CFG IP Integration Specification Addendum – CModel

Version: 19.01

Revision: A.0

CFG AMBA IP Integration Specification: CModel Addendum

About this document

This document describes the CFG AMBA C++ Model and usage model where user can use as reference when integrating the model into their target system.

Audience

This document is intended for users of NocStudio:

- NoC architects, designers and verification engineers
- SoC architects, designers and verification engineers

Prerequisites

Before proceeding, you should generally understand:

- Basics of Network on Chip technology
- AMBA4 specification and C++ modeling technique

Related documents

The following documents can be used as a reference to this document.

- CFG NocStudio Orion AMBA User Manual
- CFG Orion IP Integration Specification
- CFG Orion Technical Reference Manual

Customer support

For technical support about this product and general information, contact CFG Support.

Revision History

Revision	Date	Updates
0.0	Jan 07, 2019	Initial Release
A.0	Feb 24, 2019	No change, revision update

Contents

List of Figures

List of Tables

1 C++ NoC Model Integration

The C++ NoC model consists of a dynamically linked library containing all functions of the model, and a header file containing the APIs that are used to run the model. The C++ model is named libnocstudio.so for Linux and nocstudio.lib for Windows. The C++ header files are named nocstudio.h.

1.1 How to Simulate Model

A number of APIs are defined in the nocstudio.h file. These APIs are used to simulate the model. During simulation, flits are injected into the model and ejected from the model at various bridge interfaces.

Flit Construction for Injection

A base NocFlit header class will be provided in the nocstudio.h file. Its attributes are as follows:

```
enum FlitPos { SopEop, Sop, Eop, Middle };

struct NocFlit_base {
   brif_t src;                  // Source bridge interface id
   int qos;                     // QoS
   FlitPos pos;                 // Position of flit in packet
   void* payload;               // Pointer to payload
};

struct NocFlit : NocFlit_base {
   brif_t dest;                 // Dest bridge interface id
};
struct MCNocFlit : NocFlit_base {
   std::vector<brif_t> dest;  // Vector of dest bridge interface id
};
```

A flit object of type NocFlit or MCNocFlit must be constructed for injection into the NoC, depending on whether the Flit is Unicast or Multicast. The NocFlit is unicast and has one destination, while the MCNocFlit supports multicast transmissions through the NoC, having a vector of destinations. Flit injection process must adhere to the protocol requirements of the interface. The FlitPos field determines the position of the flit in the packet; i.e. Start of Packet (Sop), End of Packet(Eop), Middle of packet (Neither Sop nor Eop) or single flit packet (SopEop).

The source and destination of the NocFlit/MCNocFlit are represented as brif_t (typedef int), encoding the bridge id and the interface id. The brif_t (bridge_interface ID) of a source or destination is derived as (bridge_id << 2) | interface_id.

Streaming bridge interfaces a, b, c, d are mapped to integers 0, 1, 2, 3, respectively.

The payload field is a generic (void) pointer that is disregarded by the internals of the model, and returned unchanged upon flit ejection.

1.1.1 Flit Injection

The following API is used for flit injection.

```
bool inject_flit(Sim* s, const NocFlit& flit,
                 string* error_string = NULL);
bool inject_flit(Sim* s, const MCNocFlit& flit,
                 string* error_string = NULL);
```

For an injection to be successful, the flit provided must match a valid traffic hop specified in the imported script file with which the model was configured. Flit injections must adhere to following protocol:

1. SOP/EOP logic
    a. A series of flits must be started with an SOP flit
    b. An EOP flit must come at some point after an SOP to complete the series
    c. The first injection from any interface must always be an SOP flit
2. Flits for different traffic flows cannot be interleaved when injecting into the same interface
3. Only one flit may be injected per interface per cycle

During each cycle, a flit may be injected at every input interface of all bridges of the NoC by calling the appropriate inject_flit() API depending on whether the injected flit is unicast or multicast. The injection API returns true if flit injection is successful, and false if injection fails. Injection may fail because of protocol errors, invalid message, or invalid fields in the flow. It may also fail if there is flow control asserted at the injection interface. In this case, the same flit may be injected again in the next cycle.

Table 1 Injection error cause and string values

Cause	Error message

Invalid src_h	Source bridge ID <bridge id> not valid.
Invalid dest_h	Destination bridge ID <bridge id> not valid.
Invalid src_i	Source interface ID <interface id> must be in range [0-3].
Invalid dest_i	Destination interface ID <interface id> must be in range [0-3].
Invalid src_br	Cannot get a valid bridge from source bridge ID <bridge id>.
Invalid dest_br	Cannot get a valid bridge from destination bridge ID <bridge id>.
Invalid src_ifce	Cannot get a valid transmitting interface from Bridge ID <bridge id> and Interface ID <interface id>.
Invalid dest_ifce	Cannot get a valid receiving interface from Bridge ID <bridge id> and Interface ID <interface id>.
Invalid src_br (regbus)	Source bridge is Regbus Master. Currently not supported.
Invalid dest_br (regbus)	Destination bridge is Regbus Master. Currently not supported.
Invalid message	Cannot create a valid packet. No traffic flows match flit attributes.
Invalid message	No packet is being sent; need SOP to start a new packet.
Invalid message	Current packet is still transmitting; need EOP to complete.
Invalid message	Flits are being interleaved. QoS and destination of flits from same packet must match.

Invalid message	Flit offset mismatch.
Flow control	Can inject only one flit per interface every cycle.
Flow control	Cannot inject; rate limit exceeded.
Flow control	Cannot inject; fifo full.
Flow control	Flow Control.
Other reasons	Failed to inject flit into the NoC.

The following API is used for credit return by the NoC.

```
bool set_credit_return_callback(Sim* s,
                                const brif_t src,
                                function<void(const brif_t src)> cb,
                                string* error_string = NULL);
```

set_credit_return_callback() sets the callback function to be called upon the transmitting bridge returning a credit to the host via the given interface. The NocFlit should be destroyed inside this callback function.

Table 2 Credit return callback setting error cause and string values

Cause	Error message
Invalid src bridge id	Source bridge ID <bridge id> not valid.
Invalid src ifce id	Source interface ID <bridge id> must be in range [0-3].
Invalid src_br	Tx bridge for Bridge ID <bridge id> not found.

Invalid src_ifce	Tx interface for Bridge ID <bridge id> and Interface ID <interface id> not found.
Undefined callback	Callback must be defined.

1.1.2 Flit Ejection

The following event-based API is used for flit ejection. Note: The original polling-based eject_flit() and can_eject_flit() APIs have been removed.

```
bool set_eject_callback(Sim* s, brif_t dest,
                              function<void(const NocFlit&)> cb,
                               string* error_string = NULL);
bool send_credit_rxif(Sim* s, brif_t dest,
                              string* error_string = NULL);
```

set_eject_flit_callback() sets the callback function to be called upon the receiving bridge receiving a flit at the given interface. When it is called, the ownership of the NocFlit returns to the user code. The callback function is responsible for calling send_credit_rxif() to indicate to the receiving bridge that it will be able to send one more packet to the host without overflowing the host. There is no MCNocFlit version of eject callbacks because only NocFlit is ejected.

The following properties hold:

1. Flits injected from a given source to a given destination arrive (are ejected) in the order that they were injected
2. As long as the payload pointers of all injected flits of a packet are consistent with each other (i.e. they all point to the same payload), the payload pointers of all ejected flits of the packet will be consistent with each other

Table 3 Flit ejection callback setting and credit issuing error cause and string values

Cause	Error message
Invalid dbridge	Destination bridge ID <bridge id> not valid.
Invalid difce	Destination interface ID <interface id> must be in range [0-3].

Invalid dest_br	Rx bridge for Bridge ID <bridge id> not found.
Invalid dest_ifce	Rx interface for Bridge ID <bridge id> and Interface ID <interface id> not found.
Undefined callback	Callback must be defined. (Only for set_eject_flit_callback())

1.1.3 Advance Clock

The following API is used to advance the internal clock of the NoC model.

```
void advance_time(Sim* s);
```

advance_time() advances the simulation time by one clock cycle, consuming all inputs of all blocks and producing outputs during that clock. The simulation is clocked at the highest clock frequency of all the clocks domains in the NoC.

1.2 How to Include Model

To use the NoC model, users must include the .h header file and link the static library in the project. Then the model must be configured with the same command script file that was used to generate the NoC IP in NocStudio. Once model is configured, the model can be simulated by using the flit injection and flit ejection APIs defined in the header files. The Flit data structure is also defined in the same library. The model is clock cycle aware; therefore, the clock must be advanced in the model cycle by cycle by calling the advance_time() API.

An example of how to include various model files and use the NoC model APIs is provided in APITest.cpp in the NocStudio release package.

1.3 How to Configure Model

To configure the model, a command script file must be created. This file must contain the same set of commands that were used to generate the NoC IP. Alternatively, the command log file generated in NocStudio can be used.

The command script must be scrubbed, as the NoC model does not allow standard NocStudio simulator commands such as run, cont, etc. Therefore, all such commands should be removed from the command script. If not removed, these commands will be ignored while configuring the NoC model. The command that generates NoC IP, gen_rtl, is ignored, as the model library is disabled from generating RTL and verification files. Currently, all commands that write to disk are disabled, including sim_stats, gen_image, create_trace_files, etc., and are ignored during model configuration.

Once the script file is ready, the following API is used to configure the model.

```
Sim* create_sim(istream& commands, string* error_string = NULL);
```

The function takes the input stream of the commands from the script file as an argument and returns a NULL pointer in case of error(s) in script processing, mapping of the NoC, or Sim creation, and writes a description of the error in error_string if the pointer is not NULL.

Table 4 NoC model configuration error causes and string values

Cause	Error message
Invalid commands in script	Error: Failed processing commands from input
Mesh cannot be created	Error: Cannot extract a NocStudio grid from Console; new_mesh not called?
Traffic is not mapped	Error: Cannot run sim unless traffic is mapped.

1.4 Simulation Performance

The NoC model features adjustable knobs that enable users to make tradeoffs between simulation speed and statistics data collection and error checking.

1.4.1 Performance statistics logging

The following API is used to set the performance statistics logging level.

```
void set_perf_logging(api_mode_stats_collection_level_t stat_level);
```

set_perf_logging() adjusts the level of logging performed by the simulator for purposes of performance statistics generation. Lower values consume fewer resources per simulation cycle, increasing simulation speed.

Table 5 Performance statistics logging levels

Level	Logging performed
LvlNone (default)	No logging
LvlLow	Includes interface and bridge level logging
LvlMedium	Includes channel buffer level logging w/ blocked causes
LvlHigh	Includes detailed logging of tx/rx times on each packet

1.4.2 Simulation safety checks

The following API is used to set the simulation safety check level.

```
void set_sim_safety_check_level(Sim* s,
                                api_mode_sim_safety_check_level_t level);
```

set_sim_safety_check_level() adjusts the extent of safety checks performed by simulation routines. Levels are per Sim object. Lower values consume fewer resources per flit injection, increasing simulation speed.

Table 6 Simulation safety check levels

Level	Checks performed
ChkNone (default)	No checking
ChkMedium	Includes flit injection flow control checks
ChkHigh	Includes protocol checks on injected flits: • Invalid source or destination bridge/interface ID • Source or destination bridge is a regbus master (currently not supported)

	• No traffic flows match flit attributes • Mismatch of QoS and/or destination between two flits of same packet • More than one flit injected per cycle • SOP of next packet sent before EOP of current packet • Flit begins a new packet, but is not SOP

1.5 UTILITIES

The NoC model also includes utilities for modifying the NoC and querying information.

1.5.1 Run commands

The following API is used to run arbitrary commands on the NoC, as if they were typed into the NocStudio console.

```
bool run_commands(Sim* s,
                  const vector<string>& cmds,
                  ostream& cmd_out = cout,
                  string* error_string = NULL);
```

run_commands() takes a vector of strings and runs them sequentially on the NoC of the given simulator. It also accepts an optional cmd_out parameter containing a reference to an output stream, to enable users to redirect the output of the commands to a file or a parser, for instance to inspect error messages to determine why a command failed. If no output stream is given, output is written to standard output cout.

It is intended for each set of commands to either:

- Not modify NoC construction, such as merely changing the FIFO depth of a link
- End with a map command after modifying the construction of the NoC by, for instance, adding a host, changing the skip_enable property of a router, or adding traffic

Consequently, it returns true only if all commands succeeded *and* the NoC remains in a 'Mapped' state after executing them.

Table 7 Run command error cause and string values

Cause	Error message
One or more commands failed	Command(s) failed; see standard output / 'cmd_out' stream for details.
Given set of commands modified NoC construction without re-running map	NoC is no longer in 'Mapped' state after running commands.

1.5.2 Query statistics

The following API is used to query the occupancy, throughput or latency of a channel.

```
struct EventStats {
        double minimum;         // Minimum value of the stat gathered
        double maximum;         // Maximum value of the stat gathered
        double average;         // Average value of the stat gathered
        double count;           // Number of samples of the event stat
        void merge(const EventStats& e); // To merge two stats
};


EventStats query_end_to_end_latency(Sim* s,
        brif_t src, brif_t dest,
        string* error_string = NULL);

EventStats query_stats(Sim* s,
        const string& channel_name,
        api_mode_stats_t type,
        string* error_string = NULL);
```

query_end_to_end_latency() is used to get the end to end latency of the messages from a source to a destination. If either or both of the src and dest are -1, then the function returns the event stats associated with all the src/dest.

query_stats()is used to get the occupancy, throughput or latency of a specified channel. The type of statistics to collect is controllable via the third argument to the function. The allowed types are Occupancy, Throughput and Latency.

Table 8 Statistics query error cause and string values

Cause	Error message
Invalid channel name	Invalid channel name <channel name> specified.
Channel name contains a wildcard	Specified channel name must match exactly 1 channel. Wildcards are not supported.

1.5.3 Reset stats

The following API is used to reset the stats collected on a channel(s).

```
bool reset_stats(Sim* s,
                 string& channel_name,
                 api_mode_stats_t type,
                 string* error_string = NULL);
```

reset_stats() allows users to reset the logging structures associated with the Occupancy, Throughput or Latency of channel(s). The channels can be VCs of routers or Interfaces of bridges, and can include wildcards. The function returns true if the resetting of stats was successful.

1.5.4 Performance simulator verbose logging to file

The following API is used to enable or disable the writing of detailed performance simulator messages to an output stream.

```
bool set_perf_sim_log_file(Sim* s,
                           bool en,
                           ostream* os,
                           string* error_string = NULL);
```

set_perf_sim_log_file() allows users to write the verbose output displayed in NocStudio console output upon issuing a step command to a file or an output stream instead. It accepts a Boolean value of whether to enable or disable logging, and a mandatory pointer to an output stream to which the messages will be written (if enabling).

The function returns false if enabling logging, but the given output stream is null or not in a "good" state.

1.6 Summary of NoC Model APIs

The NoC model uses the following commands, summarized below:

1. **create_sim**
    a. *input:* input stream of the script file
    b. *output:* pointer to model simulator
    c. *about:* creates the sim pointer from a script
2. **advance_time**
    a. *input:* simulator pointer
    b. *about:* runs the simulator for one clock cycle
3. **inject_flit**
    a. *input:* simulator pointer, NocFlit/MCNocFlit reference, payload
    b. *output:* bool value of successful injection
    c. *about:* injects a flit
4. **set_credit_return_callback**
    a. *input*: simulator pointer, source bridge_interface ID, pointer to user-implemented callback function
    b. *output*: bool value of successful setting of callback function
    c. *about*: sets the callback function that handles a credit issued by a transmitting bridge to a host
5. **set_eject_flit_callback**
    a. *input*: simulator pointer, destination bridge_interface ID, pointer to user-implemented callback function
    b. *output*: bool value of successful setting of callback function
    c. *about*: sets the callback function that handles an ejected flit and issues a credit
6. **send_credit_rxif**
    a. *input*: simulator pointer, destination bridge_interface ID
    b. *output*: bool value of successful issuing of credit
    c. *about*: to be called by user-implemented 'eject flit' callback function to issue a credit to the receiving bridge
7. **set_perf_logging**
    a. *input*: desired level
    b. *about*: sets the performance statistics logging level
8. **set_sim_safety_check_level**
    a. *input*: desired level
    b. *about*: sets the extent of safety checks performed by simulation routines
9. **run_commands**
    a. *input*: simulator pointer, vector of commands, output stream

   b. *output*: bool value of whether all commands succeeded and NoC remains in 'Mapped' state
   c. *about*: runs a list of commands on the NoC

10. **query_stats**
    a. *input*: simulator pointer, channel name as string, logging type
    b. *output*: EventStats containing the min, max, avg and number of events of the requested statistic
    c. *about*: returns the throughput, occupancy, or latency of a channel
11. **query_end_to_end_latency**
    a. *input*: simulator pointer, src bridge_interface ID, dest bridge_interface ID
    b. *output*: EventStats containing the min, max, avg and number of events of the requested statistic
    c. *about*: returns the end to end latency of the messages from source(s) to destinations(s)
12. **reset_stats**
    a. *input*: simulator pointer, channel name as string, logging type
    b. *output*: true if the reset of stats was successful
    c. *about*: resets the stat logger associated with the channel(s)
13. **set_perf_sim_log_file**
    a. *input*: simulator pointer, bool value of whether to enable or disable logging, output stream
    b. *output*: bool value of whether given output stream is good
    c. *about*: enables or disables writing of simulator `step` output to an output stream

1.7 AXI/ACE Modeling Extensions

AXI and ACE protocols can also be modeled using extended APIs for these protocols. This is accomplished in three parts:

1. the packet format is extended
2. The flow control is re-interpreted to match AXI behavior
3. Interface IDs are re-assigned to match ACE protocol

With these extensions, AXI and ACE protocols can be modeled using the infrastructure described above.

1.7.1 AXI/ACE Flit Format

The following code snippet describes AXI Flits. Note that it inherits from NocFlit_base, so in addition to these fields, there are also source, qos, pos and payload fields. The source field is used to determine both which bridge in the model sent this flit as well as the flit type, i.e. whether the flit is carried on AR, AWW, R, B, AC, etc. More details on interface_id ↔message type mappings in the AXI/ACE interfaces section <LINK HERE>. The qos field carries the AR/AWQOS value used to differentiate different traffic classes. All qos values from 0 to 15 are allowed. Pos is used the model's implementation AMBA protocols and must be correctly specified for each flit; use SopEop for single-flit requests like AR and B and a sequence of Sop, Middle, Middle, …, Eop for multi-beat AWW and R. The payload pointer can be populated in AMBA simulations, but its behavior will be different because of the presence of in-NoC agents like LLC or CCC. A request's payload pointer will only be received at the slave if no changes to the request were made by the NoC, otherwise a null pointer will be found in this field. When the NoC generates a response, this response will have the same payload pointer value as the request, and it is recommended for slaves to maintain this property so that requests and responses can be correlated.

```
typedef uint64_t Address_t;          //! memory address
         // AxiFlits will be forwarded to the appropriate dest based on
         address and security bits
struct AxiFlit : NocFlit_base {
  Address_t address = 0;             // can be omitted on responses
  bool priv = false;                 //!< true for privileged accesses
  bool non_secure = false;           //!< true for non-secure accesses
  bool instruction = false;          //!< true for instruction accesses
  int64_t aid = 0;                   //!< AxID of this packet, needed for
                                     both requests and responses
  int data_bytes = 0;                //!< number of bytes of data to read/write
  unsigned char* data_payload = nullptr; //!< where the slave should get/put
                                     the transaction data, must be valid up to
                                     data_payload[data_bytes-1]
  uint8_t blen = 1;                  //!< burst length: number of beats of data
                                     in this request
  bool decode_error = false;         //!< set to true on responses to indicate
                                     decode error
  bool coherent_access = false;      //!< true if this is a coherent request
  bool cache_access = false;         //!< true if this is a cachable request
  bool operator==(const AxiFlit&) const;
};
```

Instead of a destination, AxiFlits have an address, which is an unsigned 64-bit value indicating the byte offset of memory to read or write. The three following fields are "security" bits that are also used in destination lookup.

The aid field is used for ordering related requests; the model guarantees that responses to requests with the same aid and request interface (ar/aw) will be presented to the master in the same order as the requests; this may require serializing the requests or reordering the responses.

The data_bytes field indicates the number of bytes of data being requested and must be specified for both reads and writes. If the data payload is null, data will be ignored for this operation; no data will be modified or returned. Otherwise the data payload must point to at least data_bytes of memory to be read from or written to. The slave (or agent) processing this transaction will read or write data to this location to complete the request. This memory should be considered reserved for this transaction until the transaction is complete.

The blen field is currently unused.

The decode_error field must be left false on requests and will be set to true when a request's address does not indicate a destination in the master bridge's routing table.

The *_access fields are for coherent and cachable requests, and indicate whether the request needs/is allowed to go to an in-NoC CCC or LLC (or a customer-provided cache added with add_cache).

Ace Flits extend AxiFlit and have a number of additional fields:

```
enum class ace_commands_t {
   ReadNoSnoop, ReadOnce, ReadShared, ReadClean, ReadNotSharedDirty,
         ReadUnique, CleanUnique, MakeUnique,
   CleanShared, CleanInvalid, MakeInvalid, WriteNoSnoop, WriteUnique,
         WriteLineUnique, WriteClean, WriteBack, Evict, EvictFromDirectory
};
enum class ace_snoop_t { ReadOnce, ReadShared, ReadClean,
         ReadNotSharedDirty, ReadUnique, CleanShared, CleanInvalid,
         MakeInvalid };

//! when sending RACK/WACK into fabric, use these command types to
//! distinguish read from write (as same ifce is used for both)
const ace_commands_t RACK_CMD = ace_commands_t::ReadClean;
const ace_commands_t WACK_CMD = ace_commands_t::WriteClean;

//! AceFlits are used for snoop requests and ace requests and responses; they
         can only be sent to and from Ace Master bridges
struct AceFlit : AxiFlit {
   bool pass_dirty = false;   //!< sent as part of snoop response and as part of
                              read response to initiating ace master
   bool is_shared = false;    //!< sent as part of snoop response and as part of
                              read response to initiating ace master
   bool has_data = false;     //!< sent as part of snoop response
   ace_commands_t ace_command;          //!< sent as part of ace requests
                                        from master to interconnect;
                                        WACK and RACK use ReadClean
                                        and WriteClean commands to
                                        indicate read/write
   ace_snoop_t snoop_command;           //!< sent as part of snoop requests

   AceFlit(const AxiFlit& af) : AxiFlit(af) {}
   AceFlit(AxiFlit&& af) : AxiFlit(af) {}
   AceFlit() {}
};
```

The two big additions are the ace and snoop command fields, but there are also a few boolean fields used for snooping.

Requests coming from an ACE master must have the ace_command set to one of the enumeration values in ace_commands_t. These correspond to the various commands defined in the ACE spec. The src brif_t must have the correct ACE interface for the command being sent; it is not allowed to send, for example, ReadClean on the AWW interface. Other ACE fields of request messages are unused.

When a snoop is sent to an ACE master, a flit will arrive on the AXI_SN interface of that master with the snoop_command field set to a value from ace_snoop_t. These values correspond to the various snoop messages in the ACE spec. Other ACE fields of snoop messages are unused.

The response to a snoop message must have the pass_dirty, is_shared and has_data flags set to indicate these details of the snoop response. If there is data, multiple flits can be sent with appropriate pos values to carry that data back to the CCC.

The response to a read request will have valid pass_dirty and is_shared values to indicate the state of this cache line in the requesting master's cache. Other ACE fields of read response (and all ACE fields of write response) messages are unused.

After receiving a coherent response, a RACK/WACK message must be sent back to the CCC to complete the exchange. As both of these messages use a single interface, the ace_command field is given either the value RACK_CMD or WACK_CMD to indicate the type of the message.

1.7.2 AXI/ACE Flow Control

The AXI and ACE protocols use a VALID/READY protocol for flow control. This scheme uses a VALID signal sent by the transmitter to indicate that the data pins are currently valid. The received sends a READY signal to indicate if it is ready to receive data. At the end of each cycle, if both VALID and READY were asserted, the sender and receiver know that data transfer happened successfully and the next data transfer can happen in the next cycle. Because NSIP uses credit-based flow control, the flow control of the Modeling API follows this nomenclature, but is adapted in behavior for AXI/ACE interfaces to match VALID/READY semantics.

First, the number of credits for an AMBA interface is always 1; this means that until a credit is returned, no more data can be sent. Second, the data transfer is considered complete only when credit is returned. In the case that VALID and READY are both asserted in the same cycle, this means that the credit will be returned in the same cycle as the data was sent, allowing more data to be sent in the next cycle. In the case that READY is deasserted during a cycle that VALID is asserted, the valid/ready protocol requires the same data to be presented on the interface until READY is asserted. This is modeled by data being injected, but it will not make progress through the NoC model until READY is sent back to the injector in the form of a credit. After data is injected, further data cannot be injected until READY is asserted (i.e. a credit is returned on that interface).

In this translation, it is not possible to assert READY before VALID, but this can be modeled equivalently by asserting READY (sending a credit) immediately after VALID (inject_flit).

1.7.3 AXI/ACE Interfaces

To indicate a source or destination id, the concept of brif is extended to give identifiers to AXI and ACE interfaces as follows:

Integer	AxiM Tx ifce AxiS Rx ifce	AxiM Rx ifce AxiS Tx ifce	AceM Tx ifce	AceM Rx ifce
0	AR		AR	
1	AWW		AWW	SN
2		R	SN RESP	R
3		B	ACK	B

To aid in programming against this interface, constants are provided to give names to these values.

```
// Axi/Ace bridge interfaces are mapped to integers according to the
        following:
const int AXI_AR = 0;            // tx on master, rx on slave
const int AXI_AWW = 1;           // tx on master, rx on slave
const int AXI_R = 2;             // rx on master, tx on slave
const int AXI_B = 3;             // rx on master, tx on slave
const int AXI_SN = 1;            // rx on master
const int AXI_SN_RESP = 2;       // tx on master
const int AXI_ACK = 3;           // tx on master
```

1.7.4 Channel Specific Implementation Notes

The AXI AW and W channels are represented in the model as a single interface, AWW. This interface transports both address and data, modeled by having the first injected flit be the AW data and all following flits be data (i.e. W). Injecting single-flit write requests is not supported.

Note: Under some rare circumstances, the NOC may present a single-flit write request and this is considered a flaw and will be fixed in a future release.

The AXI protocol permits interleaved R responses from slaves, where data/response for a different transaction can be sent before all data for the first transaction was complete (RLAST). In current release, the modeling API expects responses to be presented to the model in a contiguous fashion.

Note: When the slave host model fails to comply, the modeling API behavior is undetermined.

2200 Mission College Blvd
Santa Clara, CA 95054
www.intel.com