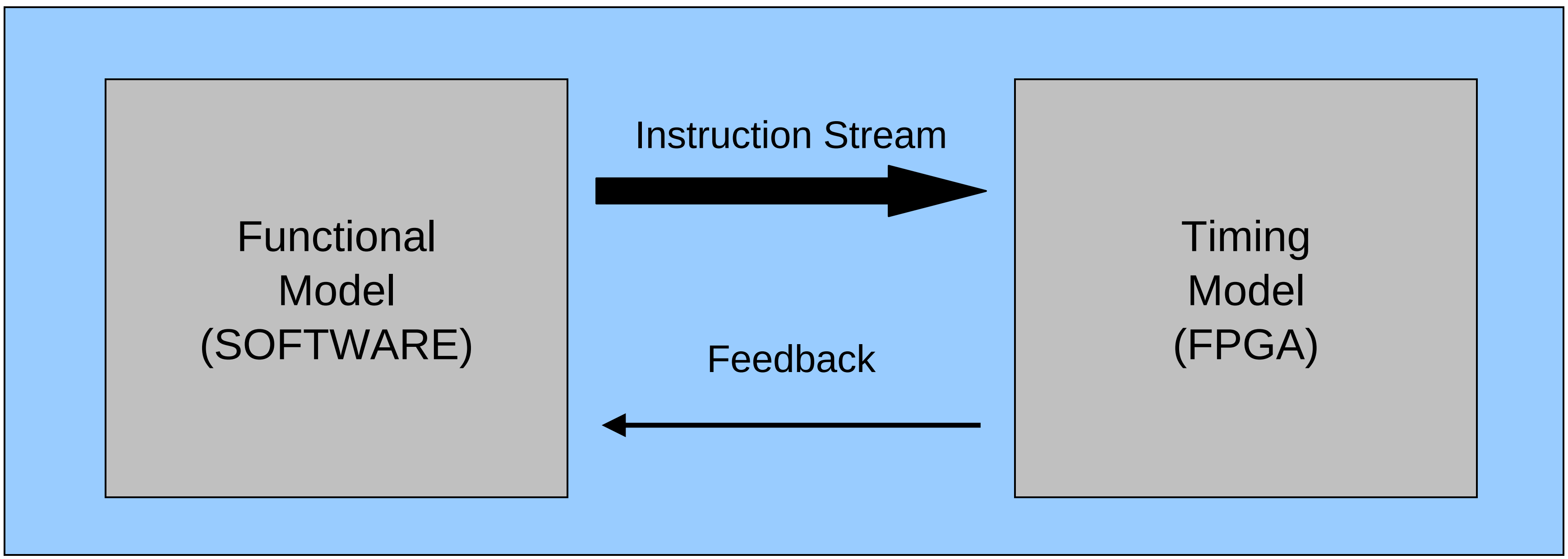
Design and Implementation of FAST Module Connectors

Bill Reinhart
UT-FAST Group
University of Texas

FAST Methodology

- FAST
  - Functional Model partition
  - Timing Model partition



The Design Challenge

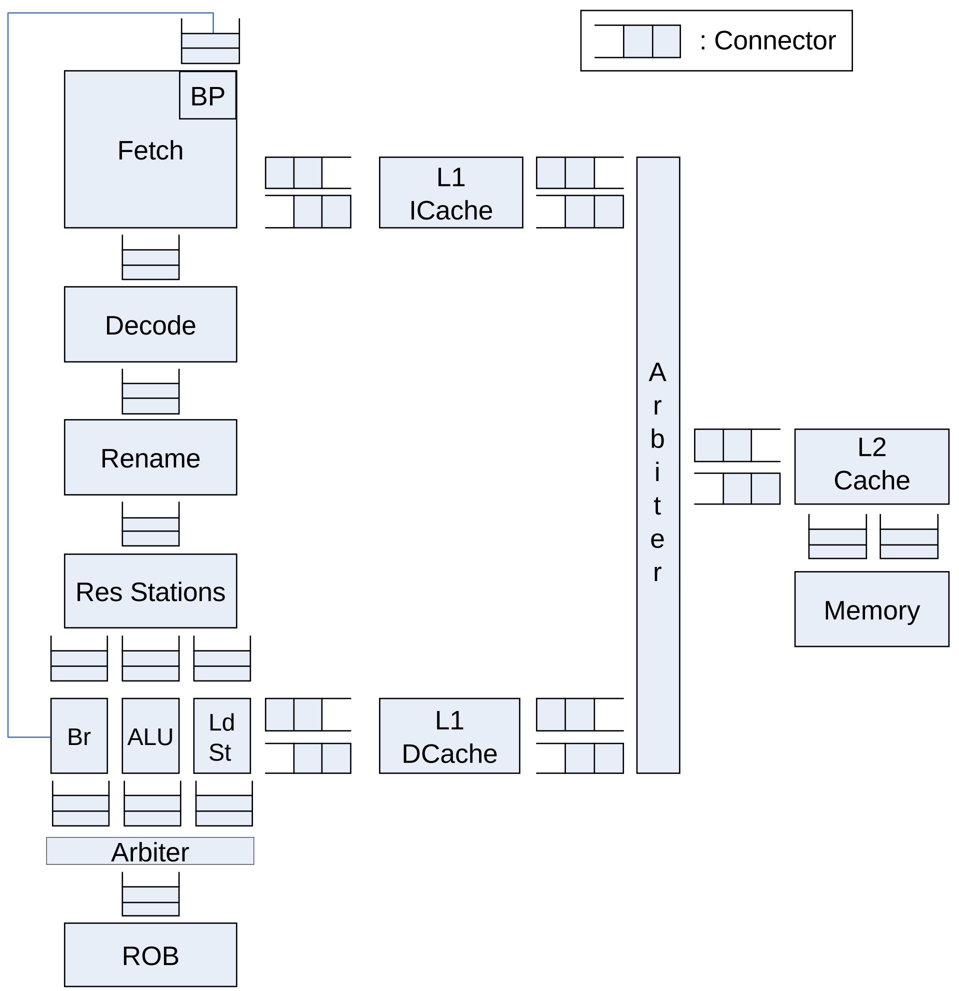
- Timing model complexity
- Reduced development time
- Relationship between Host Cycles and Target Cycles
- Statistic and Tracing Integration

➢ **FAST Connectors**

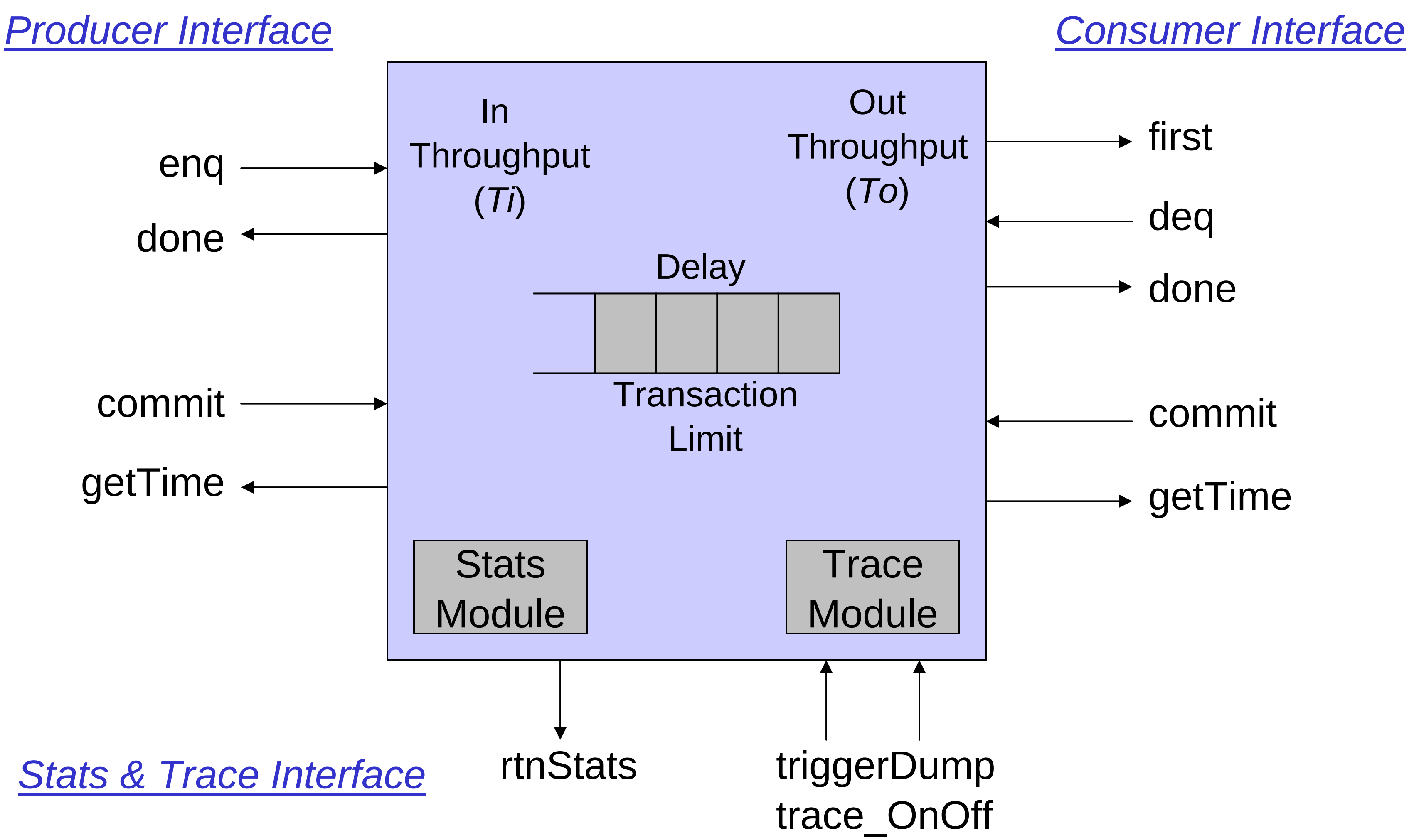
Influenced by Asim Ports and discussions with Joel and Michael

FAST Timing Model

- FAST Timing Model
  - Modules model behavior
  - Connectors abstract performance information from modules
- Connectors
  - Configurable interfaces
  - Time
    - Delay
    - Throughput
    - Outstanding Transactions
  - Stats and Tracing



FAST Connector

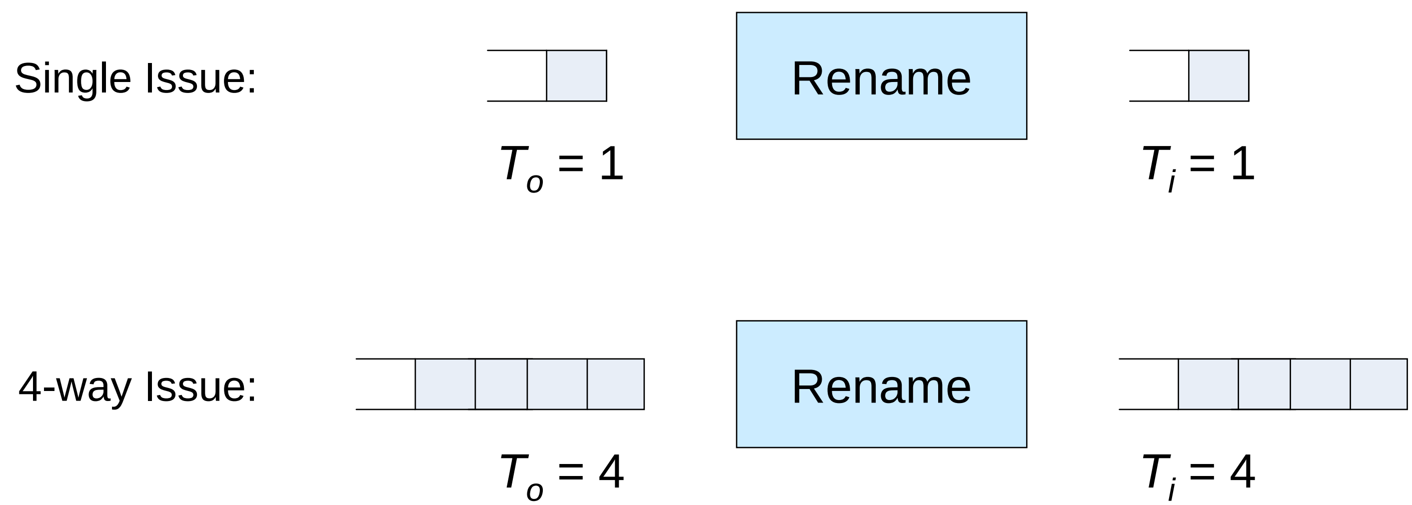


Leveraging Bluespec

- **Parameters**. Enable connector design to be reused through the design
- **Arguments**. Enable compile-time targeting/allocation of FPGA resources
- **Function Passing**. Enable Statistics and Tracing to be easily customized for each instantiation
- **Building Blocks**. Complex mechanism designed from little more than a FIFO and handful of registers
- **Conditional Execution**. Simple to enforce simulation correctness

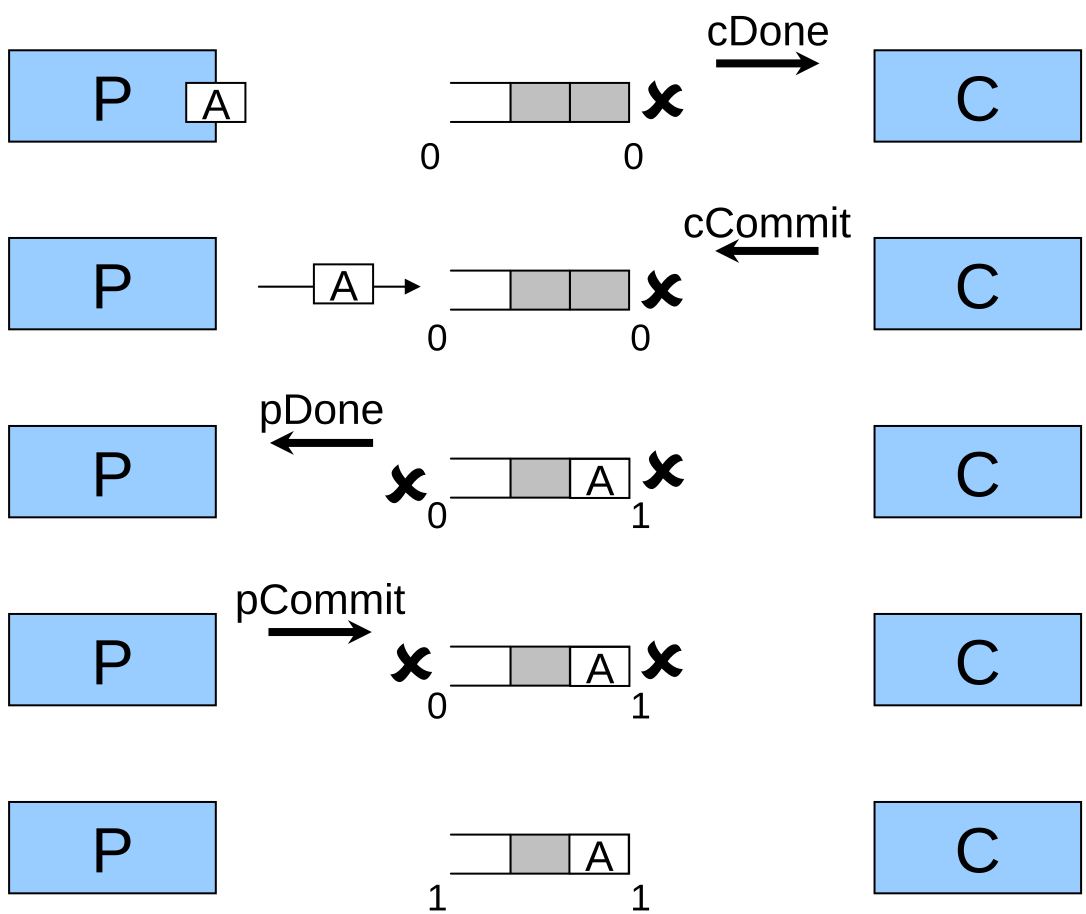
Example : Rename

- Rename module written to handle one register rename at a time
- Connectors enable the same rename module to model different architectures



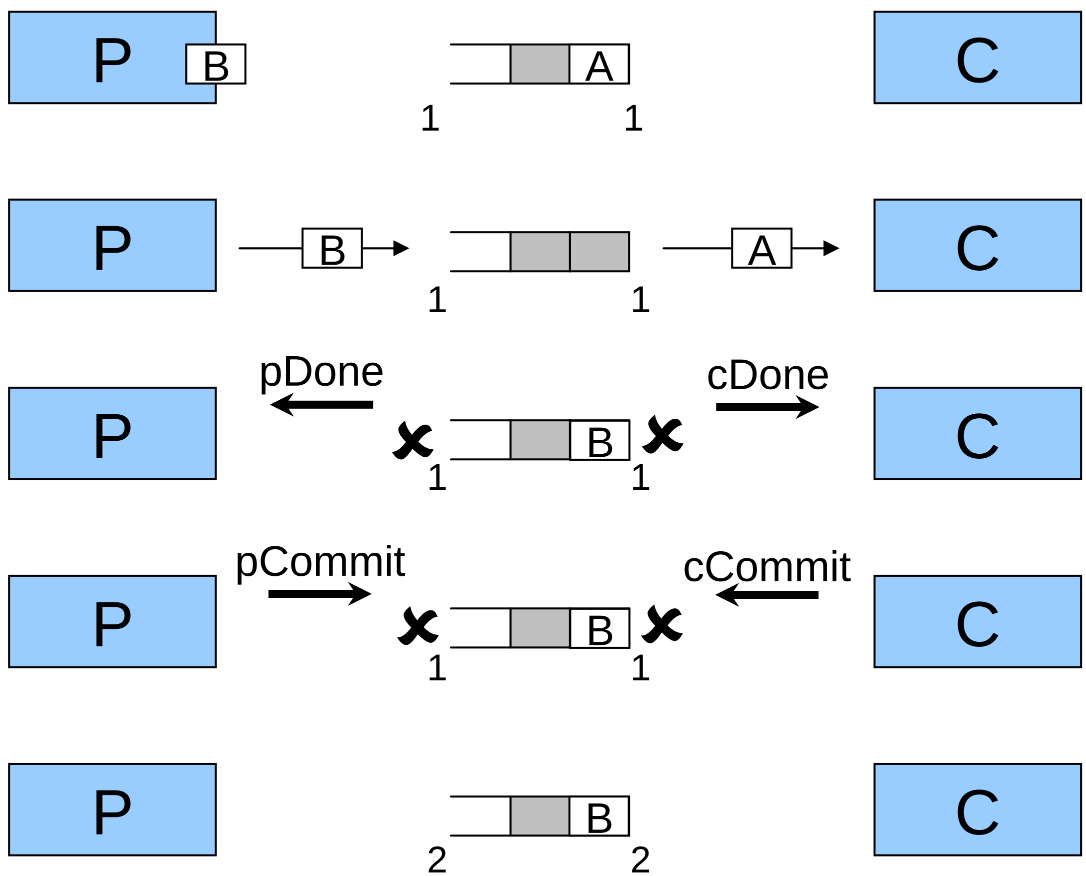
Simple Example

- $T_i = 1$, $T_o = 1$, $D = 1$, Trans = 2



Simple Example

- $T_i = 1$, $T_o = 1$, $D = 1$, Trans = 2



Stats & Tracing

- Connectors used everywhere
  - Factor in common functionality
- Statistics
  - Stats module passed into the connector
    - configured via a function for selective stats collection
- Trace
  - Buffer maintains data past its transferal through the connector
  - Log = all data, Trace = selective instructions
  - Dump of saved data can be event triggered or continuous
  - Triggered dump provides a record of events that precede the trigger event

Our Challenges

- RWires versus registers
- Need *Integer* to *Numeric Type* conversion
- Connections are specified by passing interfaces
  - Flattens the hierarchy
  - Prevents incremental synthesis
  - Move to soft connections
- Enable efficient Routing on FPGA
  - Statistics and Trace Information
  - Desire a single-point of communication outside of FPGA (possible to push back through connectors)
- *Pack* / *Unpack* for C – Bluespec conversion

FAST Connector Conclusions

- Simplifies modules by
  - Decoupling timing from behavior
  - Enables the sequential modeling of parallel actions
- Automatically introduces configurability
  - Throughput
  - Delay
  - Outstanding events
- Can relax lock-step behavior between modules
- Facilitates debug and analysis enabling statistics and trace capture

BACK UP SLIDES

FAST Connector Benefits

? **Backup** (this is mostly covered in "Design Challenge")

- Simplifies modules by
  - decoupling timing from behavior
  - ENABLES THE sequential modeling of parallel actions
- Increases flexibility of the design and enable design exploration by enabling variable delay, to include 0-delay (via a bypassed register), and variable throughput
- Increases simulation speed by removing the need for global synchronization between data processing and transferal between modules
- Facilitates debug and analysis enabling statistics and trace capture

Synchronizing Data Flow

- Normal Operation
  - connector counts ENQs (DEQs)
  - asserts '*done*' to the module and blocks buffer access if T_i (T_o) is reached;
  - on receipt of '*commit*' from the modules, increments time and unblocks buffer access
- Zero data flow modeled by …
  - Connector asserting '*done*' without enabling ENQ/DEQ
  - Module asserting '*commit*' without ENQing/DEQing
  - The connector uses the enqueued count (or time stamps) in lieu of explicit "null" messages to maintain target-cycle synchronization

Connector Code
(Lockstep, Fully Buffered Implementation)

```
module mkConnect_lockstepFB#(…
…
// internal components
   FIFOF#(Bit#(counterSize_t)) enqCntFIFO <- mkSizedFIFOF(delay);
   ConnectFIFO#(data_t, Bit#(bufferIndex_t)) buffer <- mkConnectFIFO(mem, bufferSize);
   Reg#(Bool)                        pCmt <- mkReg(False);
   Reg#(Bool)                        cCmt <- mkReg(False);
   Reg#(Bit#(counterSize_t))         toEnq <- mkReg(fromInteger(enqNum));
   Reg#(Bit#(counterSize_t))         toDeq <- mkReg(0);
   Reg#(Bit#(counterSize_t))         enqCount <- mkReg(0);
   Reg#(Bit#(counterSize_t))         deqCount <- mkReg(0);
   Reg#(Bit#(counterSize_t))         transCount <- mkReg(0);
   Reg#(UInt#(TimeWidth))            tgtTime <- mkReg(0);
   Reg#(Bit#(TLog#(MaxDelay)))       delayTime <- mkReg(fromInteger(delay));

// internal variables
  let blockENQ = (pCmt || toEnq==0);
  let blockDEQ = (cCmt || deqCount==fromInteger(deqNum) || toDeq==0);
  let blockPcmt = pCmt;
  let blockCcmt = cCmt;
  let pDone = ( !pCmt && toEnq==0);
  let cDone = ( !cCmt &&  ( deqCount==fromInteger(deqNum) || toDeq==0 ) );
```

Connector Code (cont)

```
rule cycleEndProcessing (init && pCmt && cCmt);
    // compute values for next cycle
        let ntransCount = transCount + enqCount - deqCount;
        let nEnq =  min ( fromInteger(enqNum),
                                   (fromInteger(bufferSize)-ntransCount) );
         if (transLimit!=0)
                nEnq = min(nEnq, (fromInteger(transLimit)-ntransCount));
         if (nEnq < fromInteger(enqNum))
                nEnq = 0;
        enqCntFIFO.enq(enqCount);
        let addnDeq = 0;
        if (delayTime>0) begin
                delayTime <= delayTime - 1;
                if (delayTime==1)  enqCntFIFO.deq();
        end else begin
                addnDeq = enqCntFIFO.first();
                enqCntFIFO.deq();
        end

endrule
```

```
// assign register values
        tgtTime <= tgtTime +1;
        transCount <= ntransCount;
        toEnq <= nEnq;
        toDeq <= toDeq + addnDeq;
        pCmt <= False;
        cCmt <= False;
        enqCount <= 0;
        deqCount <= 0;
```

Connector Code (cont)

```
interface ProducerPort p;

  method Action enq(data) if (init && !blockENQ);
        buffer.enq(data);
        toEnq <= toEnq - 1;
        enqCount <= enqCount + 1;
        stats <= data;
  endmethod
  method Action commit () if (init && !blockPcmt);
        pCmt <= True;
  endmethod
  method Bool done = pDone;
  method UInt#(TimeWidth) getTime = tgtTime;

endinterface
```

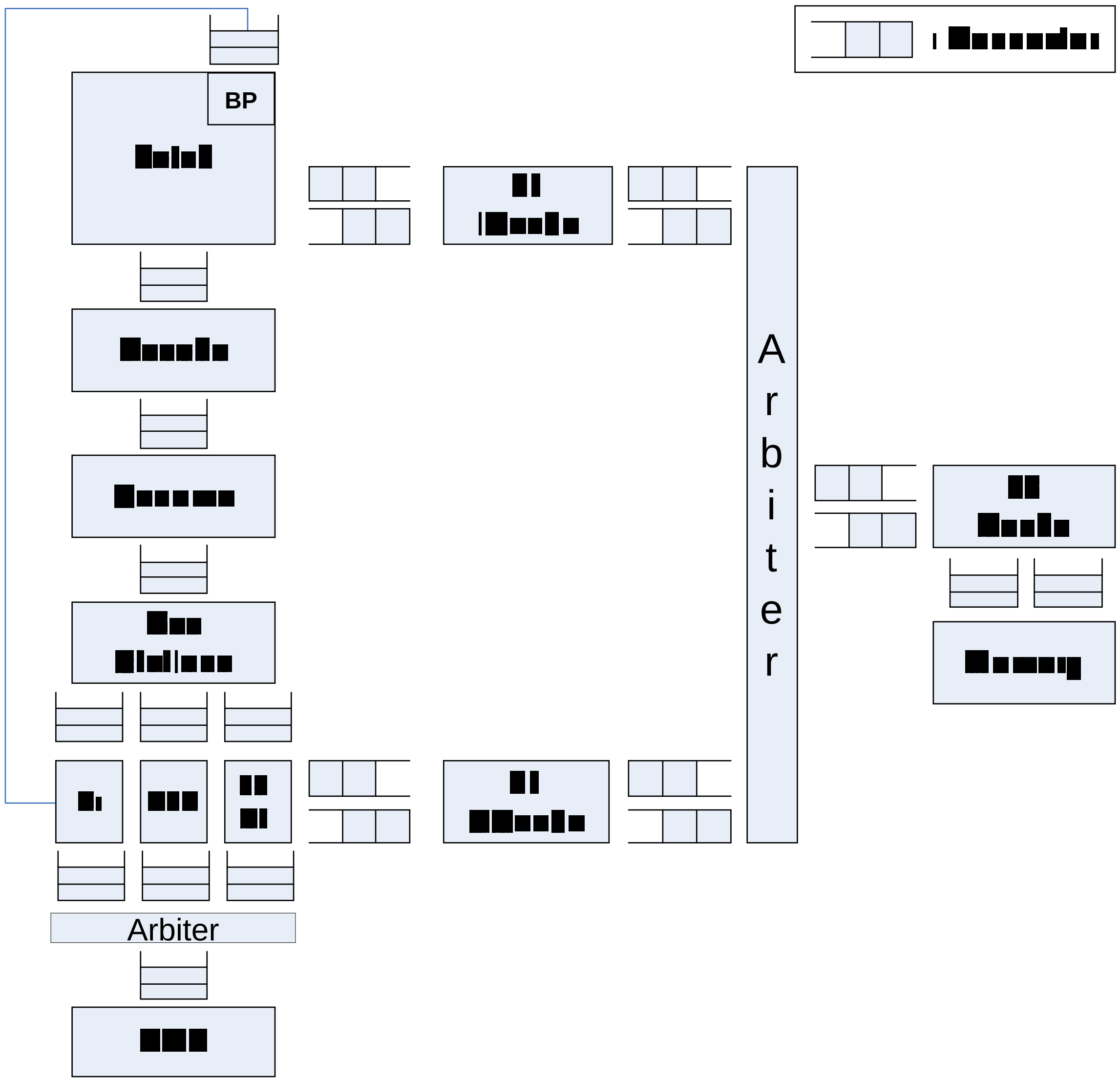
```
interface ConsumerPort c;

   method Action deq() if (init && !blockDEQ);
        buffer.deq();
        toDeq <= toDeq - 1;
        deqCount <= deqCount + 1;
   endmethod
   method data_t first() if (init && !blockDEQ);
        return buffer.first();
   endmethod
   method Action commit () if (init && !blockCcmt);
        cCmt <= True;
   endmethod
   method Bool dataReady = ( init && !blockDEQ );

   method Bool done = cDone;
    method UInt#(TimeWidth) getTime = tgtTime;

   endinterface
```

Example FAST Architecture



Modeling Time

- Two target-cycle times are LOGICALLY maintained, one at each end of the connector
- Decoupling the logical time at each end of the connector is possible to enable modules to advance as soon as data is ready
- Target-cycle times are incremented upon receipt of ‘*commit*’ thus enabling multiple host-cycles to be used in modeling the behavior