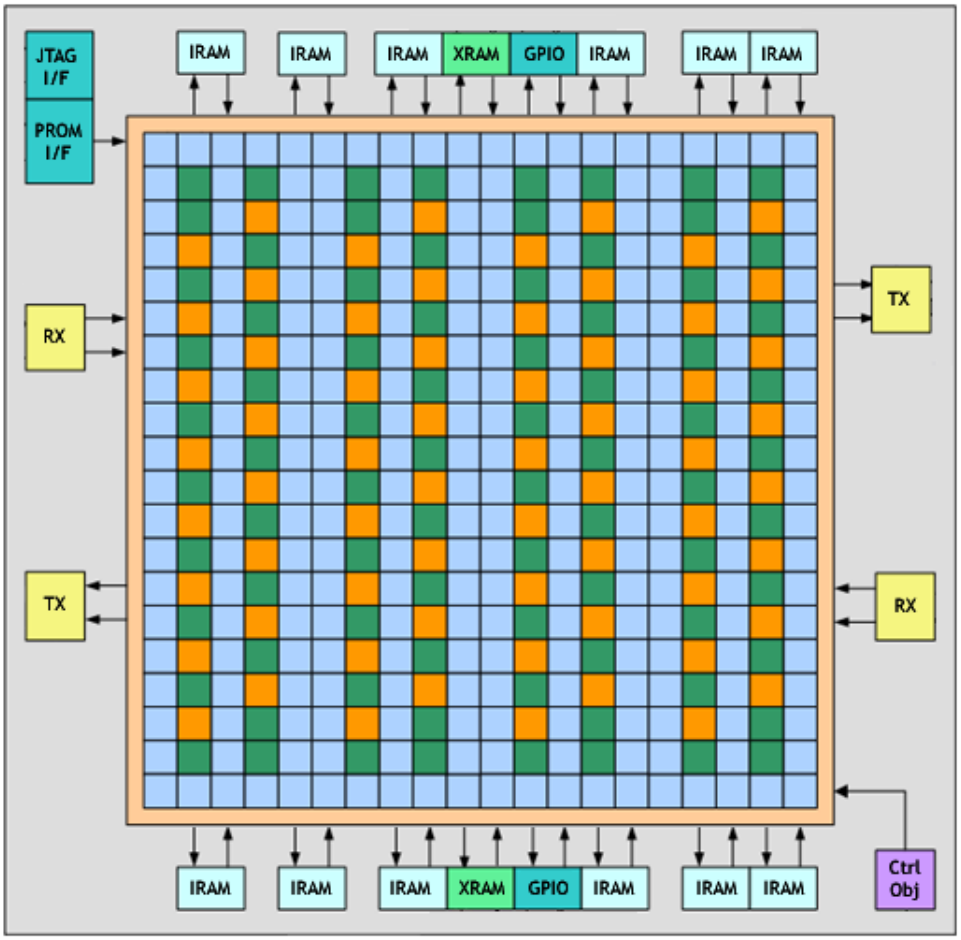
Arrix™ Family



FPOA™ Architecture Guide

Notice of Liability

Information in this document is provided in relationship with MathStar™ products. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted by this document except as provided in MathStar's Terms and Conditions of Sale for such products.

MathStar may make any changes to specifications and product descriptions at any time, without notice.

Copyright Notice

Trademarks

For customer support, visit www.mathstar.com or email us at support@mathstar.com.

Published in the United States of America.

Table of Contents

About this Guide

The Field Programmable Object Array™ (FPOA™) Architecture Guide provides a high-level description of the Arrix™ family of FPOAs at the functional block diagram level. The topics covered include the following:

- An introduction to the Field Programmable Object Array.
- A detailed description of the communication framework in the FPOA.
- A functional description of the objects within the FPOA.
- A sample implementation of an FPOA design.

Intended Audience

This guide is intended for those evaluating the FPOA architecture and its capabilities. Detailed descriptions of each object can be found in the Arrix Family Application Developer's Object Reference. Information on the external characteristics of the FPOA can be found in the Arrix Family Data Sheet.

Related Documents

The following documents provide additional information and further define the concepts covered by the FPOA Architecture Guide:

Document	Description
Arrix Family Data Sheet	This document describes the electrical and mechanical characteristics of the Arrix family FPOA, including packaging and pinout information.
Application Developer's Guide	This document uses an example application to guide you through creating an FPOA design, connecting and assigning the design, and debugging the design. It includes sample screens, typical circuits, and basic procedures based on the MathStar Design Software.
Arrix Family Application Developer's Object Reference	This document provides detailed information about each object in the Arrix family FPOA, including inputs, outputs, configuration parameters, and timing details.

Chapter 1
Introduction

Overview

MathStar™'s Field Programmable Object Array™ (FPOA™) is a breakthrough field-programmable silicon platform that supports high-speed designs. Unlike FPGAs, which implement functions at the gate level, FPOAs employ higher-order building blocks called "objects." These objects provide a much higher level of abstraction than the gates of conventional FPGAs and perform complex operations at very high clock rates.

Each Arrix™ family FPOA contains over 400 objects that are able to pass data and signals to each other through a patented, configurable communication framework. The timing of both the objects and the communication framework is fixed, operating deterministically at frequencies of up to 1 GHz. This deterministic performance eliminates the tedious timing closure design steps associated with previous silicon solutions such as FPGAs and ASICs. MathStar's FPOA allows high-level functions, algorithms, equations, and block diagrams to be quickly, directly, and efficiently realized in high-performance silicon.

The Arrix product family is comprised of six products as shown in Table 1.

Table 1 Arrix Product Family

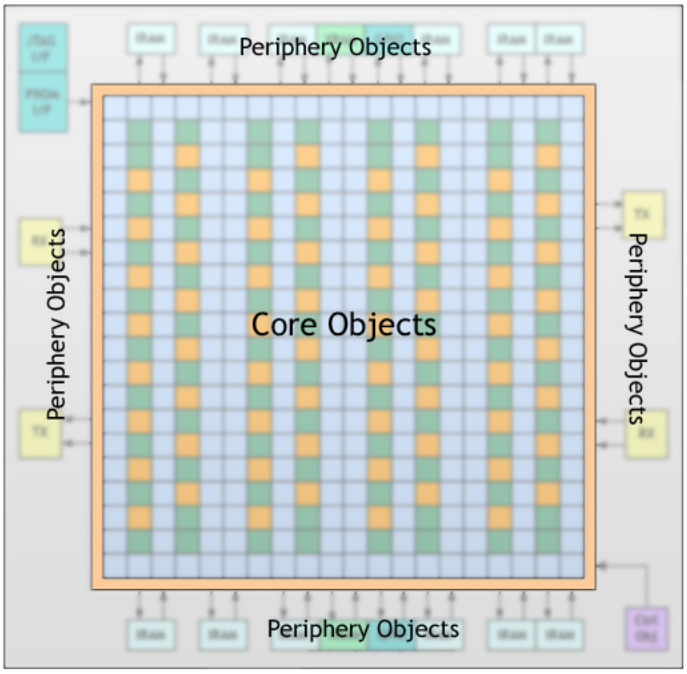
Product	Package Information	Maximum Operating Frequency
MOA2400D-10	HFCBGA-896	1 GHz
MOA2400D-08	HFCBGA-896	800 MHz
MOA2400D-04	HFCBGA-896	400 MHz
MOA2400D-10 R	HFCBGA-896, RoHS Compliant[a]	1 GHz
MOA2400D-08 R	HFCBGA-896, RoHS Compliant[a]	800 MHz
MOA2400D-04 R	HFCBGA-896, RoHS Compliant[a]	400 MHz

a. RoHS (Restriction on Hazardous Substances) compliance requires limited levels of lead (Pb), mercury, cadmium, hexavalent chromium, polybrominated biphenyl (PBB), and polybrominated diphenyl ethers (PBDE). These levels must be below thresholds proposed by the EU.

FPOA Objects

The objects of the FPOA reside in two areas: the core and the periphery. The core objects do most of the computation while the periphery objects provide additional RAM as well as move data between core objects and external devices. Figure 1-1 illustrates these two areas.

Figure 1-1: Field Programmable Object Array



Since core objects lie on a grid, they are described in terms of columns and rows. There are twenty columns and twenty rows on this grid. Some diagrams in this document (particularly those that depict core to periphery interaction) will simply refer to two columns or two rows as “0” and “1”. This is meant to distinguish two nearby columns or rows from each other and does not signify that they are the *first* two columns or rows on the grid.

Core objects communicate with each other (and with periphery objects) through the “interconnect framework.”

Interconnect Framework

There are two forms of communication within the FPOA:

- “Nearest Neighbor” communication allows a core object to communicate with any of its immediate neighbors without any clock delays.
- “Party Line” communication allows an object to communicate with objects at greater distances, or between the core and the periphery. Party line communication requires at least one clock delay.

Core Objects

Core objects perform all the computational-intensive functions for the FPOA. There are three types of core objects:

- Arithmetic Logic Unit (ALU) — This object performs logical and mathematical functions on 16-bit data, and provides general purpose truth functions for control. There are 256 ALUs in the FPOA.
- Multiply Accumulator (MAC) — This object performs 16x16 multiply operations (with a 40-bit accumulator) every clock. There are 64 MACs in the FPOA.
- Register File (RF) — This object can be programmed as RAM, as a FIFO, or as a sequential read object. It contains 64 by 20 bits (16 data bits + 4 control bits) of storage. There are 80 RFs in the FPOA.

Although all core objects operate independently, they run synchronously relative to a single clock, known as the "core clock."

Periphery Objects

Periphery objects interact with external devices and provide additional RAM for the core objects. There are five types of periphery objects:

- Internal RAM (IRAM) — Each IRAM object provides access to a single port 2k by 76-bit (4*16 data bits + 4*3 tag bits) SRAM. This SRAM can be preloaded during initialization. There are 12 IRAM objects in the periphery.
- External RAM (XRAM) — Each XRAM object provides access to 16 meg by 72-bit (4*16 data bits + 4*2 tag bits) RLDRAM-II. There are two XRAM objects in the periphery.
- General Purpose I/O (GPIO) — The GPIO object provides 48 bidirectional pins of I/O, allowing data transfer between the FPOA and external devices. There are two GPIO objects in the periphery.
- Receive (RX) Interface — The RX interface is used for parallel LVDS input to the FPOA. Each interface has a 17-bit input (16 data bits + 1 tag bit). There are two RX interfaces in the periphery.
- Transmit (TX) Interface — The TX interface is used for parallel LVDS output from the FPOA. Each interface has a 17-bit output (16 data bits + 1 tag bit). There are two TX interfaces in the periphery.

Initialization and Control

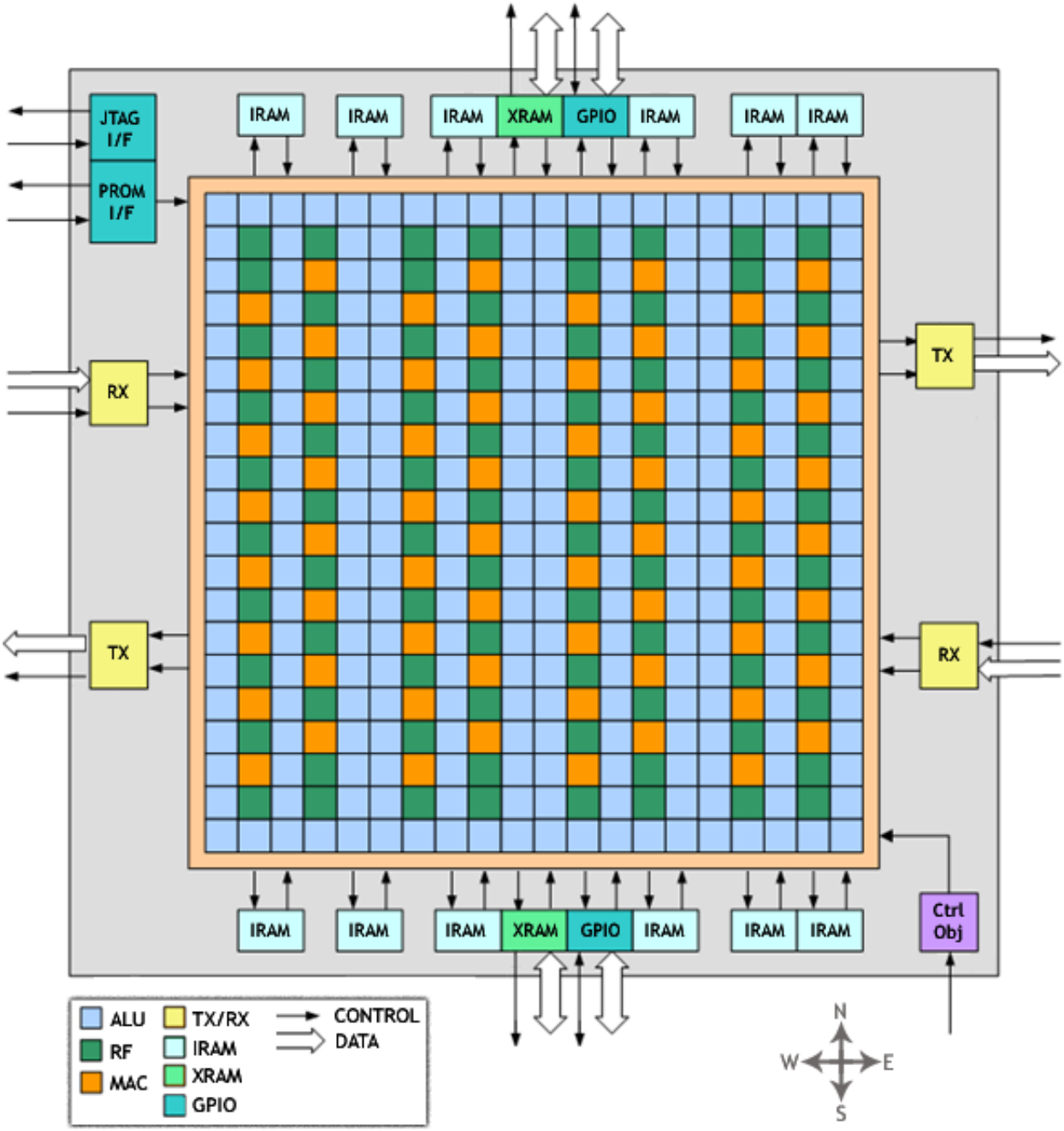
There are three interfaces involved in initialization and control:

- PROM controller — The PROM controller oversees the loading and initialization process of the FPOA. It requests data directly from an external PROM until all of the memory and configuration registers are initialized.
- JTAG controller — The JTAG controller provides an alternate way to load an FPOA configuration. It also provides a means to access memory, such as IRAM, to aid in debugging.
- Control object — This object can be used to stop the core clock. It also contains a PLL, which multiplies an external reference clock to generate the FPOA core clock. The external reference clock must be between 18.75 MHz and 37.5 MHz.

Block Diagram

The FPOA block diagram, illustrated in Figure 1-2, identifies the core objects and periphery objects discussed throughout this document.

Figure 1-2: FPOA Block Diagram



Chapter 2
Interconnect Framework

This chapter describes the interconnect framework, which is used for communication between objects in the FPOA.

Overview

The interconnect framework is a configurable mesh of connections used to transfer signals and data between objects. There are two types of connections:

- Party Line (PL) — Any object can communicate with any other object through party line communication. At 1GHz, PL communication can occur across four core objects in one clock cycle.
- Nearest Neighbor (NN) — Core objects can also communicate with each of their eight adjacent neighbors through nearest neighbor communication. There is zero latency for nearest neighbor communication.

These two types of connections, along with their associated registers and muxes, make up the interconnect framework.

Communication Channels

A communication channel is defined as a 21-bit signal comprised of 16 register data bits (R bits), 1 valid bit (V bit), and 4 control bits (C bits). Although data bits and the valid bit always travel together (and are sometimes referred to as VR bits), each C bit signal can travel independent of the VR bits and of the other C bits.

Each core object has 8 nearest neighbor channels and 10 party line channels, as shown in Figure 2-3. Data can travel in both directions at the same time. Periphery objects can communicate via the PL channel; however, they cannot use NN communication.

Figure 2-4 depicts PL and NN communication channels within the interconnect framework.

Figure 2-3: Core Object Communication Channels

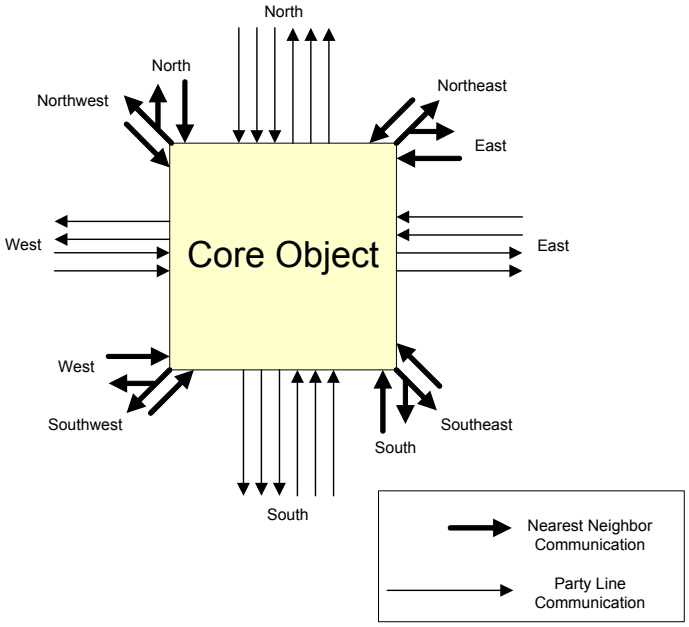
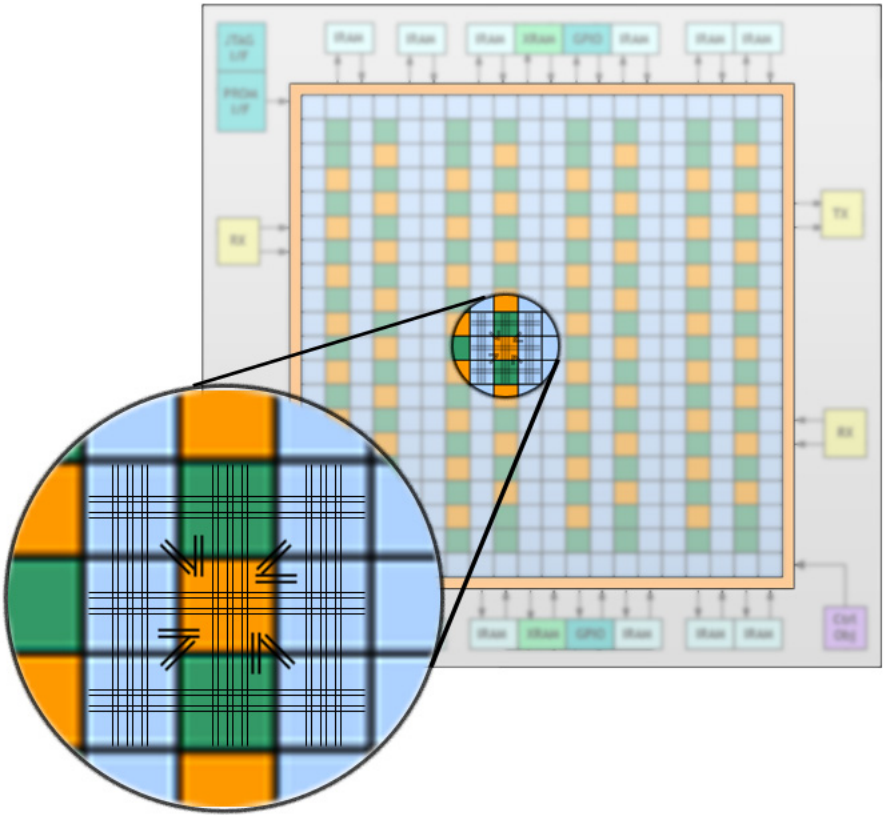


Figure 2-4: Interconnect Framework



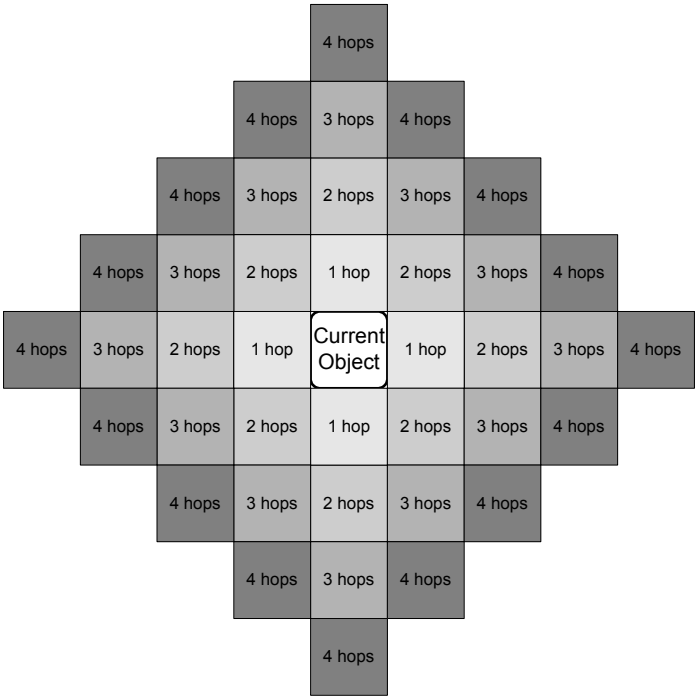
Party Line Overview

Party lines (PL) are register-to-register connections. PLs transfer data and control from one object to any other object. Distance is measured in "hops" where a hop is the movement from one object to an adjacent object that lies N, S, E, or W. (Note that diagonal movements require two hops, as shown below in Figure 2-5.) Data can travel a fixed number of hops per clock cycle. After this fixed number of hops, data must be reclocked[1] before going onto the next object. At 1GHz, core object data can move four hops in one clock cycle (with one exception specified in "Party Line Select" on page 22). Communication with periphery objects is slower and travels one hop less per clock cycle.

Slower core clock settings allow more hops between reclocking. However, a design that takes advantage of this will not be able to run if the core clock speed is increased.

Figure 2-5 illustrates all core objects that are accessible via PL communication in one clock cycle at 1 GHz.

Figure 2-5: Four Hop PL Communication



1. 'Reclocking' requires data to land in a register and then be relaunched. See "Launching and Landing Behavior" on page 20 for more information.

Table 2-1 shows the maximum number of hops per clock cycle for each Arrix product at various operating frequencies.

Table 2-1 Hop Count Table

Operating Frequency	Arrix Family Model Number		
	MOA2400D-10 (1 GHz max)	MOA2400D-08 (800 MHz max)	MOA2400D-04 (400 MHz max)
1 GHz	4 hops (3)	N/A	N/A
800 MHz	5 hops (4)	4 hops (3)	N/A
600 MHz	8 hops (7)	6 hops (5)	N/A
400 MHz	14 hops (13)	10 hops (9)	8 hops (7)
200 MHz	32 hops (31)	24 hops (23)	18 hops (17)

Notes:

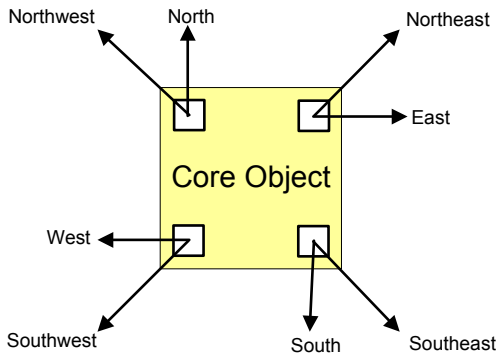
1. Values in parentheses ( ) denote the number of hops when crossing the core/periphery boundary.
2. Refer to "Party Line Select" on page 22 for a condition that can reduce these hop count values by one.

Each object in the FPOA has 10 party line channels: 3 going north, 3 going south, 2 going east, and 2 going west. A core object has five registers known as “party line launch/land” registers. Since there are twice as many PL channels as there are PL launch/land registers, each register is shared by two PL channels. Party line communication is discussed further in "Party Line Communication Details" on page 19.

Nearest Neighbor Overview

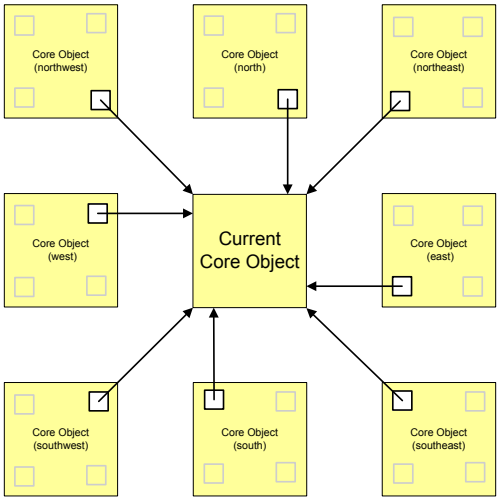
Nearest Neighbor (NN) communication provides a direct connection between physically adjacent core objects without any latency. Each core object has four NN registers. Each register can be used as inputs to two adjacent core objects as shown in Figure 2-6.

Figure 2-6: Local Nearest Neighbor Registers



A core object can also read the value from its eight adjacent NN registers without any latency. Figure 2-7 shows which NN registers can be used as inputs to a core object.

Figure 2-7: Nearest Neighbor Input



Source and Result Registers

Core objects use PL launch/land registers and NN registers as working registers for their internal functions. Inputs to the internal functions can be acquired from any of 19 “Source Registers.” These include:

- 4 local NN registers
- 5 PL launch/land registers
- 8 adjacent NN registers (located in the eight adjacent core objects)
- 2 local constant registers (which are programmed during initialization) — these are sometimes referred to as K0 and K1

Results from these internal functions can be saved to a set of fewer registers, called “Result Registers.” These include:

- 4 local NN registers
- 5 PL launch/land registers

Input Select Muxes

There are three muxes (named A, B, and M) used to select interconnect framework signals as inputs to internal functions. These signals can come from any of the source registers described in the previous section. Each type of core object uses these muxes differently. For example, the ALU object uses the A mux to select the A operand, the B mux to select the B operand, and the M mux to select the mask operand. In the MAC object, the M mux is used to load the accumulator.

Signal and Register Notation

Party Line Signals

When describing a party line signal, the following naming convention is used:

PL _ {direction of travel} _ {channel group number} _ {bit type}_{in/out}

For example, the data bits (R bits) on the north, outbound, party line 1 channel would be described as follows:

PL_N_1_R_OUT

Note that the “bit type” and the “in/out” direction are not always necessary when describing a party line signal. In addition, when the direction of the signal is apparent, the signal is sometimes described just by its numeric value: PL1, PL2, PL3.

Party Line Registers

When describing a PL launch/land register, it is labeled in terms of the direction of the two PL channels which share that particular register. For example, the launch/land register shared by PL_N_1 and PL_S_1 is described as “PL_NS_1.”

Nearest Neighbor Registers

Since it is important to distinguish the local NN registers (used for input and output) from the NN registers of the adjacent core objects (used for input only), the following convention is used to describe their direction:

- Each of the four local NN registers is defined by its two output directions: NNW (North / Northwest), ENE (East / Northeast), SSE (South / Southeast), WSW (West / Southwest).
- Each of the eight neighboring NN registers, used for input only, is defined by its direction: NW, N, NE, E, SE, S, SW, W.

When describing a nearest neighbor register, the following naming convention is used:

NN_{direction}_{bit type}

For example, the data bits (R bits) stored in the local North / Northwest register would be defined as follows:

NN_NNW_R

Whereas the data bits stored in the register that lies northeast of the current object would be defined as follows:

NN_NE_R

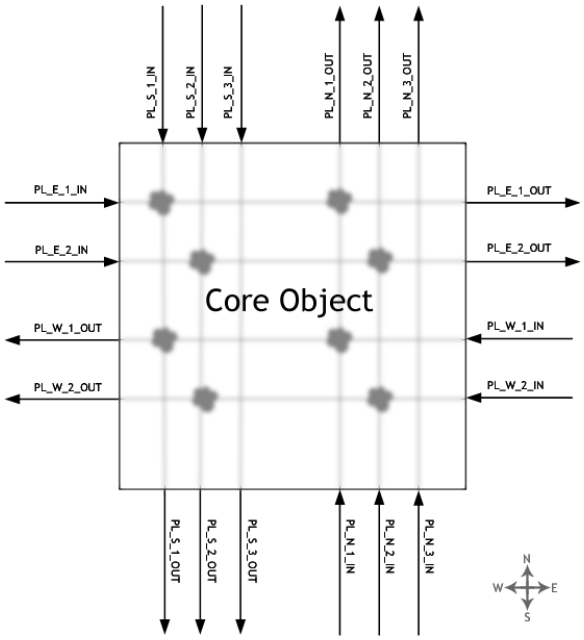
Party Line Communication Details

PL Channel Groups

Party line channels are organized into three channel groups. Groups 1 and 2 each contain channels moving north, south, east, and west. Group 3 only contains a channel moving north and south. PL channels can change directions within a particular group. When the maximum hop count has been reached, the channel must “land” and then “launch” as described in the next section.

Figure 2-8 illustrates the PL communication channels among the three groups (1, 2, and 3).

Figure 2-8: Party Line Communication Channels

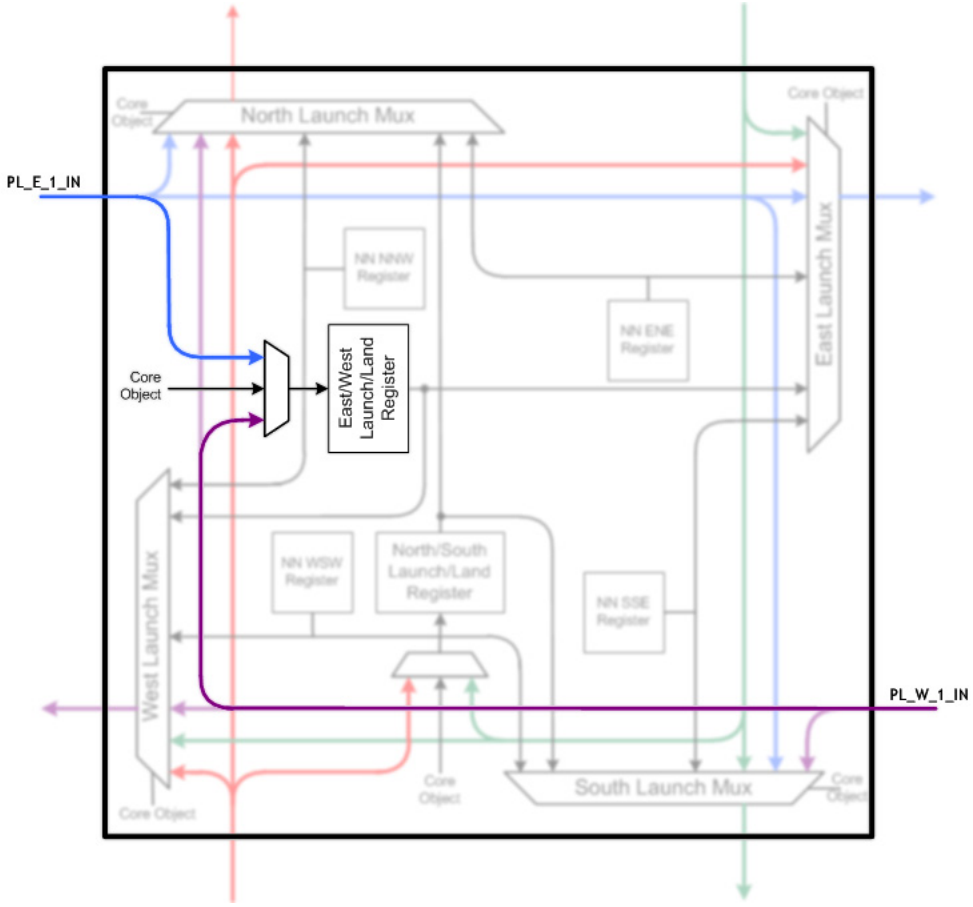


Launching and Landing Behavior

Landing

Each core object has five registers used for party line communication. These registers are called “party line launch/land” registers. A source object uses these registers to place information on the PL communication channel (“launch”) and a destination object uses these registers to receive information from the PL channel (“land”). Since there are ten party lines and only five PL registers per object, each register is shared between the two party lines traveling in opposite directions along the same channel group. Figure 2-9 illustrates the register that is shared between PL_E_1 and PL_W_1.

Figure 2-9: Party Line Launch/Land Register



The pin labeled “Core Object” in the above diagram denotes that the core object can also use the launch/land register to store data.

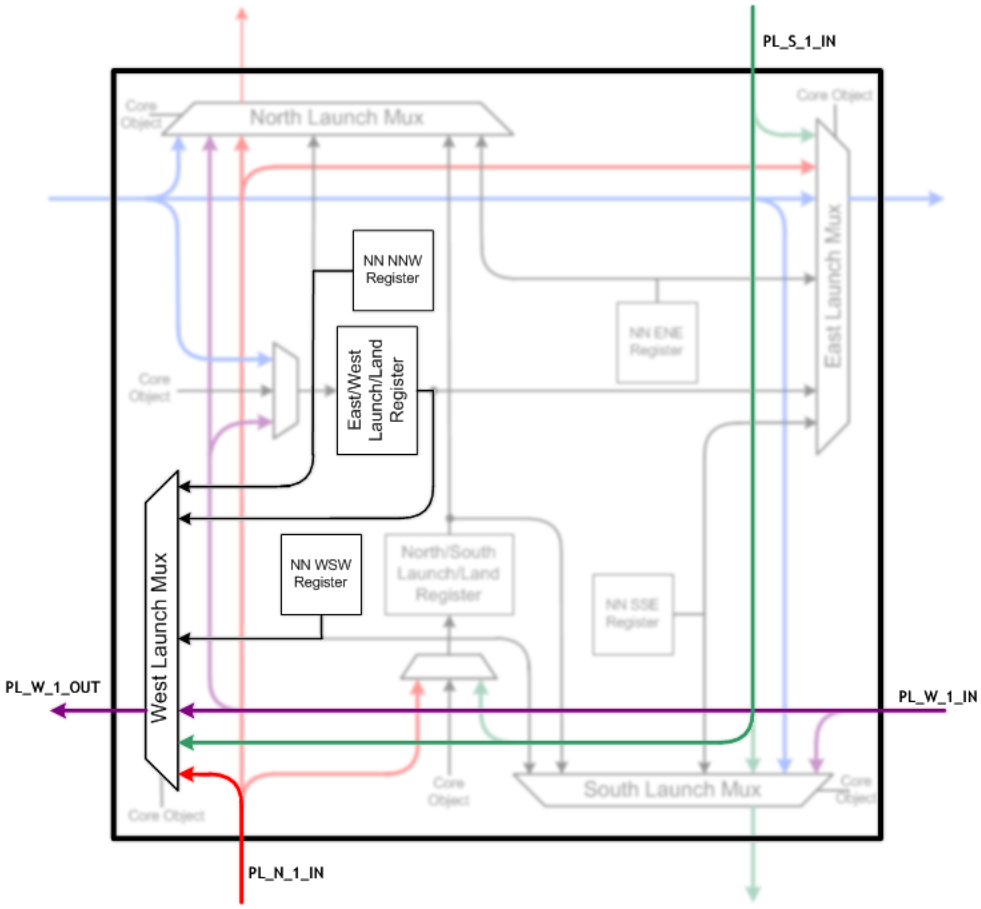
Launching

A dedicated “launch mux” launches data onto each party line. The launch mux can launch data from one of the following sources:

- A party line launch/land register
- Two nearest neighbor registers
- Up to three party line input signals

Figure 2-10 illustrates the launch mux that is dedicated to PL_W_1_OUT.

Figure 2-10: Party Line Launch MUX



Although only one mux is shown in this diagram, there are five launch muxes per party line channel: one for the data bits (VR bits) and one for each of the control bits (C bits). In total, there are 50 launch muxes per object.

Party Line Select

Every launch mux has a PL select pin as highlighted in Figure 2-11. The PL select pin controls which of the input signals is used as the output signal. This pin can be set statically at initialization time or controlled dynamically at runtime. If the PL select is changed at runtime in order to launch data from one of the local registers, then a one hop penalty is incurred (see Table 2-2). In other words, a signal travels one less hop during a clock cycle when it is launched under these circumstances. If the PL select is changed dynamically within objects that are not launching the data, then no penalty is imposed.

Figure 2-11: Party Line Select Pin

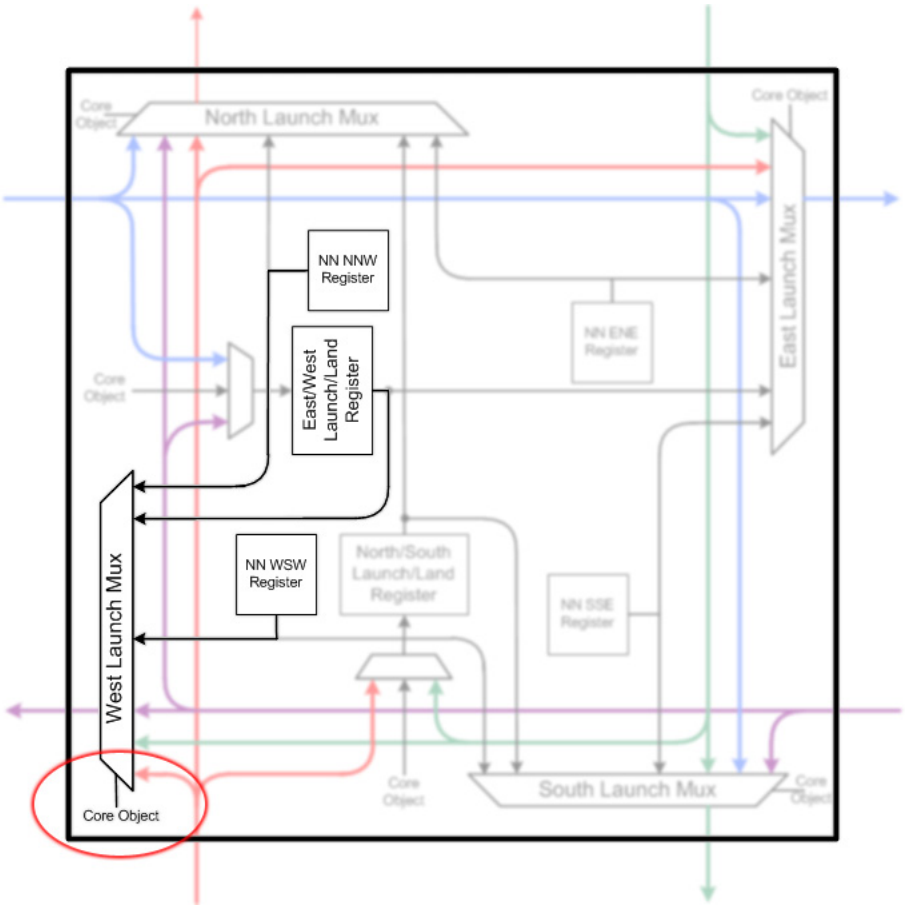


Table 2-2 shows the maximum number of hops per clock cycle when data is first launched using the PL select feature at runtime.

Table 2-2 Hop Count Table – Assuming Party Line Select in First Hop

Operating Frequency	Arrix Family Model Number		
	MOA2400D-10 (1 GHz max)	MOA2400D-08 (800 MHz max)	MOA2400D-04 (400 MHz max)
1 GHz	3 hops (2)	N/A	N/A
800 MHz	4 hops (3)	3 hops (2)	N/A
600 MHz	7 hops (6)	5 hops (4)	N/A
400 MHz	13 hops (12)	9 hops (8)	7 hops (6)
200 MHz	31 hops (30)	23 hops (22)	17 hops (16)

Notes:

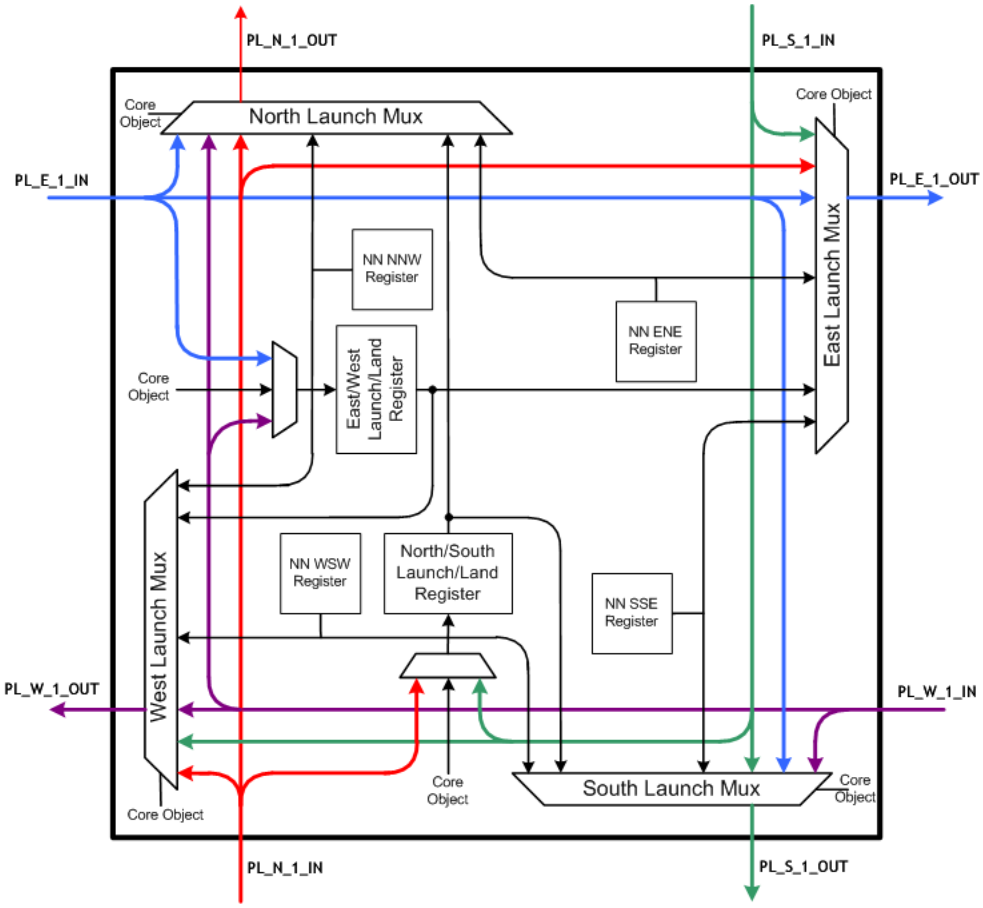
1. Values in parentheses ( ) denote the number of hops when crossing the core/periphery boundary.
2. These values are all one hop less than the standard number of hop counts, which are specified in Table 2-1 on page 16.

Note that the software tools handle this party line select functionality through a virtual mux called “SO_MUX.” The architecture itself does not contain a multiplexer for this purpose.

Party Line 1 and 2

Figure 2-12 illustrates the registers and muxes associated with party line 1 communication. Note that this diagram combines the previous three diagrams, and then adds the launch/land mechanism for the three other channels in this group.

Figure 2-12: Party Line 1

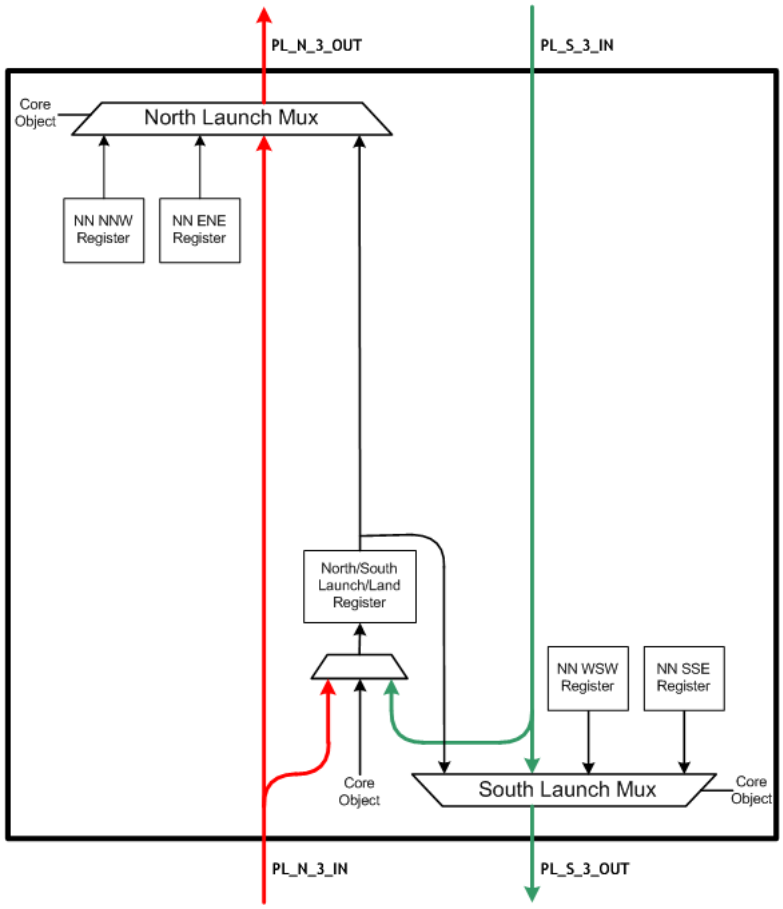


Party line 2 has the same layout as party line 1.

Party Line 3

Unlike party line 1 and 2, party line 3 only contains channels traveling north and south (i.e. there is no turn option). Figure 2-13 illustrates the layout for party line 3.

Figure 2-13: Party Line 3



Changing Party Line Channel Groups

A signal can switch from one party line group to another by landing in a launch/land register. The core object can then be programmed to move this data to another group's launch/land register.

Chapter 3
Arithmetic Logic Unit Object

This chapter describes the Arithmetic Logic Unit (ALU) core object.

Additional details can be found in the Application Developer's Object Reference.

Overview

The ALU is a programmable, multi-state core object that provides a general-purpose 16-bit arithmetic logic block for data operations, and four general-purpose truth functions for control bit operations. Each Arrix product provides 256 ALUs in the FPOA core.

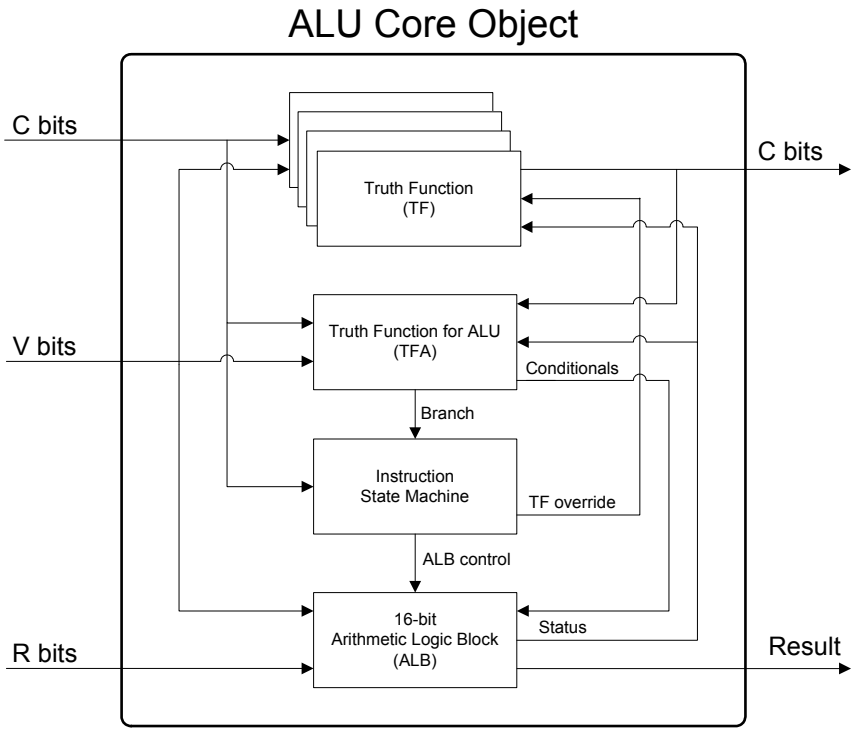
Functional Description

The ALU object is made up of four components:

- Arithmetic Logic Block (ALB) — The ALB performs a wide range of arithmetic, logical, and multiplexing functions. The ALB operates on R bits and is controlled through an instruction state machine.
- Instruction State Machine — The ALU contains an eight state instruction machine, which executes in one core clock cycle. Each state supplies the ALB input selection, an ALB operation code (opcode), and result destination information. Each state also can override truth function operations.
- Truth Functions (TFs) — The four general-purpose truth functions operate on and generate control bits. Each function can have up to four inputs that index a 16-bit truth table. Each truth table is initialized to reflect an application-defined boolean expression of the inputs. The truth table is indexed every core clock—based on the input values—to generate a single bit result. The result of the operation can be passed to other objects.
- Truth Function for the ALB (TFA) — An additional truth function provides control signals to the ALU's instruction state machine. These controls allow dynamic instruction flow control (next state determination), can override ALU instruction operation, and can block ALU result writes to destination registers.

Figure 3-14 shows these four components and the connections between them. Each component is further described in this chapter.

Figure 3-14: ALU Block Diagram

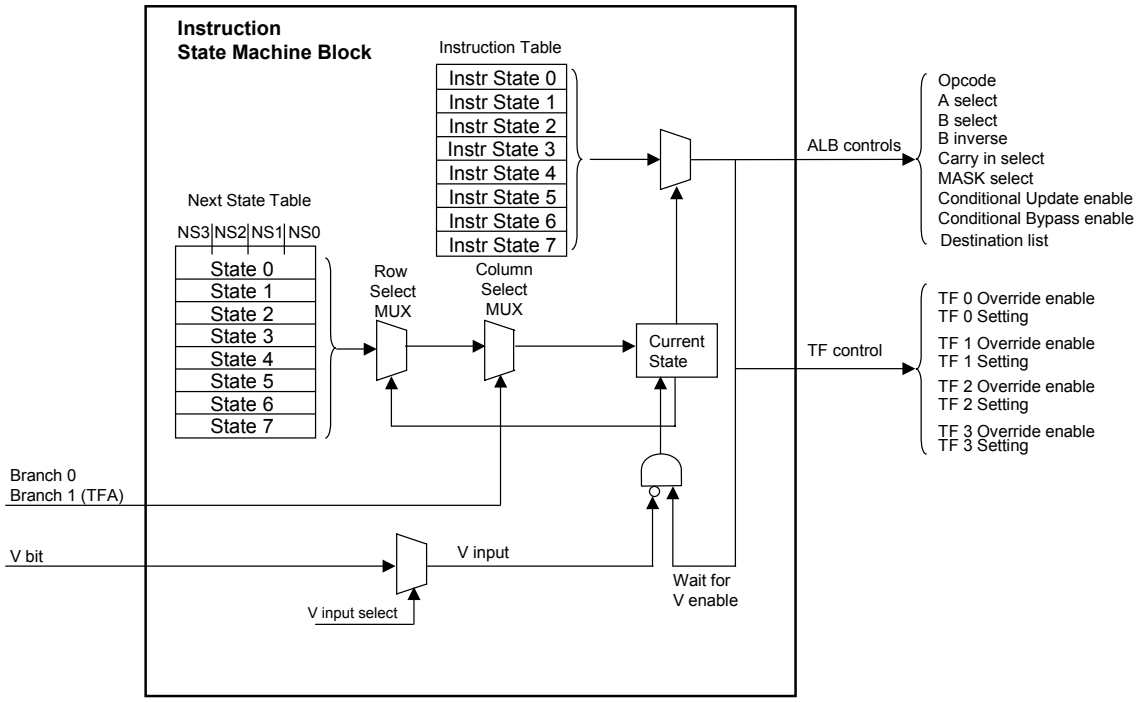


Instruction State Machine

The ALU instruction state machine provides eight programmable states. Each state executes in one core clock and provides controls for the ALB and TF functions. The TFA function provides control for the instruction state machine.

Figure 3-15 shows the instruction state machine.

Figure 3-15: Instruction State Machine



Output for the state machine includes ALB controls. The controls for the ALB include the A and B operand selection, carry in selection, and MASK input selection from the interconnect framework as described in Chapter 2. Each state also defines the ALB operation (opcode) and to which destination registers the result should be written. The default programming in each state is a NOP. Every instruction supports writing 0, 1, or many (up to all) destination registers with the result.

Each state can override the TF results. (See “TF Description” on page 35 for an explanation of how the TF generates results.) All four TF results can be overridden in each state. This feature allows control bits to be generated solely from the instruction state machine or a combination of the TF and the state machine.

Each state can optionally enable the use of the "conditional bypass” and the “conditional update” signals from the TFA function in the ALB. (See “TFA Description” below for detail on how these two bits are generated.) If a state enables the “conditional bypass” signal and the TFA asserts this signal, then the ALB instruction is cancelled and the result is set to the selected A input. If a state enables the “conditional update” signal and the TFA asserts this signal, then the ALB result will not be written to the destination registers.

A four-entry "next state" table is defined for each state that is used for branching. This table is indexed with two bits that are generated in the TFA function (branch 0 and branch 1) as described below in “TFA Description.”

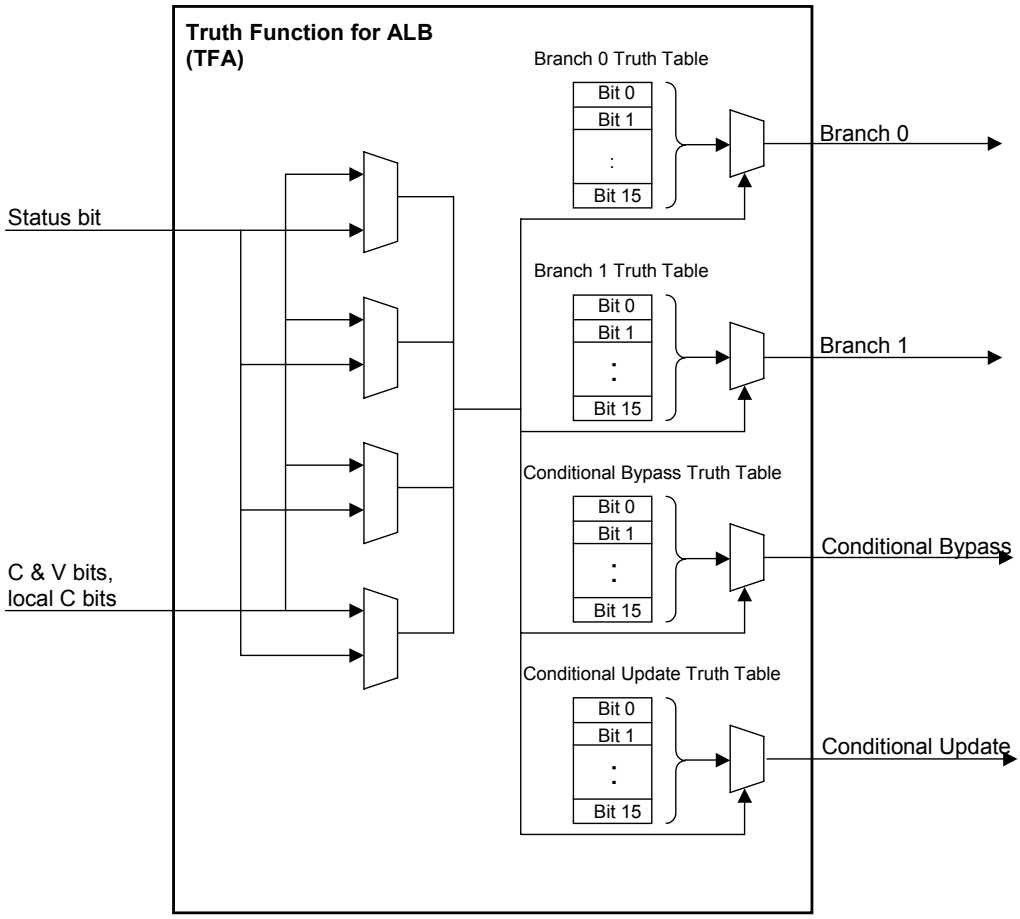
Each state can be programmed to wait for a selected V bit. When "wait for V" is enabled in a state and the selected V bit is de-asserted, then state advancement is inhibited. However, all other ALU operations generate results.

Each ALU object also has a programmable “warm reset” signal which will force the state machine to state 0.

TFA Description

The instruction state machine is controlled using the TFA. The TFA has four inputs that are selected from among the interconnect framework's C and V bits, and the ALB status bit. The four selected inputs are used to index four 16-bit truth tables. Each truth table represents a boolean expression of the inputs that generates the TFA outputs: branch 0, branch 1, “conditional bypass,” and “conditional update.” Figure 3-16 shows a block diagram of the TFA.

Figure 3-16: Truth Function for the ALB



The truth tables are set to reflect a boolean logic expression using up to four inputs. All four truth tables share the same four selected inputs. The TFA generates a result for all four outputs every core clock.

The results from the two branch truth tables are used to index the instruction state machine's current state four-entry “next state” table. This table is always indexed with the results of branch 0 and branch 1 functions to determine the next state.

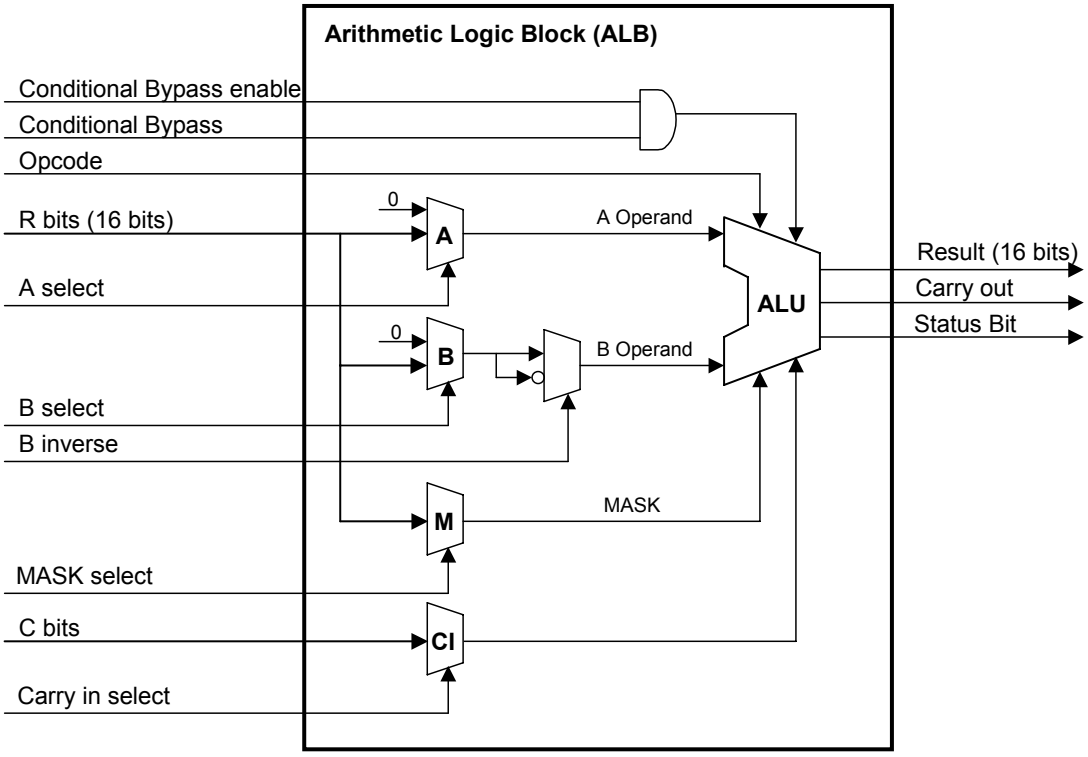
The “conditional bypass,” when enabled in the current state, overrides the current state's ALB instruction and passes the A operand through as the result.

The “conditional update,” when enabled in the current state, blocks the ALB's 16-bit result write to the destination registers. “Conditional update” does not block TF outputs.

ALB Description

The ALB provides general purpose arithmetic and logic operations on 16-bit data. All operations complete in one clock cycle. Figure 3-17 shows a block diagram of the ALB.

Figure 3-17: ALB Block Diagram



The four inputs to the ALB are selected from the interconnect framework: 16-bit A & B operands, a 16-bit mask (M), and a single bit carry in (CI). Additionally, the A and B input select muxes include a constant 0 value. The B operand also has an option to invert the selected input (allowing for a constant 1 value in additional to the values of the other source registers). The instruction state machine defines these selections for each state.

Each state in the instruction state machine provides the ALB opcode, the destination register selection, the “conditional bypass enable,” and the “conditional update enable” signals. The TFA provides the “conditional bypass” and “conditional update” signals. The ALU performs a logical *and* on the enable signals from the instruction state machine with the TFA signals. The “conditional bypass” signal will override the ALB opcode and pass the A operand through as the result. The “conditional update” signal blocks the write of the result to the selected destination registers.

The instruction state machine defines a selection of destination registers for each state. This selection can be empty, in which case the result of the operation is not saved in any register.

Along with the 16-bit result, the ALB generates a carry out bit and status bit. The status bit can be used in the TFA and TF functions. The carry out can be passed to other core objects through the interconnect framework. This allows multiple ALUs to be cascaded together for higher precision operations.

Table 3-3 shows all available ALB operations (opcodes).

Table 3-3 ALB Instruction Set

Instruction	Inputs	Carry Out (CO)	Status	Result [0:15]
ADD	A, B	Result[16]	signed overflow	A + B
AND	A, B [,M]	Retain previous CO	result word ≠ 0	(A & B) & M
AVG	A, B	Retain previous CO	Result[15]	(A + B) / 2 [a]
BMUX	A, B, M	Retain previous CO	result word ≠ 0	(A & M) | (B & ~M)
CMP	A, B	Result[16]	signed overflow	A – B
DEC	A	Result[16]	A == 0x8000	A – 1
DSHL	A, B, M	Retain previous CO	Result[16]	{A<<M[3:0], B>>(16–M[3:0])}
DSHLI	A, B, #K	Retain previous CO	Result[16]	{A<<K, B>>(16–K)}
DSHRI	A, B, #K	Retain previous CO	Result[16]	{A<<(16–K), B>>K}
HADD	A, B, CI	Result[16]	signed overflow	A + B + CI
HAVG	A, B, CI	Retain previous CO	Result[15]	(A + B + CI) / 2 [a]
HCMP	A, B, CI	Result[16]	signed overflow	A – B + CI
HDEC	A, CI	Result[16]	A == 0x8000 && C1 == 0	A – 1 + CI
HINC	A, CI	Result[16]	A == 0x7FFF && C1 == 1	A + CI
HNEG	B, CI	Result[16]	B == 0x8000 && C1 == 1	–B + CI
HSUB	A, B, CI	Result[16][b]	signed overflow	A – B + CI
HXORADD	A, B, CI, M	Result[16]	A ≠ M	A + B + CI
HXORAVG	A, B, CI, M	Retain previous CO	A ≠ M	(A + B + CI) / 2 [a]

Table 3-3 ALB Instruction Set (Continued)

Instruction	Inputs	Carry Out (CO)	Status	Result [0:15]
HXORCMP	A, B, CI, M	Result[16]	A ≠ M	A - B + CI
INC	A	Result[16]	A == 0x7FFF	A + 1
INV	B	Retain previous CO	result word ≠ 0 (B ≠ 0xFFFF)	~B
MOV	A	Retain previous CO	A ≠ 0	A
MUX	A, B (,tfa.bypass[c])	Retain previous CO	B ≠ 0	tfa.bypass? A: B
NEG	B	Result[16]	B == 0x8000	–B
OR	A, B [,M]	Retain previous CO	result word ≠ 0	(A | B) & M
ROL	A, M	Retain previous CO	Result[16]	{A<<M[3:0], A>>(16–M[3:0])}
ROLI	A, #K	Retain previous CO	Result[16]	{A<<K, A>>(16–K)}
RORI	A, #K	Retain previous CO	Result[16]	{A<<(16–K), A>>K}
SETSB		Retain previous CO	1	0xFFFF
SHL	A, M	Retain previous CO	Result[16]	A << M[3:0]
SHLI	A, #K	Retain previous CO	Result[16]	A << K
SHRI	B, #K	Retain previous CO	0	B >> K
SUB	A, B	Result[16][b]	signed overflow	A – B
XOR	A, B [,M]	Retain previous CO	result word ≠ 0	(A ^ B) & M
XORADD	A, B, M	Result[16]	A ≠ M	A + B
XORAVG	A, B, M	Retain previous CO	A ≠ M	(A + B) / 2 [a]
XORCMP	A, B, M	Result[16]	A ≠ M	A - B
ZERO		Retain previous CO	0	0

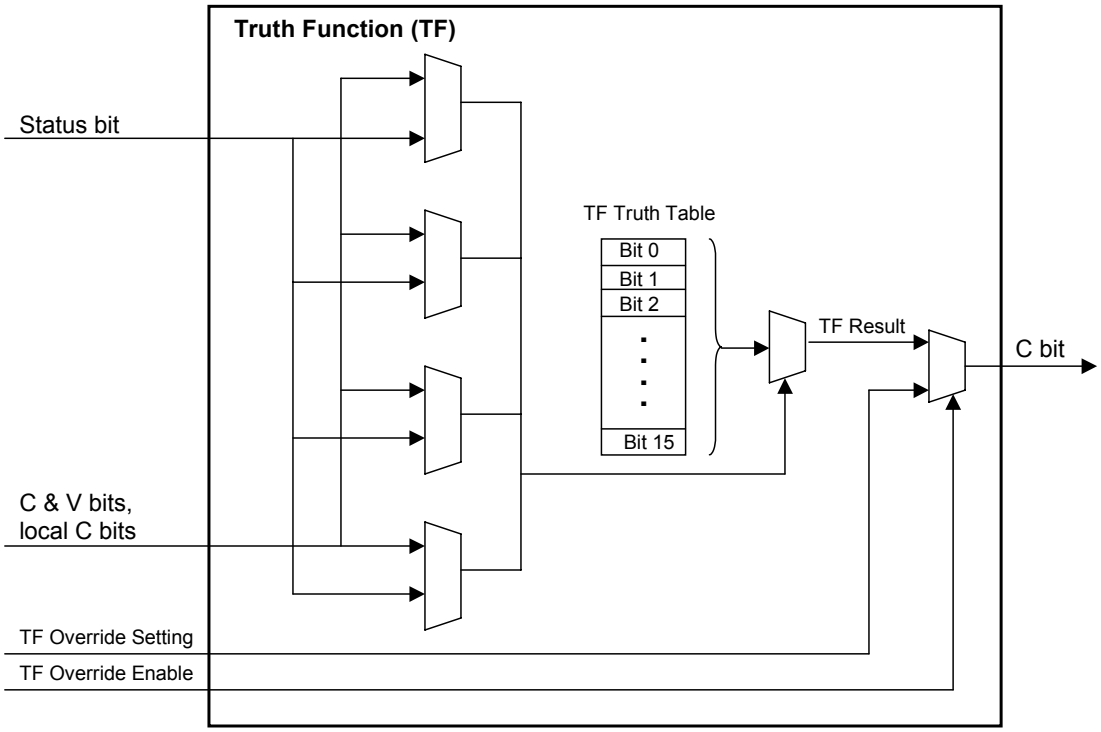
a. Result is sign-extended.
b. If B == 0, then Carry Out = 1; otherwise Carry Out = Result[16]. See “CMP” (or “HCMP”) for a similar instruction with a different carry out value.
c. See "TFA Description" on page 30 for information about the TFA’s conditional bypass signal.

TF Description

The ALU object contains four separate general-purpose truth functions (TFs). Each TF is driven by up to four inputs selected from the C and V bits (available via the interconnect framework), as well as the ALB status bit. These four inputs index a 16-bit programmable truth table. The contents of each truth table are determined by a programmable boolean expression of up to four inputs. A new output is generated from each TF every core clock.

Figure 3-18 shows a block diagram of the TF.

Figure 3-18: TF Block Diagram



Each state in the instruction state machine can override the result from the truth tables individually. This allows the C bit register to be set solely from the instruction state machine, or in combination with a truth function.

The output is saved in local C bit registers and can be sent to other core objects through the interconnect framework. These outputs are also available as selectable inputs to the TF and TFA.

Initialization

The ALU core object initialization allows presetting the four local nearest neighbor register and the four local C bit registers. The ALU's instruction state machine always initializes to state 0. The ALB carry out and status bit are initialized to 0.

Similar to initialization, when warm reset is asserted, the instruction state machine, ALB status bit, and carry out values are all forced to their initial values (0). The four local nearest neighbor registers and four local C bit registers can optionally be reloaded with the initial values. Note that after initialization or warm reset, party lines registers are always forced to 0.

Chapter 4
Multiply Accumulator Object

This chapter describes the Multiply Accumulator (MAC) core object.

Additional details can be found in the Application Developer's Object Reference.

Overview

The MAC performs multiply and accumulate functions. The multiplier function multiplies two 16-bit inputs and generates a 32-bit result plus carry. The accumulator function adds a 32-bit input to an existing number and, depending on the configuration, provides a 40-bit output result. Each Arrix product provides 64 MACs in the FPOA core.

Functional Description

Figure 4-19 illustrates the Multiply Accumulator (MAC) core object comprised of the following functional elements:

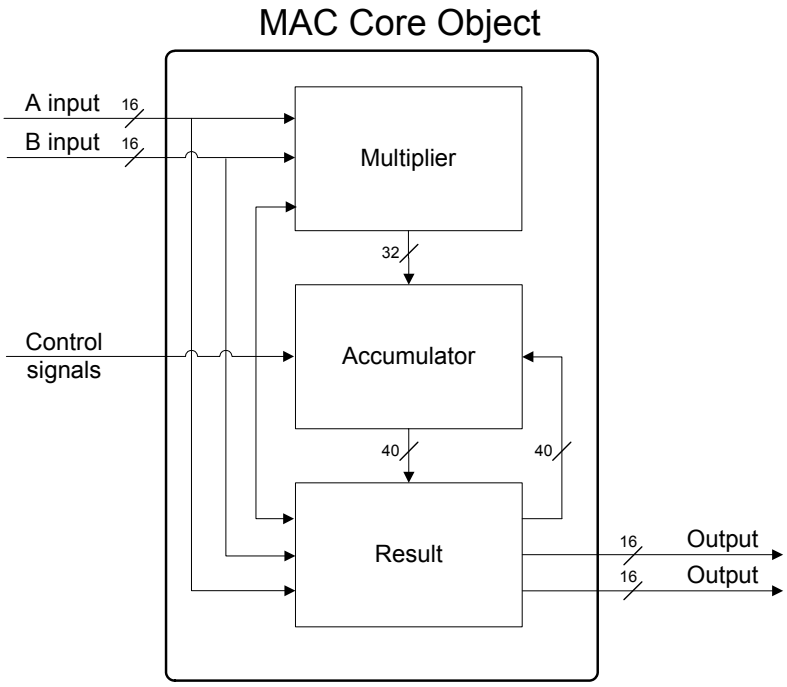
- Input mapping
- Multiplier function
- Accumulator function
- Result/output mapping

The MAC selects inputs from a range of possible interconnect framework source registers.The MAC provides a 40-bit result which is available to a range of possible result registers.

The MAC has a variety of control signals, some are statically configured and some are delivered on a dynamic, clock-by-clock basis. The dynamic control signals must be delivered to the MAC in advance of the input data. This requires some advance planning by the design engineer.

The multiplier function requires two clock cycles to complete and the accumulator function requires one clock cycle to complete. However, the MAC is fully pipelined so that every FPOA clock cycle, the MAC accepts new inputs and generates a new result.

Figure 4-19: MAC Block Diagram



Input Mapping

The MAC selects its input from any of the possible source registers discussed in "Source and Result Registers" on page 17. Each input data source may be treated as signed 16-bit two's complement integers, unsigned integers, or signed fixed fractional numbers.

Control bits (C bits) are used to dynamically control the operational mode of the multiplier and accumulator. These bits are mapped at configuration time onto the various inputs and can be used to control the operational mode of the MAC on a clock-by-clock basis. These operational modes are discussed in more detail later in the chapter.

In the MAC, the valid bit (V bit) is ignored as an input.

Multiplier Function

The multiplier generates a 32-bit product from the two 16-bit A and B operands. The MAC treats the two operands and the result as signed or unsigned depending on the operational mode. The product can be treated as signed fixed fractional numbers (Q15) if both operands are signed and the respective control signal is set to “Q15 format.” This particular mode shifts the current product left by one, padding the least significant bit of the product with a 0 bit.

The multiplier function can receive input and produce output on a clock-by-clock basis.

Accumulator Function

The accumulator adds a 32-bit input to an existing number and can provide a 40-bit output result. The accumulator function can be configured to perform three basic functions on a clock-by-clock basis:

- Load the output value of the multiply function into the accumulator. This will erase any previous value stored in the accumulator.
- Preload the accumulator with a specified value.
- Add the output value of the multiply function to the current value stored in the accumulator. At least 255 products can be accumulated without any threat of overflow or underflow. Accumulator bypass can be implemented by adding 0 to the multiplier output.

These operations are controlled by three control signals: enable, preload, and bypass. These signals are mapped into the control bits (C bits) in a method similar to the multiplier function.

The accumulator output can be configured to perform a rounding function, effectively rounding the lowest 16 bits either up or down. Logically, this is equivalent to adding 0x8000 to the result.

Output mapping

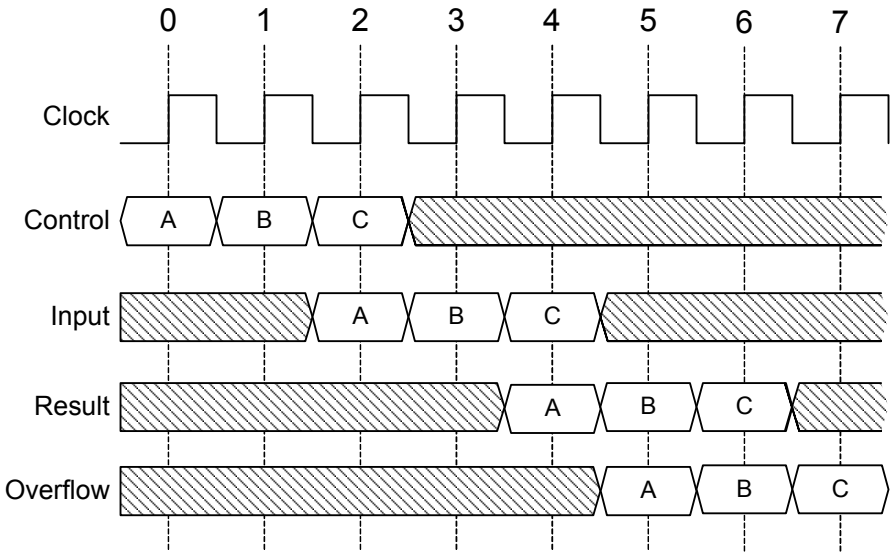
The MAC output is 40 bits and therefore requires specific control to map it to the available 16-bit output registers. The MAC allows for two of four 16-bit fields ([39:24], [33:18], [31:16], [15:0]) out of the 40-bit accumulator to be mapped into the possible output registers.

The output mapping is chosen dynamically by the control bits (C bits). Each configuration chooses to update any combination of the seven output registers.

MAC Timing and Control

Control signals for the MAC core object are unique in that they must arrive two clocks ahead of the data. This does not prevent pipelining at the full clock rate but does require some advance planning of the data and control flow through the MAC. Figure 4-20 illustrates MAC control and data timing.

Figure 4-20: MAC Control and Data Timing



The multiply and accumulate operational mode selection, MAC input mapping, and MAC result mapping control bits must be available two clocks ahead of when the 16 bit data operands arrive in their respective registers. These control signals are mapped to the control bits (C bits).

The valid bit (V bit) is used as an output to signify an underrun or overflow error in the accumulator function. Saturation can be achieved with the assistance of the ALU.

Chapter 5
Register File Object

This chapter describes the Register File (RF) core object.

Additional details can be found in the Application Developer's Object Reference.

Overview

The RF object contains 64 memory locations of 20 bits each (16 data bits + 4 control bits). The RF object supports three operating modes (RAM mode, FIFO mode, and Read Sequence mode), which are discussed in this chapter. Each Arrix product provides 80 RFs in the FPOA core.

The Register File can be configured to 20 bit single-width or 40 bit double-width read and write data paths. All modes support simultaneous read and write every clock cycle. When simultaneously writing and reading the same address, the newest data is returned.

The Register File content can be preloaded during FPOA initialization. Initial content can only be restored by reloading the chip.

RAM Mode

In RAM mode (also known as Register File mode), the RF object functions as dual-port Random Access Memory (RAM). Write requests are accepted on every clock cycle. Optional byte enable inputs indicate whether to update the low byte [7:0] of the R bits, the high byte [15:8] of the R bits, or the four tag bits. If a particular field is not written, the previous content is preserved. In double-width mode, three additional byte enables are associated with the second entry's data bytes and four tag bits.

RAM mode supports simultaneous read and write of the same address location. Read requests are accepted every clock cycle with a two clock cycle read latency.

The width of the read and write interfaces can be independently programmed to single-width or double-width interfaces. The read and write addresses presented to the RF object are always interpreted as a word address (range 0 to 63). In double-width mode only even addresses are used.

FIFO Mode

In FIFO mode the RF object manages a fixed 64-word circular buffer with two programmable watermark settings. FIFO mode supports a flush command and provides status outputs for the following states: empty, overflow or underflow error, and the two watermark levels.

Read and write interfaces can be programmed to either 20-bit or 40-bit width, but both interfaces must be programmed to the same width. Read data is prefetched, yielding an effective read latency of one clock cycle. Simultaneous reads and writes are supported.

Read Sequence Mode

In Read Sequence mode, the RF object operates in FIFO mode for read operations, and RAM mode for write operations. This mode allows the RF object to circularly read a sequence of a programmable length while accepting random address writes. Since write operations in Read Sequence mode behave similarly to RAM write operations, FIFO status flags have no meaning in this mode. Read data is not prefetched in this mode and thus has a two clock cycle latency.

The width of the read and write interfaces can be independently programmed to single-width or double-width interfaces.

Chapter 6
Internal SRAM Controller Object

This chapter describes the Internal SRAM (IRAM) controller object.

Additional details can be found in the Application Developer's Object Reference.

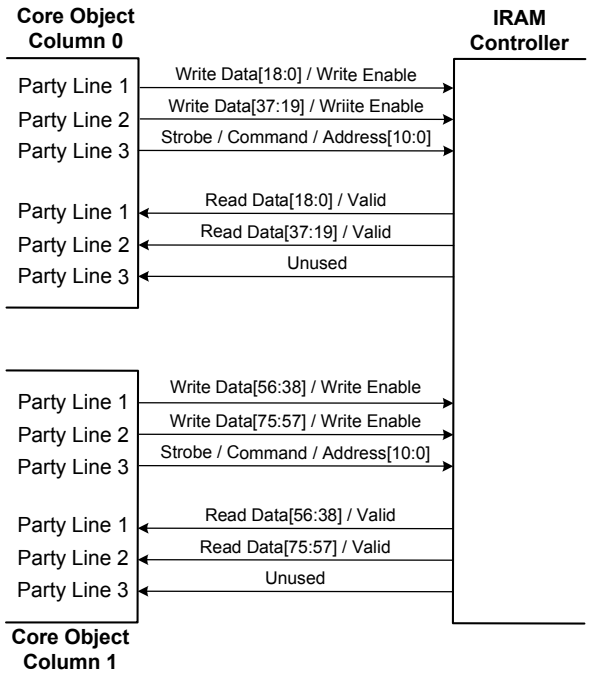
Overview

Each Arrix product provides 12 IRAM controllers, six on the north side of the periphery and six on the south side. Each IRAM controller provides access to a 2048x76-bit single-port SRAM. The IRAM controller requires two core clock cycles per access.

Functional Description

Each IRAM communicates with the core objects via two columns of six party lines. (For details on party line communication, see Chapter 2.) Figure 6-21 illustrates the inputs and outputs to the IRAM object.

Figure 6-21: IRAM Inputs and Outputs



The write and read data paths are statically configured. Address and control inputs are provided on party line 3 of either column 0 or column 1. Write data and read data are hardwired to party lines 1 and 2 of both column 0 and column 1.

The IRAM controller transfers 76 bits per command. The data is separated into four 19-bit entries, with each entry transferring over a specific party line as shown in Figure 6-21. The least significant 16 bits are R bits. The other three bits are tag bits and are statically configured to use any of the four C or V bits of that party line. The write data mapping and read data mapping of these tag bits are independent.

Control inputs (“command strobe” and “command”) can be selected from the four C bits and one V bit of party line 3. Address, control, and (optionally) the enable bits are sampled when the input signal “command strobe” is active. The application program must wait at least one clock between command strobes to ensure IRAM minimum cycle timing.

The contents of each IRAM can be preloaded via the PROM or JTAG interface during initialization.

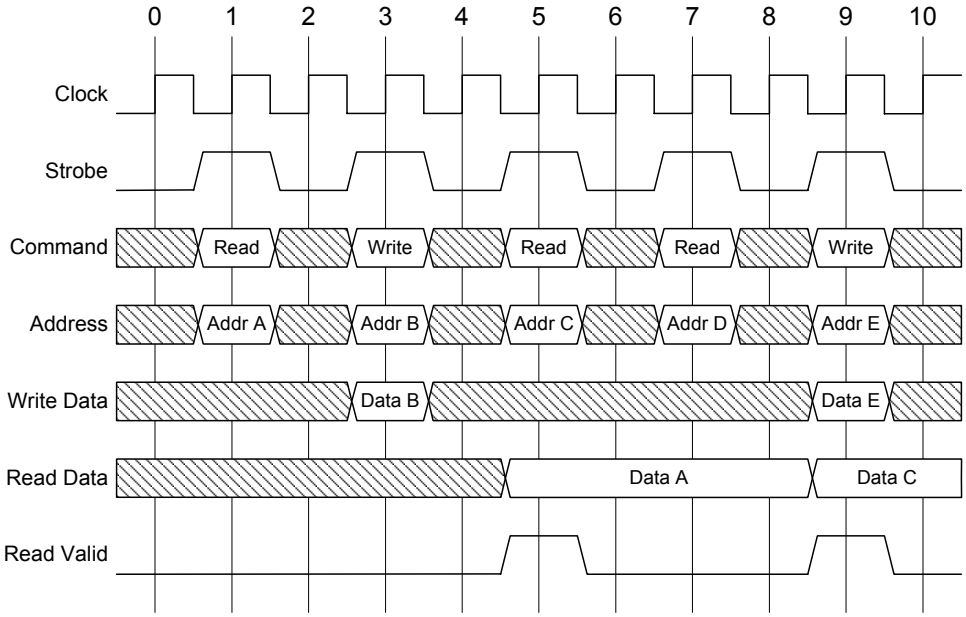
Read and Write Commands

Each IRAM can be statically configured to commit the full 76 bits to the internal SRAM for every command. Optionally, each of the four 19-bit entries can be independently written based on the “write enable” signal associated with that entry. If a particular entry is not written, the previous content is preserved. The “write enable” signal can be selected from the four C bits and one V bit of that party line.

Read commands return data four core clocks after the read request. An associated “data valid” signal indicates new output data is available. This signal can be returned on any of the four C bits or V bit of that party line. The IRAM controller holds previously read data until subsequent read operations return new data.

The timing diagram in Figure 6-22 illustrates reading from and writing to the IRAM controller.

Figure 6-22: IRAM Party Line Timing



Chapter 7
External RAM Controller Object

This chapter describes the External RAM (XRAM) controller object.

Additional details can be found in the Application Developer's Object Reference and in the Arrix Family Data Sheet.

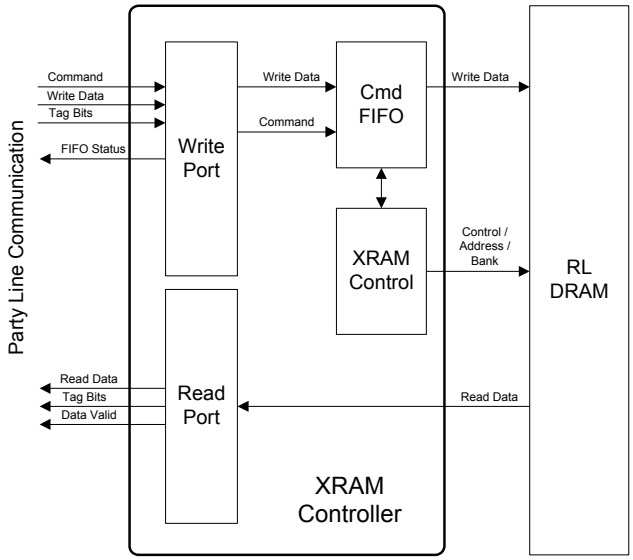
Overview

Each Arrix product provides two XRAM controllers, one on the north side of the periphery and one on the south side. Each XRAM controller provides access to external 36-bit Double-Data-Rate (DDR) Reduced Latency DRAM (RLDRAM-II) memory.

Functional Description

Figure 7-23 shows a block diagram of the XRAM controller.

Figure 7-23: XRAM Block Diagram



The XRAM controller accepts commands from party lines and produces the appropriate external memory interface signals. Commands can be sent to the controller on a clock by clock basis and are queued in command FIFO. See "Issuing Commands" on page 48 for more information.

Each XRAM controller supports three possible configurations of external RLDRAM-II with memory sizes up to 144 Mbytes. The controller contains all necessary external memory refresh logic. Refresh is handled either automatically (default operation) or manually.

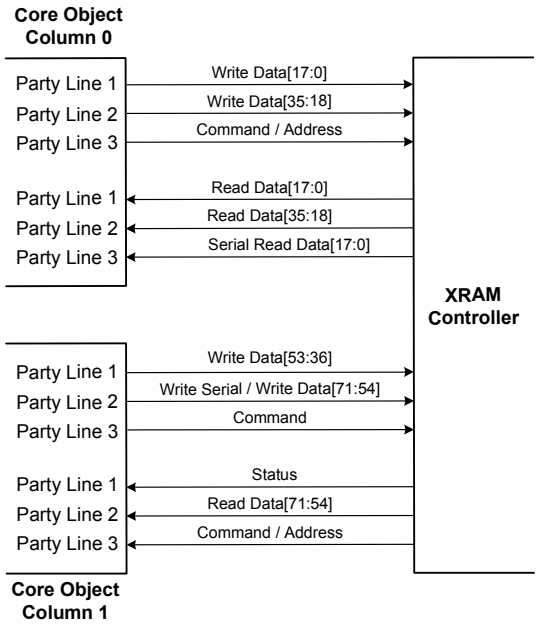
The maximum supported XRAM clock frequency is 266 MHz. This clock frequency is derived from the core clock using a divisor. At maximum frequency, 18 Gb/s (2.4 Gbytes/s) of peak bandwidth can be achieved per XRAM controller. The minimum operating frequency (imposed by the RLDRAM) is 175 MHz.

Issuing Commands

The XRAM command queue is a FIFO that can hold up to 36 commands. Each command contains the opcode address and, if necessary, write data. The command queue also bridges the core clock and XRAM clock domains. The command queue has a static "almost full" indicator, which is returned to the application for flow control when the FIFO contains 18 or more entries. A command can be issued to the XRAM controller every core clock cycle. If the command queue is full, subsequent commands are discarded.

Figure 7-24 further details the inputs and outputs to the XRAM controller.

Figure 7-24: XRAM Controller Inputs and Outputs



The write and read data paths are statically configured. Address and control inputs are provided on party line 3 of column 0 or column 1. Write data and read data are hardwired to party lines 1 and 2 of both column 0 and column 1.

The XRAM controller transfers 72 bits per command. The data is separated into four 18-bit entries, with each entry transferring over a specific party line as shown in Figure 7-24. The least significant 16 bits are R bits. The other two bits are tag bits and are statically configured to use any of the four C or V bits of that party line. The write data mapping and read data mapping of these tag bits are independent.

Control inputs (“command” and “command strobe”) can be selected from the four C bits and one V bit of party line 3. Address, control, and (optionally) the write data are sampled when “command strobe” is active.

Write Operation

The XRAM controller accepts 72 bits of data per write command from the core objects, which are then strobed out to external memory. Write data is provided on four party lines.

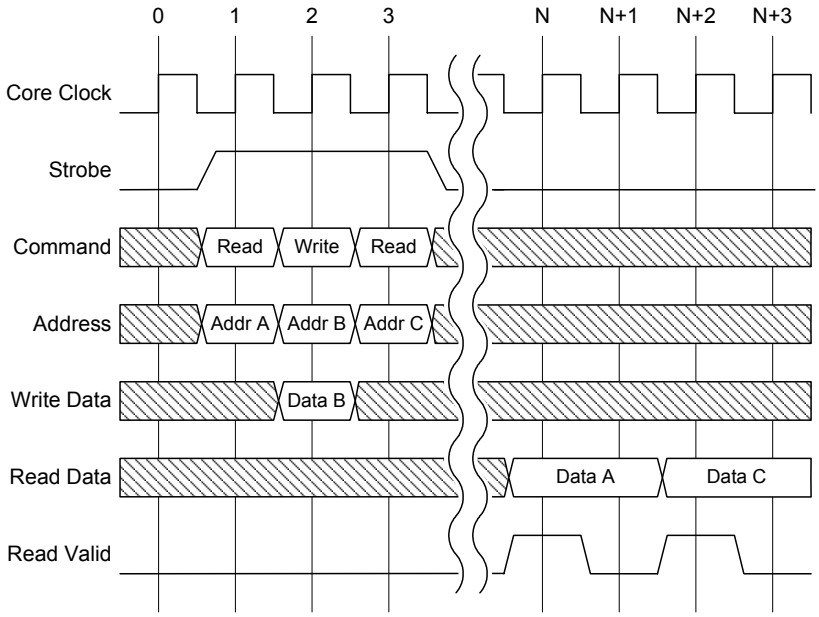
Read Operation

In general, there is a latency of 50-200 core clocks between issuing a read command to the XRAM controller and receiving the first 72 bits. After the first access, the XRAM controller will read sequential data out of external memory at a rate of 72 bits per external XRAM clock. Additional details can be found in the Arrix Family Data Sheet.

Write and Read Timing

Figure 7-25 illustrates writing to and reading from the XRAM controller. Note that read data is available 50-200 core clocks after the command is sent.

Figure 7-25: XRAM Timing Diagram



Command Codes

The XRAM controller accept commands which are represented by 3-bit opcodes. The available commands are listed in Table 7-4.

Table 7-4 XRAM Command Codes

Command	Name	Description
000	nop	No operation. Device is deselected.
001	read	Read command.
010	write	Write command.
100	ref	Refresh – refreshes one or all banks, depending on configuration.[a]
110	mrs	Mode Register Set – this command is used to set the RLDRAM configuration.[a]
111	init	Initialize – performs the RLDRAM initialization sequence.[a]

a. This command can be performed manually or it can automatically be performed by the XRAM controller. See the Application Developer's Object Reference for details on configuring this and other parameters.

RLDRAM-II Description

RLDRAM-II is dynamic RAM featuring: reduced latency, DDR data, eight internal banks per component, and non-multiplexed addressing. As outlined in Table 7-5, each XRAM memory controller supports three configurations of external memory at clock speeds up to 266 MHz DDR.

Table 7-5 RLDRAM-II Memory Configurations

Component Type	Number of Devices	Total Memory Size Per Interface
8M x 36	1	36 Mbytes
16M x 18	2	72 Mbytes
32M x 9	4	144 Mbytes

The XRAM controller supports moving 36 bits on each clock edge for a total burst length of two. Longer reads and writes can be performed by a sequence of individual commands. RLDRAM write masks are not supported.

A special consideration in RLDRAM is that multiple accesses to the same bank are slower than sequential accesses to different banks. To maximize throughput, sequential accesses should come from different banks. This is accomplished through XRAM controller configuration and data structure construction. The XRAM controller provides four choices of mapping addresses to banks.

Note that the XRAM controller only supports RLDRAM that is one rank deep. Additional details can be found in the Arrix Family Data Sheet.

Specifications for RLDRAM are available at http://www.micron.com.

Chapter 8
General Purpose I/O Interface Object

This chapter describes the General Purpose I/O (GPIO) interface object.

Additional details can be found in the Application Developer's Object Reference and in the Arrix Family Data Sheet.

Overview

The GPIO interface object facilitates communication between the FPOA and external devices. Each Arrix product provides two GPIO interfaces, one on the north side of the periphery and one on the south side. Each interface provides 48 bidirectional pins operating at LVCMOS (low voltage CMOS) signaling levels.

Functional Description

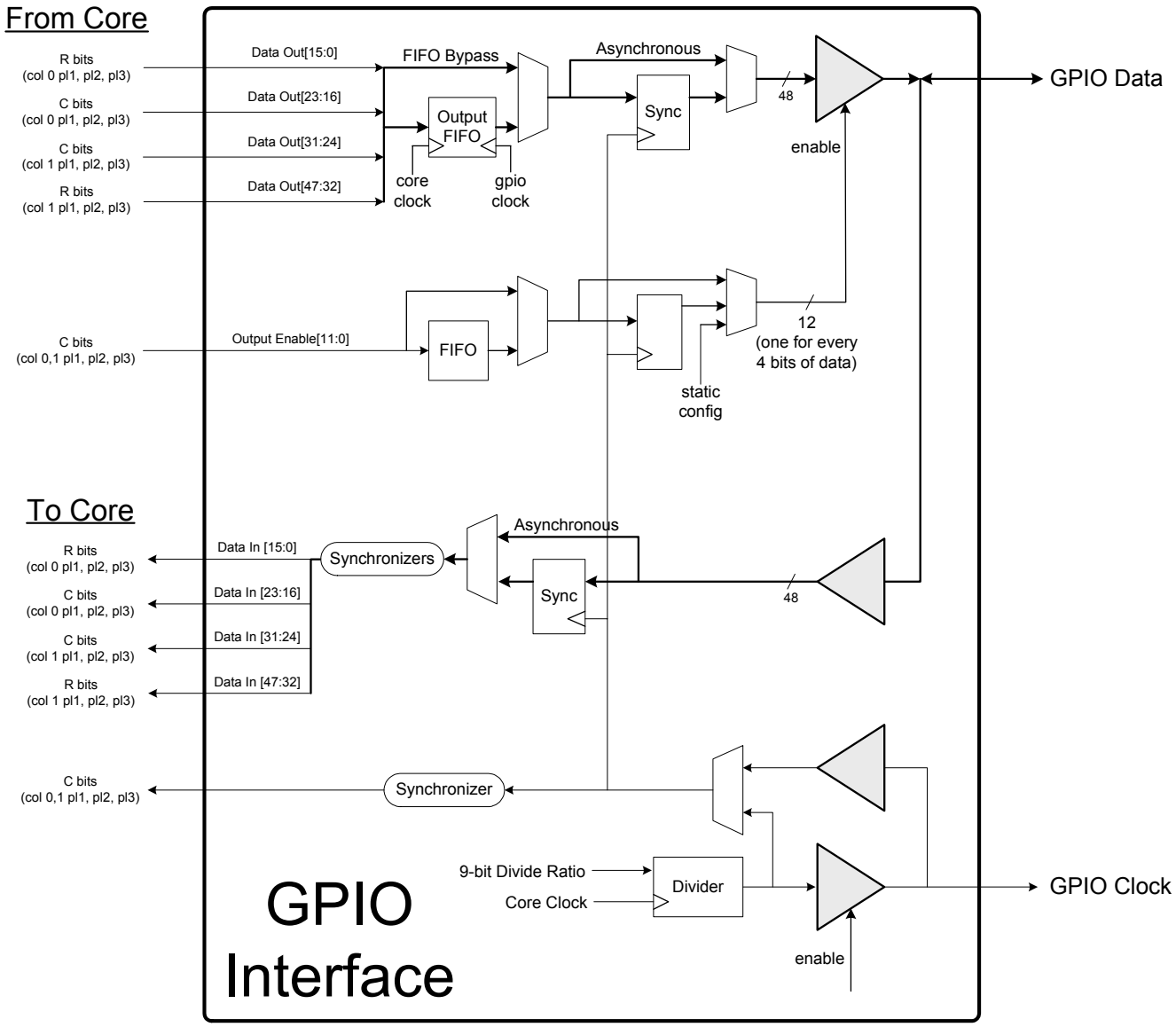
The GPIO interface, illustrated in Figure 8-26, is connected to two core objects (labeled column 0 and column 1) through six party line channels. GPIO data pins and the GPIO clock are limited to speeds less than or equal to 100 MHz for both synchronous and asynchronous operation.

For synchronous operation, a GPIO clock is used to strobe inputs, outputs, and output enables. The GPIO clock can be created by dividing the core clock. Alternatively, an external source can supply the GPIO clock. Asynchronous operations do not use the GPIO clock.

GPIO controls are provided in groups of four pins. This means that the GPIO provides 12 controls for the 48 pins. These controls include:

- "Output enable," which sets pin direction
- Asynchronous/Synchronous input setting
- Asynchronous/Synchronous output setting
- "FIFO bypass enable" for output data

Figure 8-26: GPIO Block Diagram



For both inputs and outputs, the GPIO interface is separated into two 24-bit fields (16 data bits and 8 tag bits). Data bits are controlled and observed by the party line R data path; tag bits are controlled and observed by C bits.

Output from the GPIO interface is divided into 12 four-bit fields. Each four-bit field has an associated “output enable” that is either statically configured or controlled by party line C bits.

In synchronous mode, an output FIFO handles the transition between clock domains. This FIFO is eight entries deep and contains an “almost full” signal, configured by a three-bit field. Writes to a full FIFO are discarded.

A static configuration selects either asynchronous or synchronous input. Synchronous inputs are strobed on the rising edge of the GPIO clock. Asynchronous inputs arrive independent of the GPIO clock.

Synchronous Operation

In synchronous mode, transfers through the GPIO interface occur on the rising edge of the GPIO clock. Data and tag signals must meet setup and hold requirements. Transfer rates on any GPIO pin cannot exceed 100 Mbit/sec. One data value is transferred every GPIO clock period. Delay from the input pin to the party line signal is six to seven core clock periods plus one GPIO clock period. Delay from the party line to the output pins is four core clock periods plus one GPIO clock (and a possible FIFO delay). Refer to the Application Developer's Object Reference for more information.

Asynchronous Operation

In asynchronous mode, transfers through the GPIO interface do not use the GPIO clock. Delay from the input pin to the party line signal is five to six core clock periods. Delay from the party line to the output pins is four core clock periods. Refer to the Application Developer's Object Reference for more information.

Initialization

All GPIO pins are held in tri-state until the I/O configuration is complete. If the GPIO clock is internally derived, it is enabled after configuration is complete. If the GPIO clock is externally derived, it must be valid before the application starts.

Chapter 9
Receive Interface Object

This chapter describes the Receive (RX) interface object.

Additional details can be found in the Application Developer's Object Reference and in the Arrix Family Data Sheet.

Overview

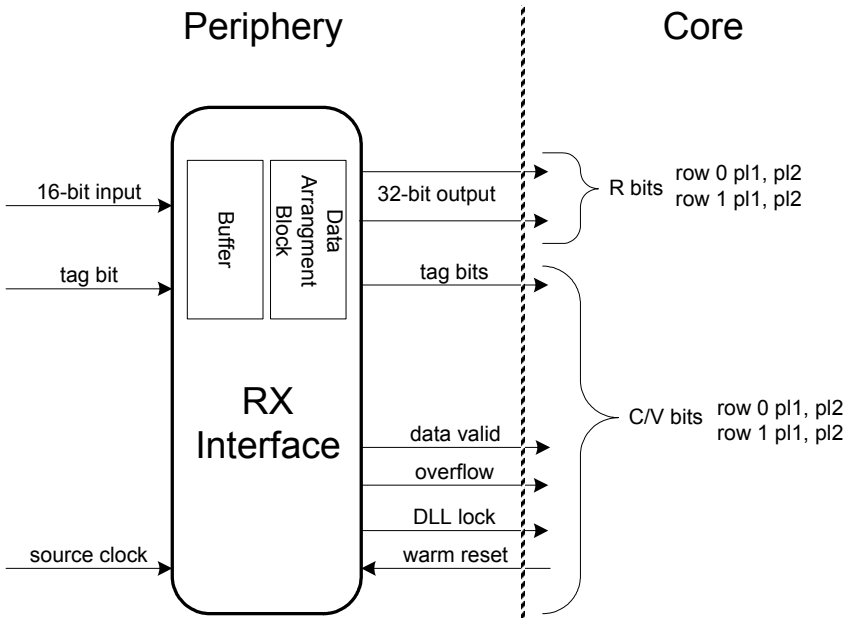
The RX interface is used for high-speed parallel LVDS input to the FPOA. Each Arrix product provides two RX interfaces, one on the east side of the periphery and one on the west side. Each interface has a 17-bit input (16 data bits + 1 tag bit) and a source-synchronous clock input. Because the clock input is source-synchronous, there is no option to use the internal clock.

The RX interface can be configured to operate in either double data rate (DDR) or single data rate (SDR). In DDR, data is sampled at the rising and falling edge of the clock. The clock frequencies supported are between 116 MHz and 500 MHz for DDR, and between 116 MHz and 640 MHz for SDR.

Interface Inputs and Outputs

Figure 9-27 illustrates the inputs and outputs to the RX interface.

Figure 9-27: RX Interface Inputs and Outputs



There are up to 16 data bits available for input, which are sent to the core objects—32 bits at a time—through the party line interconnect framework. A data buffer in the RX interface can hold up to 64 entries. A “Data Arrangement Block” can be configured to word swap, byte swap, and bit swap the incoming data. Along with data bits, there are several control bits that are sent to core objects:

- Tag bits return the values set by the input tag bits.
- The data valid bit is set when there is valid data on the line.
- The overflow bit is set when data arrives faster than can be passed to the core objects.
- The DLL lock bit is set when the DLL has initialized.

DLL Initialization Considerations

When the FPOA configuration is loaded, the periphery devices are first initialized. If the source clock for the RX interface is available during this time, there will usually be enough time for the DLL lock to occur while the core objects are initializing. However, if the external clock is not present at initialization time, or whenever a warm reset occurs, the core object that handles the RX data will need to explicitly monitor the DLL lock bit to determine when the interface is ready.

Bit Modes and Data Rates

The RX interface can operate at either single data rate (SDR) or double data rate (DDR), and can be used in 8-bit mode or 16-bit mode.

Data is buffered so that it can be transferred from the RX interface to the core objects 32 bits at a time. This may require a delay depending on the bit mode and the data rate. Table 9-6 shows the number of clock cycles required to transmit data from the RX interface to the core objects.

Table 9-6 Clock Cycles Required to Transmit Data to Core Objects

Bit Mode	Data Rate	Clock Cycles
8 bit	SDR	4
8 bit	DDR	2
16 bit	SDR	2
16 bit	DDR	1

The number of tag bits sent to the core objects is dependent on whether the interface is set to 8-bit mode or 16-bit mode. Since the resulting output is always 32 bits, more than one tag bit is sent to the core objects.Table 9-7 and Table 9-8 show the number of tag bits sent to core objects given each mode.

.

Table 9-7 Data and Tag Bits in 8-bit Mode

Sample Number	From Device		To Core	
	Data Bits	Tag Bits	Data Bits	Tag Bits
n	8	1		
n+1	8	1		
n+2	8	1		
n+3	8	1	32	4

Table 9-8 Data and Tag Bits in 16-bit Mode

Sample Number	From Device		To Core	
	Data Bits	Tag Bits	Data Bits	Tag Bits
n	16	1		
n+1	16	1	32	2

Other Features and Considerations

- There is a configuration parameter to power down the RX interface. Unused RX interfaces are powered down by default.
- There is a warm reset that can be used to dynamically set the input width. After resetting the interface, the core objects must wait for the DLL lock signal.
- The device can accept a clock that is aligned with the data, or it can be 90 degrees phase-shifted from the data.
- It takes a maximum of eight RX clock cycles plus eight core clock cycles for data to get to the party lines from the external device.

Chapter 10
Transmit Interface Object

This chapter describes the Transmit (TX) interface object.

Additional details can be found in the Application Developer's Object Reference and in the Arrix Family Data Sheet.

Overview

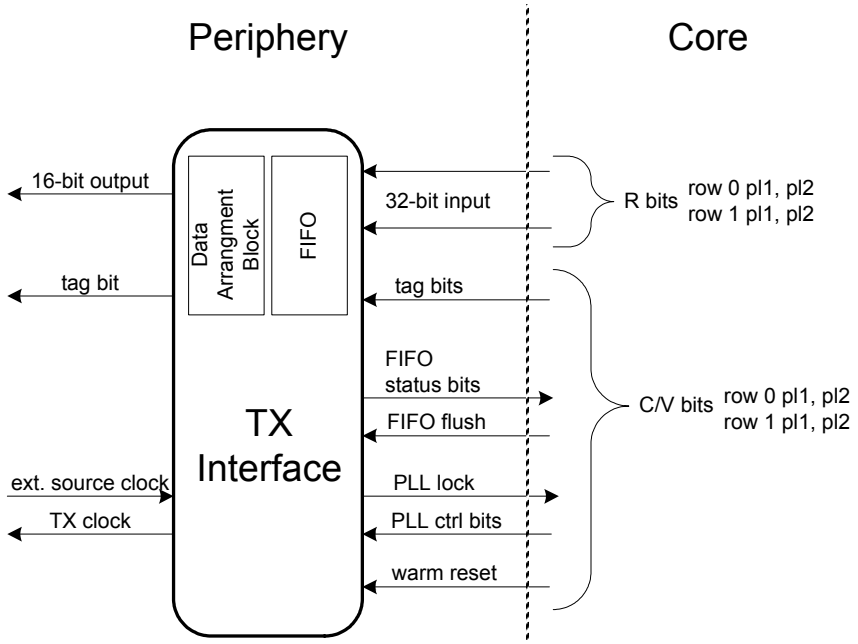
The TX interface is used for high-speed parallel LVDS output from the FPOA. Each Arrix product provides two TX interfaces, one on the east side of the periphery and one on the west side. Each interface has a 17-bit output (16 data bits + 1 tag bit) and a source-synchronous clock output. A FIFO within this interface allows for a smooth transition from the core clock domain to the TX interface clock domain.

The TX interface can be configured to operate in either double data rate (DDR) or single data rate (SDR). In DDR, data is transmitted at the rising and falling edge of the clock. The clock frequencies supported are between 18.75 MHz and 500 MHz for DDR, and between 18.75 MHz and 640 MHz for SDR.

Interface Inputs and Outputs

Figure 10-28 illustrates the inputs and outputs to the TX interface.

Figure 10-28: TX Interface Inputs and Outputs



Core objects send 32 data bits at a time to the TX interface. These data bits, along with their associated tag bits, are stored in a FIFO. The FIFO stores information in pairs (two writes) and can hold up to 64 pairs of data. This data is released in 8 or 16 bit increments to an external device. A “Data Arrangement Block” can be configured to byte swap and bit swap the outgoing data.

The FIFO releases status bits to the core objects, as described in "FIFO Behavior" on page 63.

The PLL can be controlled through the PLL control bits, as described in "Data Rates and Clock Speeds" on page 64. The PLL lock bit is discussed in "PLL Initialization Considerations" on page 64.

An external source clock can be used to derive the TX clock. Alternatively, the TX clock can be derived from the core clock.

FIFO Behavior

The TX interface's FIFO can store up to 64 pairs of data. Each data element in the pair contains 32 data bits and their associated tag bits. Since the FIFO requires pairs of data, a single write to the TX interface will not be stored in the FIFO — the data is stranded until the next write occurs. For this reason, the core objects should be sure to write to the TX interface an even number of times.

Before the FIFO receives any data, the TX interface sends a startup pattern (16 bits of data + tag bit). After the FIFO receives data, this startup pattern will never be sent again. Whenever there is no data to send (after startup), the FIFO sends an underflow pattern. These two FIFO patterns are configurable and can be identical if desired.

The TX interface sends five FIFO status bits to the core:

- "Almost empty" watermark — This bit is set when the FIFO level drops below a configurable level.
- "Almost full" watermark —This bit is set when the FIFO level rises above a configurable level.
- Empty — This bit is set when the FIFO is empty.
- Overflow — This bit is set when the core objects attempt to write to a full FIFO. In this case, the newest data is lost.
- Underflow — This bit is set when there is no data in the FIFO to send. In this case, the TX interface sends the underflow pattern.

In addition to these status bits, the core objects can flush the FIFO by setting the FIFO flush bit.

Data Rates and Clock Speeds

The TX interface can operate at either single data rate (SDR) or double data rate (DDR), and can be used in 8-bit mode or 16-bit mode. When in 8-bit mode, the upper byte of output can be configured either to repeat the first eight bits of data or to be padded with a configurable value.

A source clock can be used with a frequency between 18.75 MHz and 37.5 MHz. The PLL can increase this clock rate with a multiplier value of 1,2,4,8,16, or 32. This PLL value can be configured at design time or runtime. In addition, the core clock can be used directly. Regardless of the configuration, the clock frequency can not exceed 500 MHz for DDR and 640 MHz for SDR.

PLL Initialization Considerations

When the FPOA configuration is loaded, the periphery devices are first initialized. If the source clock for the TX interface is available during this time, there will usually be enough time for the PLL lock to occur while the core objects are initializing. However, if the external clock is not present at initialization time, or whenever the PLL multiplier is set at runtime (using a warm reset), the core object that handles the TX data will need to explicitly monitor the PLL lock bit to determine when the interface is ready.

Other Features and Considerations

- There is a configuration parameter to power down the TX interface. Unused TX interfaces are powered down by default.
- There is a warm reset that can be used to dynamically set the PLL multiplier. After resetting the interface, the core objects must wait for the PLL lock signal.
- The device can transmit a clock that is aligned with the data, or it can be 90 degrees phase-shifted from the data.

Chapter 11
Initialization and Control

This chapter describes the initialization process for the FPOA and includes an overview of the debugging support.

Additional details can be found in the Application Developer's Object Reference and in the Arrix Family Data Sheet.

Overview

The FPOA tool suite can generate a PROM or JTAG load image. The PROM image can be burned into a configuration PROM. The JTAG image can be loaded directly using the FPOA JTAG interface.

During initialization, the periphery objects are configured and enabled before the core objects. This allows periphery objects to synchronize with external clocks while the core objects are initialized. Part of this synchronization process includes starting the core clock.

After loading an image onto the FPOA (via PROM or JTAG), the JTAG interface can be used for debugging.

Initialization

PROM Initialization

The PROM load image is formatted in Intel HEX86. This image is burned into a PROM using an external PROM programmer.

After reset, the PROM controller automatically accesses the PROM (if present) to acquire the load image. An input pin notifies the FPOA that a PROM is present. If the PROM isn't present, the FPOA waits for the load image from the JTAG interface.

A CRC checksum value can be used to validate the PROM image. If a CRC error occurs, the application does not execute and a JTAG register indicates the error.

A PROM can be read at clock speeds up to 100 MHz. PROM initialization can occur in serial mode (one bit at a time) or in parallel mode (eight bits at a time). At a frequency of 100 MHz, the FPOA will load in 2 milliseconds — when in parallel mode. Additionally, each IRAM can be loaded in 0.4 milliseconds.

JTAG Initialization

The JTAG load image is passed directly to the FPOA through the JTAG interface using MathStar's FPOA tools.

Warm Reset vs. Initial Program Load

At initial program load, all core objects are released from reset at the same time. Warm reset allows individual objects to reset independent of other objects. This condition must be considered in a design. Note that after both initial program load and warm reset, party line launch/land registers are always forced to 0. NN registers can optionally revert back to initial values after a warm reset. Otherwise, they maintain their values through a warm reset.

Debugging

JTAG Interface Debugging

The JTAG interface provides access to a set of control registers, which allow control of the FPOA. These capabilities include:

- 1149.1 Boundary Scan testing support
- Displaying contents of object registers and RAM
- "Load-and-Wait", which doesn't execute the image until the start command
- "Halt-the-chip", which will stop the application from executing

Application-controlled Breakpoints

The FPOA Control object has a single bit input that — when asserted — halts execution of the FPOA. The application defines the conditions that assert this control bit. The FPOA is halted 13 clocks after assertion of this input. The FPOA tools can override the assertion of the debug signal.

After the FPOA is halted, the JTAG interface is used to display the contents of objects and RAM.

Chapter 12
Sample Application

This chapter describes, at a high level, the implementation of an 8-Tap FIR using MathStar's FPOA.

Overview

Finite Impulse Response (FIR) filters are basic building blocks used in numerous DSP applications. A FIR filter performs a weighted sum on consecutive input samples. FIR filters have no feedback and are always stable. The basic defining equation and its direct realization is shown in Figure 12-29. The difference equation shows how the input signal is related to the output signal.

Figure 12-29: FIR Filter Difference Equation

$$y(n) = b_0 x(n) + b_1 x(n-1) + \ldots + b_{p-1} x(n-(P-1))$$

where *P* is the filter order (or Taps); *x(n)* is the input signal; *y(n)* is the output signal; and b_i are the filter coefficients.

The previous equation can also be expressed as shown in Figure 12-30.

Figure 12-30: FIR Filter Difference Equation – Using Summation

$$y(n) = \sum_{i=0}^{P-1} b_i x(n-i)$$

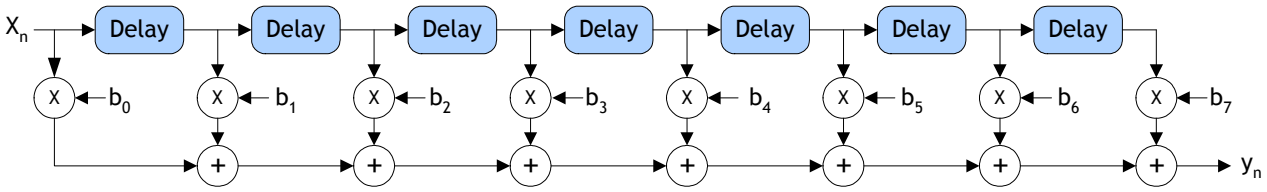
In the case of an 8-Tap filter, *P* = 8.

This chapter describes how an 8-Tap FIR filter can be mapped to FPOA objects.

Block Diagram

Figure 12-31 shows the basic logical block diagram for a FIR filter. The delays result from operating on prior input samples. The b_k values are the coefficients used for multiplication, so that the output at time *n* is the summation of all the delayed samples multiplied by the appropriate coefficients.

Figure 12-31: Logical Structure of a FIR Filter



Code Example

The following C code demonstrates a 16-bit 8-Tap FIR filter. In this example, *bArray* contains the coefficients and *xArray* is the input samples over time represented as *Delay* in Figure 12-31. The code performs two loops, the first is the FIR Multiply and the second emulates the delay.

```
/*****************************************************************
* Performs the basic fir by storing the input sample, calculating
* the output sample and moving delay line.
* Inputs:
*    x = input sample, bArray = Array of Coefficients,
*    xArray = x samples over time
*****************************************************************/
int fir(int x, const int bArray[], int xArray[])
{
    int i, NTAPS = 8, accum;

    /* store input at the beginning of the delay line */
    xArray[0] = x;

    /* calc FIR */
    accum = 0;
    for (i = 0; i < NTAPS; i++) {
        accum += bArray[i] * xArray[i];
    }

    /* shift delay line */
    for (i = NTAPS - 2; i >= 0; i--) {
        xArray[i + 1] = xArray[i];
    }
    return accum;
}
```

From Algorithm to FPOA Objects

Mapping the FIR into an FPOA requires a different approach than the C code because the delay can be accomplished using the interconnect characteristics of the FPOA. In the data flow shown in Figure 12-32, a new sample is muxed in on every eight cycles. During the other seven cycles, the data is cycled back through the FIR multiply as in the first loop of the C code from the previous section.

Figure 12-32: FIR Data Flow – Sequential Implementation

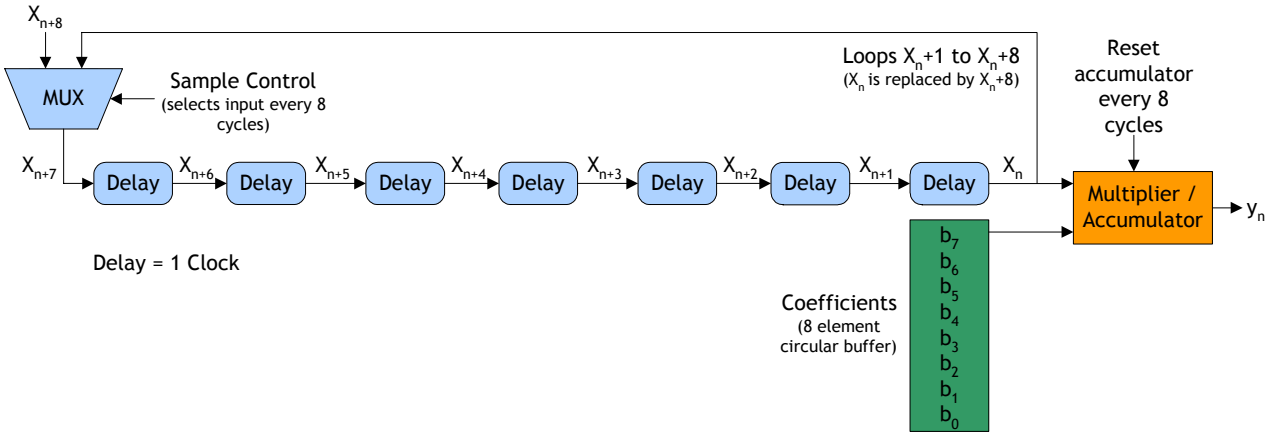
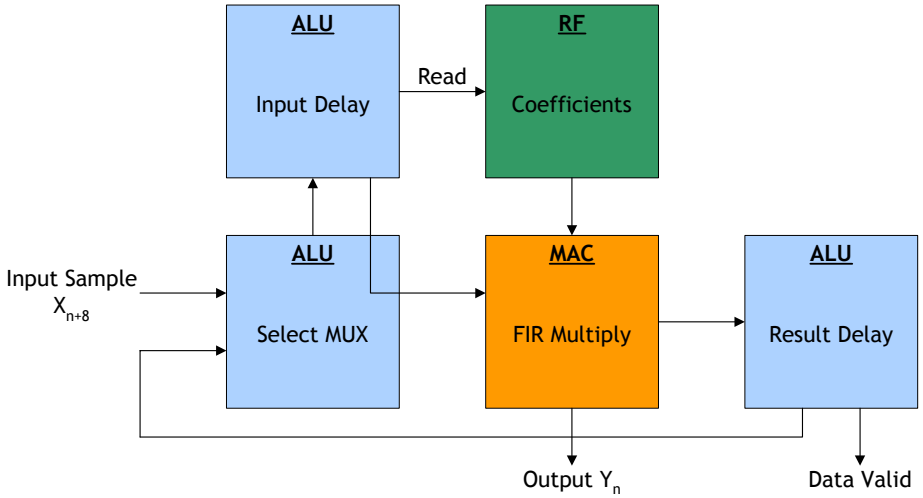


Figure 12-33 illustrates a sample FPOA tiling of the above data flow. The basic building block for the FIR filter is the 16-bit multiplier. A MAC object (labeled “FIR Multiply”) performs this operation. An RF object stores the required coefficients. Both the multiplexer and the delays are performed by ALU objects. This implementation generates one sample every eight cycles.

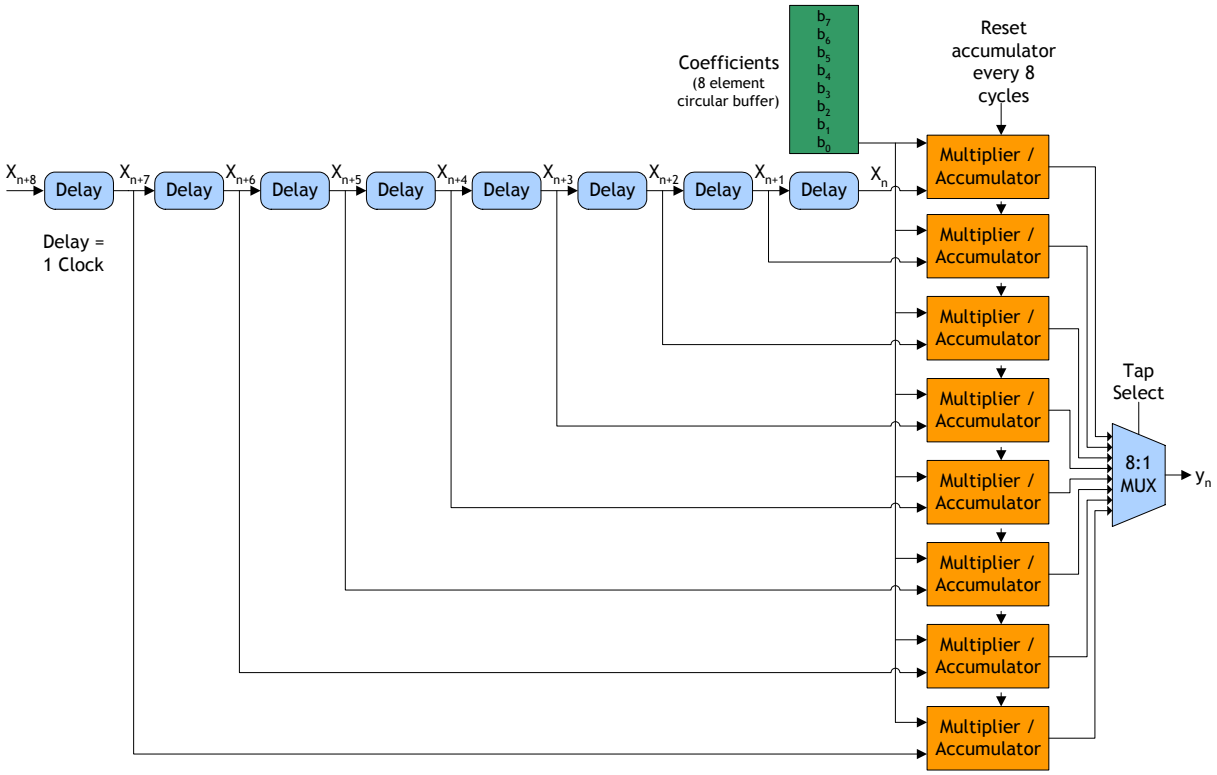
Figure 12-33: FIR Tiling



Alternate Implementation

As with any architecture, the design trade-offs are usually size versus speed. The previous implementation produces a result every eight cycles. A faster version of the algorithm can be constructed to produce a result on every clock cycle. Figure 12-34 shows a data flow for a parallelized version of the FIR in an FPOA.

Figure 12-34: FIR Data Flow – Parallel Implementation



These two implementations demonstrate the trade-offs between resource and throughput. The sequential version uses fewer objects and generates one sample every eight MAC operations. The parallel version uses eight MACs (rather than one) and generates eight results every clock. The parallel version also needs more objects to route data to all eight MACs and to convert the parallel eight results into sequential data.

Conclusion

This chapter described two ways of implementing an 8-Tap FIR filter using the FPOA. This information is presented as an overview to FPOA design. For detailed information on using MathStar's tools to produce FPOA designs, see MathStar's Application Developer's Guide.

Acronyms and Abbreviations

Shortened term	Expanded term
ALU	Arithmetic Logic Unit
ALB	Arithmetic Logic Block
C bit	Control bit
DDR	Double Data Rate
DLL	Delay Lock Loop
FPOA	Field Programmable Object Array
GPIO	General Purpose I/O
IRAM	Internal RAM
K0, K1	Constant values from the two constant registers in each core object
LVDS	Low Voltage Differential Signaling
MAC	Multiply Accumulator
MUX	Multiplexer
NN	Nearest Neighbor
PL	Party Line
PLL	Phase Lock Loop
R bits	Register data bits
RF	Register File
RX Interface	Receive Interface
RLDRAM	Reduced Latency DRAM
TF	General Purpose Truth Function
TFA	Branch Control Truth Function for the ALU
TX Interface	Transmit Interface
V bit	Valid bit
VR bits	V bit + R bits
XRAM	External RAM

Index of Terms

H

I

J

L

M

N

O

P

R

S

T

V

W

X

List of Figures