APPENDIX B

THE FPGA FRAME API

Field-programmable gate arrays (FPGAs) have steadily evolved over the last decade as a means of accelerating a number of computational tasks through the use of reconfigurable hardware. Given the potential for this technology it is beneficial to investigate methods by which FPGAs can be integrated into the cluster computer architecture and efficiently utilized by end applications. Unfortunately integrating an FPGA into a cluster can be extremely challenging due to the limited types of resources these cards employ. Most commercial FPGA cards employ one or more FPGAs, a cache of on-card memory, and a simple PCI controller. Because these cards often lack a general purpose CPU, it is often necessary to construct a state machine in the FPGA that serves as an interface between a user's computational circuits and external resources such as on-card memory or the host CPU.

In order to integrate the Celoxica RC-1000 FPGA card into a cluster computer utilizing the GRIM communication library, it was necessary to design and implement a block of FPGA circuitry that managed interactions between the FPGA's computational circuits and end applications. This block of logic is known as the FPGA's static *frame* because it allows a *canvas* of user-defined computational circuits to be insulated from the card-specific features of the RC-1000 device. This section describes the low-level mechanics of the frame and provides an API by which end users can interact with the FPGA device. While the frame is designed to operate specifically with the RC-1000 card, it is possible to adapt this work for use with other similar FPGA cards.

B.1 Architecture Overview

As depicted in Figure B.1, the RC-1000 implementation of a GRIM communication endpoint is divided into two contexts: the *static frame* unit and the *dynamic circuit canvas*. The frame serves as a reusable block of hardware that allows different computational circuits to be dynamically plugged into one of the cluster's FPGA devices. The frame provides three specific interfaces to insulate a user's circuits from the device specific characteristics of the target FPGA card. First, the frame implements a communication library API that is responsible for handling messages coming from or going to the communication library. Second, the frame provides an interface to the dynamic circuit canvas that allows multiple user-defined circuits to be connected to the frame. Finally, the frame provides an interface that allows applications to access a region of on-card memory known as the scratchpad.

B.1.1 Data Path of the Frame

A simplified view of the Celoxica RC-1000 frame's low-level data path is depicted in Figure B.2. The four SRAM banks available on the RC-1000 are allocated as follows. Bank 0 houses incoming message queues for the communication library as well as runtime information for the frame. SRAM banks 1 and 2 are utilized as scratchpad memory for storing application data. SRAM bank 3 houses the outgoing messages for the communication library. The control/status port on the RC-1000 provides a simple means of transferring

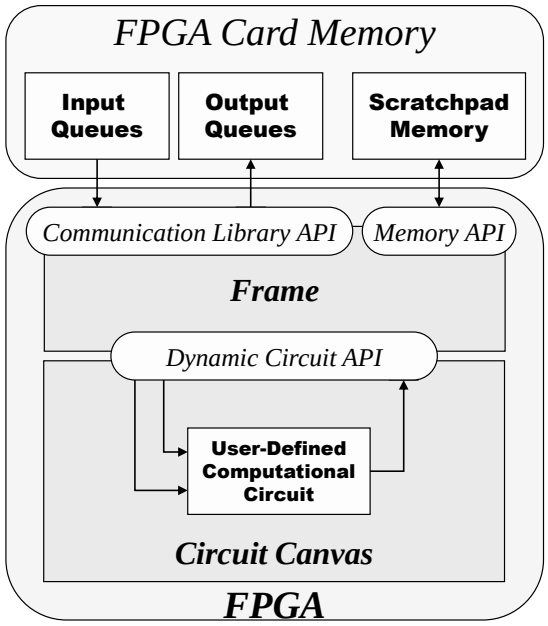

Figure B.1: The three interfaces managed by the FPGA frame.

8-bit data values with the host. This port can be configured to transmit an interrupt to the host and is used to pass simple state information between the host and card.

The individual units in the frame architecture are described as follows:

- **Fetch/Decode unit**: This unit fetches the next message to be processed by the frame and establishes the necessary data paths through the frame to process the message. A message can originate from either the endpoint's message queues (housed in SRAM bank 0) or from a recycle buffer which contains the previously generated outgoing message.

- **Scratchpad controller unit**: This unit is used to exchange data with the scratchpad memory (SRAM banks 1 and 2). A single SRAM bank can supply both input vectors and accept the output vector of the user-defined circuit if needed. Vector data is fetched and stored linearly starting at memory offsets provided in the incoming message's header.

- **Results cache**: The results cache is used to buffer the output of a computation until the frame is able to write the data into its proper destination. The cache is utilized only when an operation needs to fetch and store data with the same scratchpad memory bank, or when input data is fetched from an incoming message and output is written to the recycle buffer.

- **Message generator**: This unit takes results generated by the computational circuit, formats the data into an outgoing message, and inserts the data into an outgoing message queue (located in SRAM bank 3).

- **Vector data ports**: The frame provides three vector data ports, to which all user-defined circuits are connected. Ports A and B provide input streams to the circuits while port C receives output data generated by the circuits.

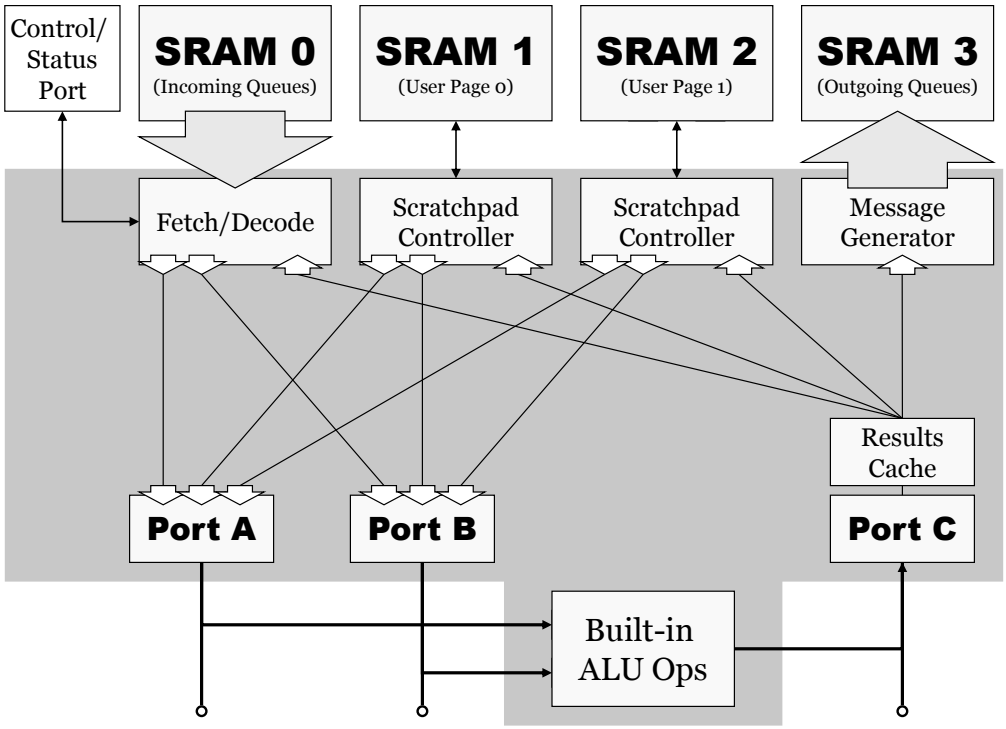

Figure B.2: The internal structure of the frame for the RC-100 implementation.

- **Built-in Ops**: The frame provides a simple built-in computational circuit that can perform a variety of common vector operations, including add, subtract, multiply, min, max, invert, and pass.

B.2 Communication Library Interface

The first interface that the frame provides allows the FPGA to interact with the communication library. This interface is responsible for managing incoming and outgoing message queues, and must be designed to work with the message format specified for a given communication library. The RC-1000 implementation of the frame utilizes messages formatted for the GRIM communication library, although it is possible to adjust the implementation to operate with other libraries.

B.2.1 GRIM Message Format

The RC-1000 implementation of the frame processes messages that are formatted for the GRIM communication library. Like other communication endpoints found in GRIM, information included in the header of each message is used to specify how the RC-1000 should process a message. The active message function handler identifier for the RC-1000 corresponds directly to the globally unique user-defined circuit that is used to process the message. Because of the flexibility that the frame provides in processing a message, it is necessary to encapsulate additional information in the message header. This information resides in the arguments section of the active message header and is used to configure the frame's data paths to meet an application's needs. The fields used to configure the FPGA

Table B.1: The data fields in an active message header that control the operation of the frame and the corresponding bit lengths.

Arg[0]	Forwarding ID (8)	A Driver (1)	B Driver (1)	C Driver (1)	Reserved (7)	Sub Op (4)	Op Length (10)
Arg[1]	Port A Virtual Address (29)						
Arg[2]	Port B Virtual Address (29) *or* Port B Constant (32)						
Arg[3]	Port C Virtual Address (29)						

are listed in Table B.1.

B.2.2 Message Queues

The frame implements three different types of message queues. The first category of message queue is used to house incoming messages for the card. These queues are located in SRAM bank 0 and adhere to the append-style of queuing utilized throughout GRIM. The current implementation of the frame provides two separate incoming message queues, with the intention that one queue is for the host CPU and the other for the NI. The frame periodically polls each of these queues to determine if new messages are available. This polling operation takes place every 300 FPGA clock cycles and requires less than a dozen clock cycles to poll for new data.

The second place where messages can be stored is in the recycle buffer. This buffer is housed in SRAM bank 0 and has room for exactly one message. This buffer is utilized when an application needs the FPGA to perform a series of operations on a set of data. Users can specify a message be recycled by setting the C-Driver bit in the message's header to zero. The frame will then route the results of the computation into the recycle buffer and insert the proper header from the forwarding table. The frame provides a guarantee that if a message is placed in the recycle buffer, it will be selected as the next message processed by the FPGA. This guarantee is necessary to prevent multiple messages from being inserted into the recycle buffer. Therefore it is imperative that users prevent endless recycling loops in the forwarding table.

The third type of message queue controlled by the frame is for outgoing messages. Currently there are two outgoing message queues that are housed in SRAM bank 3. In order to simplify the task of managing these queues, the frame implements a slotted queuing system. Because the FPGA card cannot directly trigger the DMA engines, the frame must notify the host when data is available in the message queues. This operation is performed by updating the RC-1000's status register, which the host periodically polls. When the host detects new messages in the card's outgoing message queues, it can perform the necessary transfer of data to the proper endpoint.

B.2.3 Forwarding Registers

A key design point for the RC-1000 frame is that it is able to process incoming messages and generate outgoing messages. This allows the card to be utilized as an intermediate computational stage as opposed to simply a unit that sinks data. Therefore mechanisms

have to be present in the frame to allow messages to be ejected by the frame into the communication library. Like other peripheral devices in the GRIM environment, it is expected that the card will generate messages only in response to a stimuli, such as the detection of a new incoming message. The hardship in implementing such a system is providing an interface where users can specify the types of response messages that the frame generates. The implementation of the RC-1000 frame utilizes a set of forwarding registers to solve this problem.

The term forwarding registers in the GRIM environment refers to a database in a communication endpoint that contains information used to format outgoing messages. For the RC-1000 frame this database is implemented as a large table of user-programmed message headers. All incoming messages have a field in the message header that specifies which table entry (if any) the frame should reference to generate an outgoing message. The frame copies the information from the specified entry to the outgoing message and places the results of the computation in the payload section of the message. Users can adjust the forwarding register table entries through a set of built-in active message handlers for the frame. The set_pipeline handler simply copies 64 bytes of payload into the specified forwarding register entry. It is the responsibility of the user to allocate and manage forwarding registers in this table.

The forwarding registers for the RC-1000 are located in SRAM bank 0 starting at address 0 in the current frame implementation. The table contains 256 entries, with each entry housing a single message header (64-bytes). The frame is designed to reference the forwarding registers only when a header needs to be placed on a message that is generated. For these situations the frame operates as follows. First, the frame processes a message in a normal manner. The message header is fetched, the frame data paths are established, and data is streamed through a specified computational unit. The results of this computation are routed to the payload section of the generated message, whether the generated message is assembled in an outgoing message queue slot or the recycle buffer. Next, the frame uses information from the incoming message to generate an index into the forwarding register table. The message header located at this entry is then streamed into the header section of the generated message. Finally, the frame updates the sender id and the payload length fields of the message header to guarantee that the generated message is properly identified.

B.2.4 Active Message Circuit Identification

Once the frame receives an incoming message it must determine which user-defined circuit is utilized to process the data. Conceptually, user-defined circuits are similar to function handlers found in any other communication endpoint in GRIM. Therefore each user-defined circuit is labeled with a unique active message handler identifier that end applications can reference to perform a desired computation. However, unlike other function handlers used in GRIM, user-defined circuits are statically assigned active message identifiers. When creating a new circuit, a user must define a new static active message handler identifier for the circuit in the `grim_handlers.h` file. This file contains a static list of handler IDs for various functions utilized in the GRIM library. Once identified, users can reference a user-defined circuit with a simple constant as opposed to locating an identifier for the circuit through the runtime handler database. An advantage of this approach is that it simplifies the task of forwarding data between FPGA computational circuits because all circuit identifiers are known in advance.

At runtime the frame must be able to determine which user-defined circuit is utilized

to process an incoming message. In the RC-1000 implementation of the frame this is accomplished by comparing an incoming message's active message handler id to a list of the FPGA's user-defined circuits. This list is managed by the host and updated whenever the FPGA's configuration is updated. Specifically, the host stores the list of a configuration's user-defined circuits in the card's SRAM before a configuration is loaded into an FPGA. After the FPGA is reset, the frame loads this list of functions from SRAM into an internal set of registers. When the frame observes an incoming message, it compares the active message handler to the list of available circuits. If the requested circuit is available the frame establishes the data path necessary to connect the user-defined circuit to process the message. If the circuit is not available, the host is notified of the problem with a function fault.

B.2.5 Function Faults

A function fault is when an incoming message requests an active message handler that cannot be satisfied with the user-defined circuits that are currently available in an FPGA. The first phase in a function fault is for the FPGA to store all of its runtime state information in on-card SRAM. This data includes the id of the function that caused the fault as well as the frame's current set of message queue pointers. Future versions of the frame may also include the runtime state information of individual user-defined circuits in this operation. After runtime information is stored the frame notifies the host of the function fault through the card's status register. The frame then suspends operation until the host passes an activation signal to the frame through the control register.

Once the host detects a function fault it must load the id of the missing user-defined circuit and determine how the FPGA should be reconfigured. In the current implementation the FPGA is reconfigured in its entirety. Therefore the host simply locates an FPGA configuration in its database that features the missing hardware and loads the configuration onto the FPGA. This process includes writing the new FPGA configuration's list of user-defined circuits to the card's SRAM, loading the FPGA with the new configuration, and triggering the FPGA reset. The FPGA then loads its runtime state information from SRAM and continues processing where it left off.

B.3 Computational Circuit Interface

The frame allows multiple user-defined computational circuits to exist in the dynamic circuit canvas, as illustrated in Figure B.3. Each user-defined circuit is connected with two vector inputs (labeled as ports A and B) and one vector output (labeled as port C). For simplicity the frame is designed to allow only one user-defined circuit to be active at any given time. When the frame detects a new incoming message, it sends an activation signal to the user-defined circuit specified in the message's header and then routes data into and out of the vector data ports. Vector data ports are asynchronous and provide sequential streams of data using a simple control protocol. Circuit designers are free to utilize these ports in any manner they desire, as long as unused ports are properly grounded.

B.3.1 Vector Data Port Signaling

Vector data ports are designed to operate in an asynchronous fashion. A simple valid/acknowledge handshaking protocol is utilized to allow either the sender or receiver of a vector data port to stall the passing of data. By design the sender and receiver of a vector data

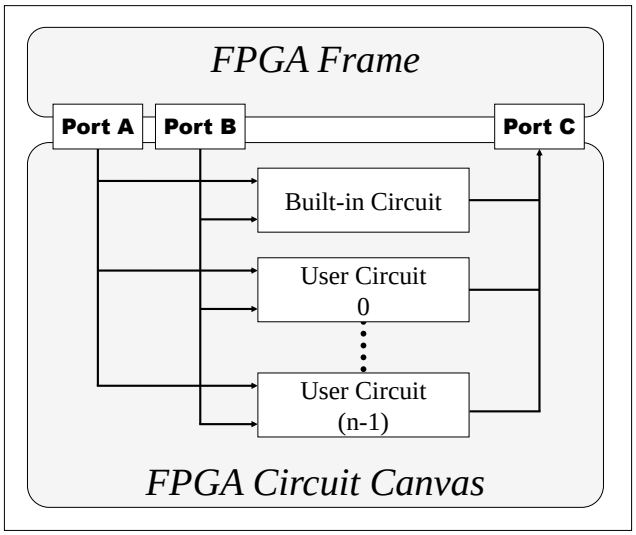

Figure B.3: The interface between the FPGA's frame and circuit canvas.

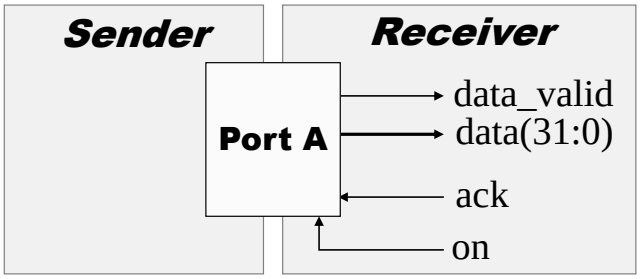

Figure B.4: The signals for a vector data port.

port operate on opposite clock edges. The frame inverts the clock supplied to user-defined circuits so both units have the appearance of operating on the rising edge of the clock. Because the sender and receiver are on opposite clock edges it is possible for a new data value to be transferred every clock period. The interface for a port is depicted in Figure B.4.

The signals for a vector data port are as follows.

- **On:** The user-defined circuit must assert the on signal for the entire time it needs to transfer data with the port. Therefore the first action a user-defined circuit must perform when it is activated by the frame is asserting the on signals for all data ports that will be used when processing a message. Once the on signal is turned off the frame will stop attempting to transfer data with the port. All vector port on signals must be set to low before job_done can be triggered.
- **Valid:** The transmitter for a port signifies that the next word from the vector port has been placed on the data lines. Valid remains high until the receiver of a port asserts an acknowledgement. Note that because sender and receiver are on opposite clocks, it is possible for the valid signal to remain high for multiple clock periods if the receiver can accept data every clock signal and assert the acknowledgement signal.
- **Data:** The data lines provide the next 32-bit data value when valid is asserted.