Credit-Based Flow Control Link

Christopher Fletcher
Hartej Dhami
UC Berkeley

May 23, 2008

Abstract

This report discusses the motivation and design for a credit-based flow control link to be integrated into the RAMP host system. First, we briefly introduce the channel model and what the link strives to accomplish. Next, we present a FIFO-based implementation for the link, along with the advantages and use guidelines that come with the design. Finally, we discuss future work on the link that can be undertaken to deal with efficiency issues in the design.

Contents

1 Introduction

In this section we will provide a brief overview of the RAMP channel model and introduce the link that was implemented in this project.

1.1 The Channel Model

Communication between units in a RAMP target system is accomplished through channels. Specifically, channels must provide a physical means of transport for messages, ordered/unidirectional delivery of messages, and support for the RAMP timing model [1]. Transport for messages and ordered send/receive capabilities are critical to maintain the message-passing behavior of the channel. Adherence to the timing model models allows for performance testing to be done using different 'flavors[1]' of channels.

[1] Channels differ in terms of their latency, bandwidth, and fragment capacity.

This project is concerned with the interaction between the channel, sending unit, and receiving unit. The sending unit passes messages to the channel and the channel passes the said messages to the receiving unit. Since the channel is unidirectional, the sending unit never becomes the receiving unit, and visa versa.

The channel has four parameters (BIT_WIDTH, FORWARD_LATENCY, BUFFER_SIZE, and BACKWARD_LATENCY)[2] that impact the channel timing model. Specifically, all messages passed by the sending unit to the channel are broken into BIT_WIDTH wide message fragments. Each fragment that enters the channel takes FORWARD_LATENCY target cycles to reach the receive side of the channel. When all of a message's fragments reach the receive side, they are reassembled into the original message. The original message is then stored until the receiving unit decides to read it (which may happen after an arbitrary amount of time, making the channel time-insensitive). When the receiving unit does indeed read the assembled message, the sending unit is notified after BACKWARD_LATENCY more target cycles. This notification is important because of the last parameter: BUFFER_SIZE. BUFFER_SIZE indicates how many message fragments can be sent without the sending unit receiving any notification. From the channel's perspective, BUFFER_SIZE represents capacity: once the link is "full" of fragments, a message must be read by the receiving unit (which leads, after BACKWARD_LATENCY target cycles, to the sending unit receiving notification) before any more messages can be sent by the sending unit.

1.2 Further reading

The above was a massive simplification of the RAMP channel and timing model. For a more detailed account of this material, please refer to [1].

1.3 The Link

Channels are entities in a target system, and are therefore not actual hardware. Instead, their behavior is mimicked by a member of the host system (which implements the target system as a whole) called the link. The link's implementation is not restricted, given that it performs the tasks stipulated by the channel model. We present one possible implementation of a link in this report, based on a credit-based flow control system (to model BUFFER_SIZE) and FIFOs (to handle buffering).

Credit-based flow control allows for a relatively simple implementation of the BUFFER_SIZE parameter used by the link. The link begins operation with a certain number of credits (determined by BUFFER_SIZE). With each fragment that enters the "channel" (the target system abstraction of the link), the credit count decrements by one. Whenever a message is read by the receiving unit, the credit count increments by however many message fragments composed the read message, after BACKWARD_LATENCY target cycles. When the credit count is zero, no messages may be sent by the sending unit. Hence, the BUFFER_SIZE functionality is realized with a simple accumulator, receiving delayed orders to increment or decrement.

The concept of delay in both passing fragments through the "channel" and sending read notifications from the receiving unit to the sending unit is accomplished through FIFOs. The FIFO latency parameter specifies the delay associated with each of these actions. Specifically, when a piece of data enters the FIFO, it cannot be read out for at least a certain number of cycles (as specified by the FORWARD_LATENCY and BACKWARD_LATENCY parameters, for passing fragments and read notifications, respectively). Because FIFO latency stipulates that the read head of the FIFO must always be a certain minimum distance behind the write head, the data can potentially get stuck in the FIFO if writes cease and reads continue. To remedy this problem, the FIFOs are sent an *idle fragment* during each cycle of operation regardless of whether or not they are to pass valid data for that cycle. This ensures that the FIFO never underflows (the case where writes cease and reads continue) and that message fragments/notifications do not ever get stuck in the FIFO, regardless of latency.

[2] As a shorthand for describing different specifications of the timing model parameters, we will henceforth describe them using tuples of the form <BIT_WIDTH, FORWARD_LATENCY, BUFFER_SIZE, BACKWARD_LATENCY>.

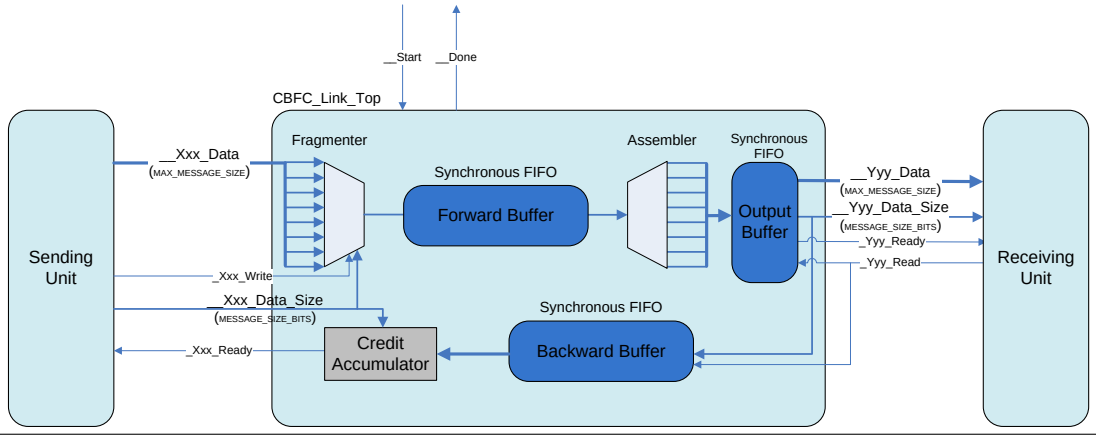
1.4 Decoupled clocks, __Start, and __Done

The link runs on the host (or physical) clock, but must simulate the timing model in target (or simulated) cycles. Each target cycle commences with the __Start signal being pulsed by an external wrapper, beyond the link's control. The target cycle can end when all units and links following that target cycle have asserted their __Done signals. This is an important point: the link proposed in this report has its own __Done signal. There is no restriction on when __Done must be asserted: it may be on the same cycle as __Start, or many cycles afterwards.

2 Design

In this section, we will go over detailed design for the entire link and the components that compose it. **In its present state, the link is coded, correctly transports messages, and abides by the RAMP timing model.** See figure 1 for a top-level block diagram of the entire link, and the basic interaction between the sub-modules that compose it.

Figure 1 CBFC_Link_Top I/O block diagram



2.1 CBFC_Link_Top

CBFC_Link_Top is the top-level module for the link. Table 1 shows the link's input/output specification. This interface conforms to that described in [1], but adds __Done, __Xxx_DataSize, and __Yyy_DataSize lines. The __Done signal is synonymous to a unit's __Done signal. The ...DataSize lines indicate an incoming/outgoing message's size in fragments. Message size is used to increment and decrement credits. The apparatus for fragment transport and credit backflow is shown in figure 1.

CBFC_Link_Top also allows the user to configure parameters to tune the timing model parameters and message bounds. The user-configurable parameters for the top-level module are shown in table 2.

Figure 2 shows normal operation for a <4, 2, 6, 1> configured channel handling 3-fragment messages. For those interested in passing 1-fragment messages with a foward_latency of 1, a $< 4, 1, X, 3 >$ configured channel is shown in figure 3, where X indicates that BUFFER_SIZE is large enough to ignore for the example.

2.2 Fragmenter

The [Message] Fragmenter is a parallel-in serial-out shift register that shifts an incoming message (least-significant fragment to most-significant fragment) on __Xxx_Data into the Forward Buffer FIFO.

Signal	Direction	Width	Description
Clock	I	1	The host clock
Reset	I	1	Resets the link
__Start	I	1	Indicates that the link is to perform one target cycle's worth of work
__Xxx_Write	I	1	Indicates that the data on __Xxx_Data is valid and forces the link to transport the message on __Xxx_Data
__Xxx_Data	I	MAX_MESSAGE_SIZE	The message at the input
__Xxx_Data_Size	I	MESSAGE_SIZE_BITS	The number of message fragments composing __Xxx_Data
__Yyy_Read	I	1	Indicates that the message at __Yyy_Data was read
__Done	O	1	Indicates that the link has finished one target cycle's worth of work
__Xxx_Ready	O	1	Indicates that the link is ready for a new message
__Yyy_Ready	O	1	Indicates that __Yyy_Data is valid
__Yyy_Data	O	MAX_MESSAGE_SIZE	The message at the output
__Yyy_Data_Size	O	MESSAGE_SIZE_BITS	The number of message fragments composing __Yyy_Data

Table 1: CBFC_Link_Top input/output table.

Parameter	Description
MAX_MESSAGE_SIZE	The width of the largest message that can be passed to the link (in bits)
BIT_WIDTH	Timing model parameter (see section 1.1)
FORWARD_LATENCY	" "
BACKWARD_LATENCY	" "
BUFFER_SIZE	" "

Table 2: CBFC_Link_Top user configurable parameters.

The serial-out width is given by BIT_WIDTH and is transparent to the first fragment. The Fragmenter has

Figure 2 Timing model: normal operation $< 4, 2, 6, 1 >$

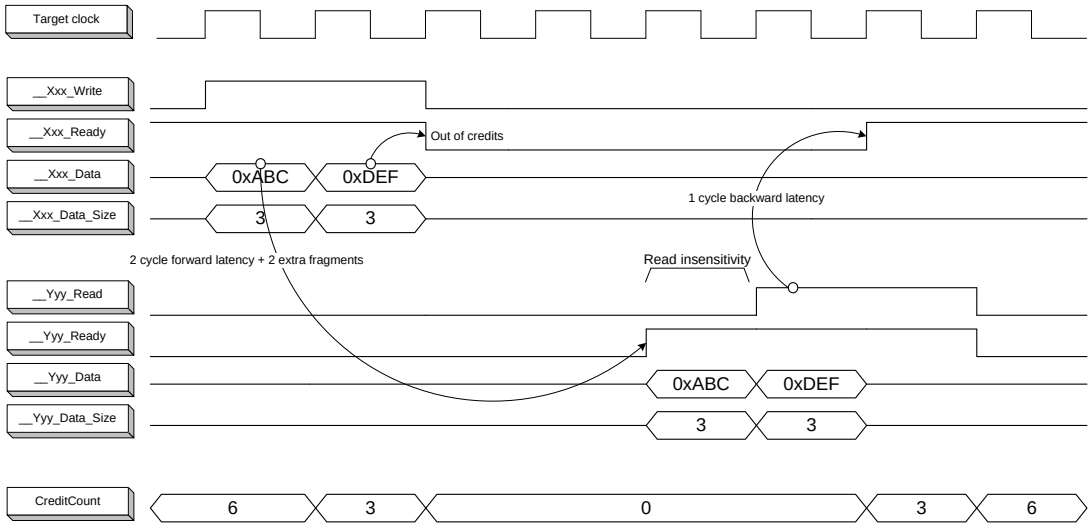
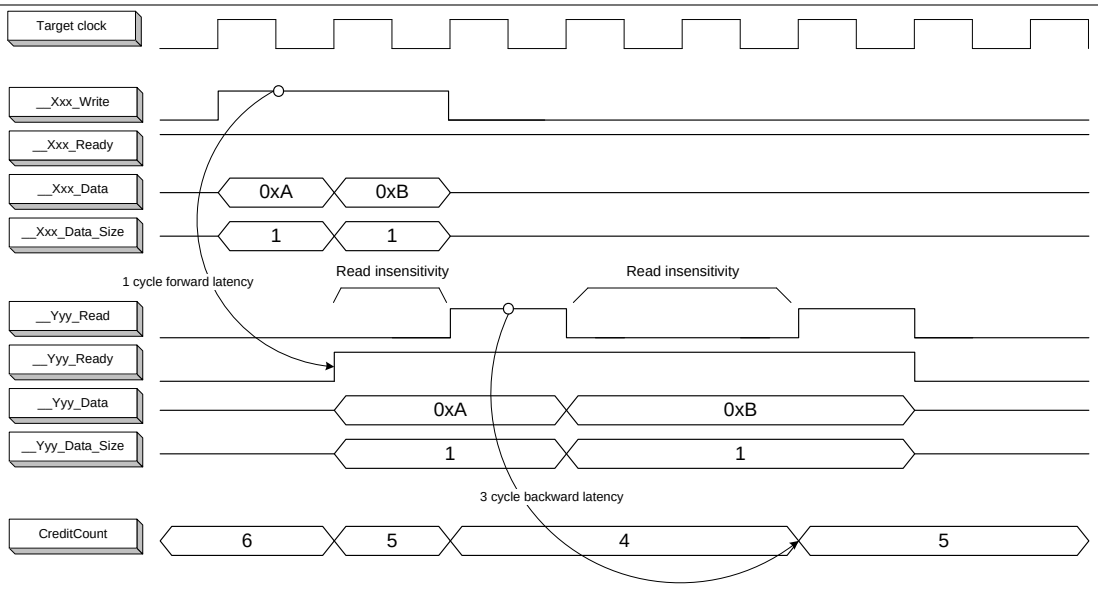


Figure 3 Timing model: 1 fragment messages $< 4, 1, X, 3 >$



a __Done signal that will not pulse until the Fragmenter has shifted the entire message that it is servicing into the Forward Buffer (which will take a number of host cycles equal to however many fragments compose the message). The Fragmenter input/output table is shown in table 3. A typical session for the Fragmenter is shown in figure 4.

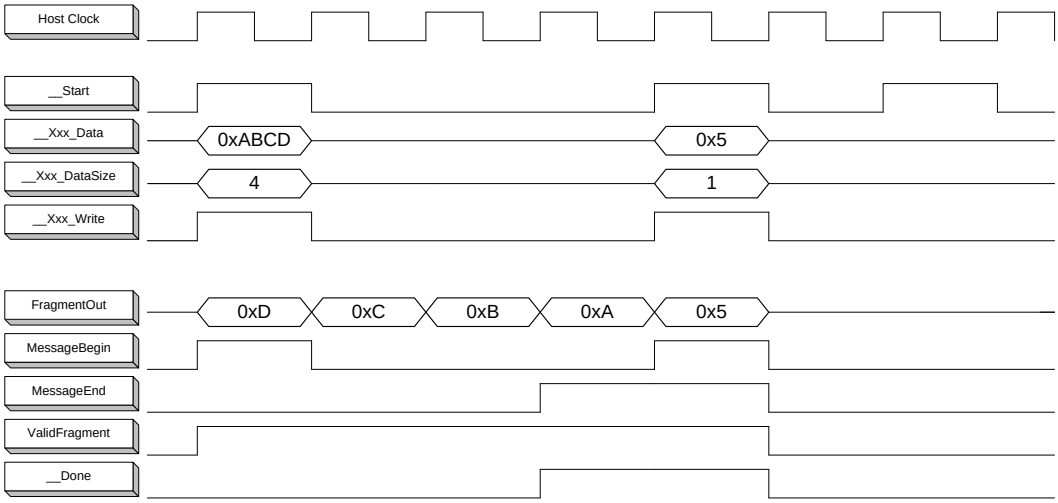
Signal	Direction	Width	Description
Clock	I	1	The host clock
Reset	I	1	Resets the fragmenter
__Start	I	1	Wrapper-controlled __Start signal
__Xxx_Data	I	MAX_MESSAGE_SIZE	Message to be passed out in fragments
__Xxx_Data_Size	I	MESSAGE_SIZE_BITS	Size (in fragments) of the message at the input
__Xxx_Write	I	1	The message at the input is valid
FragmentOut	O	BIT_WIDTH	The current fragment at the output
MessageBegin	O	1	FragmentOut is the LEAST-significant fragment
MessageEnd	O	1	FragmentOut is the MOST-significant fragment
ValidFragment	O	1	FragmentOut is valid
__Done	O	1	The fragmenter is done shifting out a message or is idling

Table 3: Fragmenter input/output table.

2.3 Forward Buffer

The Forward Buffer is a FIFO that provides the forward transport for message fragments from the send side to the receive side of the link. The Forward Buffer has latency equal to the FORWARD_LATENCY parameter. Regardless of writes and reads to the Forward Buffer from the Fragmenter and Assembler (see section 2.4), the Forward Buffer will always have a fragment written to it each host cycle. If the fragment is not a part of a message, it is considered an *idle fragment*. On the receive side, the Forward Buffer will either read out, and present to the Assembler, a valid fragment or read out an *idle fragment*. If the latter, the Forward Buffer will repeatedly read out fragments (without writing to the Assembler)

Figure 4 Timing: Message Fragmenter



until a valid fragment is at the output. This means that any *idle fragments* in the Forward Buffer are removed each target cycle. This maintains the timing model and allows consecutive messages to be written to the link on consecutive target cycles. Note that since the Forward Buffer may have to read out multiple *idle fragments* at a time, it also has a `__Done` signal (created artificially in `CBPC_Link_Top`, as the Forward Buffer is just a FIFO).

2.4 Assembler

The [Message] Assembler is a demux that will receive one or zero fragments per target cycle from the Forward Buffer (it will not process *idle fragments*). On the cycle that the a message's last fragment is received, the entire message is presented at the output of the Assembler. Since the Assembler only takes (at most) one fragment / cycle, it does not need a `__Done` signal. The Fragmenter input/output table is shown in table 4. A typical session for the Assembler is shown in figure 5.

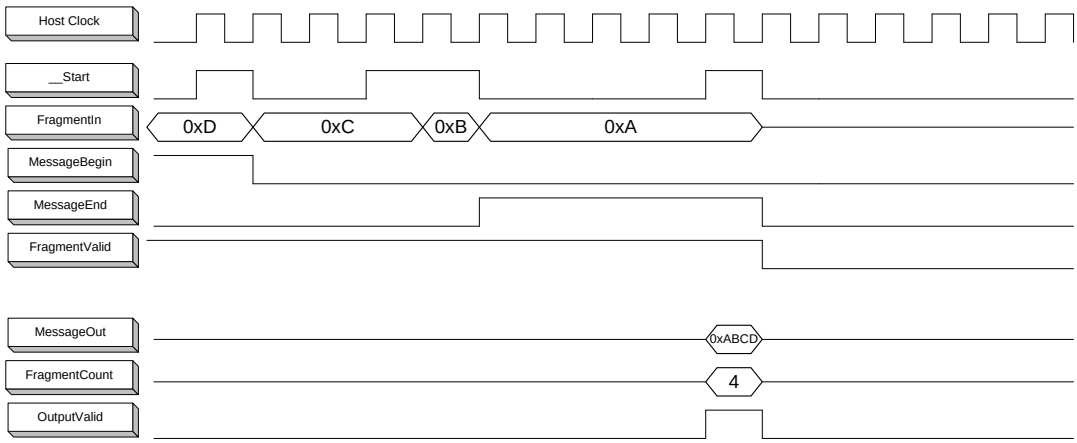
Signal	Direction	Width	Description
`Clock`	I	1	The host clock
`Reset`	I	1	Resets the assembler
__Start	I	1	Wrapper-controlled __Start signal
FragmentIn	I	`BIT_WIDTH`	The fragment at the input
MessageBegin	I	1	`FragmentIn` is the LEAST significant fragment of the message
MessageEnd	I	1	`FragmentIn` is the MOST significant fragment of the message
FragmentValid	I	1	`FragmentIn` is valid
MessageOut	O	`MAX_MESSAGE_SIZE`	The reformed message
FragmentCount	O	1	The number of fragments in `MessageOut`
OutputValid	O	1	`MessageOut` is valid

Table 4: Assembler input/output table.

2.5 Output Buffer

When messages are reassembled by the Assembler, they are stored in the Output Buffer. The Output Buffer is a zero latency FIFO that enables read-insensitivity on the receive side of the link. When messages are read by the receiving unit, they are read from the Output Buffer.

Figure 5 Timing: Message Assembler



2.6 Backward Buffer

The Backward Buffer is a FIFO that provides the credit backflow latency as specified by `BACKWARD_LATENCY`. If a message is read during a particular cycle on the receive side, the outgoing message's size in fragments will be written to the Backward Buffer. If nothing is read, an *idle fragment* will be written. On the read side of the Backward Buffer, the Credit Accumulator (see section 2.7) will increment by whatever value is on the output of the Backward Buffer (zero if the output is an *idle fragment*). The Backward Buffer does not need a `__Done` signal as it writes/reads exactly one value every target cycle.

2.7 Credit Accumulator

The Credit Accumulator records and updates the link's credit count each target cycle. This realizes the `BUFFER_SIZE` parameter. Since a message can be written to the link during the same target cycle that a message's size can be seen on the output of the Backward Buffer, the credit counter changes its value based on the difference of the number of credits it should increment by and the number of credits that it should decrement by.

3 Usage

Summarizing the earlier sections, there are two implementation specific protocols that must be followed in order for the link to function as intended:

1. The sending unit must present the link with all incoming messages' sizes (in number of fragments)[3]
2. The unit controlling `__Start` must listen for the link's `__Done` signal (which is pulsed) before issuing another `__Start`

4 Future Work

The Output Buffer is currently necessary because of a constraint on the link, namely that the link must be able to receive two messages on two consecutive target cycles. This forces the Fragmenter to push whole messages into the Forward Buffer in a single target cycle (as the Fragmenter must be ready to receive a new message on every target cycle). This makes the Fragmenter unable to accommodate the timing model, as it doesn't push one fragment into the Forward Buffer every target cycle. Instead, the Assembler handles the timing model, as it only every processes one fragment each target cycle. (This ensures, for example, that a 5 fragment message with 2 forward latency is ready on the receive side after $4 + 2 = 6$ cycles).

[3]The receiving unit gets the benefit of this additional information on the receiving side, whether it should ever need to use it or not.

The consequence to these design decisions is that the output of the Assembler must be stored in the Output Buffer until it is read by the receiving unit. **As the Output Buffer must be deep enough to accomodate a potential `BUFFER_SIZE` messages (for messages that are one fragment in size), it must be as wide as the `MAX_MESSAGE_SIZE` to accomodate the largest messages. As such, the Output Buffer is impractically memory intensive and cannot exist if the final link implementation is to be efficient in space.** In defense of its existance, if the Output Buffer wasn't present, then after a message was read from the Assembler (after an arbitrary amount of time, to accommodate the time insensitivity of the link), it would take a certain number of target cycles for the message before the first message to be re-assembled. Consider the case where two 5 fragment messages are written to the link (with 2 forward latency) in two consecutive target cycles. If, after many target cycles, the first message is read, the timing model stipulates that the next message be ready IMMEDIATELY (assuming that the first message is read after a long time). It won't be ready until 5 target cycles later, however, because the Assembler can only re-assemble one fragment per target cycle.

5 Acknowledgments

We would like to acknowledge Greg Gibeling for all of his advice and mentorship over the course of this project. Not only was Greg always responsive to questions, his utility modules are the most reliable around, and to no end make life easier than it would be otherwise.

We would also like to thank the rest of the RAMP Undergraduates who worked alongside us on similar projects throughout the semester. Through direct feedback and being a part of discussions, we progressed in our design a great deal.

References

[1] Greg Gibeling. RDLC2: The RAMP Model, Compiler & Description Language. Masters thesis, University of California Berkeley, 2008. **URL: http://ramp.eecs.berkeley.edu/Publications/Greg%20Gibeling%20Masters.pdf**