UVM Error Injection
Using a Two-Phase Slave Sequence

Mona Beimers
Xilinx Inc.

April 21, 2017
Canada

Agenda

UVM Error Injection

Design Challenge: Interlaken Protocol

Interlaken UVM Verification Environment

Common Error Injection Methods

Error Injection Mechanism Using a Two-Phase Slave Sequence

Advantages and Conclusions

Agenda

UVM Error Injection

Design Challenge: Interlaken Protocol

Interlaken UVM Verification Environment

Common Error Injection Methods

Error Injection Mechanism Using a Two-Phase Slave Sequence

Advantages and Conclusions

UVM Error Injection

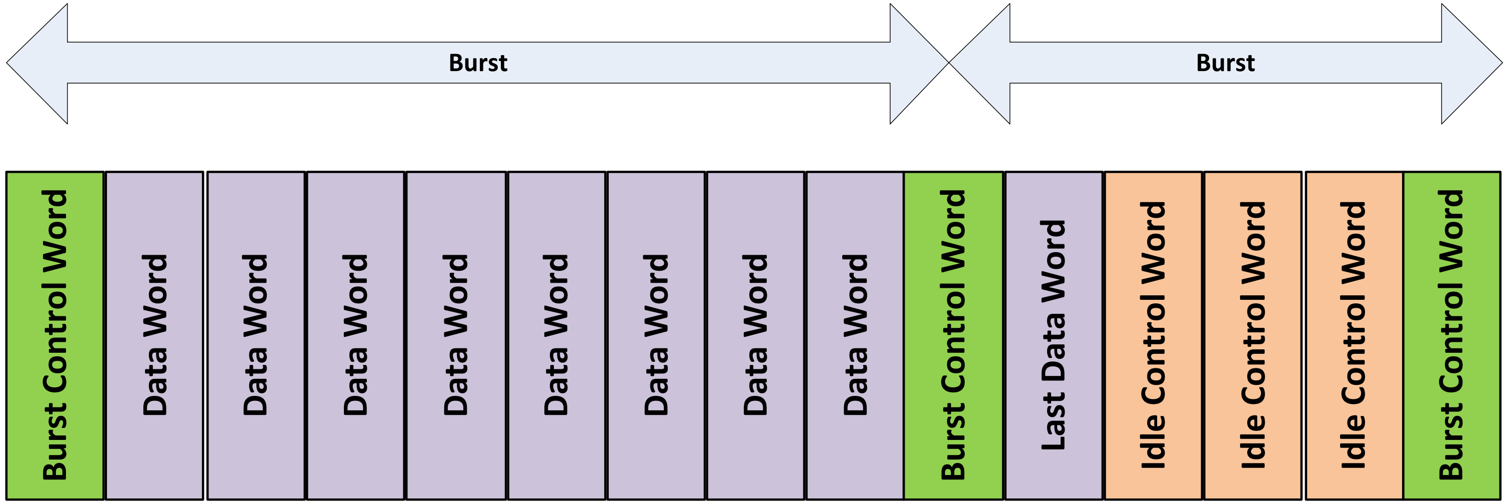
- Error injection consists of injecting stimulus with errors and verifying that the device under test (DUT) is able to handle it, report it and continue operating as documented in the functional specification
- There is no “recipe” for implementing it
- Possible sources of inspiration are:
  - UVM documentation
  - Online resources like Solvnet, SNUG, blogs or forums
  - Previous company projects
- Examples are few, simplistic and target mostly errors associated with the fields of a packet, like corrupted start-of-packet, end-of-packet or CRC

Agenda

- UVM Error Injection
- Design Challenge: Interlaken Protocol
- Interlaken Verification Environment
- Common Error Injection Methods
- Error Injection Mechanism Using a Two-Phase Slave Sequence
- Advantages and Conclusions

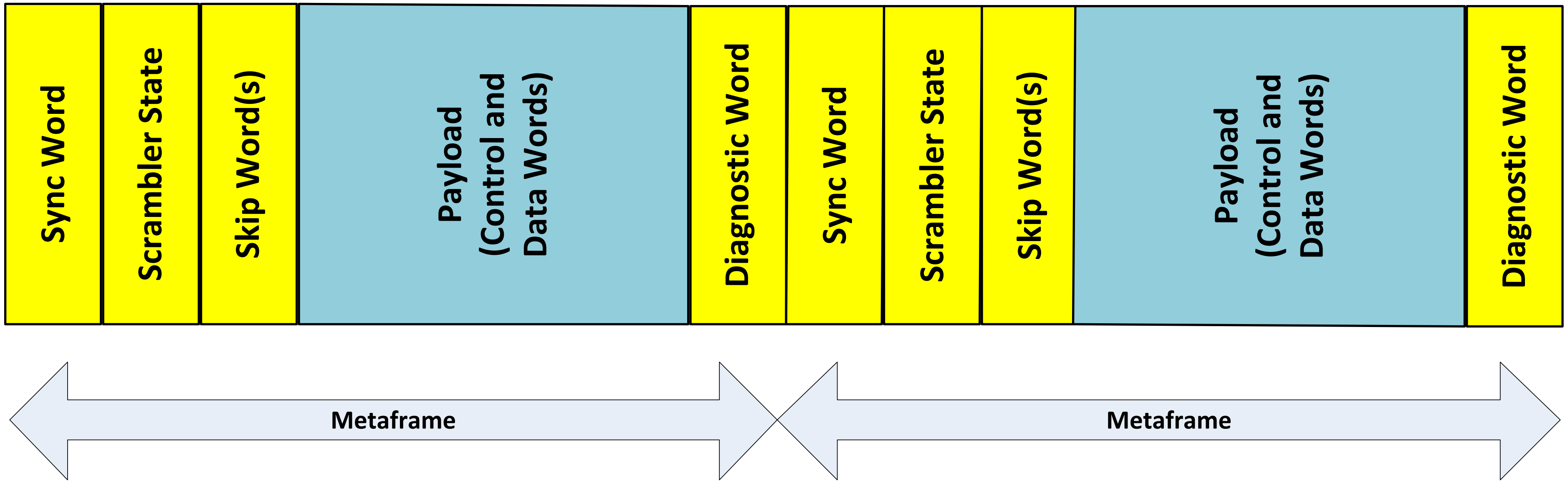
Design Challenge: Interlaken Protocol

- The Interlaken protocol is a high-speed protocol used to transfer packets over serial links
- The packets are broken up into bursts of sizes determined by BurstMax, BurstMin and BurstShort parameters and transmitted in burst interleaved or non-interleaved mode
- The bursts are data words delineated by control words
- The basic unit of data is an 8-byte word which is used for 64B/67B encoding



Interlaken Protocol (continued)

- The data and control words are striped sequentially over all the lanes of the interface
- On each lane they are encapsulated in metaframes which include following framing words: sync, scrambler-state, skip and diagnostic
- Metaframes are injected into the traffic at a programmable regular interval



Agenda

UVM Error Injection

Design Challenge: Interlaken Protocol

Interlaken UVM Verification Environment

Common Error Injection Methods

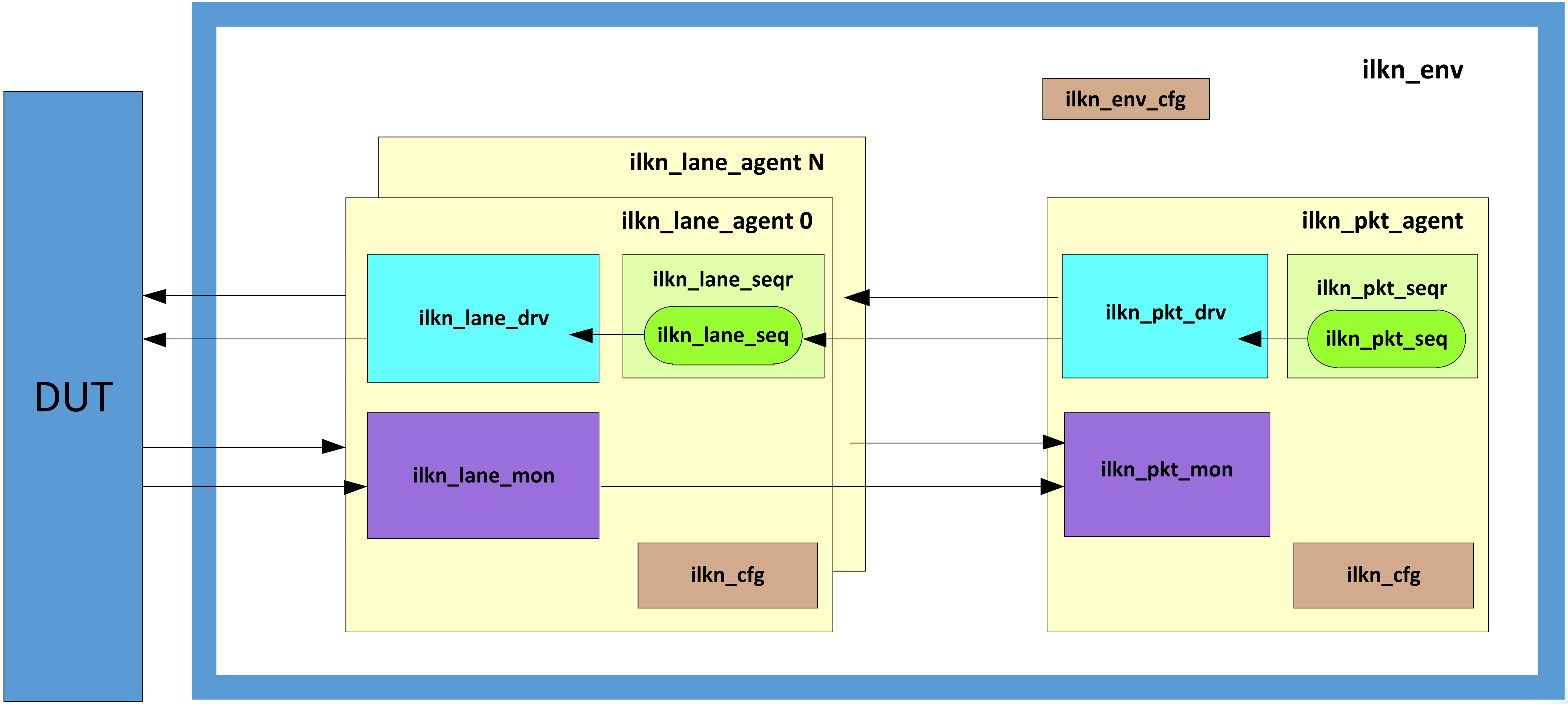
Error Injection Mechanism Using a Two-Phase Slave Sequence

Advantages and Conclusions

Interlaken UVM Verification Environment

- Consists of a packet agent and several lane agents
- The packet driver
  - Breaks up packets into bursts, and each burst into data words
  - Adds idle or burst control words to separate the bursts
  - At regular interval injects metaframes by adding framing words into the data stream
  - Implemented as a layering driver which stripes the stream of words (data, burst, idle, framing words) over the lower-level lane drivers which will ultimately drive the words into the DUT

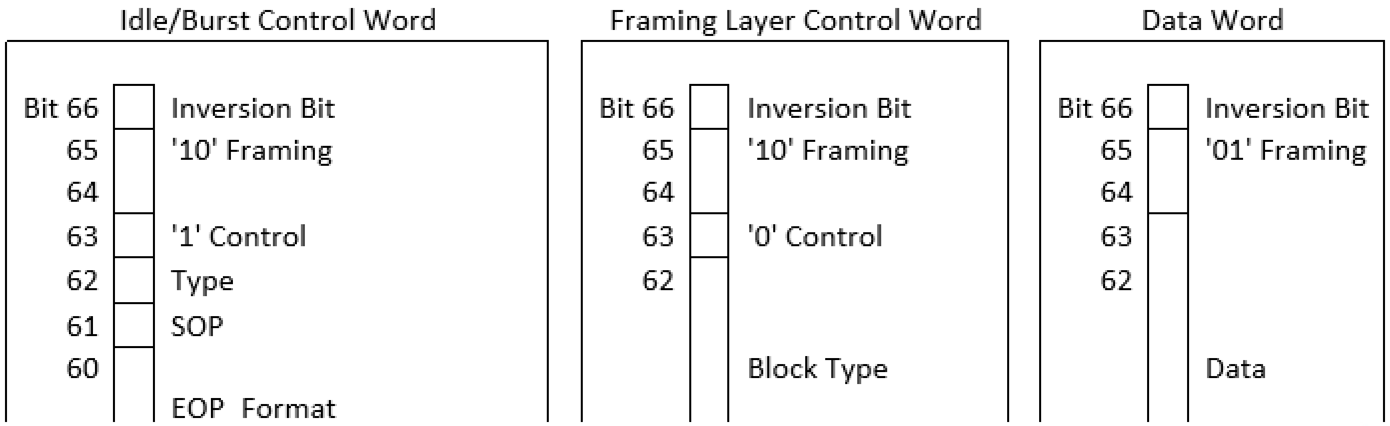
Interlaken UVM Verification Environment



Interlaken UVM Classes

Interlaken Word

```
class ilkn_word_c extends uvm_sequence_item;
  `uvm_object_utils(ilkn_word_c)

  bit [66:0]          ilkn_word_data;
  ilkn_word_type_e    word_type;      // DATA_WORD, BURST_WORD, IDLE_WORD,
                                      // SYNC_WORD, SCRAM_WORD, SKIP_WORD,
                                      // DIAG_WORD, UNKNOWN_WORD
…
  extern virtual function void set_inversion_bit(bit inversion);
  extern virtual function bit  get_inversion_bit();
```



Interlaken UVM Classes

Interlaken Packet

```
class ilkn_pkt_c extends uvm_sequence_item;
  `uvm_object_utils(ilkn_pkt_c)

  rand int          pkt_length;
  rand int unsigned chan;
  rand byte         data_bytes[];
  rand int unsigned burstmax;
  rand int unsigned burstshort;
  rand int unsigned burstmin;

  ilkn_word_c  ilkn_word_data[$];   // data words making up packet
…
  constraint pkt_length_cons { pkt_length inside { MIN_LEN, MAX_LEN }; }
…
```

Interlaken UVM Classes

Interlaken Packet Sequence

```
class ilkn_pkt_seq_c extends uvm_sequence #(ilkn_pkt_c);
  `uvm_object_utils(ilkn_pkt_seq_c)
  `uvm_declare_p_sequencer(ilkn_pkt_seqr_c)
…
  virtual task body();
…
    if( !ilkn_pkt.randomize() with {
                           burstmax   == p_sequencer.ilkn_cfg.burstmax;
                           burstmin   == p_sequencer.ilkn_cfg.burstmin;
                           burstshort == p_sequencer.ilkn_cfg.burstshort;
    …
                           })
```

Agenda

- UVM Error Injection
- Design Challenge: Interlaken Protocol
- Interlaken UVM Verification Environment
- Common Error Injection Methods
- Error Injection Mechanism Using a Two-Phase Slave Sequence
- Advantages and Conclusions

Error Injection Methods

- Error injection on packet fields
  - Extend the packet
  - Extend the packet sequence
- Error injection on configurations
  - Extend the configuration
- Error injection associated with flags inside the packet class
  - Error flags added as extra fields in the packet class
- Error injection on control, data and framing words
  - Use callbacks
  - Use proposed UVM error agent with a two-phase slave sequence

Error Injection on Packet Fields

Extending the Packet Class

```
class ilkn_pkt_extended_length_c extends ilkn_pkt_c;
  `uvm_object_utils(ilkn_pkt_extended_length_c)
   // overwrite base class constraint (legal values: MIN_LEN..MAX_LEN)
   constraint pkt_length_cons {pkt_length inside {[0:MAX_LEN+100]};}
   // new()
endclass // ilkn_pkt_extended_length_c

class test_ilkn_err_illegal_size_pkts_c extends test_base_c;
  `uvm_component_utils(test_ilkn_err_illegal_size_pkts_c)
  // new()
  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    ilkn_pkt_c::type_id::set_type_override(ilkn_pkt_extended_length_c::get_type());
  endfunction // build_phase
endclass // test_ilkn_err_illegal_size_pkts_c
```

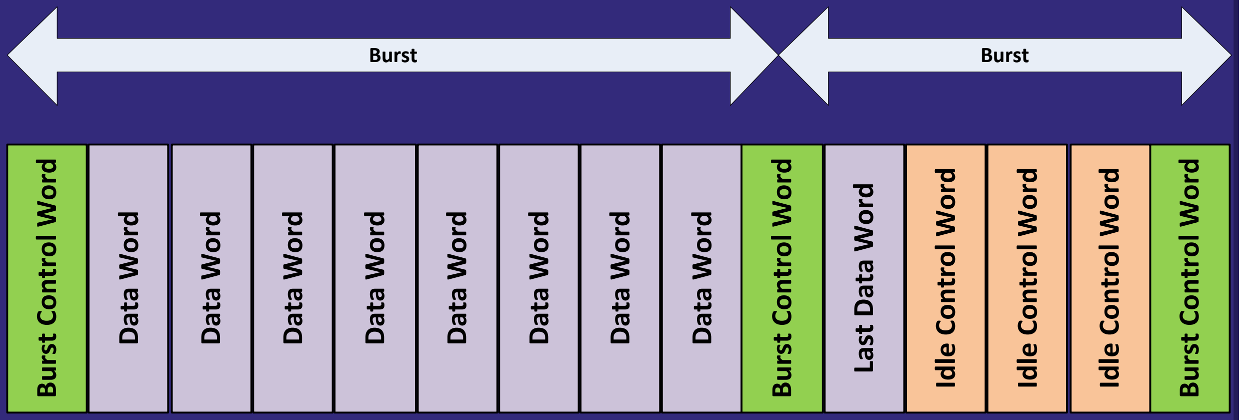
Error Injection on Packet Fields

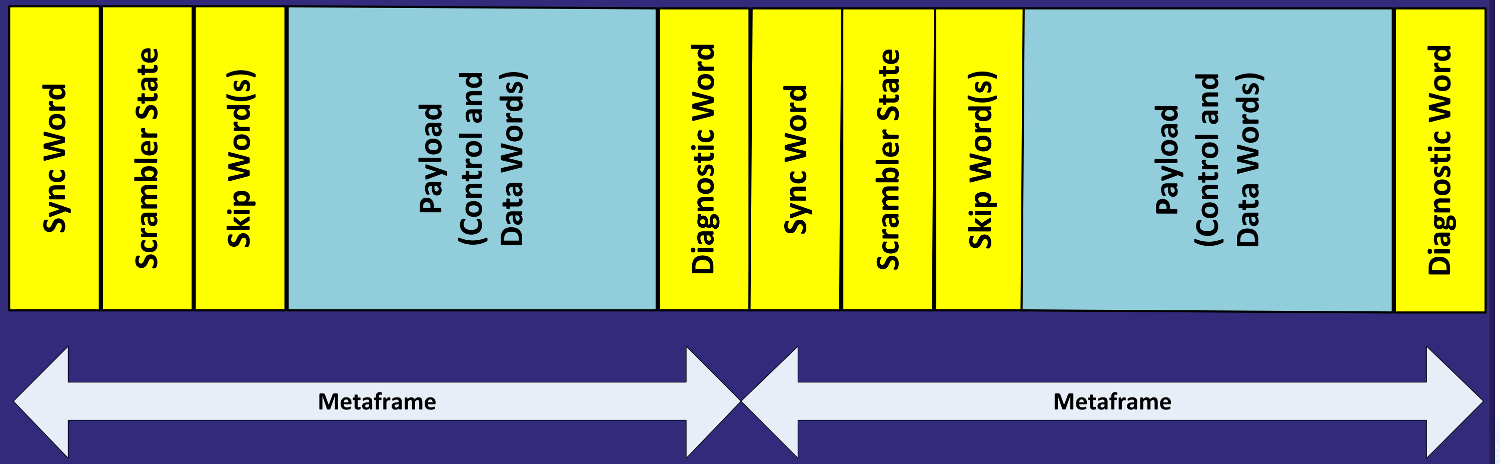
Extending the Packet Sequence Class

```
class ilkn_pkt_burstmax_violation_seq_c extends ilkn_pkt_seq_c;
  `uvm_object_utils(ilkn_pkt_burstmax_violation_seq_c)
  `uvm_declare_p_sequencer(ilkn_pkt_seqr_c)

  rand   int burstmax_add_val;  // 90% correct, 10% incorrect
  constraint burstmax_add_val_cons { burstmax_add_val dist { 0 := 90, [1:10] :/ 10 }; }
  ...
  virtual task body();
    ...
    if(!randomize()) `uvm_error(get_name(), $sformatf("can't randomize seq"));
    if(!ilkn_pkt.randomize() with {
      burstmax   == p_sequencer.ilkn_cfg.burstmax + burstmax_add_val;
      burstmin   == p_sequencer.ilkn_cfg.burstmin;
      burstshort == p_sequencer.ilkn_cfg.burstshort;
    ...
endclass // ilkn_pkt_burstmax_violation_seq_c
```

Error Injection Using Flags Inside the Packet Class

```
class ilkn_pkt_c extends uvm_sequence_item;
  `uvm_object_utils(ilkn_pkt_c)

  rand int          pkt_length;
  rand int unsigned chan;
  ...
  rand bit          corrupt_sop;
  rand bit          corrupt_eop;
  rand bit          corrupt_crc;
  ...
  constraint errors_off_cons {
    corrupt_sop == 0;
    corrupt_eop == 0;
    corrupt_crc == 0;
  }
```

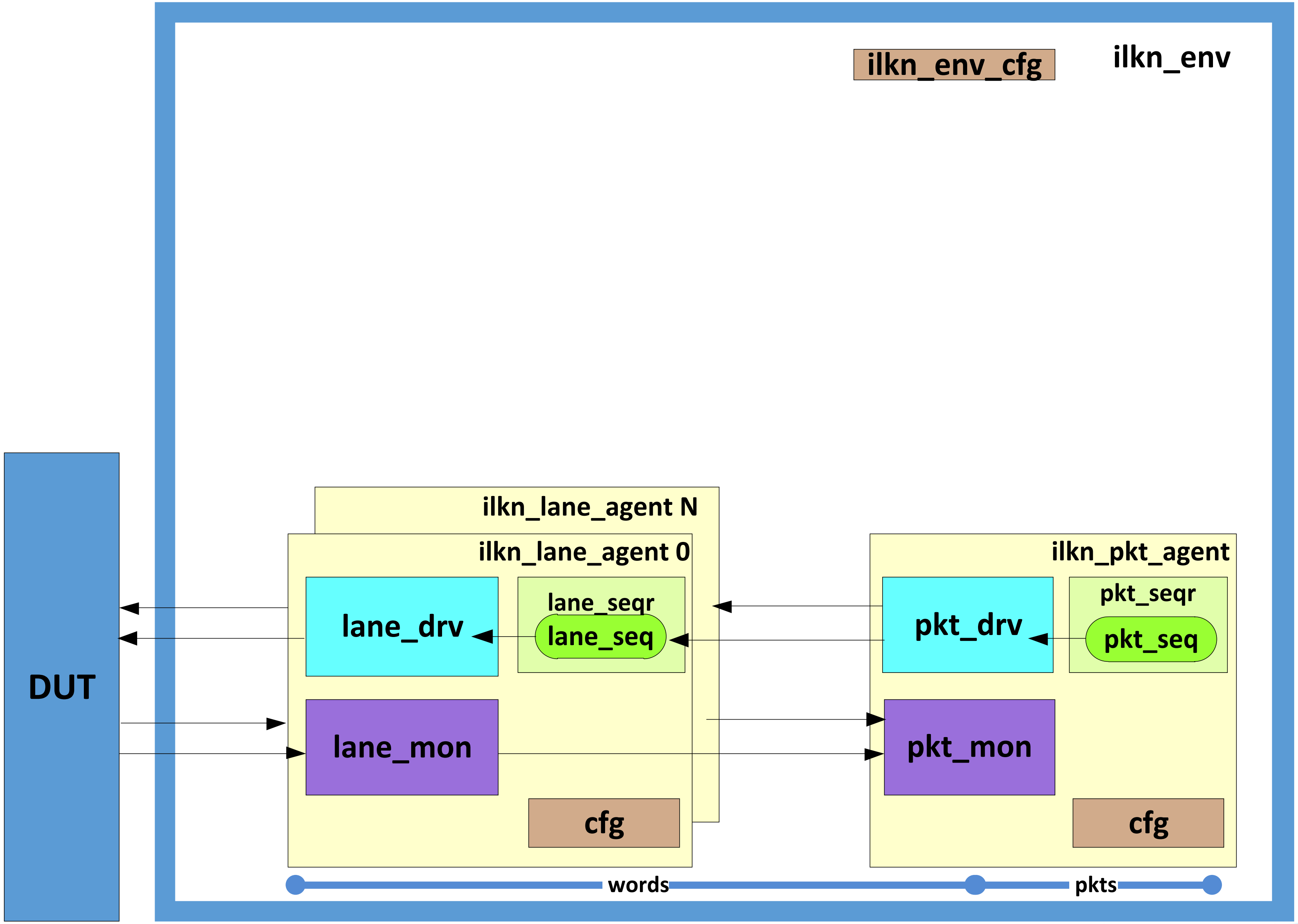




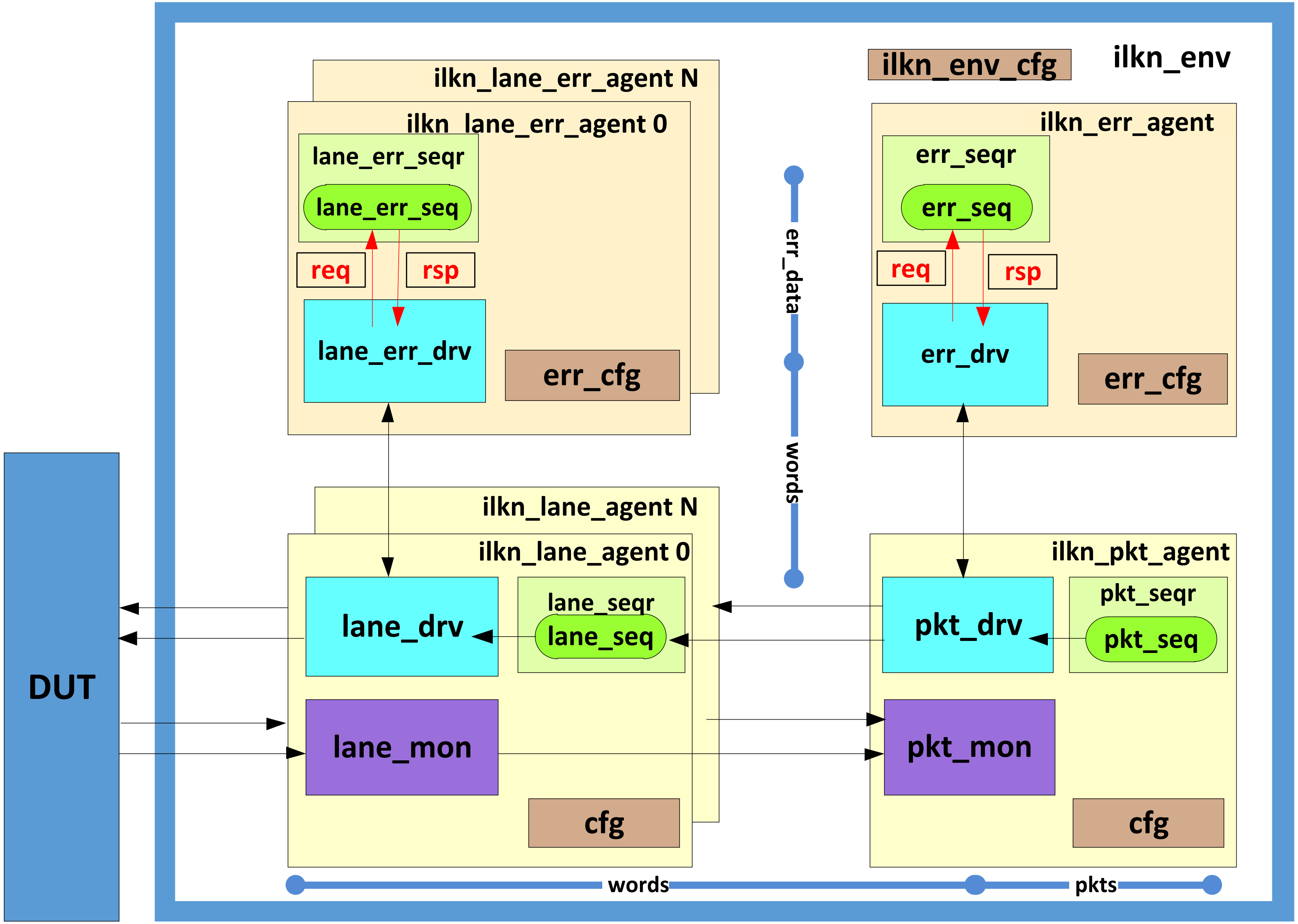
Agenda

- UVM Error Injection
- Design Challenge: Interlaken Protocol
- Interlaken UVM Verification Environment
- Common Error Injection Methods
- Error Injection Mechanism Using a Two-Phase Slave Sequence
- Advantages and Conclusions

Interlaken UVM Verification Environment

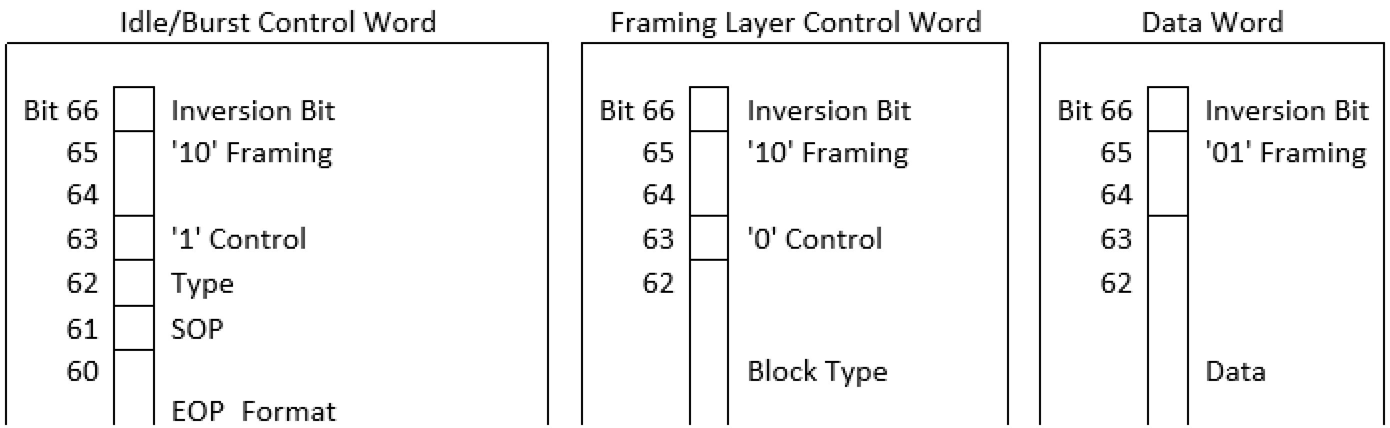


Error Injection Mechanism Using a Two-Phase Slave Sequence



Error Flags and Groups of Errors

- **error_type** - controls the individual errors to be injected
  - ILKN_INVERSION_BIT_ERR,
  - ILKN_WRONG_FRAMING_BITS_ERR or ILKN_ILLEGAL_FRAMING_BITS_ERR
  - ILKN_MISSING_SOP_ERR or ILKN_BOGUS_SOP_ERR etc.
- **ilkn_error_group** - controls the valid error_types per Interlaken word
  - CW_ERROR, DW_ERROR, LANE_ERROR



Interlaken UVM Error Class

```
typedef enum { ILKN_NO_ERR, ILKN_INVERSION_BIT_ERR,
               ILKN_WRONG_FRAMING_BITS_ERR, ILKN_ILLEGAL_FRAMING_BITS_ERR,
                ...} ilkn_err_type_e;

typedef enum { CW_ERROR, DW_ERROR, LANE_ERROR } ilkn_err_group_e;

class ilkn_error_data_c extends uvm_sequence_item_c;
  `uvm_object_utils(ilkn_error_data_c)

  rand ilkn_err_group_e ilkn_err_group;  // CW_ERROR, DW_ERROR, LANE_ERR
       ilkn_word_type_e word_type;       // set from inside the sequence
  rand ilkn_err_type_e  error_type;      // errors to be injected
  ...

  constraint cw_errors_cons;      // errors injected on control words
                                  // passing through the ilkn_pkt_drv
  constraint dw_errors_cons;      // errors injected on data words
                                  // passing through the ilkn_pkt_drv
  constraint lane_errors_cons;    // errors injected on all words passing
                                  // through the ilkn_lane_drv
```

Interlaken UVM Error Class

continued

```
constraint ilkn_error_data_c::cw_errors_cons
{
  if(ilkn_err_group == CW_ERROR)  // control word errors
  {
    if(word_type == BURST_WORD)
      {
        error_type inside { ILKN_NO_ERR,
                            ILKN_TYPE_ERR,
                            ILKN_BOGUS_SOP_ERR,
                            ILKN_MISSING_SOP_ERR ... };
      }
    else if(word_type == IDLE_WORD)
      {
        error_type inside { ILKN_NO_ERR,
                            ILKN_TYPE_ERR,
                            ILKN_BOGUS_EOP_ERR,
                            ILKN_MISSING_EOP_ERR ... };
      }
  }
}
```

Handshake

ilkn_pkt_drv::run_phase()	ilkn_err_drv::corrupt_cw/_dw(…)	ilkn_err_seq::body()	ilkn_err_data
For each word it processes			
word (DW? CW?)	task corrupt_cw(word_type, ref word)		
	if(ok_to_inject_errors())	forever	
	// A - randomize error		
	// phase 1		
	Set word_type, error_group in req		req
	// phase 2	randomize rsp based on req	
	Get randomized error_type in rsp		rsp
	// B – apply error corruption		
	case (rsp.error_type)		
	// corrupt word		
	endcase		
	else		
	// leave word uncorrupted		
	endtask (return word)		
Continue thread Send word to lower level			
Process next word			

Two-Phase Error Sequence

<table>
<tr><th>Error Driver</th><th>Error Sequence</th></tr>
<tr><td>

```
class ilkn_error_drv_c extends uvm_driver
                        #(ilkn_error_data_c);
 `uvm_component_utils(ilkn_error_drv_c)


  // function new(string name="");


  task corrupt_control_word(
  // or corrupt_data_word(
  [... some arguments omitted ]
    ilkn_word_type_e word_type,
    ref ilkn_word_c  ilkn_word,
        …
);
    ilkn_error_data_c req;
    ilkn_error_data_c rsp;
```

</td><td>

```
class ilkn_error_seq_c extends uvm_sequence
                        #(ilkn_error_data_c);
  `uvm_object_utils(ilkn_error_seq_c)
  `uvm_declare_p_sequencer(ilkn_error_seqr_c)

  // function new(string name="");


  virtual task body();
    ilkn_error_data_c  req;
    ilkn_error_data_c  rsp;

    […] // create req, rsp
```

</td></tr>
</table>

Two-Phase Error Sequence

Error Driver	Error Sequence
`if(ok_to_inject_errors())` `begin`	`forever` `begin`
Set word_type, error_group in req	
PHASE 1	1) `start_item(req);`
2) `seq_item_port.get_next_item(req);`	
	`finish_item(req);`

3) `req.ilkn_err_group = CW_ERROR;`
`// or req.ilkn_err_group = DW_ERROR;`
`req.word_type = word_type;`	

4) `seq_item_port.item_done();`	

Two-Phase Error Sequence

Error Driver	Error Sequence
← Get randomized error_type in rsp	
PHASE 2	

6) seq_item_port.get_next_item(rsp);	5) start_item(rsp);
	rsp.copy(req);
	
	// randomization of errors in rsp
	// based on req fields
	
	if(!rsp.randomize() with
	{ilkn_err_group ==req.ilkn_err_group;})
	`uvm_error(...);
	
	finish_item(rsp);

Two-Phase Error Sequence

Error Driver	Error Sequence

`7)     // executes the corruption code`
`       case(rsp.error_type)`
`         // error flag: corruption code`
`         // error flag: corruption code`
`       endcase // case (rsp.error_type)`
 	
`8)     seq_item_port.item_done();`
`       …`
`     end // if (ok_to_inject_errors())`
`endtask // corrupt_control_word`
	`    end // forever begin`
	`  endtask // body`
	`endclass // ilkn_error_seq_c`

Agenda

UVM Error Injection

Design Challenge: Interlaken Protocol

Interlaken UVM Verification Environment

Common Error Injection Methods

Error Injection Mechanism Using a Two-Phase Slave Sequence

Advantages and Conclusions

Advantages

ilkn_pkt_drv::run_phase()	ilkn_err_drv::corrupt_cw/_dw(…)	ilkn_err_seq::body()	ilkn_err_data
For each word it processes			
word (DW? CW?)	task corrupt_cw(word_type, ref word)		
	if(ok_to_inject_errors())	forever	
	// A - randomize error		
	// phase 1		
	Set word_type, error_group in req		req
	// phase 2	randomize rsp based on req	
	Get randomized error_type in rsp		rsp
	// B – apply error corruption		
	case (rsp.error_type)		
	// corrupt word		
	endcase		
	else		
	// leave word uncorrupted		
	endtask (return word)		
Continue thread Send word to lower level			
Process next word			

Advantages and Conclusions

- Advantages of proposed method
  - No pollution of the packet class with error injection flags
  - No pollution of the main stimulus driver with error injection code
  - Error code isolated in the error driver
  - Error type generated is guaranteed to be applicable to the current word
    - Due to the two-phase slave sequence
  - Error tests easy to create by extending the error data class
  - More complicated error tests can be created by extending the error sequence
  - Error sequence and error data are generic and used in all error agents

- Conclusion
  - Scalable solution that can be applied at any layer of a protocol

Thank You