Advanced Matchlib Topics 2021

Stuart Swan | Platform Architect
Siemens EDA, a part of Siemens Digital Industries Software

SIEMENS

Catapult SystemC / Matchlib Resources

This presentation assumes you have reviewed the slides and examples below:

Matchlib training slides

- $MGC_HOME/shared/examples/matchlib/toolkit/doc/matchlib_customer_training.ppt

Matchlib example kit in Catapult install

- $MGC_HOME/shared/examples/matchlib/toolkit

Brief Review of Matchlib Interfaces

- Message Passing Interfaces : Push/Pop, PushNB/PopNB
  - $MGC_HOME/shared/examples/matchlib/toolkit/examples/05_push_pop
- Data + Valid Protocol :
  - …toolkit/examples/15_data_valid_protocol
- Event Pulse :
  - …toolkit/examples/94_event_pulse
- Process Sync:
  - …toolkit/examples/12_ping_pong_mem
- External Dual Port RAM:
  - …toolkit/examples/12_ping_pong_mem
- ARM AXI4:
  - …toolkit/examples/08_dma
- ARM APB:
  - …toolkit/examples/52_apb

Brief Review of Matchlib Interfaces (continued)

- Native SystemC signals :
  - …toolkit/examples/04_ready_valid
- Toggle Protocol :
  - …toolkit/examples/13_toggle_protocol
- FIFO instantiation between modules :
  - …toolkit/examples/10_fifo_channel_hier
- FIFO instantiation between processes within a module :
  - …toolkit/examples/11_fifo_channel_flat

Brief Review of Primary Matchlib Features

- Throughput accurate modeling in Pre-HLS simulation
  - …toolkit/examples/06_thruput_accurate
- Waveform generation in pre-HLS simulation
  - …toolkit/examples/06_thruput_accurate
- Channel log generation in pre-HLS simulation with auto-comparison utilities
  - …toolkit/doc/matchlib_soc_debug_tutorial.pdf
- Supports multiple clocks, sync/async resets with automatic consistency checking
  - …toolkit/examples/95_multi_clk
- Mixed language simulation and debug support in Questa, VCS, and Xcelium
  - …toolkit/examples/45_vlog_tb_dma_dut

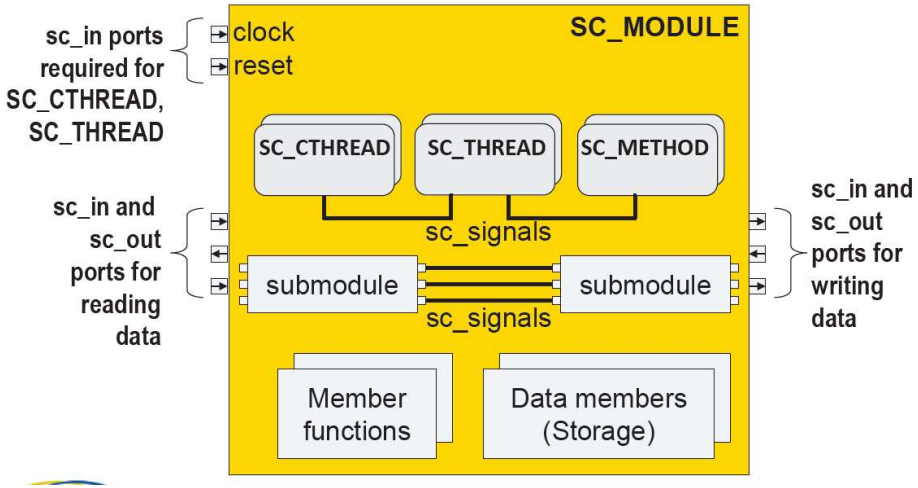
Brief Review of Primary Matchlib Features (continued)

- ~30X RTL simulation speed in ACCURATE_SIM mode
- ~300X RTL simulation speed in FAST_SIM mode
- Random stall injection in pre-HLS model (Matchlib DUT and TB.)
  - …toolkit/doc/matchlib_soc_debug_tutorial.pdf
- Random stall injection in post-HLS model (Catapult RTL via STALL_FLAG_SV)
- Thruput and latency backannotation
  - …toolkit/examples/71_annotate_simple

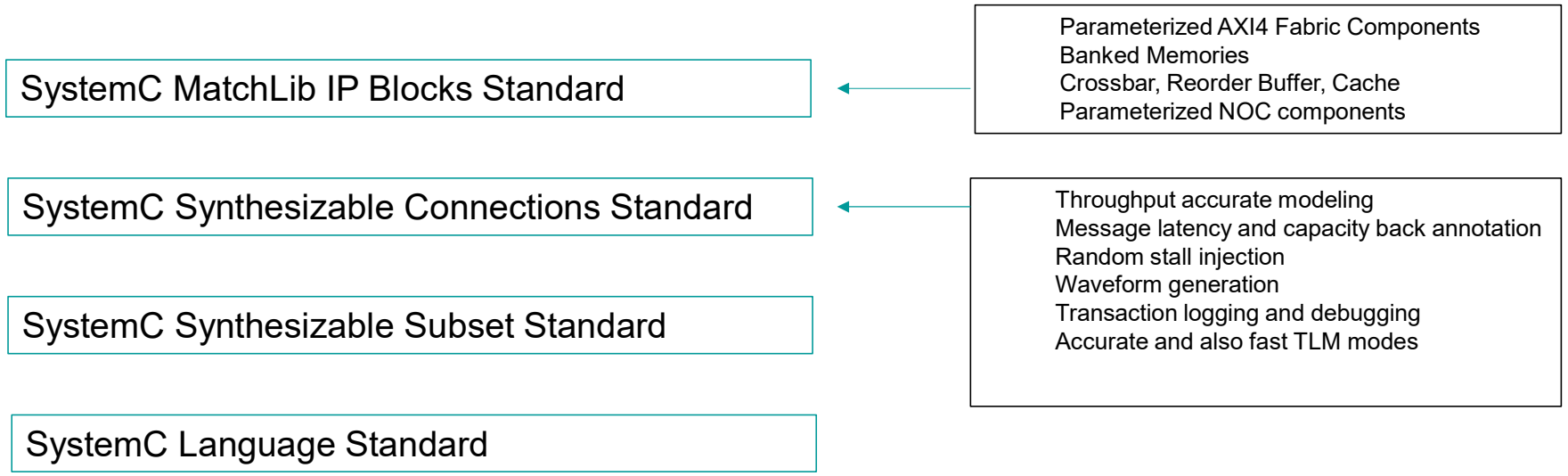
MatchLib Relationship to SystemC Standards

Accellera SystemC Synthesizeable Subset Standard focuses on what's in diagram below

- Modules, Ports, Processes, clocks, resets, signal IO, datatypes



Proposed SystemC HLS Standards Layers



Matchlib Key Coding Guidelines:
When do you need to use Non-blocking IO?

- Arbitration requires Peek or PopNB to all arbitrated inputs.
- Time-based splitting and merging of transaction streams requires PushNB and PopNB (respectively)
- Even when using NB IO, it is best to have at least one blocking IO call in each iteration of a loop
  - This ensures Catapult understands a clear relationship between II of loop and IO behavior.
- There are cases where a process will need ALL non-blocking IO (all PopNB and PushNB), but it should be pretty rare.
- In this case the process should be kept as small / simple as possible, ideally communicating with other processes that follow the guidelines above.
- With all non-blocking IO, you will likely be modeling at very close to RTL level, and most likely HLS will just be translating SC RTL into Verilog RTL.

Matchlib: Always prefer to use blocking IO over nonblocking IO.

- Your models will be simpler and more likely to have a good process structure.
- 100% blocking IO is called KPN (Kahn Process Networks)
  - KPN is deterministic
  - easier to verify.
- Non-blocking IO is sometimes needed, but introduces timing dependent behavior, and can make verification more difficult in some cases if the timing dependent behavior is externally visible.
- Summary: always ask yourself if use of non-blocking IO is really justified.

Key Guidelines for Coding for Good QOR in SC

- Simulate and debug your pre-HLS model before you synthesize (!)
  - Verify performance and functionality in your pre-HLS model
- In SystemC HLS, you should always have:
  - At least a rough idea of what your HW implementation will be (e.g. pipeline characteristics)
  - An exact idea of what your module pin level interfaces are
- Refine your architecture in your pre-HLS model
  - Usually this is by far the most effective way to improve QOR
- If functionality can easily be split into smaller processes, it is usually better to do so
  - Control FSMs generated by Catapult will be smaller
  - Smaller processes may be able to run in parallel

Can I use “feature X” with Catapult SystemC Flow?

- AC Datatypes – Yes
- AC Math – Yes
- AC DSP – Functions only, no hierarchy or instantiated AC channels
- AC channel – No
- “Interface synthesis” – Generally, No
- “Interface synthesis for C arrays in single process” - Yes
- Ccores – Yes
- Blackboxes – Yes
- Catapult Directives – Generally, Yes
- Coupled_IO - Yes

Matchlib Accuracy

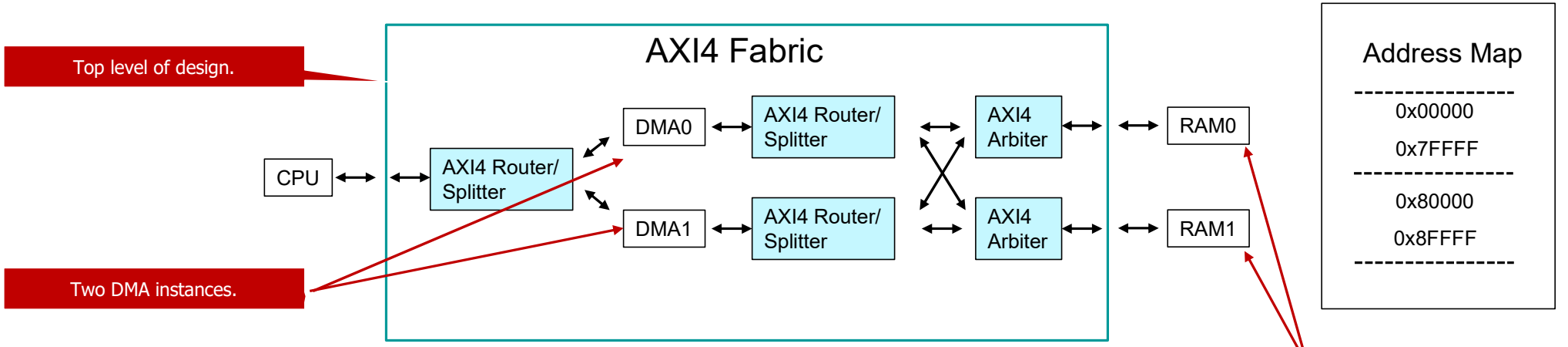
- Goal of Matchlib is to be “thoughput accurate” in pre-HLS models, even as you scale up size of systems.
- Goal is NOT to be precisely cycle accurate wrt RTL
- Even with some differences in accuracy wrt RTL, there is big value in being able to model time/performance in the pre-HLS models.
- It is useful to understand where sources of inaccuracies may arise and how to address them if desired.

Small Example

- Matchlib assumes all processes are pipelined with an II=1.
- Matchlib assumes stall_mode=flush.
- Matchlib assumes coupled_io=false.
- Matchlib assumes pipeline latency = 1
- Pre-HLS sim of model on right will Pop and Push one transaction on every clock.
  - Output will appear one clock after input.
- Post-HLS RTL will have same behavior except latency will increase.
  - ac_sqrt likely to require several clock cycles latency in RTL.

```
12   Connections::Out<uint32> CCS_INIT_S1(out1);
13   Connections::In <uint32> CCS_INIT_S1(in1);
14
15   SC_CTOR(dut) {
16     SC_THREAD(main);
17     sensitive << clk.pos();
18     async_reset_signal_is(rst_bar, false);
19   }
20
21 private:
22
23   void main() {
24     out1.Reset();
25     in1.Reset();
26     wait();
27 #pragma hls_pipeline_init_interval 1
28 #pragma pipeline_stall_mode flush
29     while (1) {
30       uint32_t i = in1.Pop();
31       uint32_t o = ac_sqrt(i);
32       out1.Push(o);
33     }
34   }
35 };
36
```

Larger Example: AXI4 Bus Fabric using MatchLib




```
/**
 *  * \brief fabric module
*/
#pragma hls_design top
class fabric : public sc_module, public local_axi {
public:
  sc_in<bool> INIT_S1(clk);
  sc_in<bool> INIT_S1(rst_bar);

  r_master INIT_S1(r_master0);
  w_master INIT_S1(w_master0);
  r_master INIT_S1(r_master1);
  w_master INIT_S1(w_master1);
  r_slave  INIT_S1(r_slave0);
  w_slave  INIT_S1(w_slave0);
  Connections::Out<bool> INIT_S1(dma0_done);
  Connections::Out<bool> INIT_S1(dma1_done);
```

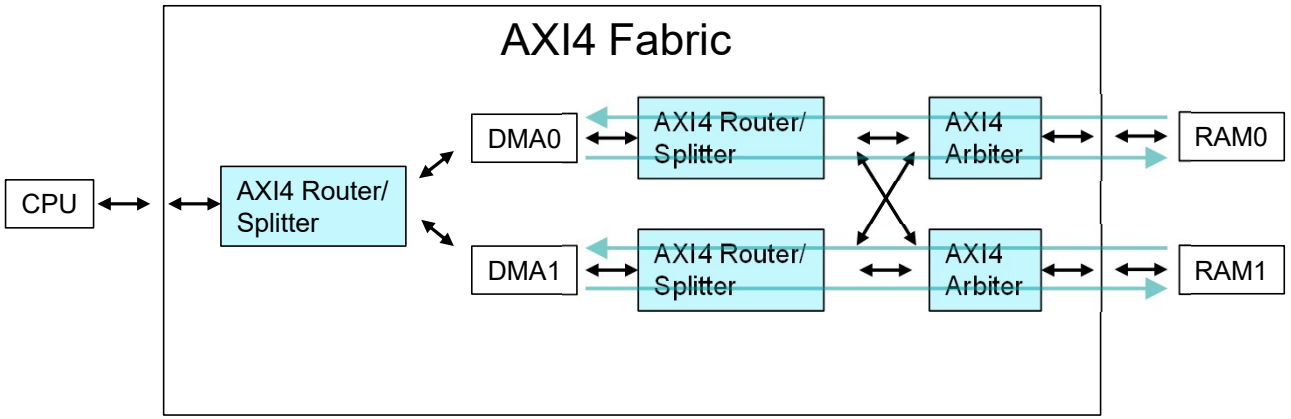
This example is in the Catapult Matchlib examples kit as 09_axi_fabric

Blue boxes are MatchLib Components

= top level of design

DMA0 and DMA1 are two instances of DMA shown earlier

AXI4 Bus Fabric using MatchLib – Test #0



RAM0 and RAM1 each have one read and one write port

Test #0: Concurrently,
DMA0 reads/writes 320 beats to RAM0
DMA1 reads/writes 320 beats to RAM1

(a beat is the data chunk transferred on the bus in 1 clock cycle)

AXI4 Bus Fabric Test #0 simulation logs

BEFORE HLS (SystemC simulation)

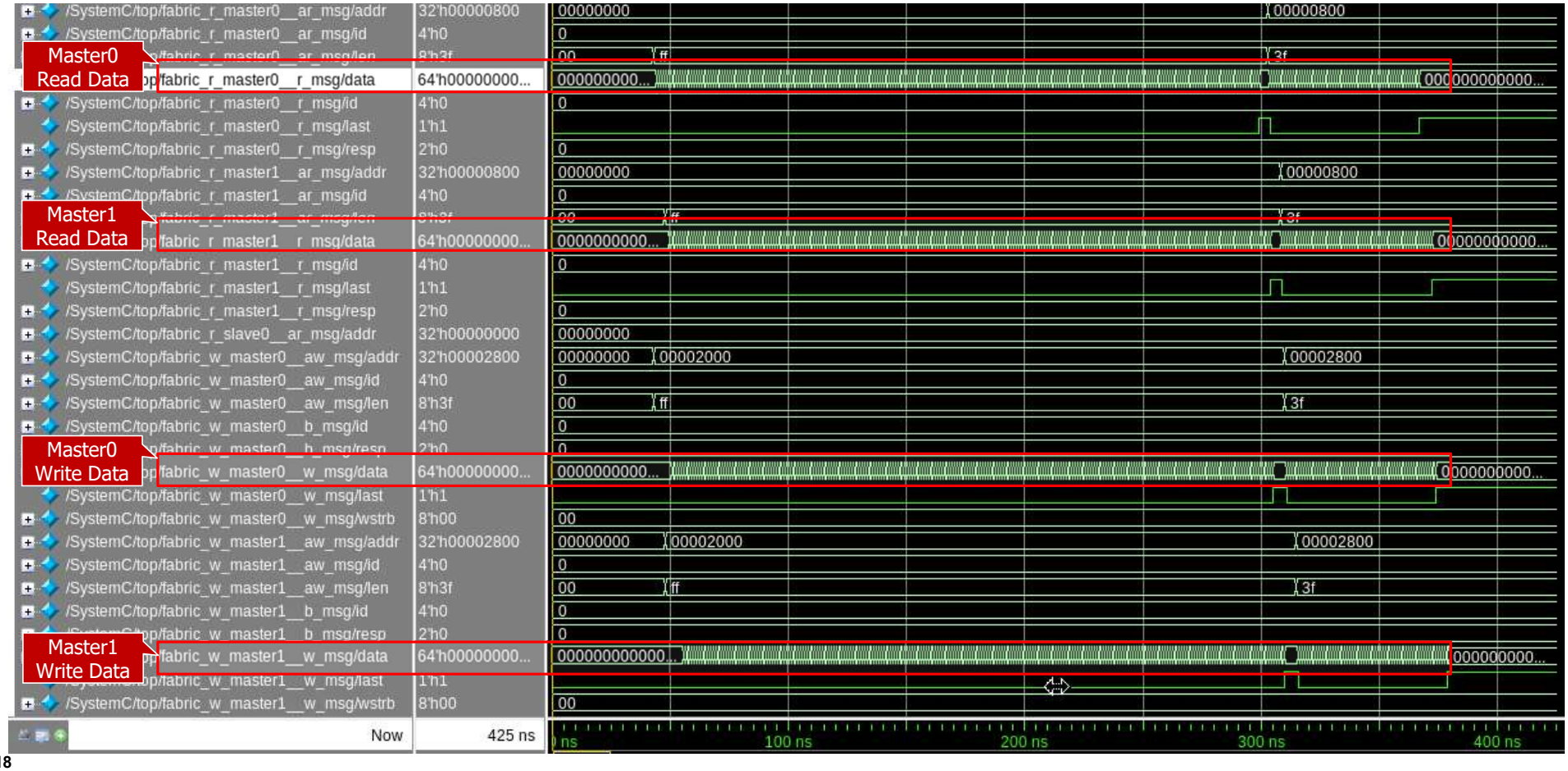
```
0 s top Stimulus started
6 ns top Running FABRIC_TEST # : 0
44 ns top.ram0 ram read  addr: 000000000 len: 0ff
44 ns top.ram0 ram write addr: 000002000 len: 0ff
49 ns top.ram1 ram write addr: 000002000 len: 0ff
49 ns top.ram1 ram read  addr: 000000000 len: 0ff
304 ns top.ram0 ram read  addr: 000000800 len: 03f
309 ns top.ram1 ram read  addr: 000000800 len: 03f
311 ns top.ram0 ram write addr: 000002800 len: 03f
316 ns top.ram1 ram write addr: 000002800 len: 03f
385 ns top dma_done detected. 1 1
385 ns top start_time: 46 ns end_time: 385 ns
385 ns top axi beats (dec): 320
385 ns top elapsed time: 339 ns
385 ns top beat rate: 1059 ps
385 ns top clock period: 1 ns
425 ns top finished checking memory contents
```

AFTER HLS (Verilog RTL simulation)

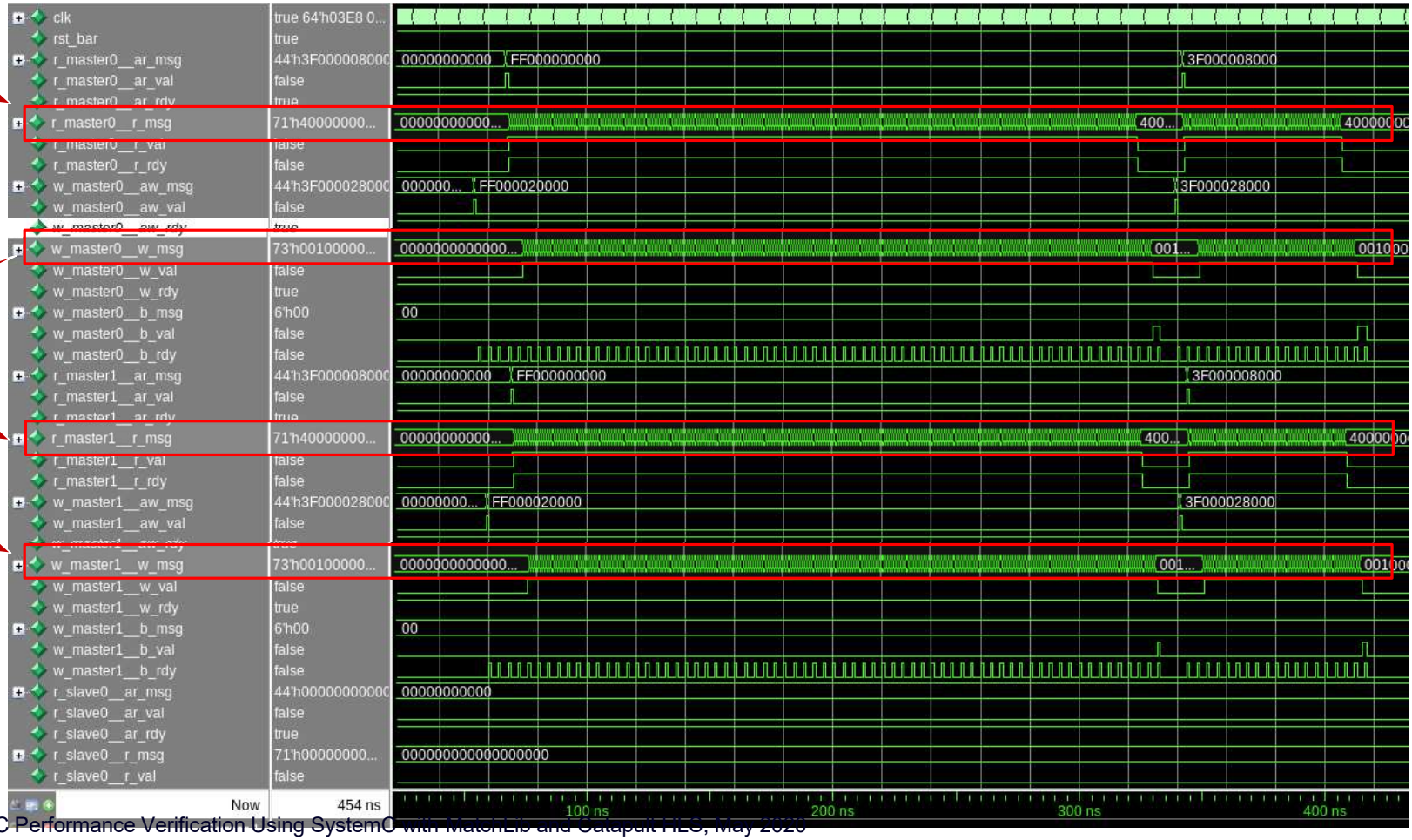
```
# 0 s top Stimulus started
# 6 ns top Running FABRIC_TEST # : 0
# 55 ns top/ram0 ram write addr: 000002000 len: 0ff
# 60 ns top/ram1 ram write addr: 000002000 len: 0ff
# 68 ns top/ram0 ram read  addr: 000000000 len: 0ff
# 70 ns top/ram1 ram read  addr: 000000000 len: 0ff
# 340 ns top/ram0 ram write addr: 000002800 len: 03f
# 342 ns top/ram1 ram write addr: 000002800 len: 03f
# 343 ns top/ram0 ram read  addr: 000000800 len: 03f
# 345 ns top/ram1 ram read  addr: 000000800 len: 03f
# 414 ns top dma_done detected. 1 1
# 414 ns top start_time: 55 ns end_time: 414 ns
# 414 ns top axi beats (dec): 320
# 414 ns top elapsed time: 359 ns
# 414 ns top beat rate: 1122 ps
# 414 ns top clock period: 1 ns
# 454 ns top finished checking memory contents
```

Before and after HLS we get nearly one beat per clock cycle

AXI4 Fabric Waveforms Before Catapult HLS–Test #0 (SystemC)

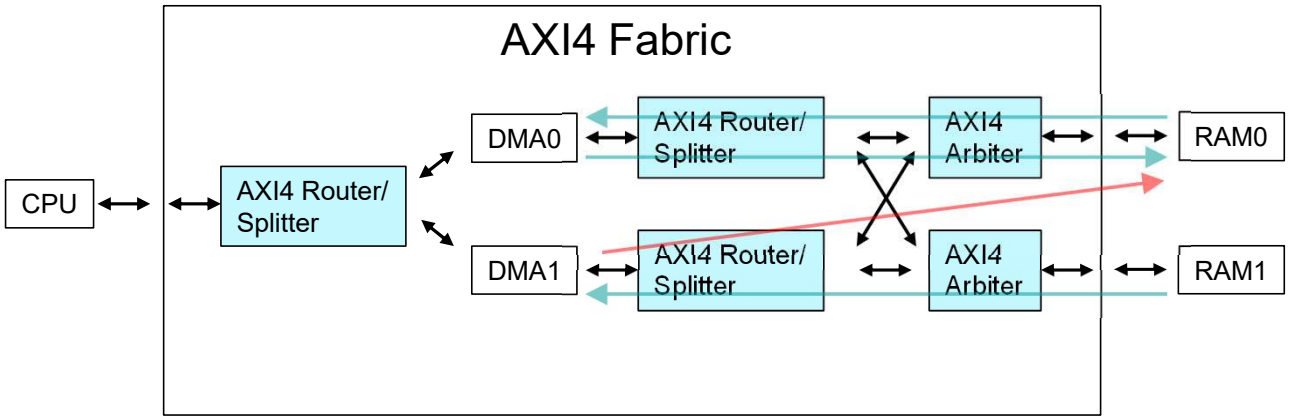


AXI4 Fabric Waveforms After Catapult HLS – Test #0 (Verilog)



Thoughput in RTL Matches SystemC

AXI4 Bus Fabric using MatchLib – Test #1



RAM0 and RAM1 each have one read and one write port

Test #1: Concurrently,
DMA0 reads/writes 320 beats to RAM0
DMA1 reads 320 beats from RAM1 and writes to RAM0
Note contention on RAM0 writes

AXI4 Bus Fabric Test #1 simulation logs

BEFORE HLS (SystemC simulation)

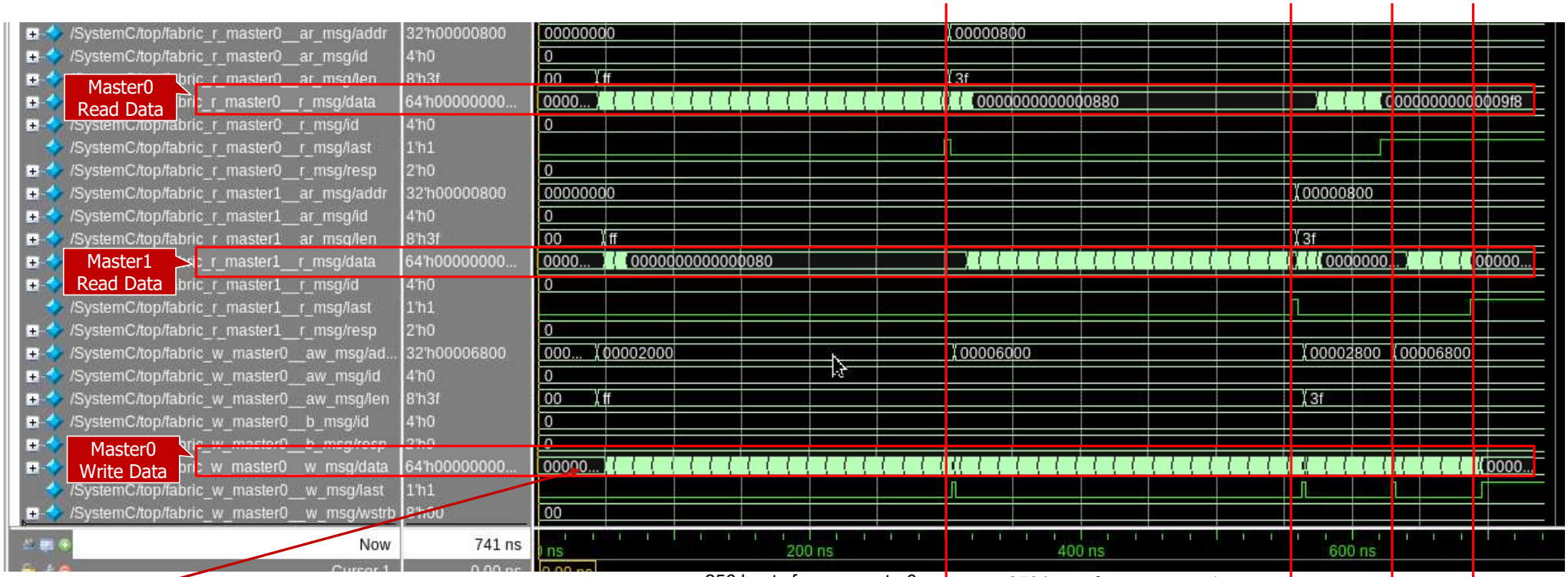
```
0 s top Stimulus started
6 ns top Running FABRIC_TEST # : 1
44 ns top.ram0 ram read  addr: 000000000 len: 0ff
44 ns top.ram0 ram write addr: 000002000 len: 0ff
49 ns top.ram1 ram read  addr: 000000000 len: 0ff
304 ns top.ram0 ram read  addr: 000000800 len: 03f
308 ns top.ram0 ram write addr: 000006000 len: 0ff
560 ns top.ram1 ram read  addr: 000000800 len: 03f
566 ns top.ram0 ram write addr: 000002800 len: 03f
632 ns top.ram0 ram write addr: 000006800 len: 03f
701 ns top dma_done detected. 1 1
701 ns top start_time: 46 ns end_time: 701 ns
701 ns top axi beats (dec): 320
701 ns top elapsed time: 655 ns
701 ns top beat rate: 2047 ps
701 ns top clock period: 1 ns
741 ns top finished checking memory contents
```

AFTER HLS (Verilog RTL simulation)

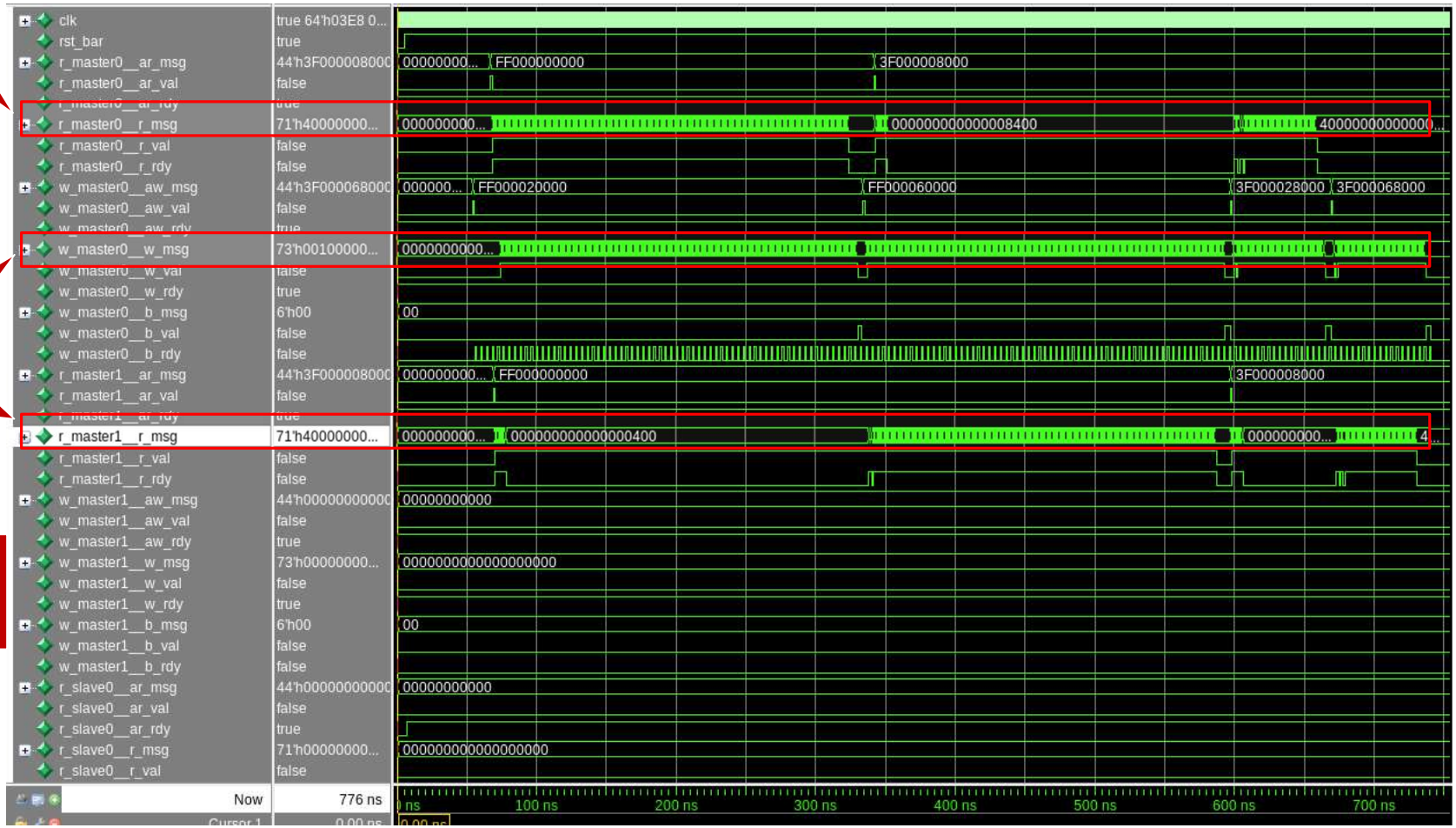
```
# 0 s top Stimulus started
# 6 ns top Running FABRIC_TEST # : 1
# 55 ns top/ram0 ram write addr: 000002000 len: 0ff
# 68 ns top/ram0 ram read  addr: 000000000 len: 0ff
# 70 ns top/ram1 ram read  addr: 000000000 len: 0ff
# 335 ns top/ram0 ram write addr: 000006000 len: 0ff
# 343 ns top/ram0 ram read  addr: 000000800 len: 03f
# 598 ns top/ram1 ram read  addr: 000000800 len: 03f
# 598 ns top/ram0 ram write addr: 000002800 len: 03f
# 670 ns top/ram0 ram write addr: 000006800 len: 03f
# 736 ns top dma_done detected. 1 1
# 736 ns top start_time: 55 ns end_time: 736 ns
# 736 ns top axi beats (dec): 320
# 736 ns top elapsed time: 681 ns
# 736 ns top beat rate: 2128 ps
# 736 ns top clock period: 1 ns
# 776 ns top finished checking memory contents
```

Two concurrent writes to RAM0 cause beat rate to be above two clock cycles.

AXI4 Fabric Waveforms Before Catapult HLS –Test#1 (SystemC)



AXI4 Fabric Waveforms After Catapult HLS – Test #1 (Verilog)



SS, Early SOC Performance Verification Using SystemC with MatchLib and Catapult HLS, May 2020

Key Observations

- “Throughput accuracy” of Matchlib pre-HLS models is generally retained even as you scale up size of systems.
- Ability to view, analyze, and debug time-based behaviors and waveforms in pre-HLS models is very valuable to:
  - HW architects
  - HLS engineers
  - DV engineers
  - RTL designers on project who will “never touch” HLS or C++/SC models.

Matchlib “throughput accuracy” is high for:

- Feedforward data streams
- Feedback data streams
- Systems with multiple clocks and multiple data stream rates
- Divergent data streams
- Arbitration of data streams (for any reasonably fair arbitration scheme)
  - e.g. round robin

Design aspects which require extra handling:

- If II != 1, then need to manually insert waits into pre-HLS model to account for this.
- Currently memory accesses for both internal and external memories are not accounted for in Matchlib timing
  - In other words, Matchlib currently assumes that low level memory accesses themselves are not primary bottlenecks.
  - If they are, then that should be explicitly modeled using Push/Pop or wait() statements.
  - This may be improved in a future release.
- As mentioned earlier, if you use PushNB/PopNB in your DUT or TB, you may introduce timing dependent behavior that can cause pre-HLS vs post-HLS mismatches.
  - But PushNB/PopNB do not introduce timing inaccuracies per se.

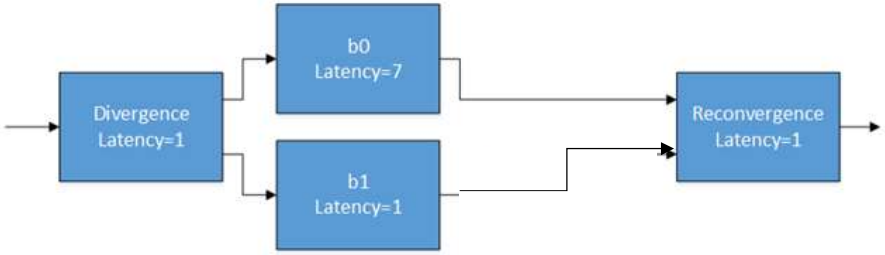
```
  void main() {
    out1.Reset();
    in1.Reset();
    wait();
#pragma hls_pipeline_init_interval 4
#pragma pipeline_stall_mode flush
    while (1) {
#ifndef __SYNTHESIS__
      wait();
      wait();
      wait();
#endif
      uint32_t t = in1.Pop();
      out1.Push(t + 0x100);
    }
  }
};
```

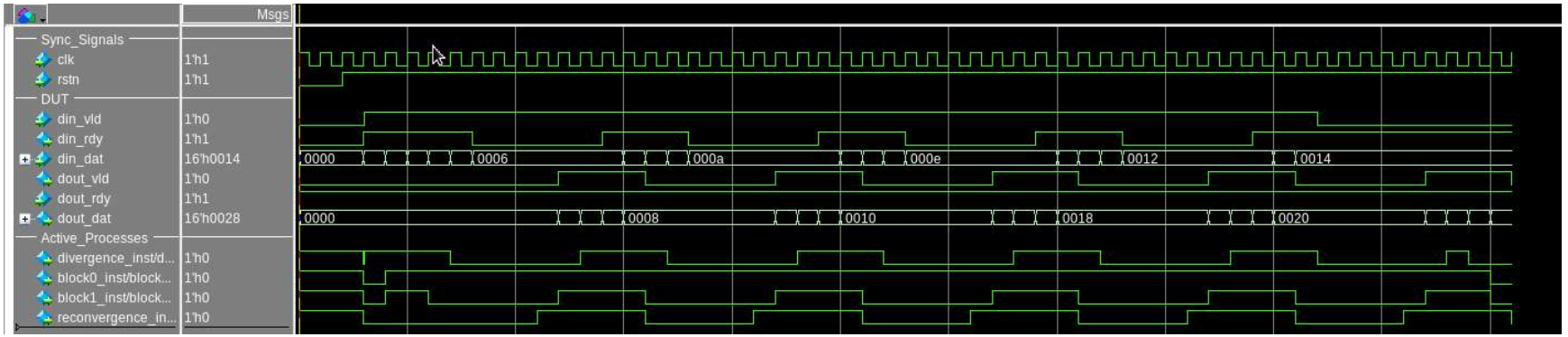
How Catapult changes latency and capacity of design…

- When Catapult synthesizes processes/blocks it typically adds latency
  - In other words, post-HLS model will usually have higher latency than pre-HLS model.
- If loop pipelining is used, then it typically also adds capacity.
  - In effect, the HW pipeline is like a HW fifo with the functionality “folded in”.
  - The HW pipeline acts very much like a HW fifo, especially if stall_mode=flush.
- If coupled_io_mode=true is used, then this removes a small amount of capacity and elasticity as compared to pre-HLS model.
- Matchlib supports a “latency and capacity” back-annotation feature to easily back-annotate post-HLS values back into pre-HLS model.
  - This increases accuracy significantly of the pre-HLS sim versus the post-HLS sim.
  - However, you should still not expect exact matches in timing behavior.

Stuttering Design Example (72* : bagel shop example)

- II=1 for all blocks
- Latency of b0 is 1 pre-HLS (by default)
- Latency of b0 is 7 post-HLS
- Post-HLS waveforms below show stuttering due to reconverging data streams with unbalanced latencies and capacities.

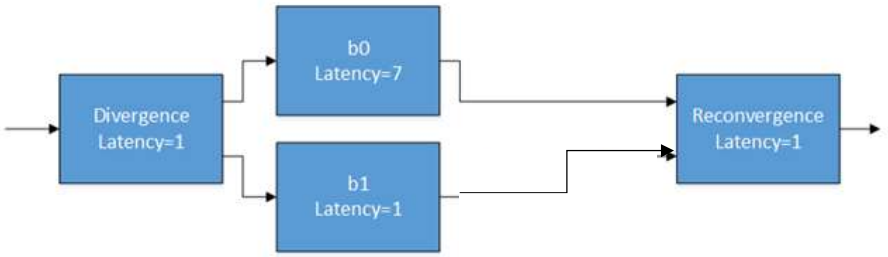


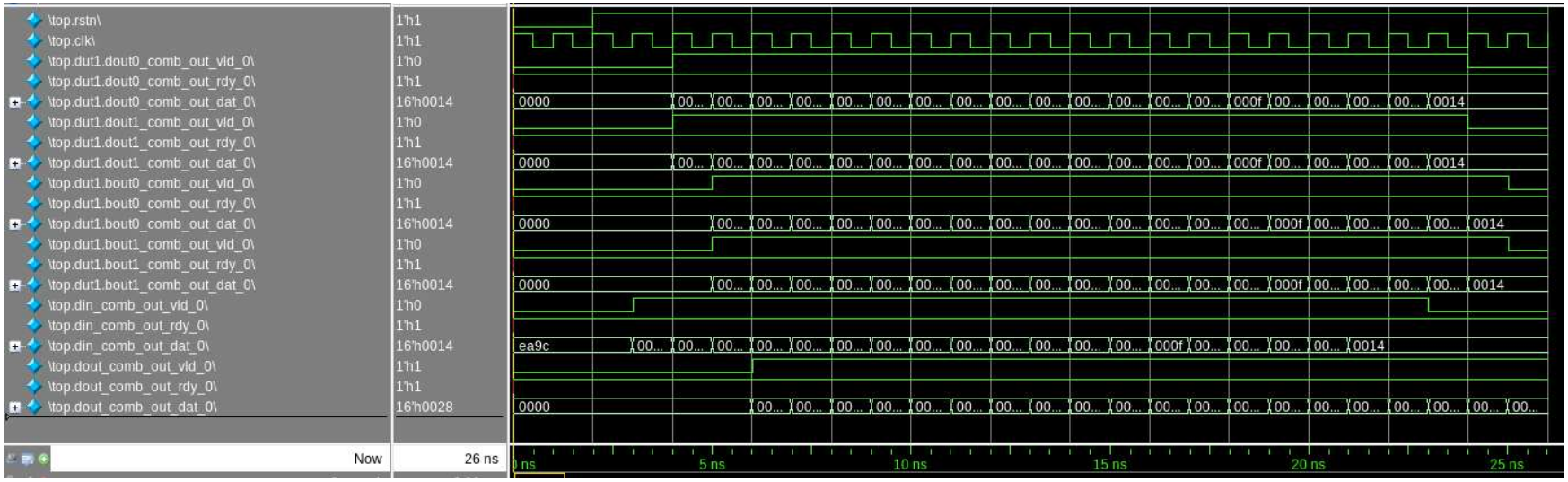


SIEMENS

Stuttering Design Example Pre-HLS Default Waveforms

- Latency of b0 is 1 pre-HLS (by default)
- Default pre-HLS waveforms below show no stuttering – new output is seen on every clock.



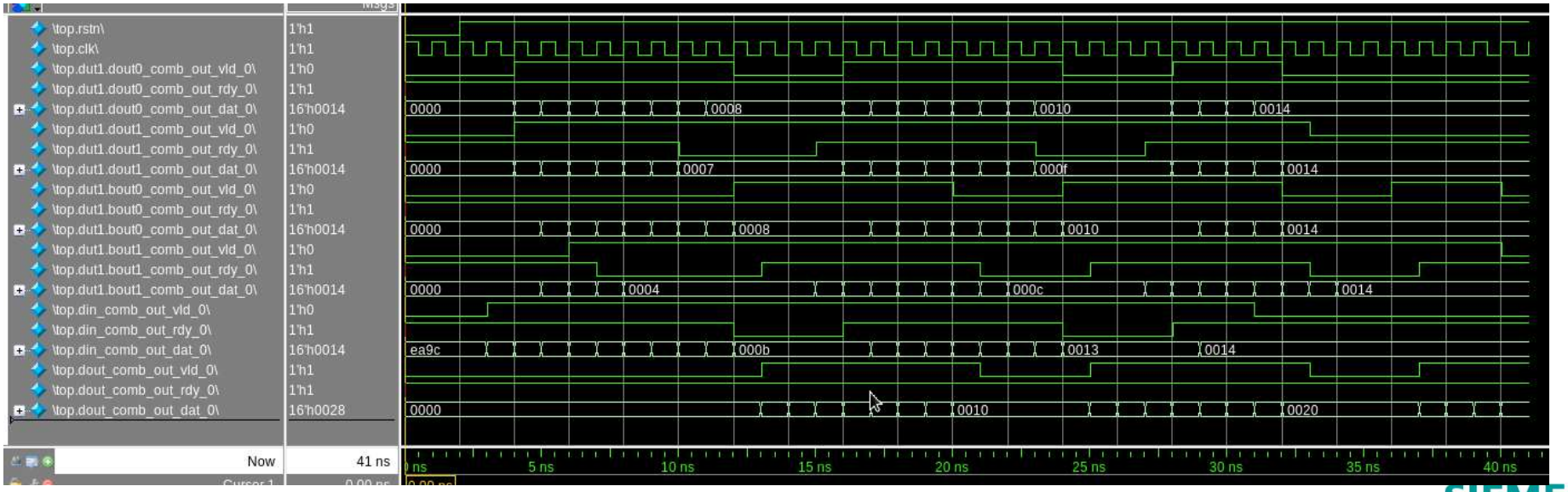


SIEMENS

Stuttering Design Example Pre-HLS with back-annotation

- Now we back-annotate latency and capacity of b0 and b1
- Pre-HLS waveforms below now show stuttering

```
"dut1.bout0_comb_BA": {
    "latency": 7,
    "capacity": 8,
    "src_name": "dut1.block0_inst.dout_vld",
    "dest_name": "dut1.reconvergence_inst.din0_vld"
},
"dut1.bout1_comb_BA": {
    "latency": 1,
    "capacity": 2,
    "src_name": "dut1.block1_inst.dout_vld",
    "dest_name": "dut1.reconvergence_inst.din1_vld"
},
```



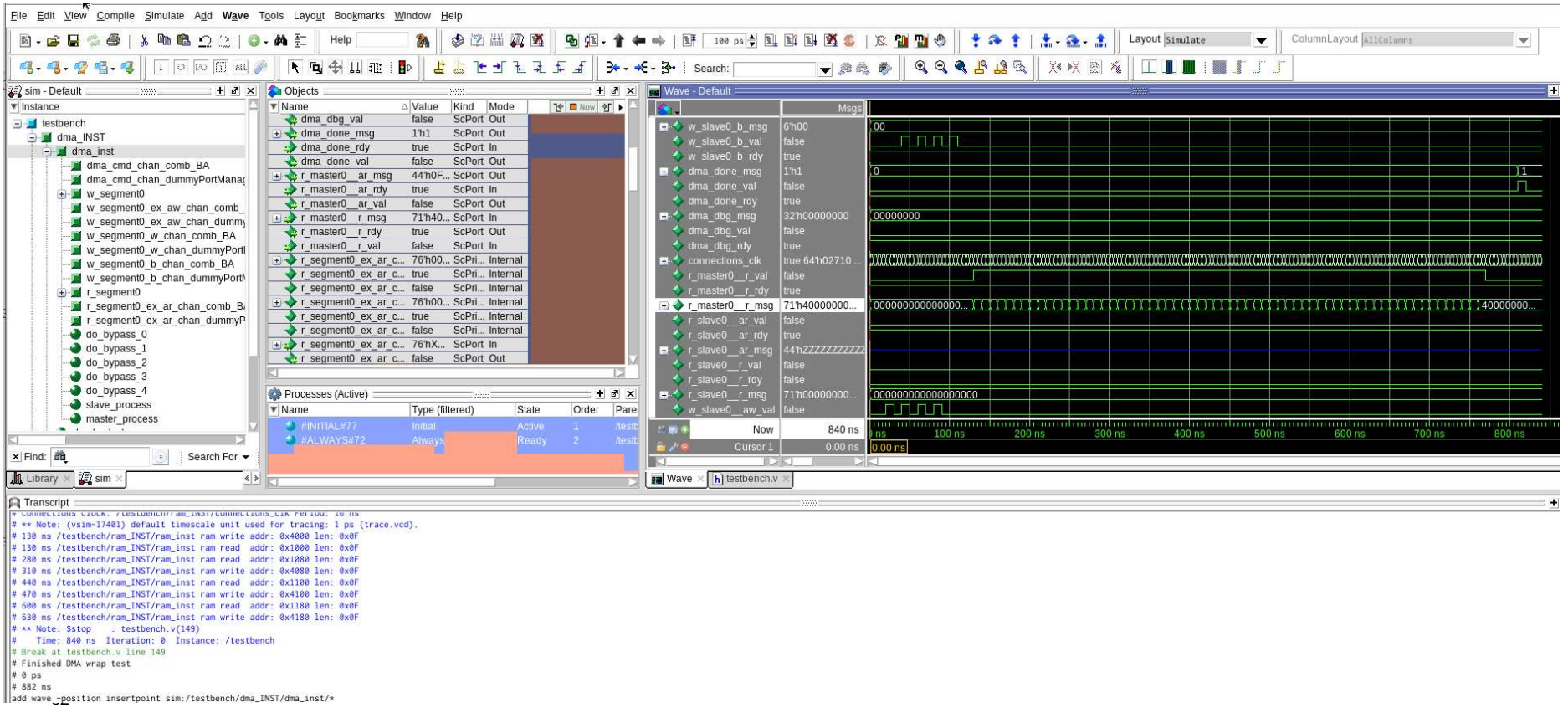
SIEMENS

Areas where Matchlib “throughput accuracy” is harder

- Stuttering due to re-convergent data streams (the example just shown)
  - Back annotation can fairly easily show existence of issue and point to the solution
  - Achieving exact match with timing behavior of post-HLS model is harder because:
    - Overall thruput is highly dependent on local latencies and capacities in various blocks
    - Even small things like coupled_io=true or stall_mode=stall can have a big impact on stuttering in RTL
- Thorough validation that system has no deadlock behaviors
  - Matchlib is useful in finding and resolving many deadlock scenarios in pre-HLS model
  - However, some deadlock scenarios are highly dependent on detailed cycle level behavior and RTL latencies/capacities, and must be verified in the RTL.
- A useful strategy for both of the above scenarios is to “sweep” the latency/capacity values in the pre-HLS back-annotation to stress test the model
  - This will increase the chances that the RTL has no bugs
  - It will also be much easier to debug any issues that are found (as compared to debugging in RTL).

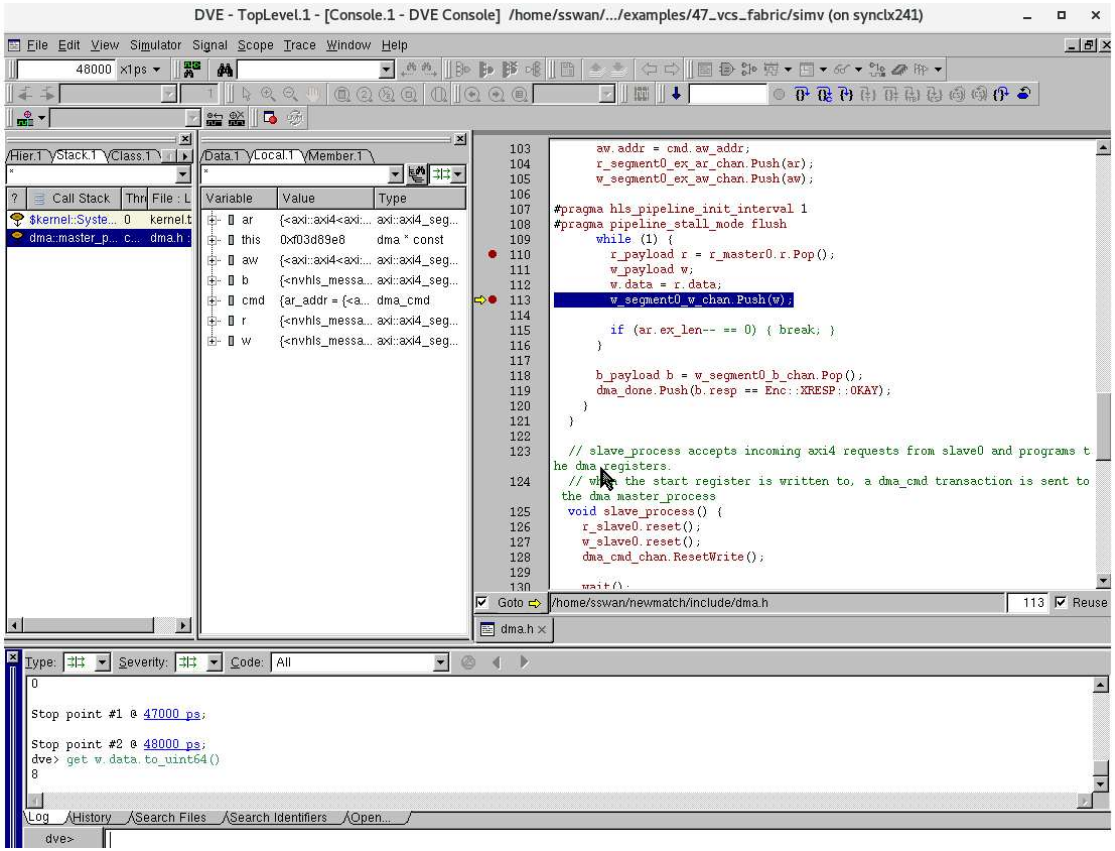
Pre-HLS Debugging in Questa, VCS, Xcelium

- HW – aware debugging in pre-HLS model, can support mixed language (e.g. SV UVM TB + SC DUT)

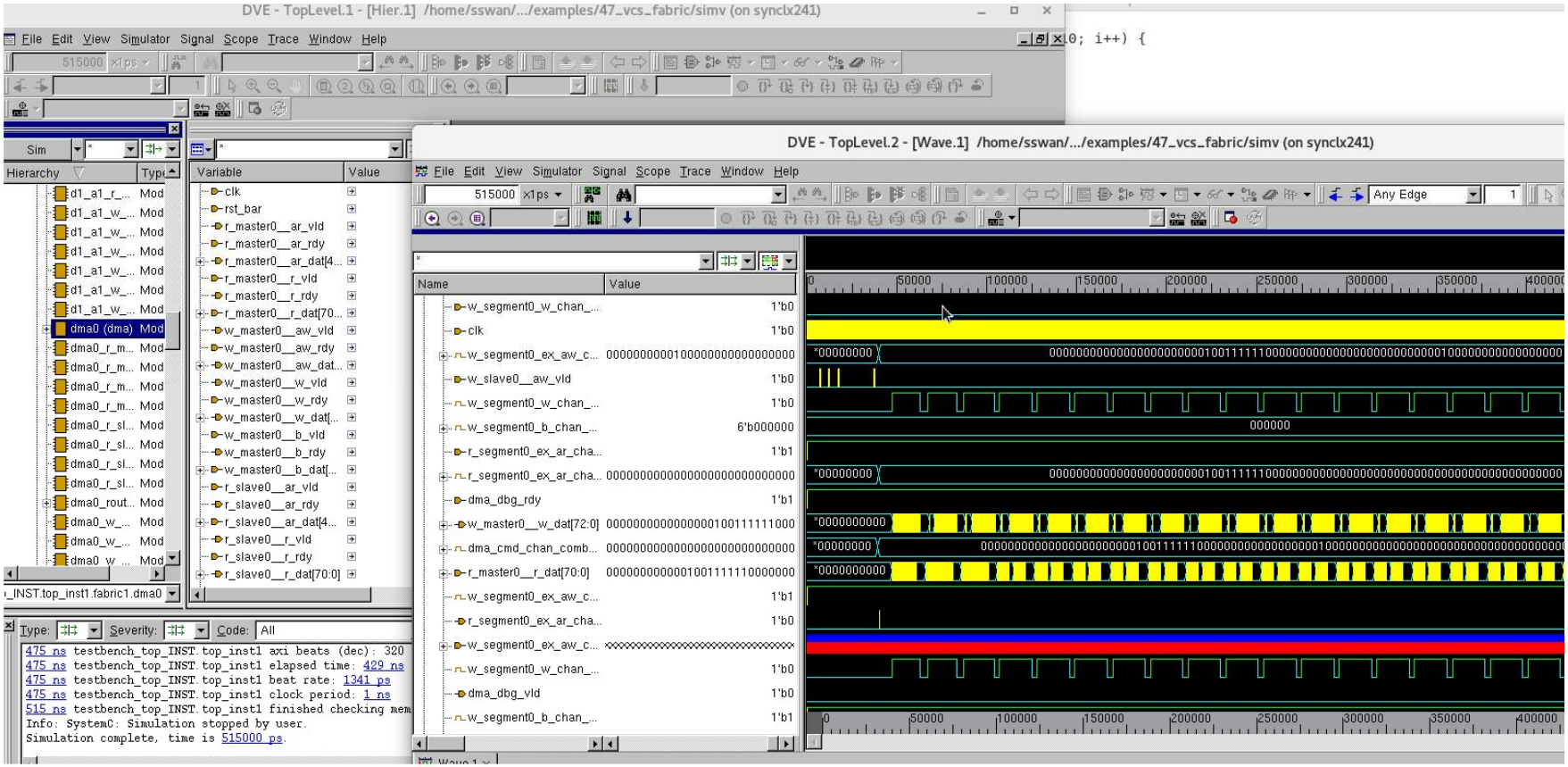


Pre-HLS Debugging in Questa, VCS, Xcelium

- Key advantage: HW aware debugging and throughput accurate modeling with quick edit/debug turnaround..



Pre-HLS Debugging in Questa, VCS, Xcelium (cont)



Supporting Custom Protocols in Catapult SystemC & Matchlib

- First question: Do you really need a new custom protocol?
  - Most modern on-chip bus protocols use message passing (e.g. AXI3/4, NOC protocols, etc).
  - Message passing means "ready/valid/data" signaling.
  - Message passing protocols are easily layered on top of Matchlib Connections
  - Existing AXI4 transactors are built using this approach:
    - See: $MGC_HOME/shared/examples/matchlib/toolkit/doc/matchlib_customer_training.ppt

- Second Question: Is it a very simple signal level protocol?
  - Very simple signal level protocols can be easily built by combining Connections::In/Out with sc_signals
  - See $MGC_HOME/shared/examples/matchlib/toolkit/examples/13_toggle_protocol

Non-Trivial Custom Protocol Example: ARM APB (52_apb)

- **Step 1**: Study the protocol spec (structural interface and state descriptions)
  - See: https://developer.arm.com/documentation/ihi0024/c

Signal	Source	Description
PCLK	Clock source	Clock. The rising edge of **PCLK** times all transfers on the APB.
PRESETn	System bus equivalent	Reset. The APB reset signal is active LOW. This signal is normally connected directly to the system bus reset signal.
PADDR	APB bridge	Address. This is the APB address bus. It can be up to 32 bits wide and is driven by the peripheral bus bridge unit.
PPROT	APB bridge	Protection type. This signal indicates the normal, privileged, or secure protection level of the transaction and whether the transaction is a data access or an instruction access.
PSELx	APB bridge	Select. The APB bridge unit generates this signal to each peripheral bus slave. It indicates that the slave device is selected and that a data transfer is required. There is a **PSELx** signal for each slave.
PENABLE	APB bridge	Enable. This signal indicates the second and subsequent cycles of an APB transfer.
PWRITE	APB bridge	Direction. This signal indicates an APB write access when HIGH and an APB read access when LOW.
PWDATA	APB bridge	Write data. This bus is driven by the peripheral bus bridge unit during write cycles when **PWRITE** is HIGH. This bus can be up to 32 bits wide.
PSTRB	APB bridge	Write strobes. This signal indicates which byte lanes to update during a write transfer. There is one write strobe for each eight bits of the write data bus. Therefore, **PSTRB[n]** corresponds to **PWDATA[(8n + 7):(8n)]**. Write strobes must not be active during a read transfer.
PREADY	Slave interface	Ready. The slave uses this signal to extend an APB transfer.
PRDATA	Slave interface	Read Data. The selected slave drives this bus during read cycles when **PWRITE** is LOW. This bus can be up to 32-bits wide.
PSLVERR	Slave interface	This signal indicates a transfer failure. APB peripherals are not required to support the **PSLVERR** pin. This is true for both existing and new APB peripheral designs. Where a peripheral does not include this pin then the appropriate input to the APB bridge is tied LOW.

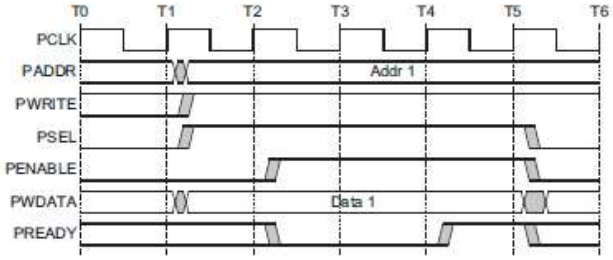


Figure 3-2 Write transfer with wait states

PREADY can take any value when **PENABLE** is LOW. This ensures that peripherals that have a fixed two cycle access can tie **PREADY** HIGH.

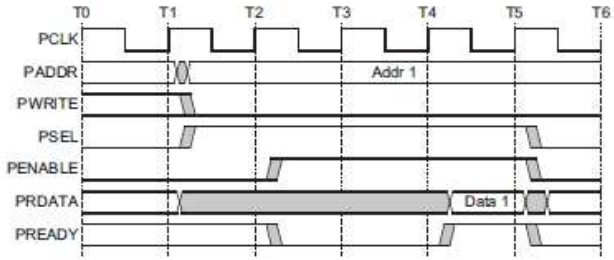


Figure 3-5 Read transfer with wait states

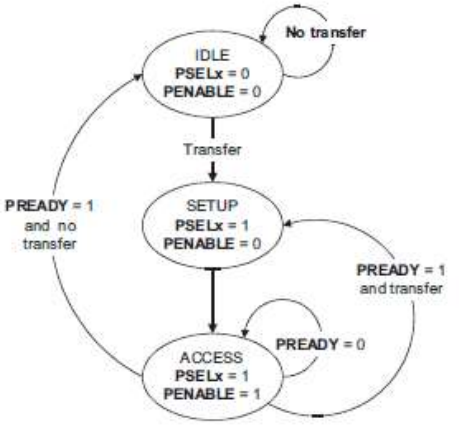


Figure 4-1 State diagram

Note: FSM perspective is bus, not master or slave. So it is a bit misleading..

SIEMENS

Non-Trivial Custom Protocol Example: Step 2

- Abstract the protocol up to the message passing level.
  - Goal: User's HLS model uses message passing interfaces only (Connections::In/Out).
  - All timing details and signal handshakes are hidden inside transactor.
  - We need to define transaction types and the number of message passing interfaces that user's model will see.
  - To get good QOR and be as flexible as possible, the strategy is to keep the message passing interfaces "as close to the HW protocol" as possible while still abstracting away timing behaviors.
- User's HLS model will almost always use blocking message passing calls exclusively.
- Protocol transactor will almost always use non-blocking message passing calls exclusively.
  - It does this while simultaneously managing the signal level protocol and detailed timing.
- Note: Catapult does not support Stratus "protocol regions", and we do not need it to support them.
  - What we do instead is push the "protocol region" into its own thread (in the transactor) and use message passing to communicate with it.

Non-Trivial Custom Protocol Example: Step 2 (cont.)

- For protocols such as AHB which support pipelined behaviors and bursts, it is important to consider how to design the interfaces and transactors to support full throughput.
  - Message passing interfaces make this easier to achieve since they are easily pipelined.

- APB is a simple protocol with no bursts and no simultaneous reads and writes.
- We can simply have a single "req" channel which contains the request, and a single "rsp" channel which contains the response.
  - These two channels are shared between reads and writes because the APB wires for reads and writes are also shared.
  - For a HW protocol which had fully separate wires for reads and writes we would want fully separate wr_req/wr_rsp and rd_req/rd_rsp channels.

Non-Trivial Custom Protocol Example: Step 3

- Create classes to represent APB channels and ports

```
struct apb_sig_chan
{
  apb_sig_chan(const char* name)
   : PSEL(nvhls_concat(name, "_PSEL"))
   , PADDR(nvhls_concat(name, "_PADDR"))
   , PWRITE(nvhls_concat(name, "_PWRITE"))
   , PENABLE(nvhls_concat(name, "_PENABLE"))
   , PWDATA(nvhls_concat(name, "_PWDATA"))
   , PSTRB(nvhls_concat(name, "_PSTRB"))
   , PPROT(nvhls_concat(name, "_PPROT"))
   , PRDATA(nvhls_concat(name, "_PRDATA"))
   , PSLVERR(nvhls_concat(name, "_PSLVERR"))
   , PREADY(nvhls_concat(name, "_PREADY"))
   {}

  SC_SIG(bool,     PSEL);
  SC_SIG(Addr,     PADDR);
  SC_SIG(bool,     PWRITE);
  SC_SIG(bool,     PENABLE);
  SC_SIG(Data,     PWDATA);
  SC_SIG(Wstrb,    PSTRB);
  SC_SIG(Prot_t,   PPROT);
  SC_SIG(Data,     PRDATA);
  SC_SIG(bool,     PSLVERR);
  SC_SIG(bool,     PREADY);
};
```

```
struct apb_master_ports
{
  apb_master_ports(const char* name)
   : PSEL(nvhls_concat(name, "_PSEL"))
   , PADDR(nvhls_concat(name, "_PADDR"))
   , PWRITE(nvhls_concat(name, "_PWRITE"))
   , PENABLE(nvhls_concat(name, "_PENABLE"))
   , PWDATA(nvhls_concat(name, "_PWDATA"))
   , PSTRB(nvhls_concat(name, "_PSTRB"))
   , PPROT(nvhls_concat(name, "_PPROT"))
   , PRDATA(nvhls_concat(name, "_PRDATA"))
   , PSLVERR(nvhls_concat(name, "_PSLVERR"))
   , PREADY(nvhls_concat(name, "_PREADY"))
   {}

  sc_out<bool> PSEL;
  sc_out<Addr> PADDR;
  sc_out<bool> PWRITE;
  sc_out<bool> PENABLE;
  sc_out<Data> PWDATA;
  sc_out<Wstrb> PSTRB;
  sc_out<Prot_t> PPROT;
  sc_in<Data>  PRDATA;
  sc_in<bool>  PSLVERR;
  sc_in<bool>  PREADY;

  template <class C>
  void operator()(C &c) {
        PADDR(c.PADDR);
```

```
struct apb_slave_ports
{
  apb_slave_ports(const char* name)
   : PADDR(nvhls_concat(name, "_PADDR"))
   , PWRITE(nvhls_concat(name, "_PWRITE"))
   , PENABLE(nvhls_concat(name, "_PENABLE"))
   , PSEL(nvhls_concat(name, "_PSEL"))
   , PWDATA(nvhls_concat(name, "_PWDATA"))
   , PSTRB(nvhls_concat(name, "_PSTRB"))
   , PPROT(nvhls_concat(name, "_PPROT"))
   , PRDATA(nvhls_concat(name, "_PRDATA"))
   , PSLVERR(nvhls_concat(name, "_PSLVERR"))
   , PREADY(nvhls_concat(name, "_PREADY"))
   {}

  sc_in<Addr> PADDR;
  sc_in<bool> PWRITE;
  sc_in<bool> PENABLE;
  sc_in<bool> PSEL;
  sc_in<Data> PWDATA;
  sc_in<Wstrb> PSTRB;
  sc_in<Prot_t> PPROT;
  sc_out<Data> PRDATA;
  sc_out<bool> PSLVERR;
  sc_out<bool> PREADY;

  template <class C>
  void operator()(C &c) {
        PADDR(c.PADDR);
```


SIEMENS

Non-Trivial Custom Protocol Example: Step 4

- Define transaction message classes

```
struct apb_req : public nvhls_message {
 bool         is_write { false };
 addr_payload addr;
 w_payload    w;

  static const unsigned int width = 1 + addr_payload::
  template <unsigned int Size> void Marshall(Marshalle
    m &is_write;
    m &addr;
    m &w;
  }
  inline friend void sc_trace(sc_trace_file *tf, const
    sc_trace(tf,v.is_write,  NAME + ".is_write");
    sc_trace(tf,v.addr,  NAME + ".addr");
    sc_trace(tf,v.w,  NAME + ".w");
    }
  inline friend std::ostream &operator<<(ostream &os,
    os << rhs.is_write << " ";
    os << rhs.addr << " ";
    os << rhs.w << " ";
    return os;
  }
};
```

```
struct apb_rsp : public nvhls_message {
 r_payload    r; // if req was a write, then write resp

  static const unsigned int width = r_payload::width;
  template <unsigned int Size> void Marshall(Marshaller<
    m &r;
  }
  inline friend void sc_trace(sc_trace_file *tf, const a
    sc_trace(tf,v.r,  NAME + ".r");
    }
  inline friend std::ostream &operator<<(ostream &os, co
    os << rhs.r << " ";
    return os;
  }
};
```


SIEMENS

Non-Trivial Custom Protocol Example: Step 5

- Create the protocol transactor code
  - Strategy: Use PopNB/PushNB to set “got_req” and “got_rsp” flags while simultaneously managing the signal level protocol.

```
190 template <Connections::connections_port_t PortType = AUTO_PORT>
191 class apb_master_xactor : public sc_module {
192 public:
193   sc_in<bool> CCS_INIT_S1(clk);
194   sc_in<bool> CCS_INIT_S1(rst_bar);
195
196   Connections::In<apb_req, PortType>   CCS_INIT_S1(req_port);
197   Connections::Out<apb_rsp, PortType>  CCS_INIT_S1(rsp_port);
198   sc_out<Addr>     CCS_INIT_S1(PADDR);
199   sc_out<bool>     CCS_INIT_S1(PWRITE);
```

```
230   void main() {
231     // reset all sigs here..
232     wait();
233
234     bool got_req = false;
235     bool got_rsp = false;
236     apb_req req;
237     apb_rsp rsp;
238
239     while (1) {
240      // See ARM APB spec for state description.
241      do {
242       // IDLE state
243       wait();
244       if (!got_req)
245       {
246        got_req = req_port.PopNB(req);
247       }
248
249       if (got_rsp)
250        if (rsp_port.PushNB(rsp))
251        {
252          got_rsp = 0;
253        }
254
255      } while (got_rsp || !got_req);
256
```

```
258      do {
259       wait();
260       if (!got_req)
261       {
262        got_req = req_port.PopNB(req);
263       }
264
265       if (got_rsp)
266        if (rsp_port.PushNB(rsp))
267        {
268          got_rsp = 0;
269        }
270       // SETUP state
271       PSEL = 1;
272       PENABLE = 0;
273       PADDR = req.addr.addr.to_uint64();
274       PWRITE = req.is_write;
275       if (req.is_write) {
276         PWDATA = req.w.data.to_uint64();
277         PSTRB = req.w.wstrb.to_uint64();
278       }
279       else {
280         PWDATA = 0;
281         PSTRB = 0;
282       }
283
284       wait();
285
286       got_req = 0;
287       PSEL = 1;
288       PENABLE = 1;
289
290       do {
291         wait();
292       } while (PREADY == 0);
293       // ACCESS state
294
295       rsp.r.data = PRDATA.read().to_uint64();
296       rsp.r.resp = PSLVERR.read();
```

Non-Trivial Custom Protocol Example: Step 5 (cont.)

- Create the protocol transactor code (apb_slave_transactor here)

```
341  void main() {
348    // reset all sigs here..
349    bool got_req = false;
350    bool got_rsp = false;
351    bool pending_read = false;
352    apb_req req;
353    apb_rsp rsp;
354
355    wait();
356
357    while (1) {
358     // See ARM APB spec for state description.
359
360     do {
361      wait();
362     } while (PSEL == 0);
363
364     req.is_write = PWRITE.read();
365     req.w.data = PWDATA.read().to_uint64();
366     req.w.wstrb = PSTRB.read().to_uint64();
367     req.addr.addr = PADDR.read().to_uint64();
368     got_req = 1;
369
370     if (req.is_write)
371       pending_read = false;
372     else
373       pending_read = true;
374
375     do {
376      if (got_req)
377       if (req_port.PushNB(req))
378       {
379        got_req = 0;
380       }
381       wait();
382     } while (got_req);
383
```

```
306 template <Connections::connections_port_t PortType = AUTO_PORT>
307 class apb_slave_xactor : public sc_module {
308 public:
309   sc_in<bool> CCS_INIT_S1(clk);
310   sc_in<bool> CCS_INIT_S1(rst_bar);
311
312   Connections::Out<apb_req, PortType> CCS_INIT_S1(req_port);
313   Connections::In<apb_rsp, PortType>  CCS_INIT_S1(rsp_port);
314   sc_in<Addr>       CCS_INIT_S1(PADDR);
315   sc_in<bool>       CCS_INIT_S1(PWRITE);
316   sc_in<bool>       CCS_INIT_S1(PENABLE);
317   sc_in<bool>       CCS_INIT_S1(PSEL);
318   sc_in<Data>       CCS_INIT_S1(PWDATA);
319   sc_in<Wstrb >     CCS_INIT_S1(PSTRB);
```

```
384     do {
385      wait();
386      if (!got_rsp)
387      {
388        got_rsp = rsp_port.PopNB(rsp);
389      }
390     } while ((PENABLE == 0) || !got_rsp);
391
392     PREADY = 1;
393     if (pending_read)
394     {
395       PRDATA = rsp.r.data.to_uint64();
396       pending_read = false;
397     }
398
399     PSLVERR = rsp.r.resp;  // works for both reads and writes..
400     wait();
401     PREADY = 0;
402     PSLVERR = 0;
403     got_rsp = 0;
404    }
405  }
```


SIEMENS

Non-Trivial Custom Protocol Example: Step 6

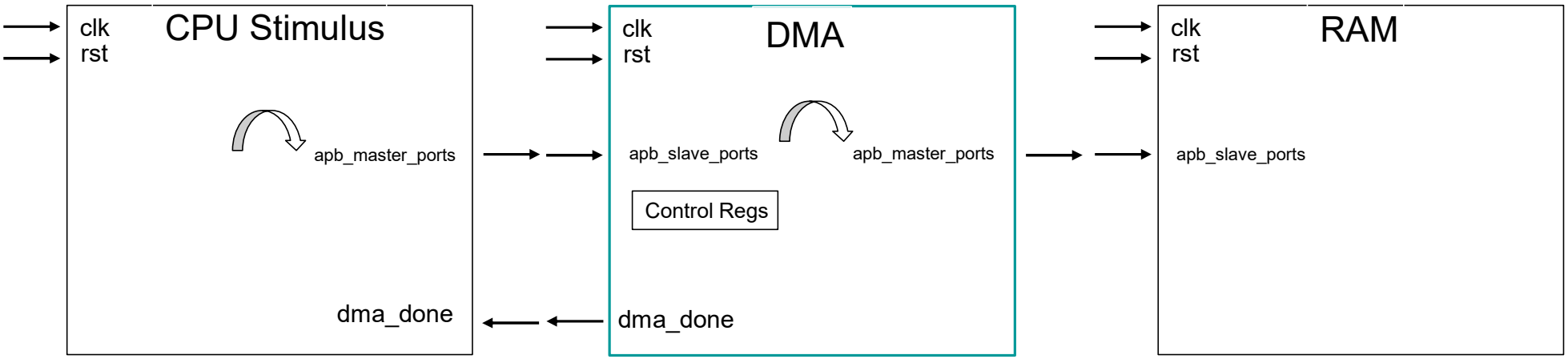
- Instantiate transactor in user's HLS model

```
50 class dma : public sc_module, public local_apb {
51 public:
52   sc_in<bool> CCS_INIT_S1(clk);
53   sc_in<bool> CCS_INIT_S1(rst_bar);
54
55   apb_master_xactor<> CCS_INIT_S1(master0_xactor);
56   apb_master_ports<> CCS_INIT_S1(master0_ports);
57   apb_req_chan CCS_INIT_S1(master0_req_chan);
58   apb_rsp_chan CCS_INIT_S1(master0_rsp_chan);
59
60   apb_slave_xactor<>  CCS_INIT_S1(slave0_xactor);
61   apb_slave_ports<> CCS_INIT_S1(slave0_ports);
62   apb_req_chan CCS_INIT_S1(slave0_req_chan);
63   apb_rsp_chan CCS_INIT_S1(slave0_rsp_chan);
64
65   Connections::Out<bool> CCS_INIT_S1(dma_done);
66   Connections::Out<uint32_t> CCS_INIT_S1(dma_dbg);
67
68
69   SC_CTOR(dma)
70   {
71     SC_THREAD(slave_process);
72     sensitive << clk.pos();
73     async_reset_signal_is(rst_bar, false);
74
75     SC_THREAD(master_process);
76     sensitive << clk.pos();
77     async_reset_signal_is(rst_bar, false);
78
79     slave0_xactor.clk(clk);
80     slave0_xactor.rst_bar(rst_bar);
81     slave0_xactor.req_port(slave0_req_chan);
82     slave0_xactor.rsp_port(slave0_rsp_chan);
83     slave0_xactor(slave0_ports);
84
85     master0_xactor.clk(clk);
86     master0_xactor.rst_bar(rst_bar);
87     master0_xactor.req_port(master0_req_chan);
88     master0_xactor.rsp_port(master0_rsp_chan);
89     master0_xactor(master0_ports);
90   }
```

```
 99 void master_process() {
100
101   dma_cmd_chan.ResetRead();
102   dma_dbg.Reset();
103   dma_done.Reset();
104
105   master0_req_chan.ResetWrite();
106   master0_rsp_chan.ResetRead();
107
108   wait();
109
110   while(1) {
111     dma_cmd cmd = dma_cmd_chan.Pop();
112     bool status = Enc::XRESP::OKAY;
113     while (1)
114     {
115       apb_req req;
116       apb_rsp rsp;
117
118       req.is_write = false;
119       req.addr.addr = cmd.ar_addr;
120       master0_req_chan.Push(req);
121       rsp = master0_rsp_chan.Pop();
122
123       req.is_write = true;
124       req.addr.addr = cmd.aw_addr;
125       req.w.data = rsp.r.data;
126       master0_req_chan.Push(req);
127       rsp = master0_rsp_chan.Pop();
128
129       if (rsp.r.resp != Enc::XRESP::OKAY)
130         status = 0;
131
132       if (cmd.len-- == 0)
133         break;
134
135       cmd.aw_addr += bytesPerBeat;
136       cmd.ar_addr += bytesPerBeat;
137     }
138     dma_done.Push(status);
139   }
```


SIEMENS

52_apb Design Example



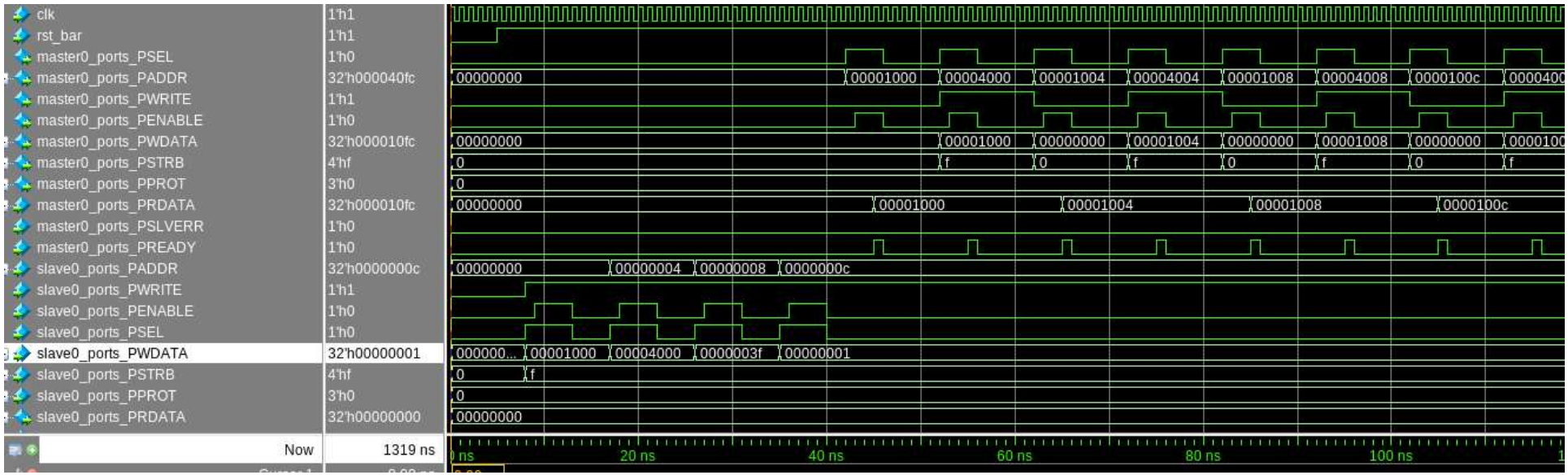

```
#pragma hls_design top
class dma : public sc_module, public local_apb {
public:
  sc_in<bool> CCS_INIT_S1(clk);
  sc_in<bool> CCS_INIT_S1(rst_bar);

  apb_master_xactor<> CCS_INIT_S1(master0_xactor);
  apb_master_ports<> CCS_INIT_S1(master0_ports);
  apb_req_chan CCS_INIT_S1(master0_req_chan);
  apb_rsp_chan CCS_INIT_S1(master0_rsp_chan);

  apb_slave_xactor<>  CCS_INIT_S1(slave0_xactor);
  apb_slave_ports<> CCS_INIT_S1(slave0_ports);
  apb_req_chan CCS_INIT_S1(slave0_req_chan);
  apb_rsp_chan CCS_INIT_S1(slave0_rsp_chan);

  Connections::Out<bool> CCS_INIT_S1(dma_done);
  Connections::Out<uint32_t> CCS_INIT_S1(dma_dbg);
```

= top level of design

SIEMENS

Running 52_apb Example RTL

Note clean preservation of all APB signal names in the Catapult generated Verilog RTL.



SIEMENS

Custom Protocol Concluding Thoughts

- If you are more comfortable writing Verilog RTL than SystemC, one approach is to write transactors in Verilog RTL first and then just translate to SC Matchlib afterwards.
  - After all, we are really just writing RTL in the transactors.

Shared Memories in SystemC Designs – Approach #1

Definition: A shared memory is an array in HLS that is preserved thru HLS that is accessed by more than one thread.

Approach #1: “Make the shared memory go away”

- This approach is what many Matchlib users do in practice.
- Use only preserved arrays which are accessed by a single thread
- Use architectural directives in Catapult to allocate ports for those rams, etc.
- Use Matchlib components such as Scratchpad, ArbitratedScratchpad, etc., to route requests from other threads to those arrays.
- If throughput is top concern, and a few extra cycles of latency is OK, this is a very robust and flexible approach.
- Pre-HLS sim will be throughput accurate under assumption that mem rd wr access to arrays within threads are not the bottleneck

Shared Memories in SystemC Designs – Approach #2

Approach #2: Use memory model from Catapult Memory Generator

- This is demonstrated in 12_ping_pong_mem
- Each thread accessing memory needs a dedicated RAM port
- There needs to be some synchronization scheme to avoid data races between threads
  - 12_ping_pong_mem uses Connections::SyncChannel
- Catapult memory generator SC models currently need “wait(0.3, SC_NS)” statements deleted from them so they work properly in Matchlib sims
- Catapult memory generator SC models currently do not participate in Matchlib “thruput accurate” simulation mechanism, so sim will not be thruput accurate if thread has > 1 mem accesses per clock

```
43   void thread2() {
44     bool ping_pong = false;
45     out1.Reset();
46     sync1.reset_sync_in();
47     wait();
48
49       #pragma hls_pipeline_init_interval 1
50       #pragma pipeline_stall_mode flush
51     while (1) {
52       sync1.sync_in();
53
54       for (int i=0; i < 8; i++)
55         out1.Push(mem[i + (8 * ping_pong)]);
56
57       ping_pong = !ping_pong;
58     }
59   }
60
61 private:
62   Connections::SyncChannel INIT_S1(sync1); // memory synchronization between threads
63   RAM_1R1W_model<>::mem<ac_int<16>,128> INIT_S1(mem);//Ping-pong shared memory
64 };
```

Memory read operation

Memory instance

Shared Memories in SystemC Designs – Approach #2 (cont.)

Example 12_ping_pong_mem shows shared memory shared by threads in same module.

- In this case, you do not need to explicitly code the mem read and write ports

You can also have memory ports on modules, so that memories can be external and/or shared between multiple modules.

- In this case, you do need to explicitly code the mem read and write ports on module interfaces

```
#include "RAM_1R1W.h"

typedef NVUINTW(32) design_T;

typedef RAM_1R1W_model<>::mem<design_T,16> mem_t;
typedef RAM_1R1W_model<>::rd0_port<design_T,16> mem_rd_t;


#pragma hls_design top
class dut : public sc_module {
public:
  sc_in<bool> INIT_S1(clk);
  sc_in<bool> INIT_S1(rst_bar);

  Connections::Out<NVUINTW(32)> INIT_S1(out1);
  Connections::In <NVUINTW(32)> INIT_S1(in1);
  mem_rd_t  INIT_S1(mem_rd_port0);

  ac_channel<design_T> chan_in1, chan_out1;

  SC_CTOR(dut)
  {
    chan_in1.bind(in1);
    chan_out1.bind(out1);

    SC_THREAD(main);
    sensitive << clk.pos();
    async_reset_signal_is(rst_bar, false);

  }
```

Memory types

Memory read port

```
class Top : public sc_module {
public:
  NVHLS_DESIGN(dut) INIT_S1(dut1);

  sc_clock clk;
  SC_SIG(bool, rst_bar);

  Connections::Combinational<NVUINTW(32)>        INIT_S1(out1);
  Connections::Combinational<NVUINTW(32)>        INIT_S1(in1);

  mem_t INIT_S1(mem0);

  SC_CTOR(Top)
     :   clk("clk", 1, SC_NS, 0.5,0,SC_NS,true)
  {
    Connections::set_sim_clk(&clk);
    sc_object_tracer<sc_clock> trace_clk(clk);

    dut1.clk(clk);
    dut1.rst_bar(rst_bar);
    dut1.out1(out1);
    dut1.in1(in1);
    dut1.mem_rd_port0(mem0);

    mem0.CK(clk);

    for (int i=0; i < 16; i++)
      mem0[i] = i;
```

Bind memory read port to memory instance

External Catapult SystemC / Matchlib Resources

Catapult SystemC MatchLib On-Demand Training (ODT)

- https://eda.learn.sw.siemens.com/training/courses/catapult-high-level-synthesis-library

2021 Catapult Virtual Seminar (ML Accelerator using Matchlib):

- https://event.on24.com/wcc/r/3187802/BF0CAE586A768CDB1314C956F64ABA35

Catapult Matchlib Webinars:

- https://webinars.sw.siemens.com/how-matchlib-and-systemc-enables
- https://webinars.sw.siemens.com/nvidia-design-and-verification-of-a-1
- https://webinars.sw.siemens.com/early-axi-soc-performance

Youtube video from NVidia:

- https://www.youtube.com/watch?v=n8_G-CaSSPU

Accellera SystemC Evolution Day 2020 Matchlib Presentation and Open Source Example Kit:

- https://forums.accellera.org/files/category/2-systemc/

Thank you!

SIEMENS