A Simulator Backplane Proposal

William McSpadden
Seagate Technologies
bill.mcspadden@seagate.com

Agenda

- The problems we need to solve.
- A proposal for a base layer API
  - Implementation examples of the base layer
- A discussion about building higher abstraction layers on top of the base layer
  - Example of a higher level API

The Problem(s)

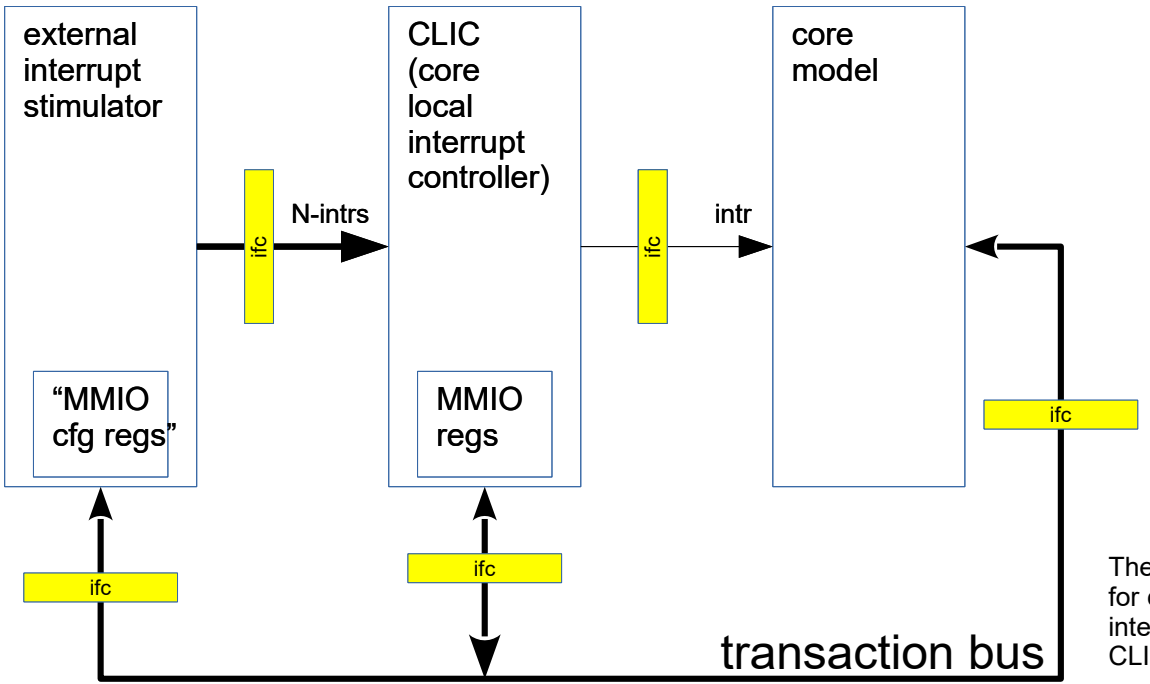
- How to get dissimilar simulators to talk to each other (ie – Co-Sim)
  - OR .... How can we use and re-use reference models, arch tests, stimulus models, etc
- How to apply the same stimulus to different models?
- How to analyze output from different models with the same analyzer?
- How to synchronize simulators? (perhaps an implementation detail)

Stimulates hw interrupts to the CLIC. Supports edge and level triggered interrupts. Need the ability to clear level triggered interrupts.

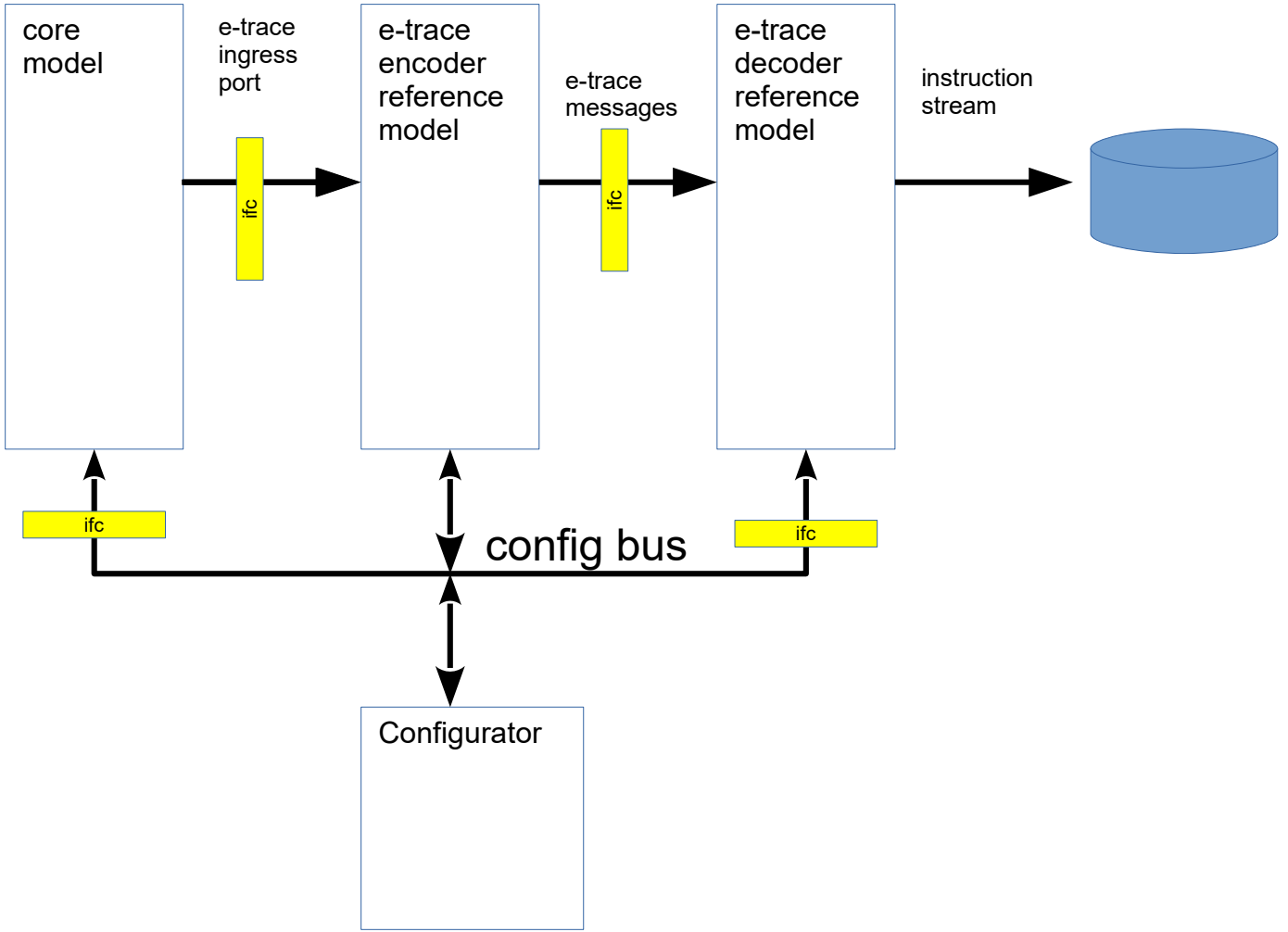
Reference model of the CLIC….

… OR …

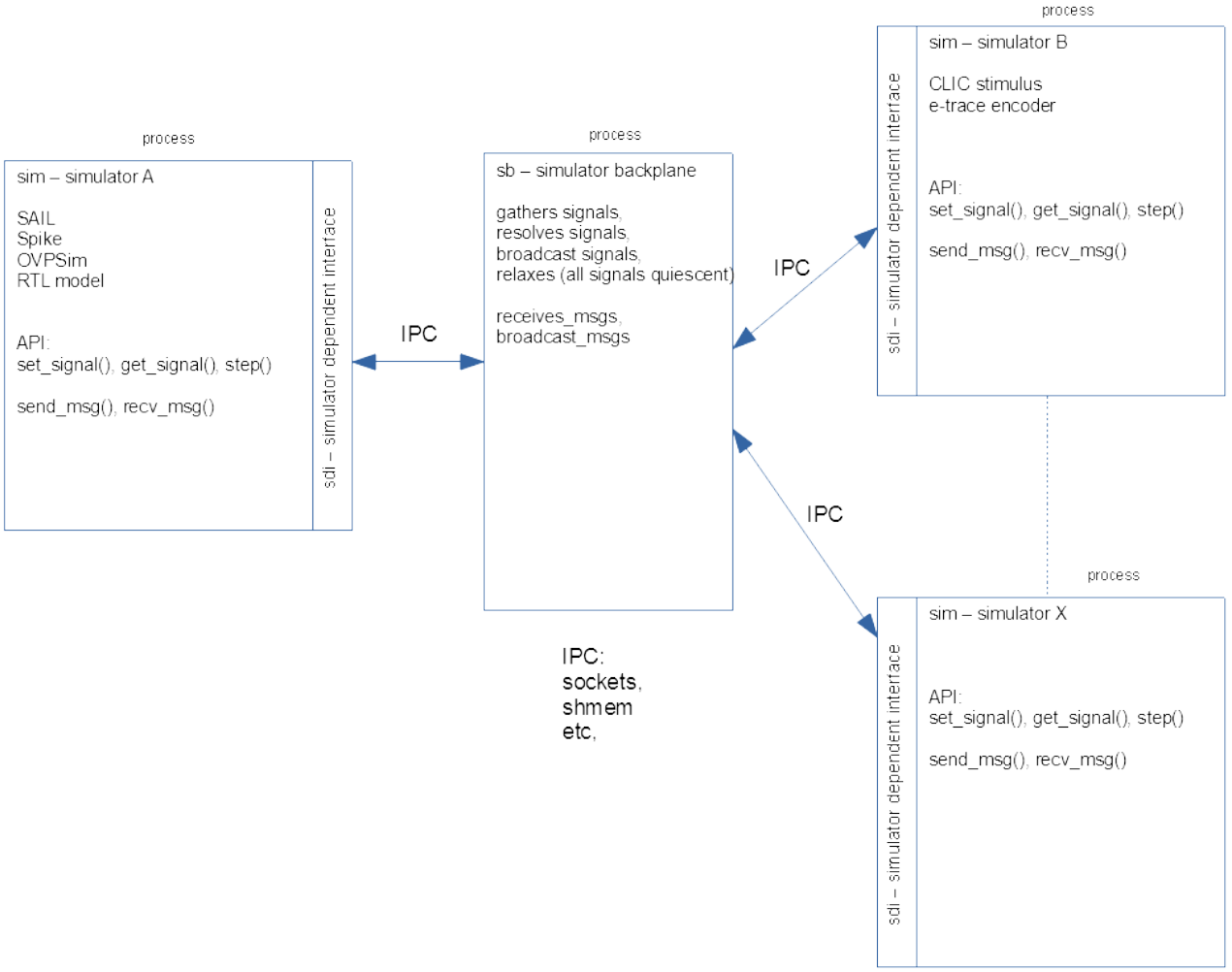
an RTL implementation of the CLIC.



The transaction bus is needed for configuring the external interrupt stimulator and the CLIC.


core model
e-trace ingress port
ifc
e-trace encoder reference model
e-trace messages
ifc
e-trace decoder reference model
instruction stream
ifc
config bus
ifc
Configurator

A Proposal for a Base Layer


process
sim – simulator A
SAIL
Spike
OVPSim
RTL model
API:
set_signal(), get_signal(), step()
send_msg(), recv_msg()
sdi – simulator dependent interface
IPC
process
sb – simulator backplane
gathers signals,
resolves signals,
broadcast signals,
relaxes (all signals quiescent)
receives_msgs,
broadcast_msgs
IPC
IPC
IPC:
sockets,
shmem
etc,
process
sim – simulator B
CLIC stimulus
e-trace encoder
API:
set_signal(), get_signal(), step()
send_msg(), recv_msg()
sdi – simulator dependent interface
process
sim – simulator X
API:
set_signal(), get_signal(), step()
send_msg(), recv_msg()
sdi – simulator dependent interface

```
interface:  intr_stimulus

signal_def: xz01

output:
intr_req[47:0]
mmio_rd_data[63:0]

input:
mmio_addr[9:0]
mmio_rd
mmio_wr
mmio_wr_data[63:0]

inout:
```

```
interface: clic_ifc

signal_def: four_value

input:
intr_req[47:0]
mmio_addr[9:0]
mmio_rd
mmio_wr
mmio_wr_data[63:0]

output:
intr
mmio_rd_data[63:0]

inout:
```

```
interface:  mmio_stimulus

signal_def: four_value

output:
intr_req[47:0]
mmio_rd_data[63:0]

input:
mmio_addr[9:0]
mmio_rd
mmio_wr
mmio_wr_data[63:0]

inout:
```

```
interface: sail_ifc

signal_def: four_value

input:
intr_req
mmio_rd_data[63:0]

output:
mmio_addr[9:0]
mmio_rd
mmio_wr
mmio_wr_data[63:0]

inout:
```

```
connector:
intr_stimulus
clic_ifc

connections:
clic_ifc.intr_req = intr_stimulus.intr_req
sail_ifc.intr_req = clic_ifc.intr
```

signal_def

- verilog
- vhdl, std_logic: UX01ZWLH-
- four_value: xz01
- two_value: 01

```
// External interrupt stimulator model, pseudo-code

ORD64       trigger_ctrl;           // 1: level   0: edge
ORD64       assert_intr;            // writing a 1 to a bit asserts intr on that line
ORD64       assert_intr_reg;
ORD64       deassert_intr;          // writing a 1 to a bit deasserts intr; only useful for level trig
ORD64       start_test_reg          // if 1, start test
int         edge_trig_pulse_width[NUM_INTRS];

main()
      forever
            if ( sdi_rising_edge(mmio_wr))
                  if (addr == TRIGGER_CTRL)
                        trigger_ctrl = sdi_get_signal(mmio_wr_data);
                  if (addr == ASSERT_INTR)
                        sdi_set_signal(sdi_get_signal(mmio_wr_data));


            if (sdi_rising_edge (mmio_rd))
                  if (addr == TRIGGER_CTRL )
                        sdi_set_signal(mmio_rd_data, control_trigger_edge_level);

            // TODO: Handle edge_triggered pulse width


            sdi_cycle();
```

```
// "The Backplane", pseudocode

main()
      // Read in connection file

      // Build IPC connections


      // loop

      forever
          do
            foreach simulator
                  gather_signals()

            resolve_signals()

            foreach simulator
                  broadcast_resolved_signals()

          while (! relax() && !timeout)
```

Why Relaxation?

Relaxation: ensures that all signals are at a quiescent state. Consider....

Zero delay loops

All simulators do some form of relaxation

For the simulator backplane, you need to relax for the same reasons that RTL simulators need to relax.

Primarily for multi-driver (INOUT) signalling

```
// "The Test", random generation of interrupts, pseudo-code

main()
      while (read_reg(START_TEST_REG) == 0)
            sdi_cycle()

      // Configure the external interrupt stimulator
      write_reg(TRIGGER_CTRL)       // write_reg():  function API written on top of set_signal; see below
      write_reg(PULSE_WIDTH)

      for (i = 0; i < NUM_INTRS ; i++)
            sdi_cycle(rand() % 100)       // random delay of assertion
            write_reg(ASSERT_INTR)

            // Do not write DEASSERT REG to clear intr.  Let the core clear the interupt.
            //    That's the point of this test.

      // Let simulation quiesce
      sdi_cycle(FINISH_UP)


write_reg(reg, value)
      sdi_set_signal(reg, value)
      sdi_set(mmio_wr, 1)
      sdi_cycle(NUM_CYCLES)
      sdi_set(mmio_wr, 0)
      return
```

```
// RISC-V assembly language test
#define CLEAR_INTERRUPT_REG   0x1000

// Initialize....
      <RISC-V initialization>


// Tell interrupt stimulator to start
      <equivalent of write_reg(TEST_START)

// Loop with various instructions to be interrupted

loop:
      add
      sub
      lw
      sw
      li
      wfi
      br    loop


interrupt_service_routine:
      <do interrupt stuff>
      .
      .
      // Clear the interrupt register (if level triggered it needs to be cleared)
      <equvalent of write_reg(DEASSERT_INTR)>

      // Return from interrupt
```

```
// An example SDI,  for verilog

// VPI code written in C
//    Question: should DPI be used?

function()
    forever
      // get signals at sdi interface
      for (signal_list)
          vpi_get_signal
          sdi_set_signal(signal, value)

      sdi_cycle()

      for (input_signal_list)
          sdi_get_signal(signal, value)
          vpi_put_signal
```

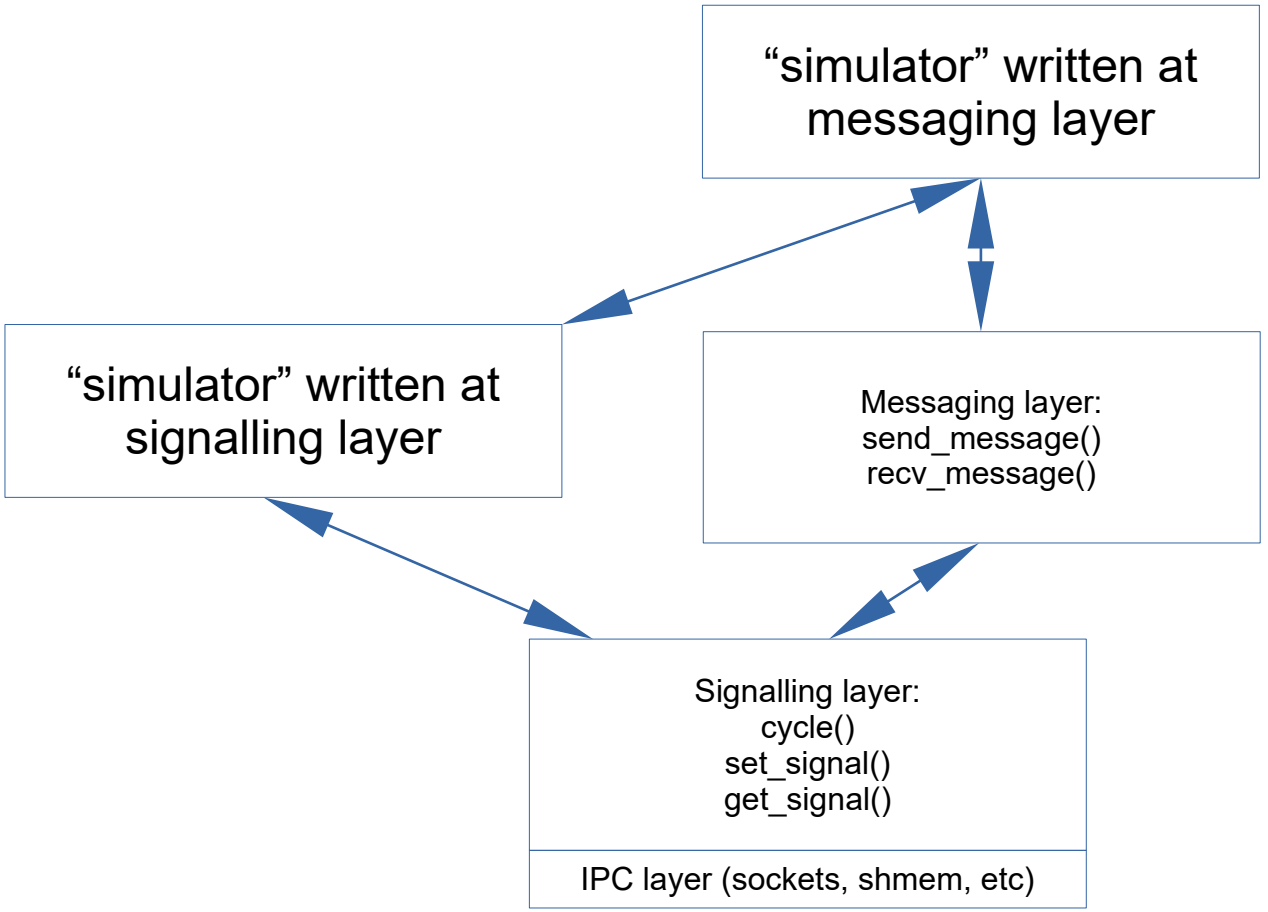
```
// Verilog/SystemVerilog code

interface   sdi_interface

      initial
            begin
            <gather signals>
            #0; #0; #0; #1;          // Put at end of relax step
            end
```

Building Abtraction Layers

- Many of our RISCV specs describe packets of information (messages) rather than a physical interface definition. It would be useful if we could build these messages on top of the signalling layer AND implement it as send/recv API.


“simulator” written at messaging layer
“simulator” written at signalling layer
Messaging layer:
send_message()
recv_message()
Signalling layer:
cycle()
set_signal()
get_signal()
IPC layer (sockets, shmem, etc)

Example: e-trace spec

Table 2.1: Basic Control

Field	W	G	RW	Rst	Description
Active	1	M	RW	0	Master enable for trace system. When 0, the trace system may have clocks gated off or be powered down, and other register locations may be inaccessible. Hardware may take arbitrarily long to process power-up or power-down and will indicate completion when the read value of this bit matches the value written.
teEnable	1	M	RW	0	Master trace enable. Trace can be enabled via **iEnable** or **dEnable** when 1. Setting to 0 flushes any queued trace data to the designated sink.
iTracing	1	M	RW	0	Instruction trace enable. When 1, trace will be generated, subject to any optional filtering. May be written by software, or via triggers if **iTrigEnable** is 1.
dTracing	1	MD	RW	0	Data trace enable. When 1, trace will be generated, subject to any optional filtering. May be written by software, or via triggers if **dTrigEnable** is 1.
iTrigEnable	1	O	RW	0	When 1, allows iTracing to be set or cleared by *trace-on* and *trace-off* Debug module triggers respectively (see 4.2.4).
dTrigEnable	1	OD	RW	0	When 1, allows dTracing to be set or cleared by *trace-on* and *trace-off* Debug module triggers respectively (see 4.2.4).
stallEnable	1	O	RW	0	When 0, if the encoder cannot accept trace input from the RISC-V hart, trace is lost, and is indicated via the *Support* trace packet (see table 7.3). When 1, the **stall** output signal is asserted to stall the RISC-V hart until trace can be accepted (see table 4.8).
Empty	1	O	R	1	Reads as 1 when all trace has been emitted. Note: this status is also indicated via the *Support* trace packet (see table 7.3).
ResyncMode	2	M	RW	SD	Selects the resynchronization mechanism. At least one non-zero mechanism must be implemented. 0: Off 1: Count trace packets 2: Count clock cycles 3: Count instruction (16-bit) half-words
ResyncMax	4	O	RW	SD	The maximum interval (in units determined by **ResyncMode**) between synchronization packets (see tables 7.2 and 7.3) is $2^{\textbf{ResyncMax}+4}$.

7.2 Format 3 subformat 0 - Synchronisation

This packet contains all the information the decoder needs to fully identify an instruction. It is sent for the first traced instruction (unless that instruction also happens to be the first in an exception handler), and when resynchronization has been scheduled by expiry of the resynchronisation timer.

Table 7.1: Packet format 3, subformat 0

Field name	Bits	Description
format	2	11 (sync): synchronisation
subformat	2	00 (start): Start of tracing, or resync
branch	1	Set to 0 if the address points to a branch instruction, and the branch was taken. Set to 1 if the instruction is not a branch or if the branch is not taken.
privilege	*privilege_width_p*	The privilege level of the reported instruction
time	*time_width_p* or 0 if *notime_p* is 1	The time value.
context	*context_width_p*, or 0 if *nocontext_p* is 1	The instruction context.
address	*iaddress_width_p* - *iaddress_lsb_p*	Full instruction address. Address alignment is determined by *iaddress_lsb_p* Address must be left shifted in order to recreate original byte address.