# TLM Extensions

**Adam Rose**
**Stuart Swan**
**John Pierce**
**April 2004**

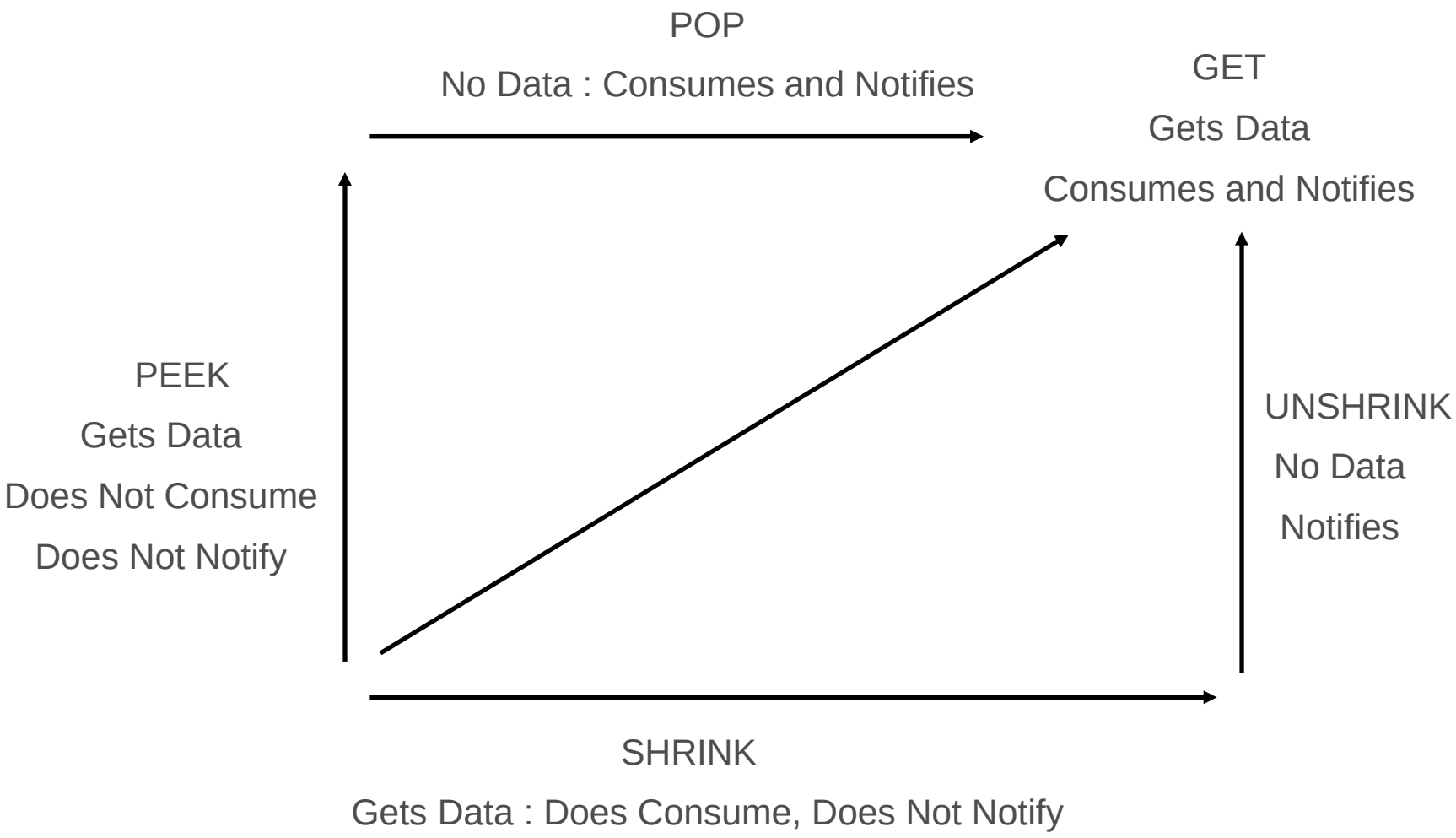
# Motivation

- The are various non standard ways that people want to access tlm_fifo
  - usually to model close-to-cycle-accurate behaviour of buses.
- The most important example of this is the OCP moded read / release mechanism
  - An important requirement is that tlm_req_rsp_channel ends up replicating all of the functionality of the OCP Level 2 channel.
- Within OSCI, similar issues have been raised
  - The need to peek at data without reading it
  - The need to have a fifo whose initial state is both full and empty
  - The need to write over old data before it's been read
  - etc
- Infinite sized fifos
  - For Modelling, we nearly always require finite fifos
  - But for Verification we nearly always need infinite fifos

# The Existing Interfaces


- Put does three things ( assuming it's not full )
  - It moves data into the fifo
  - It queues data
  - It notifies ok_to_get()
- Get does three things ( assuming it's not empty )
  - It gets data out of the fifo
  - It consumes the data
  - It notifies ok_to_put()
- The Moded interfaces separate out these three basic features of put and get

# Two New Routes to Get



# Use cases for moded gets

- Typical use for peek / pop is multiple slave decode
  - All slaves peek at transaction
    - Because we're not consuming the transaction, all slaves will see the same transaction
  - The slave that successfully decodes the transaction issues a pop
    - See decode_pop example
- Typical use for shrink / unshrink is to allow slave to control master timing
  - Slaves consumes the data with shrink
    - Slave processes transaction as it would for get()
    - Successive calls to shrink work their way through the transactions in the fifo. Each call to shrink gets a different transaction.
  - Slave then notifies ok_to_put at a time of its choosing with unshrink
    - Master thinks get happened later than it actually did
    - See timed_unshrink example
  - Can use dataless nb_unshrink() to give slave full control of timing
    - See preshrink example

# Moded Puts

- Only one new mode for put – overwrite
- Overwrite notifies but does not queue
  - Intent is to allow master to write over old data even if slave has not read it.
    - Transaction most recently written is the one that is overwritten
  - A fifo size one with overwrite on one side and peek on the other is equivalent to an sc_signal
    - See clocked_overwrite_peek example
- A put which queues but does not notify has NOT been implemented
  - The implementation for a fifo size n is quite painful
  - A possible use for this is for PVT !

# Moded Put and Get Interfaces


```
enum tlm_get_type {
  NORMAL_GET ,                 // consumes and notifies
  SHRINK ,                     // consumes but doesn't notify
  PEEK ,                       // neither consumes nor notifies
};

enum tlm_finish_get_type {
  UNSHRINK ,   // notifies
  POP          // consumes and notifies
};

template < typename T >
class tlm_moded_get_if : public virtual sc_interface
{
public:
  virtual bool get( T & , tlm_get_type ) = 0;
  virtual bool nb_get( T & , tlm_get_type ) = 0;

  virtual bool notify_got( tlm_finish_get_type ) = 0;
  virtual bool nb_notify_got( tlm_finish_get_type ) = 0;
  virtual bool nb_notify_got( const sc_time & , tlm_finish_get_type ) = 0;
};
```

```
enum tlm_put_type {
  NORMAL_PUT ,
  OVERWRITE
};

template < typename T >
class tlm_moded_put_if : public virtual sc_interface
{
public:
  virtual bool put( const T & , tlm_put_type ) = 0;
  virtual bool nb_put( const T & , tlm_put_type ) = 0;
};
```

## Notes

All methods, including blocking methods, return a bool

- Because some things just don't make sense eg
  - A shrink on a fifo size zero
  - An overwriting put when there's no data in the fifo
- Because implementations other than tlm_fifo may not implement all modes

nb_notify_got obeys event semantics in each mode

- an earlier pop cancels a later pop
- A pop now cancels any outstanding pops
- Same for unshrink
- But shrink events and pop events have no effect on each other

# Convenient combined interfaces

```
template < typename T >
class tlm_blocking_extended_put_if :
  public virtual tlm_blocking_put_if<T> ,
  public virtual tlm_blocking_moded_put_if<T>
{
  tlm_blocking_put_if<T>::put;
  tlm_blocking_moded_put_if<T>::put;
};

template < typename T >
class tlm_nonblocking_extended_put_if :
  public virtual tlm_nonblocking_put_if<T> ,
  public virtual tlm_nonblocking_moded_put_if<T>
{
  tlm_nonblocking_put_if<T>::nb_put;
  tlm_nonblocking_moded_put_if<T>::nb_put;
};

template < typename T >
class tlm_extended_put_if :
  public virtual tlm_blocking_extended_put_if<T> ,
  public virtual tlm_nonblocking_extended_put_if<T>
{};
```

```
template < typename REQ , typename RSP >
class tlm_master_if :
  public virtual tlm_extended_put_if< REQ > ,
  public virtual tlm_extended_get_if< RSP > {};

template < typename REQ , typename RSP >
class tlm_slave_if :
  public virtual tlm_extended_put_if< RSP > ,
  public virtual tlm_extended_get_if< REQ > {};
```

- The **extended** interfaces combine the moded interfaces and the normal interfaces
  - and resolve some name clashes
- The **master** interface is
  - an extended put request interface
  - and an extended get response interface
- The **slave** interface is
  - an extended get request interface
  - and an extended put response interface

# Fifo Specific Interfaces - Debug Interface

- used() and size() are self explanatory
- nb_peek( T &t , int n ) provides debug access to any element in the fifo
  - No notification, consumption or delta delay
- nb_poke( const T &, int n ) allows write access to any element in the fifo
  - No notification, queuing or delta delay
  - Very useful for deliberate injection of faults in verification
- nb_shrink() is used to pre-shrink a fifo
  - This disables puts until the slave does an unshrink.

# Fifo Specific Interfaces - Size


- A negative size parameter in the constructor creates an infinite fifo
  - The absolute value of the size parameter determines the initial size of the circular buffer
  - If required, this buffer will be dynamically resized
- There are a number of resize functions implemented in tlm_fifo but not currently made available in any sc_interface
  - nb_expand expands a finite fifo to any finite size
  - nb_reduce reduces the size of a finite fifo
  - nb_unbound makes a fifo infinite
  - nb_bound makes an infinite fifo finite
- These functions could be made available in the debug interface or a separate resize interface if required although :
  - Any user can do this in a subclass if they want
  - The functions can already be accessed via the public interface of tlm_fifo

# TLM Unidirectional Interfaces