Overview

- Introduction - Harry
- Language - Yaron
- Methodology - Erich
- Break
- Tools - Erich
- Protocol Modeling - Bernard
- Conclusion - Harry
Protocol Modeling

• Accurate communication between design units is vital
  – historically it’s the cause of many ‘bugs’
  – true for internal protocols as well as external standards like PCI
• Examples presented earlier stress properties for verification of protocol fragments
  – checking the response to an AHB BUSY or IDLE cycle is always ‘zero-wait-state’
  – checking a target never inserts more than 16 ‘not-ready’ cycles
• We will examine a different approach to protocol modeling.
  – If you describe all the legal behaviors, by inference anything that doesn’t fit the pattern must be a fault
Conventional Protocol Description

• Traditionally described with waveform diagrams and ‘English’ explanations
• Consider the AHB protocol:
  – Section 3.9.4 says “If a slave provides an ERROR response then the master may choose to cancel the remaining transfers in the burst. However, this is not a strict requirement and it is also acceptable for the master to continue the remaining transfers in the burst.”
• The AHB Specification doesn’t contain waveforms for either alternative of this complex behavior
  – probably because of limited space – the combinations grow fast!
  – this is not unique to ARM – all protocol specs are limited to a few ‘representative’ diagrams!
• The door is open to ‘bugs’
  – teams designing blocks have different interpretations of the protocol
AHB Burst Termination Example?

What happens to this address if the burst terminates on ERROR? Is it used? If not, what transaction is performed during the next data phase?

I assume this is changed to IDLE to invalidate the address, but it’s easy to make assumptions....
Compare This with PSL/Sugar

- The languages we use are standardized using BNF descriptions
- BNF’s are effective because they show every legal possibility at every stage in a sequence of tokens

SERE ::= Boolean | Sequence | SERE @ clock_Boolean | SERE ; SERE | Sequence : Sequence | Sequence AndOrOp Sequence | SERE [ * [ Count ] ] | [ * [ Count ] ] | SERE [ + ] | [ + ] | Boolean [ = Count ] | Boolean [ -> [ positive_Count ] ]

Sequence ::= { SERE } | sequence_Name [ (Actual_Parameter_List ) ]
What are the Advantages?

• Accurate protocol descriptions of all legal alternatives improve quality by eliminating ambiguity
• We can use these descriptions directly as assertions

```plaintext
sequence BurstModeRead = {
    {ReadFirst} ;
    {ReadNext}[*]
};

.....

assert {HREADY} |=> {
    BurstModeRead
    | BurstModeWrite
    | SingleRead
    | SingleWrite
    | Inactive
    | Reset
};
```

These are sequences of booleans

Every ‘start of transaction’ must be followed by a legal transaction
Describing a Simple APB Transaction

( (PADDR==addr_arg) && !PWRITE && PSEL && !PENABLE );

( (PADDR==addr_arg) && !PWRITE && PSEL && PENABLE && (PRDATA == data_arg) )

( (PADDR==addr_arg) && PWRITE && PSEL && !PENABLE && (PWDATA == data_arg) );

( (PADDR==addr_arg) && PWRITE && PSEL && PENABLE && (PWDATA == data_arg) )

PSL/Sugar Tutorial - DVCon 2003
Non-determinism

(HRESP == OKAY) && !HREADY)*; (HRESP == OKAY) && HREADY

Repeat this step as many times as necessary.
Constructs such as [* 0 .. 8] bounds the repeats

( (HRESP == OKAY) && !HREADY)[*]; (HRESP == OKAY) && HREADY)
Alternative Outcomes - Example

One-cycle delay, two-cycle error response

One-cycle delay, one-cycle okay response

Zero-cycle delay, one-cycle okay response

One-cycle delay, two-cycle error response
Alternative Outcomes - 1

- AHB protocol has two outcomes for a transaction
  - OKAY completions take one cycle
  - ERROR completions take two cycles
- Alternatives are easily described using PSL

```plaintext
( (HRESP == OKAY) && !HREADY)[*];

{   // this path is a normal completion
    { ( (HRESP == OKAY) && HREADY && (HRDATA == data_arg) && (HRESP == resp_val)) } \\

|   // this path is an error completion
  { ( (HRESP == ERROR) && !HREADY) ;
    ( (HRESP == ERROR) && HREADY && (HRDATA == data_arg) && (HRESP == resp_val)) }   }
```
Alternative Outcomes -2

• Multiple transaction outcomes, for example, ‘Master Abort’, ‘Parity Error’ and ‘normal’ are alternative sequences (paths) in one Regular Expression
  - not dispersed across various sub-sections and pages of ‘English’
• Non-deterministic ordering is supported using alternative paths
  - PCI Target completing a handshake before the Initiator, rather than vice versa
• Repeated sequences described using [*n] notation
  - provides support for burst-mode transactions
  - supports complex concepts such as unlimited RETRY or unlimited SPLIT but no alternating RETRY & SPLIT, that take careful coding in procedural languages
Why is Protocol Ambiguity Important?

- Hardware must drive the same signals when the alternative path isn’t known
- PSL is highly expressive
  - many paths evaluated concurrently until a match is found

(!sel && !rdy) ;
{ {(!rdy)[*..3]; (sel && xtnd && !rdy)[*..2] ; (rdy && (data = arg))}
| {(sel && !rdy)[*..4]; (sel && rdy && (data = arg))} }

- if we simplify this expression the problem is clearer. If ‘xtnd’ is an output that has setup requirements, when is it set to be true?

(!sel && !rdy) ; (sel && !rdy)[*..3];
{ {(xtnd && !rdy)[*..2] ; (rdy && (data = arg))}
| {(sel && !rdy)[*..1]; (sel && rdy && (data = arg))} }
A Typical Protocol Description

• A typical protocol for AHB includes:
  - Read & Write transactions (the sequences are a little different)
  - Burst-read and Burst-write transactions
  - Idle, Reset and ‘not-selected’ behavior
• The AHB description is an order of magnitude smaller than procedural languages
  - a few pages of Regular Expressions rather than 20+ pages of procedural C++ (or similar)
• Abstracted and simplified by defining “boolean” expressions

boolean SingleTransferAddress =
  ( Addr_FSM_ENABLE && (HADDR = addr_arg) && (HTRANS = NONSEQ) && (HBURST = SINGLE) && (HSIZE = size_arg) && (HPROT = prot_arg));
A Fragment of the AHB Protocol

```c
transaction single_read
{
  (SingleTransferAddress && !HWRITE && (HRESP == OKAY) && !HREADY)[*];
  {
    (SingleTransferAddress && !HWRITE && (HRESP == RETRY) && !HREADY;
    (SingleTransferIdleAddress && !HWRITE && (HRESP == RETRY) && !HREADY);
    // we may relinquish the bus, it depends on relative priority
    (HGRANT && !SingleTransferDefaultAddress && !HWRITE)[*];
    (HGRANT && !SingleTransferDefaultAddress && !HWRITE && !HREADY)[*];
    (HGRANT && !SingleTransferDefaultAddress && !HWRITE && HREADY)
  }[*];
  (SingleTransferAddress && !HWRITE && (HRESP == OKAY) && !HREADY)[*];
  (SingleTransferAddress && !HWRITE && (HRESP == OKAY) && HREADY);
  (SingleTransferAddress && !HWRITE && (HRESP == OKAY) && !HREADY)[*]
}[*]
```
What Do I Get From Protocol Modeling?

• Raising the level of abstraction to transaction-level enables you to create tests, and debug the results, at the level at which you think
  – memory read/write transactions rather than signal interactions
• A ‘transducer’ expands abstract transaction records to multi-cycle signal-level design interaction
  – re-useable BFM’s / Transactors / TVM’s convert streams of abstract transaction records to/from detailed signal protocols
• Protocol descriptions are a VERY effective source for verification IP for Transaction-based Verification
  – Regular expressions describe interaction between parties
  – tools build the interfaces automatically in minutes
  – the interface is accurate because the description is proven
Where Does This Fit in Verification?

- Stimulus generator
- Master Transactor
- Independent Monitor Transactor
- Slave Transactor
- Response checker
- Transaction recorder
- Response provider
- S-o-C
- Testbench threads

Higher-level testbench
How is This Related to PSL/Sugar?

- As we’ve seen from the previous examples, the descriptions are based on SERE’s
  - The regular expressions at the heart of PSL / Sugar
- The SERE’s are extended to support
  - Control of signal direction – properties & assertions are used for monitoring, and don’t drive signals in simulation
  - Assignment of arguments to/from signals
  - Parameters & expressions for non-deterministic repeats
- There’s a different top level
  - Replacing the PSL property / verification layers to describe transactions
  - Wrapping the transaction set to define the full Protocol
What’s Inside a Transactor?

• The translation of the PSL transaction description to an FSM is feasible
  – Extended BNF is used to generate automata-based language recognizers
  – PSL is the source for Assertion-Based Verification

• Master transactors are relatively straightforward
  – Masters are given a transaction stream by the stimulus generator

• Slaves and Monitors perform real time “transaction recognition”
  – Slaves have to respond to an unpredictable transaction stream

• Transactors are more complex than language recognizers
  – Look-ahead and back-tracking aren’t feasible in real time
What is possible automatically?

- Automatic ambiguity checks on the protocol
- Editable source language output for
  - Verilog, VHDL, Testbuilder 1.3, Testbuilder-SC / SCV, SystemC and other verification environments
- A wide range of protocols can be supported
  - Parallel protocols - ARM APB, PCI, PCI-X, UTOPIA, ISA/EISA, SCSI, CardBUS, OCP, IBM CoreConnect, PowerPC/MPX
  - Serial protocols to ‘phy’ interface – USB, Ethernet, Firewire, RapidIO, Infiniband, HyperTransport, PCI-Express
- Enhanced generation scheme for pipeline architectures
  - AMBA AHB, ASB, PowerPC/MPX, Pentium etc.
Recording Facilities

- creation of both Transaction & Waveform records

<table>
<thead>
<tr>
<th>Cursor = 0</th>
<th>Baseline = 0</th>
<th>Baseline = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Master-Stream [0]</strong></td>
<td><strong>Transaction</strong></td>
<td><strong>Transaction</strong></td>
</tr>
<tr>
<td></td>
<td><code>addr_arg</code></td>
<td><code>addr_arg</code></td>
</tr>
<tr>
<td></td>
<td><code>d7</code></td>
<td><code>d7</code></td>
</tr>
<tr>
<td><strong>Monitor-Stream [0]</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Slave-Stream [0]</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| **PCLK** | 1 | |
| **PADDR** | `'h00000000` | `00003DEA 00003C33 0000385F 00003A52 00003EDD` |
| **PWRITE** | 0 | |
| **PENABLE** | 0 | |
| **PWDATA** | `'h00000000` | `32FF902` |
| **FSEL** | 0 | |
| **PRESP** | `'h1` | 0 |
| **PRDATA** | `'h00000000` | `AAAA9742 AAAA969A AAAA92F6 AAAA90FA AAAA94` |
| **PREady** | 1 | |
Protocol-Specific Coverage is Vital

- Getting a simulation to pass is not enough!
- You need to know
  - you haven’t simulated a higher priority master taking the bus during RETRY
  - you haven’t simulated the Slave inserting a wait-cycle when you re-gain the bus after SPLIT

```c
transaction single_read
{
    (SingleTransferAddress && HWRITE && (HRESP == OKAY) && !HREADY)[*],
    {
        (SingleTransferAddress && HWRITE && (HRESP == RETRY) && !HREADY),
        (SingleTransferAddress && HWRITE && (HRESP == RETRY) && !HREADY),
        
        // we may relinquish the bus, it depends on relative priority
        (HGRANT && SingleTransferDefaultAddress && !HWRITE)[*],
        (HGRANT && SingleTransferDefaultAddress && !HWRITE)[*],
        (HGRANT && SingleTransferDefaultAddress && !HWRITE && !HREADY)
    }[*..1],
    (SingleTransferAddress && HWRITE && (HRESP == OKAY) && !HREADY)[*],
    (SingleTransferAddress && HWRITE && (HRESP == OKAY) && !HREADY),
    (SingleTransferAddress && HWRITE && (HRESP == OKAY) && !HREADY)[*]
}
```

PSL/Sugar Tutorial - DVCon 2003
Where Else is the Description Used?

- Validation of the protocol prior to construction
  - automatically perform ambiguity checks
  - Master can be formally checked against Slave to detect lockup and other faults
- ‘regular’ PSL properties
  - Either manually or by post processing the protocol descriptions to remove the signal direction controls and generate ‘fragment’ assertions
- Transaction-level verification