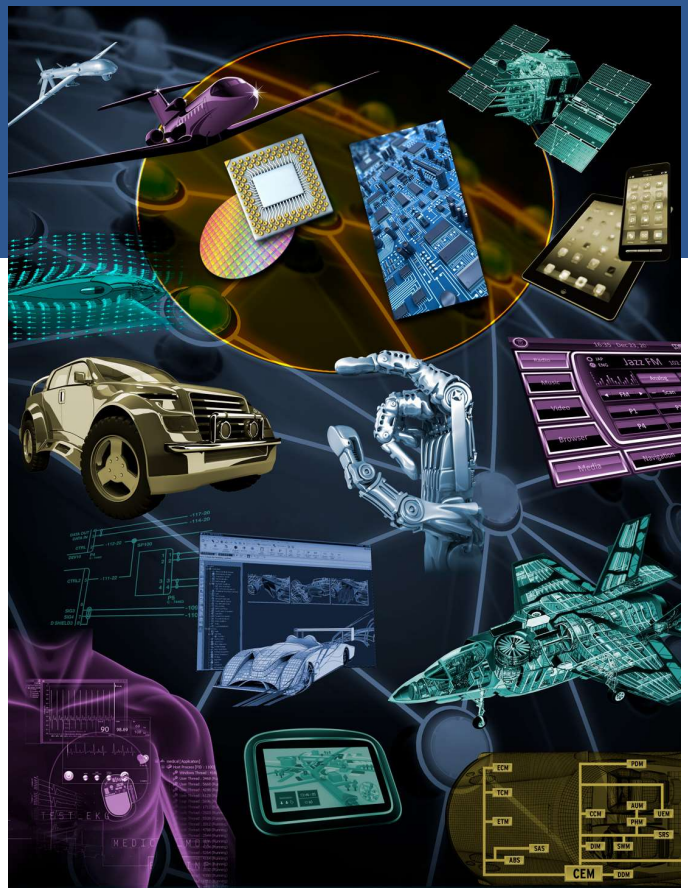


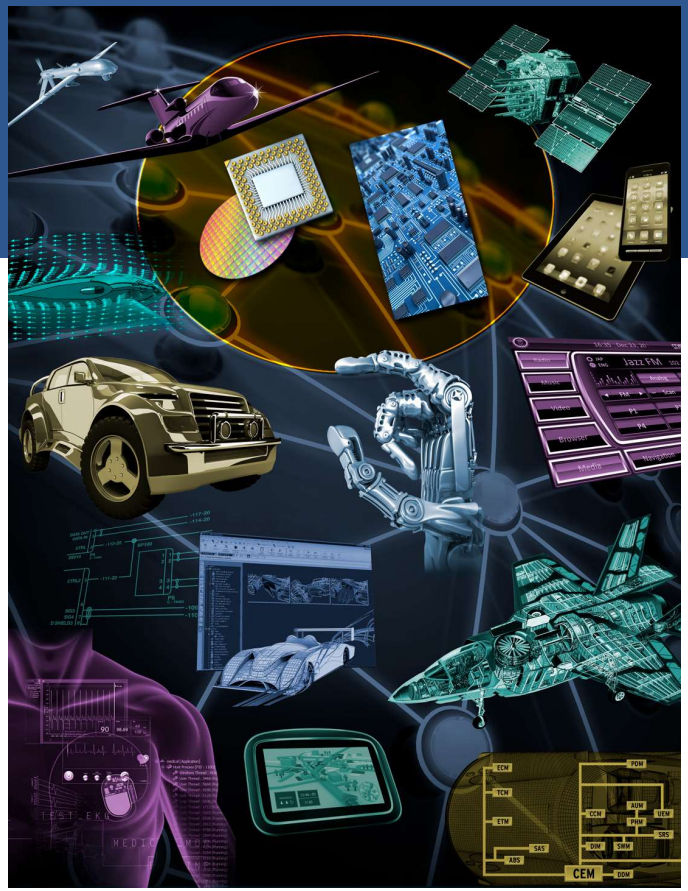
Catapult SystemC Matchlib Training July 2020

Stuart Swan

HLS IP/Platform Architect

July 2020

Mentor®
A Siemens Business



Basic SystemC and Matchlib Training

Stuart Swan

HLS IP/Platform Architect

July 2020

Mentor®
A Siemens Business

Why use SystemC + Matchlib for HLS?

- SystemC provides HDL semantics on top of C++ language
  - time, module structure, hierarchy, channels, HW semantics, resets, signals
- SC enables time based behaviors to be cleanly modeled and verified prior to synthesis (as compared to pure C++ HLS)
- SC integrates nicely with HDL simulators
- Matchlib enables models to be "throughput accurate" pre-HLS
- Matchlib provides commonly used components ready for HLS

How does Catapult SC synthesis differ from C++ synthesis ?

- Primary difference is in input language
- Once code is read in by Catapult, 90% of flow and usage is same.

Important to understand how HLS sees your SystemC model:

- Set of processes communicating **only** thru signals (sc_signal<>)
- For synthesis purposes, hierarchy is irrelevant (it is preserved in the RTL however)
- HLS synthesis occurs on each process one at a time, in complete isolation
  - No analysis/optimizations across processes
  - Huge designs can be synthesized thru HLS, as long as each process is not too large
- Each combinational process (SC_METHOD) becomes combinational logic in RTL
- Each sequential process becomes exactly one FSM + Datapath in RTL

How HLS sees your SC model (continued)

- Primary optimizations done by HLS are in construction of the FSM and Datapath (e.g. resource sharing).
- Unlike RTL synthesis, HLS is (usually) allowed to add clock cycles (latency) into design to improve QOR
  - e.g. often loops are pipelined to maintain thruput while latency is added by HLS
- Properly constructed models and proper usage of HLS should show no functional differences (aside from latency differences) pre and post HLS.
- For SC HLS, reset behaviors and IO protocols are present in pre-HLS model and synthesized into RTL together with rest of model

Catapult supports two SystemC Coding Styles

- "RTL in SystemC" coding style
- "Matchlib SystemC" coding style
- Each has advantages/disadvantages
- The two coding styles can be mixed in same design, even in same block or process

"RTL in SystemC": example 01: SC Combinational Process

```
3  #pragma once
4
5  #include "stdlib.h"
6  #include "mc_trace.h"
7
8  #pragma hls_design top
9  class and_gate : public sc_module {
10 public:
11   sc_in<bool>  INIT_S1(in1);
12   sc_in<bool>  INIT_S1(in2);
13   sc_out<bool> INIT_S1(out1);
14
15   SC_CTOR(and_gate)
16   {
17     SC_METHOD(run);
18     sensitive << in1 << in2;
19   }
20
21   void run() {
22      out1 = in1.read() & in2.read();
23   }
24 };
```

HLS Input

```
10 //
11 // ------------------------------------------------------------------
12 //  Design Unit:    and_gate
13 // ------------------------------------------------------------------
14
15
16 module and_gate (
17   in1, in2, out1
18 );
19   input in1;
20   input in2;
21   output out1;
22
23
24
25   // Interconnect Declarations for Component Instantiations
26   assign out1 = in1 & in2;
27 endmodule
28
29
```

HLS Output

“RTL in SystemC”: example 02: SC Sequential Process

```
10 #pragma hls_design top
11 class flop : public sc_module {
12 public:
13   sc_in<bool>      INIT_S1(clk);
14   sc_in<bool>      INIT_S1(rst_bar);
15   sc_in<uint32_t>  INIT_S1(in1);
16   sc_out<uint32_t> INIT_S1(out1);
17
18   SC_CTOR(flop)
19   {
20     SC_THREAD(process);
21     sensitive << clk.pos();
22     async_reset_signal_is(rst_bar, false);
23   }
24
25   void process() {
26     // this is the reset state:
27     out1 = 0;
28     wait();
29
30     // this is the non-reset state:
31     while (1) {
32        out1 = in1.read();
33        wait();
34     }
35   }
36 };
```

HLS Input

```
16 module flop_process (
17   clk, rst_bar, in1, out1
18 );
19   input clk;
20   input rst_bar;
21   input [31:0] in1;
22   output [31:0] out1;
23   reg [31:0] out1;
24
25
26
27   // Interconnect Declarations for Component Instantiations
28   always @(posedge clk or negedge rst_bar) begin
29     if ( ~ rst_bar ) begin
30       out1 <= 32'b00000000000000000000000000000000;
31     end
32     else begin
33       out1 <= in1;
34     end
35   end
36 endmodule
```

HLS Output

Mentor A Siemens Business

"RTL in SystemC": example 04: SC signal level protocols

```
 9 #pragma hls_design top
10 class dut : public sc_module {
11 public:
12   sc_in<bool> INIT_S1(clk);
13   sc_in<bool> INIT_S1(rst_bar);
14
15   sc_out<bool> INIT_S1(in1_rdy);
16   sc_in<bool>  INIT_S1(in1_vld);
17   sc_in<sc_uint<32>>  INIT_S1(in1_data);
18
19   sc_in<bool>  INIT_S1(out1_rdy);
20   sc_out<bool> INIT_S1(out1_vld);
21   sc_out<sc_uint<32>> INIT_S1(out1_data);
22
23   SC_CTOR(dut)
24   {
25     SC_THREAD(main);
26     sensitive << clk.pos();
27     async_reset_signal_is(rst_bar, false);
28   }
29
```

Question: What's the throughput?

```
32   sc_uint<32> in1_Pop()
33   {
34     sc_uint<32> i1;
35
36     in1_rdy = 1;
37
38     do { wait(); } while (!in1_vld);
39
40     i1 = in1_data;
41     in1_rdy = 0;
42
43     return i1;
44   }
45
46   void out1_Push(sc_uint<32> o)
47   {
48     out1_vld = 1;
49     out1_data = o;
50
51     do { wait(); } while (!out1_rdy);
52
53     out1_vld = 0;
54   }
55
56   void main() {
57     in1_rdy = 0;
58     out1_vld = 0;
59     out1_data = 0;
60
61     wait();
62
63     while(1) {
64       sc_uint<32> i1;
65       i1 = in1_Pop();
66       out1_Push(i1 + 0x100);
67     }
68   }
```


Mentor A Siemens Business

"RTL in SystemC" Rules

- Code up to first wait() in process models the reset state.
  - Every sc_out should be assigned reset value here.
- Clock and reset use sc_in<bool>
- All data IO uses sc_in<>, sc_out<>
- Every loop has a wait() statement (unless fully unrolled)
  - Catapult will add an implicit wait() statement if needed.
- Some of the added value over real RTL:
  - Catapult loop pipelining still works
  - Any AC + SC datatypes can be used (ac_fixed, ac_float, etc).
  - Any AC functions can be used (e.g. AC Math)
- Module hierarchy and IO in RTL will be the same as in SystemC

"RTL in SystemC" Rules (continued)

- By default,
  - Signal reads occur at preceding wait statement in SC source
  - Signal writes take effect at next wait statement in SC source
  - Catapult is free to insert scheduler states between SC wait statements
- Use IO_MODE=fixed if you don't want Catapult to add any states
- Use #pragma hls_direct_input if you don't want Catapult to insert pipeline registers for signal reads
  - E.g. useful for CSRs which are held stable after block reset (better QOR)
- Keep in mind that SC TB and DUT are modeling HW, not SW
  - e.g. shared variable between 2 processes in TB or DUT may cause races
  - Use sc_signal, Matchlib Connections, sc_fifo, tlm_fifo, etc., to avoid races.

"RTL in SystemC" Advantages/Disadvantages

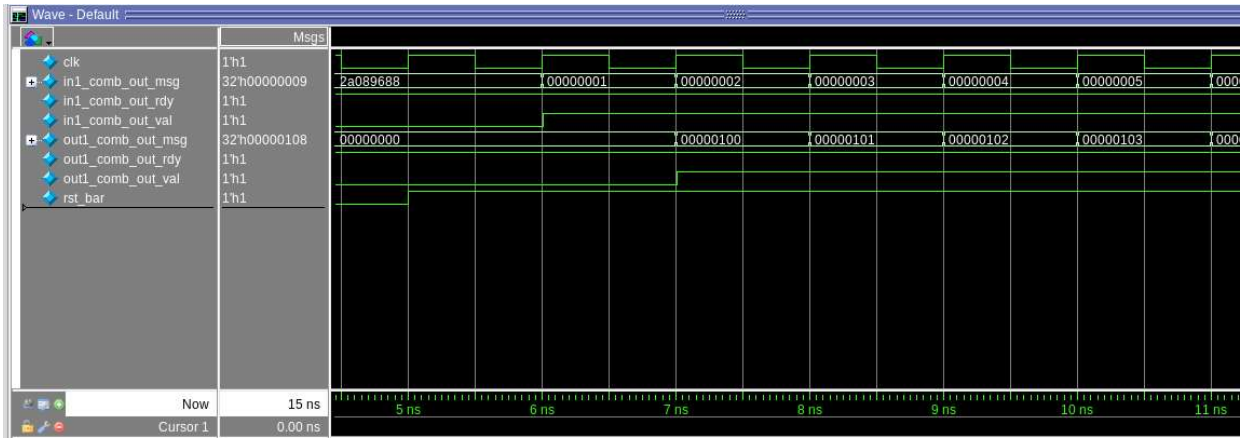
- Advantages:
  - Useful if RTL pin level protocols are "fixed"
  - Useful for integrating with existing Verilog/VHDL RTL designs
  - Useful for data_valid protocol (i.e. no backpressure) (see example 15*)
    - common in time-domain signal processing
  - May be a better fit for "RTL-centric" customers
- Disadvantages:
  - Low level of abstraction
  - Larger models
  - No fast TLM simulation mode
  - Less automation for SV UVM verification
  - Less automation for debug (e.g. no rand stall and transaction logging)

“Matchlib Coding Style”: example 05: Pop/Push

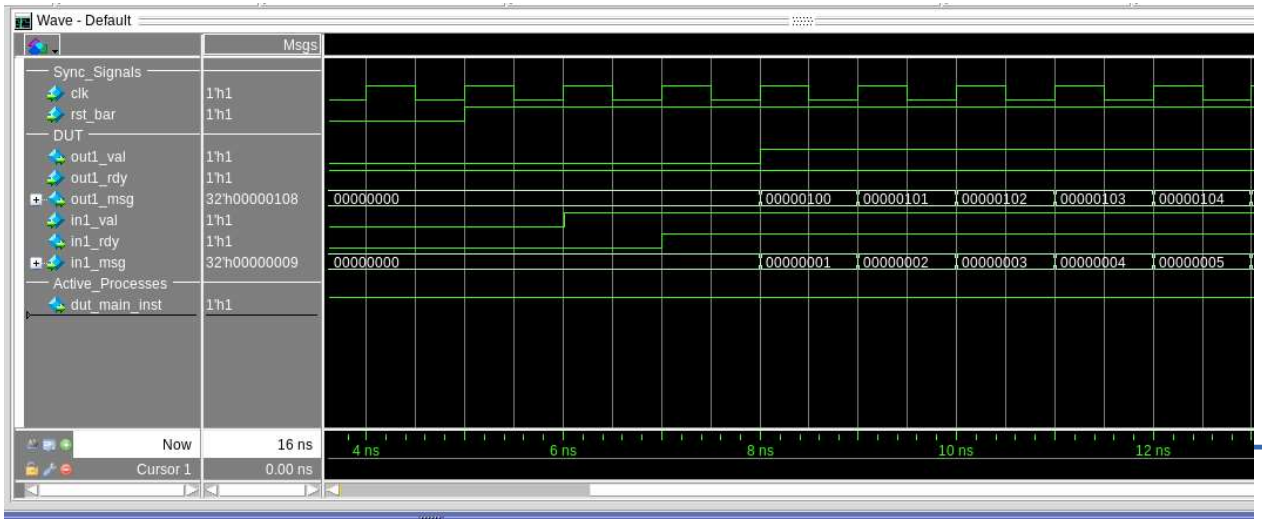
```
 2 #include <mc_connections.h>
 3
 4 #pragma hls_design top
 5 class dut : public sc_module {
 6 public:
 7   sc_in<bool> INIT_S1(clk);
 8   sc_in<bool> INIT_S1(rst_bar);
 9
10   Connections::Out<uint32> INIT_S1(out1);
11   Connections::In <uint32> INIT_S1(in1);
12
13   SC_CTOR(dut)
14   {
15     SC_THREAD(main);
16     sensitive << clk.pos();
17     async_reset_signal_is(rst_bar, false);
18   }
19
20 private:
21
22   void main() {
23     out1.Reset();
24     in1.Reset();
25
26     wait();
27
28     #pragma hls_pipeline_init_interval 1
29     #pragma pipeline_stall_mode flush
30     while(1) {
31       uint32_t t = in1.Pop();
32       out1.Push(t + 0x100);
33     }
34   }
35 };
```

05 Waveforms

Question:
What's the throughput?



Pre-HLS



Post-HLS

Mentor
A Siemens Business

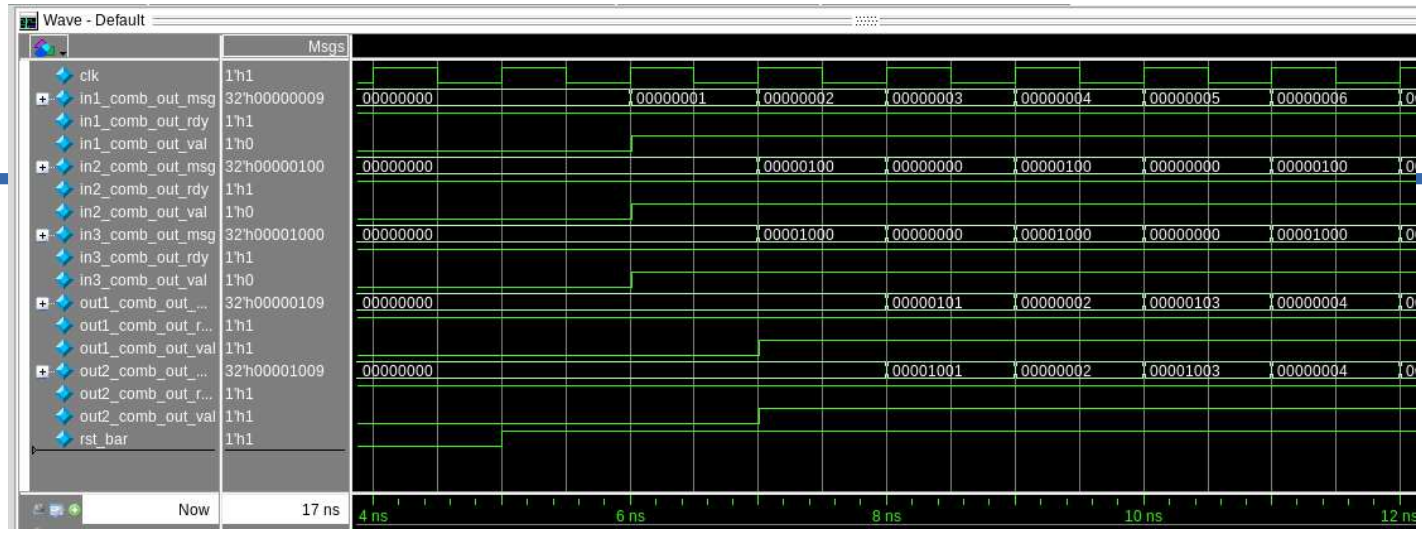
“Matchlib Coding Style”: example 06: Throughput Accurate

```
 2 #include <mc_connections.h>
 3
 4 #pragma hls_design top
 5 class dut : public sc_module {
 6 public:
 7   sc_in<bool> INIT_S1(clk);
 8   sc_in<bool> INIT_S1(rst_bar);
 9
10   Connections::In <sc_uint<32>> INIT_S1(in1);
11   Connections::In <sc_uint<32>> INIT_S1(in2);
12   Connections::In <sc_uint<32>> INIT_S1(in3);
13   Connections::Out<sc_uint<32>> INIT_S1(out1);
14   Connections::Out<sc_uint<32>> INIT_S1(out2);
15
16   SC_CTOR(dut)
17   {
18     SC_THREAD(main);
19     sensitive << clk.pos();
20     async_reset_signal_is(rst_bar, false);
21   }
22
23 private:
24
25   void main() {
26     in1.Reset();
27     in2.Reset();
28     in3.Reset();
29     out1.Reset();
30     out2.Reset();
31
32     wait();
33
34     #pragma hls_pipeline_init_interval 1
35     #pragma pipeline_stall_mode flush
36     while(1) {
37       uint32_t i1 = in1.Pop();
38       uint32_t i2 = in2.Pop();
39       uint32_t i3 = in3.Pop();
40       out1.Push(i1 + i2);
41       out2.Push(i1 + i3);
42     }
43   }
44 };
```


Mentor® A Siemens Business

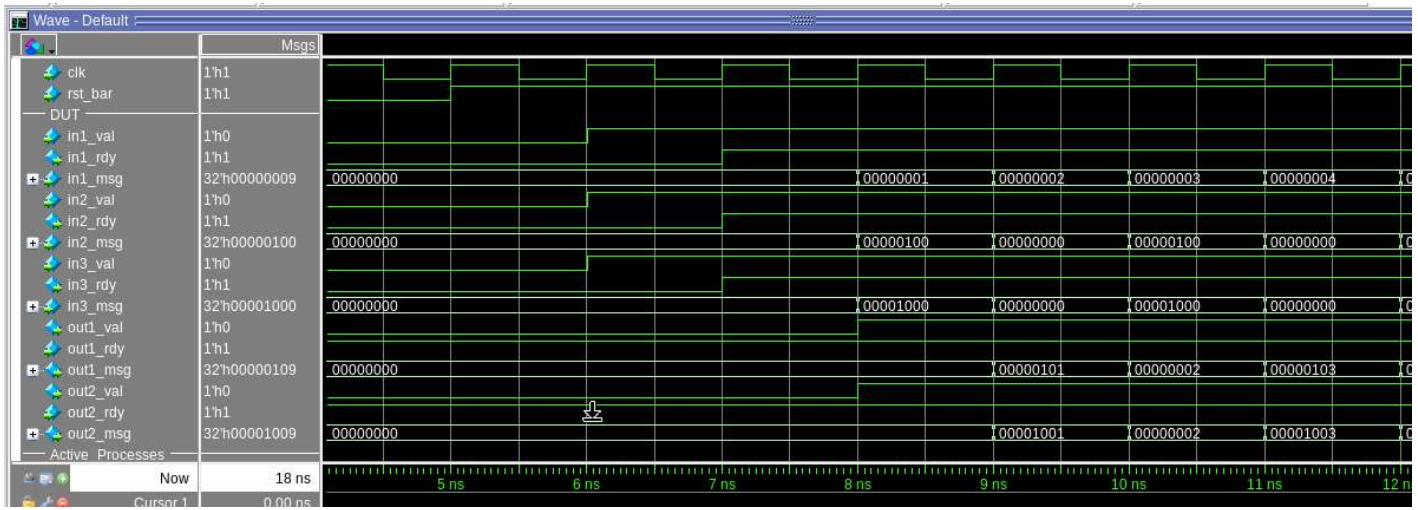
06 Waveforms

Pre-HLS



Question:
What's the throughput?

Post-HLS



"Matchlib SystemC" Coding Style Rules

- Code up to first wait() in process models the reset state.
  - Reset() methods on all ports should be called (both In and Out ports).
- Clock and reset use sc_in<bool>
- All data IO uses In<>, Out<>, or other Matchlib transaction ports
- Only use of wait() is for:
  - Reset state
  - Loops with all non-blocking Push/Pop and no blocking Push/Pop
- Use default Catapult scheduling modes only
- Only use coding style documented in Mentor Matchlib examples

"Matchlib SystemC" Coding Style Rules (continued)

- Only use sc_in, sc_out, sc_signal when:
  1. You need to mix "RTL in SystemC" coding style with Matchlib blocks/processes (try to keep these cases isolated).
  2. You want to model CSRs which are held stable after block reset
     - Enables efficient way to distribute CSR data to multiple blocks without synchronization overhead.
     - Usually also want to use #pragma hls_direct_input for these CSR signal inputs

Matchlib: 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)
- 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.

Matchlib: Methods for user-defined transactions (07*)

```
struct engine_t
{
  static const int plugs = 4;
  sc_uint<16> engine;
  spark_plug_t spark_plugs[plugs];

  static const unsigned int width = 16 + (spark_plug_t::width * plugs);
  template <unsigned int Size> void Marshall(Marshaller<Size> &m) {
    m &engine;
    for (int i=0; i<plugs; i++)
      m &spark_plugs[i];
  }
  inline friend void sc_trace(sc_trace_file *tf, const engine_t& v, const std::string& NAME ) {
      sc_trace(tf,v.engine,  NAME + ".engine");
      for (int i=0; i<plugs; i++)
        sc_trace(tf,v.spark_plugs[i],  NAME + ".spark_plug" + std::to_string(i));
  }

  inline friend std::ostream& operator<<(ostream& os, const engine_t& rhs)
  {
    os << rhs.engine << " ";
    for (int i=0; i<plugs; i++)
      os << rhs.spark_plugs[i] << " ";
    return os;
  }
};
```

User's struct/transaction

Declare HW bitwidth

Pack/Unpack to bits

SystemC standard tracing (see LRM)

Stream to text, used for transaction logging

Matchlib "gotchas"

```
4  class Top : public sc_module {
5  public:
6    CCS_DESIGN(dut) INIT_S1(dut1);
7
8    sc_clock clk;
9    SC_SIG(bool, rst_bar);
10
11   Connections::Combinational<uint32>        INIT_S1(out1);
12   Connections::Combinational<uint32>        INIT_S1(in1);
13
14   SC_CTOR(Top)
15       :   clk("clk", 1, SC_NS, 0.5,0,SC_NS,true)
16   {
17     Connections::set_sim_clk(&clk);
18     sc_object_tracer<sc_clock> trace_clk(clk);
19
20     dut1.clk(clk);
21     dut1.rst_bar(rst_bar);
22     dut1.out1(out1);
23     dut1.in1(in1);
24
25     SC_CTHREAD(reset, clk);
26
27     SC_THREAD(stim);
28     sensitive << clk.posedge_event();
29     async_reset_signal_is(rst_bar, false);
30
31     SC_THREAD(resp);
32     sensitive << clk.posedge_event();
33     async_reset_signal_is(rst_bar, false);
34   }
35
36
37   void stim() {
38     LOG("Stimulus started");
39     in1.ResetWrite();
40     wait();
41
42     for (int i = 0; i < 10; i++)
43       in1.Push(i);
44
45     sc_stop();
```

Multiple clocks and multiple resets are not currently supported.

Keep your code clean by using convenience macros for naming ports, signals, and module instances (uses -std=c++11)

Always call set_sim_clk exactly once to identify your clock to Matchlib (may get bad pre-HLS sim results if you forget currently)

All processes in both DUT and TB must use exact same clock and reset signals. TB processes must be sensitive to reset.

Always use wait(), wait(int), never wait(sc_time) in both TB and DUT. Matchlib TB and DUTs are "cycle models".

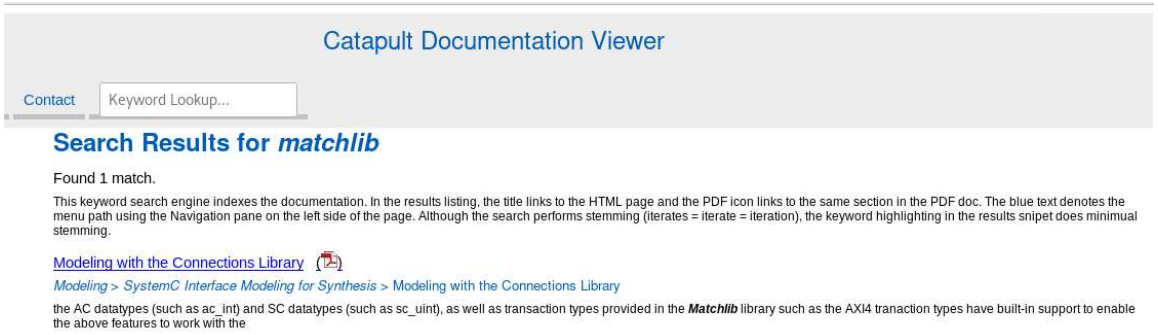
All transaction IO methods must be called exactly on clock edges. (may get bad pre-HLS sim results if you forget currently)

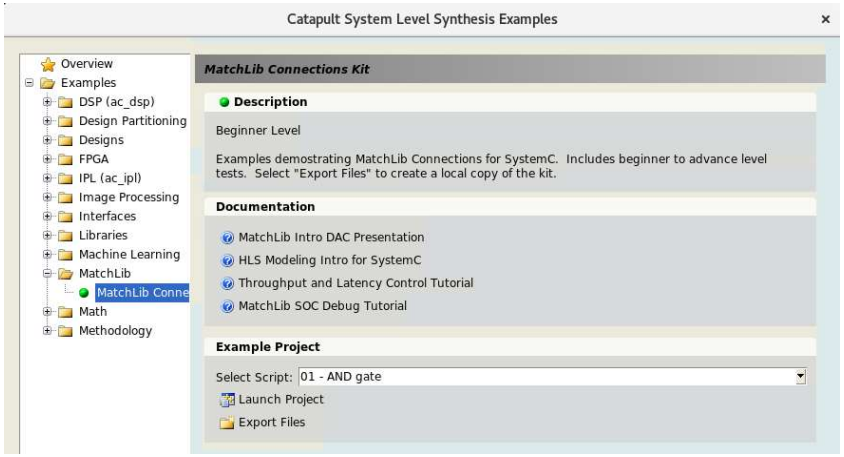
"Matchlib Coding Style" Advantages/Disadvantages

- Advantages:
  - "Throughput accurate" modeling in pre-HLS simulations for complex models
  - Higher level of abstraction than "RTL in SystemC"
  - Models are smaller
  - Fast TLM simulation mode available
  - Automation available for SV UVM verification flow
  - Automation available for debug (rand stall injection, transaction logging)
  - Growing Matchlib IP models available (AXI4, NOC, reorder buffers, etc)
  - Verification flow for Matchlib enables pre-HLS model to be "stress tested"
- Disadvantages:
  - Learning curve may be longer than "RTL in SystemC" flow
  - C++ compiler error messages can sometimes be difficult to decipher (use an IDE)
  - Signal level protocols not as flexible as "RTL in SystemC", however you can mix in "RTL in SystemC" where needed

Catapult SystemC / Matchlib Collateral

- General SystemC HLS and Matchlib Tutorials available here:
  - $MGC_HOME/shared/examples/matchlib/toolkit
  - Also see below:





How do I model “X” in Matchlib?

- How do I model a Fifo in Matchlib?
  — 10*, 11*
- How do I model a sync (aka "barrier")
  — 12*
- How do I model a dual port RAM shared between two processes?
  — 12*

Does Mentor support everything in Matchlib?

- No. Mentor supports all constructs documented in Catapult documentation.
- Mentor supports all constructs used in examples included in Catapult.
- Everything else in Matchlib is currently "open source" code that is not directly supported by Mentor
- Over time, Mentor is likely to directly support a larger subset of Matchlib

Timing Accuracy of Matchlib Models vs Catapult RTL

- Currently:
  - Intent is to be nearly throughput accurate for pipelined II=1 processes
    - A bit of special handling needed for II != 1 (insert extra wait statements)
  - Intent is NOT to be cycle accurate
  - Latency and buffer storage capacity backannotation can be used to increase timing accuracy of simulation of pre-HLS models

Matchlib QOR

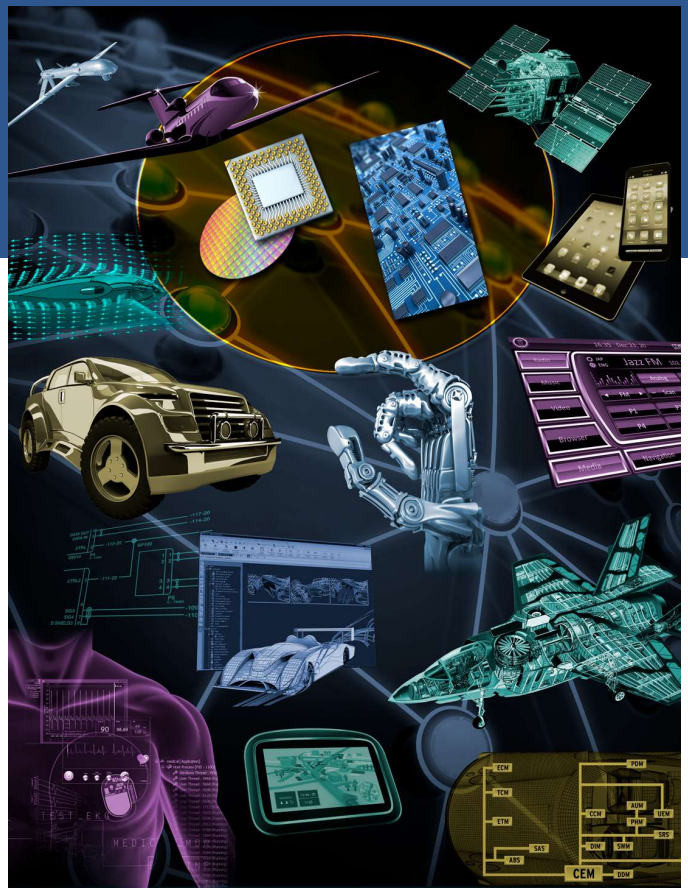
- As of Catapult 10.6, expectation is that QOR of Matchlib designs is in general equivalent to Catapult C++ flow.

General 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



AXI4 Introduction and Matchlib AXI4 Models

Stuart Swan

HLS IP/Platform Architect

July 2020

Mentor®
A Siemens Business

Why should I learn about AXI4?

- Very common on-chip bus interface
- Representative of modern bus protocols, even representative "network on chip" bus protocols
- Enables Catapult users to build bus based HW accelerators
- Enables Catapult users to build bus fabrics in HLS
- The Matchlib AXI4 models are high quality and in use in by Catapult users
- AXI4 models and examples show how to mix control and dataflow while keeping good QOR

AXI4 is a Message Passing Protocol

- Message Passing is very common, goes by many names:
  - Kahn Process Networks (KPN)
  - "Latency Insensitive"
- Matchlib Connections is also a message passing framework
  - So is ac_channel
- Message passing is good because:
  - Deterministic
  - Scalable
  - Latency insensitive
  - Can add pipeline registers (e.g. for long paths across chips)
  - Catapult is really good at pipelining message passing models

AXI4 Protocol

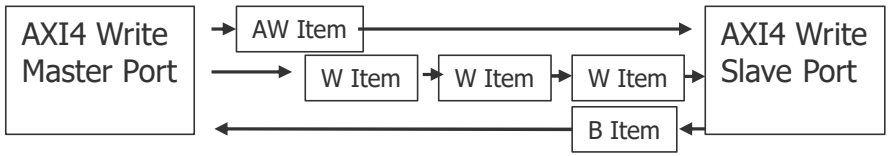
- AXI4 is a "point-to-point" protocol between a Master Port and a Slave Port
  - Master initiates reads or writes
  - Slave responds
- Routers, Arbiters, multi-layer bus fabrics etc can be built using AXI4 Master/Slave Ports
  - Matchlib contains a number of these components
- The read interfaces are completely separate from the write interfaces
  - Possible (and common) to have blocks that only have read interface but not write interface (or vice-versa)
- All addresses in AXI4 are **byte** addresses

AXI4 Read Channels

- AR is "Address Read" Payload
  - Specifies address, burst length, beat width, etc.
- R is "Read" Payload
  - Returns read data from Slave Port
  - Each read item is called a "beat"
- Protocol is:
  - Master sends AR item that specifies # of beats
  - Slave always returns specified # of beats
    - Even if there is an error! (e.g. if address is invalid)
  - Read beats are returned in order

AXI4 Write Channels

- AW is "Address Write" Payload
  - Specifies address, burst length, beat width etc.
- W is "Write" Payload
  - Contains write data to send to Slave Port
  - Includes write strobes (aka "byte enables")
  - Each write item is called a "beat"
- B is "Write response" Payload (tells if the write succeeded)
- Protocol is:
  - Master sends AW item that specifies # of write beats
  - Master sends exactly # of W items specified, in order
    - Even if there is an error! (e.g. if address is invalid)
  - Slave port returns exactly one B item per AW item.



The Plot Thickens…

- AXI4 allows a maximum of 256 beats per AW or AR burst
- AXI4 forbids a AR or AW burst from spanning a 4K address boundary
- The last beat of every W and R item in a burst must have its "last bit" set.
- These restrictions keep the size of counters and adders within bus fabric components (e.g. routers, arbiters) and slaves small and fast
- Also, the 4k boundary restriction simplifies routers, assuming all slaves are located on 4k boundaries within address map.
- In practice, these restrictions improve area and do not reduce thruput
- However, if these AXI4 aspects are visible in user's HLS models, the HLS models become complex and error-prone.
  - So, the Matchlib AXI4 components are designed to (mostly) hide these details from user's HLS models.
  - Big value-add over RTL models that use AXI4.

Matchlib Automatic Burst Segmentation

- The Matchlib Automatic Burst Segmenters provide:
  - Automatic burst segmentation so that AXI bursts be specified as having up to 2^32 beats
  - Automatic segmentation at 4k address boundaries
  - Automatic setting of the "last bit" in W items
- There are separate AXI4_W_SEGMENT() and AXI4_R_SEGMENT() modules
- These are instantiated in blocks with AXI4 master interfaces
  - The effect is that the AXI4 protocol viewed within this block is extended as noted above
- Usage of these segmenters is invisible to slaves and bus fabric components
  - This is because in all simulation modes, the AXI4 protocol is strictly followed on all module interfaces

Encoding of AXI4 Beat Lengths

- AXI4 specifies beat lengths in AR and AW payloads
  - Uses an 8 bit unsigned integer
  - But, we allow lengths from 1 -> 256
  - So, the AXI4 encoding is:
    - "0" means "1" beat
    - "1" means "2" beats
    - "0xff" means "256" beats
  - This encoding is visible in user models (to hide it would cost QOR).
- When Matchlib AXI4 Burst segmenter is used, encoding remains the same:
  - "0" means "1" beat
  - "0x1000" means "0x1001" beats
  - Reasons: consistency, and QOR

Don’t forget…

- “0” encoding always means “1” beat
- Every Push(aw) must correspond to exactly one b.Pop()
- If you are using burst segmentation, don’t “reach around” it and try to access the native AXI4 master ports
  - Exception: r items are read from native AXI4 master read port even when read segmenter is used.
- Number of beats is same regardless of success or failure:
  - masters must supply number of write beats they specify in aw, slaves must consume them
  - masters must read number of read beats they specified in ar, slaves must produce them

$MGC_HOME/shared/examples/matchlib/toolkit/include/ram.h

```
#include "stdlib.h"
#include "axi4_segment.h"


typedef axi::axi4_segment<axi::cfg::standard> local_axi;

/**
 *  \brief A simple RAM module with 1 axi4 read slave and 1 axi4 write slave
*/

class ram : public sc_module, public local_axi {
public:
  sc_in<bool> INIT_S1(clk);
  sc_in<bool> INIT_S1(rst_bar);
  r_slave<AUTO_PORT>     INIT_S1(r_slave0);
  w_slave<AUTO_PORT>     INIT_S1(w_slave0);

  static const int sz = 0x10000; // size in axi_cfg::dataWidth words

  typedef NVUINTW(axi_cfg::dataWidth) arr_t;
  arr_t* array {0};

  SC_CTOR(ram)
  {
    array = new arr_t[sz];

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

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

    for (int i=0; i < sz; i++)
     array[i] = i * bytesPerBeat;
  }
```

Default AXI4 configuration is 64 bit data width

Ram has 1 read slave port and 1 write slave

Ram stores 64k words (64 bits each)

Read and Write slaves are fully independent and each get their own thread

$MGC_HOME/shared/examples/matchlib/toolkit/include/ram.h(cont)

```
54  void slave_r_process() {
55    r_slave0.reset();
56
57
58    wait();
59
60    while(1) {
61      ar_payload ar;
62      r_slave0.start_multi_read(ar);
63
64      LOG("ram read  addr: " << std::hex << ar.addr << " len: " << ar.len);
65
66      while (1) {
67        r_payload r;
68
69        if (ar.addr >= (sz * bytesPerBeat))
70        {
71          SC_REPORT_ERROR("ram", "invalid addr");
72          r.resp = Enc::XRESP::SLVERR;
73        }
74        else
75        {
76          r.data = array[ar.addr / bytesPerBeat];
77        }
78
79        if (!r_slave0.next_multi_read(ar, r))
80          break;
81      }
82    }
83  }
```

Need to call reset() methods for signals this process drives

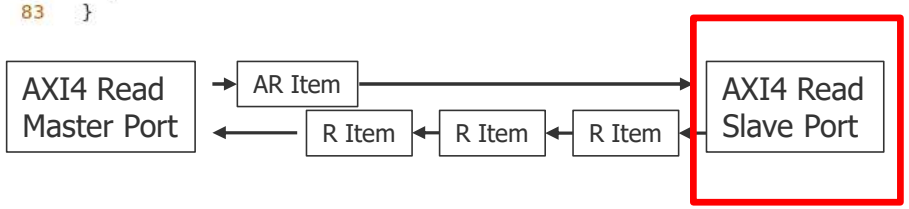
start_multi_read() is a convenience method implemented in the AXI4 read slave port. Start waiting for any length burst read ("multi"). Result is returned in "ar"

If address is invalid we still need to return an "r" item, but mark it with an error code

Map byte address to word address and read data

Send read item back to master, continue if more data to read, else break out of loop if done Automatically sets "last bit" on r item when needed

ar.addr will be updated to next read address if we continue



$MGC_HOME/shared/examples/matchlib/toolkit/include/ram.h(cont)

```
85  void slave_w_process() {
86    w_slave0.reset();
87    wait();
88
89    while(1) {
90      aw_payload aw;
91      b_payload b;
92
93      w_slave0.start_multi_write(aw, b);
94
95      LOG("ram write addr: " << std::hex << aw.addr << " len: " << aw.len);
96
97      while (1) {
98        w_payload w = w_slave0.w.Pop();
99
100       if (aw.addr >= (sz * bytesPerBeat))
101       {
102         SC_REPORT_ERROR("ram", "invalid addr");
103         b.resp = Enc::XRESP::SLVERR;
104       }
105       else
106       {
107         decltype(w.wstrb) all_on{~0};
108
109         if (w.wstrb == all_on)
110           array[aw.addr / bytesPerBeat] = w.data.to_uint64();
111         // ommitted code for write strobe handling..
112       }
113
114       if (!w_slave0.next_multi_write(aw))
115         break;
116     }
117
118     w_slave0.b.Push(b);
119   }
120 }
```

Call reset for signals this process drives

Start waiting for any length burst write
Result is returned in aw
B is initialized to "success"

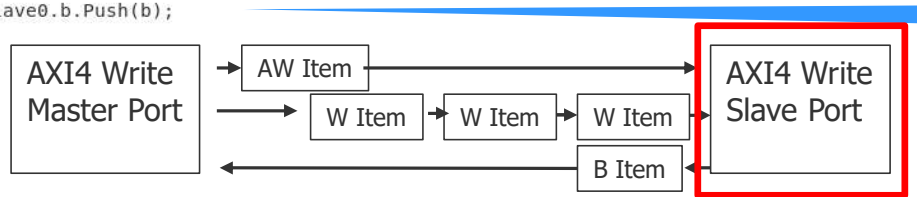
Pop a w item

On invalid address set write response to an error

Write the data into the ram array

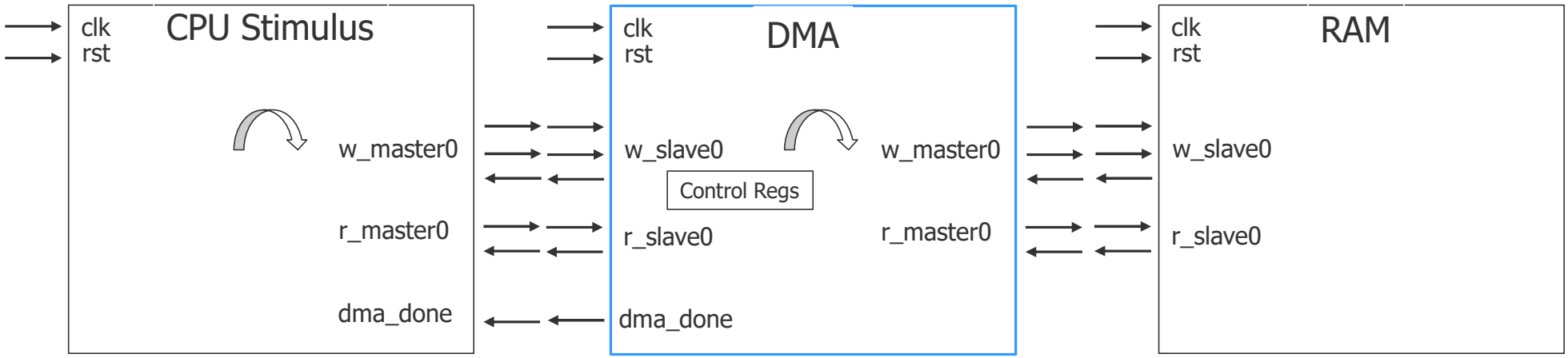
If more, update aw.addr and continue, else finish

Push write response



Mentor A Siemens Business

Simple Example: AXI4 DMA using MatchLib




```
/**
 *  * \brief dma module
*/
#pragma hls_design top
class dma : 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_slave  INIT_S1(r_slave0);
  w_slave  INIT_S1(w_slave0);
  Connections::Out<bool> INIT_S1(dma_done);
```

= top level of design

Mentor A Siemens Business

The DMA performs a memory copy using AXI4 bursts

Entire AXI4 DMA C++ is 170 lines
RTL after HLS is 6000 lines

```
85  void master_process() {
86    AXI4_W_SEGMENT_RESET(w_segment0, w_master0);
87    AXI4_R_SEGMENT_RESET(r_segment0, r_master0);
88
89    dma_cmd_chan.ResetRead();
90    dma_dbg.Reset();
91    dma_done.Reset();
92
93    wait();
94
95    while(1) {
96      ex_ar_payload ar;
97      ex_aw_payload aw;
98
99      dma_cmd cmd = dma_cmd_chan.Pop();
100     ar.ex_len = cmd.len;
101     aw.ex_len = cmd.len;
102     ar.addr = cmd.ar_addr;
103     aw.addr = cmd.aw_addr;
104     r_segment0_ex_ar_chan.Push(ar);
105     w_segment0_ex_aw_chan.Push(aw);
106
107     #pragma hls_pipeline_init_interval 1
108     #pragma pipeline_stall_mode flush
109     while (1) {
110      r_payload r = r_master0.r.Pop();
111      w_payload w;
112      w.data = r.data;
113      w_segment0_w_chan.Push(w);
114
115      if (ar.ex_len-- == 0)
116       break;
117     }
118
119     b_payload b;
120     b = w_segment0_b_chan.Pop();
121     dma_done.Push(true);
122   }
123 }
```

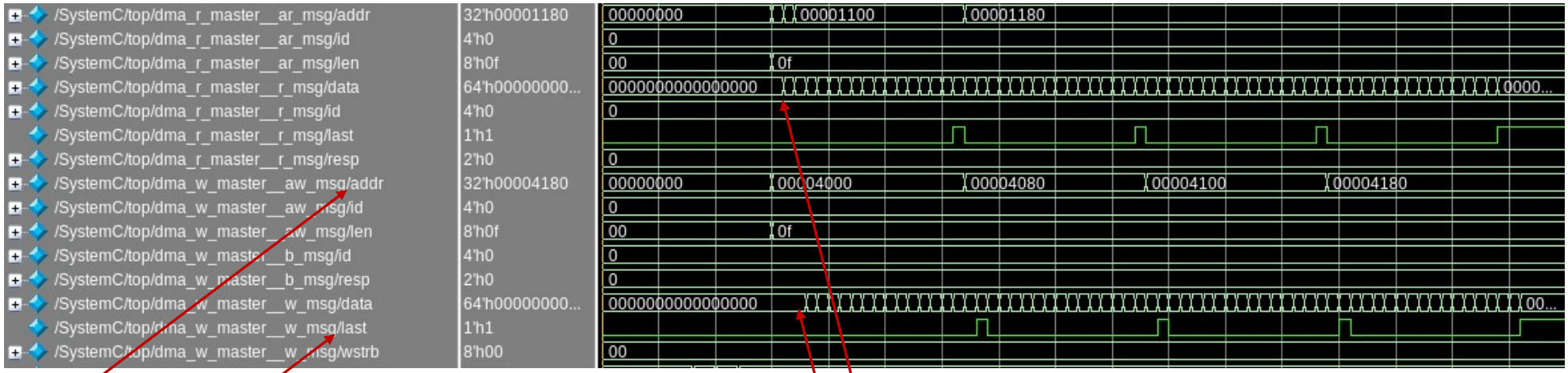
The only clock/wait is for reset state

This IO is in parallel

Main compute loop gets pipelined in HLS

This IO is in parallel

AXI4 DMA Waveforms Before HLS (SystemC simulation)



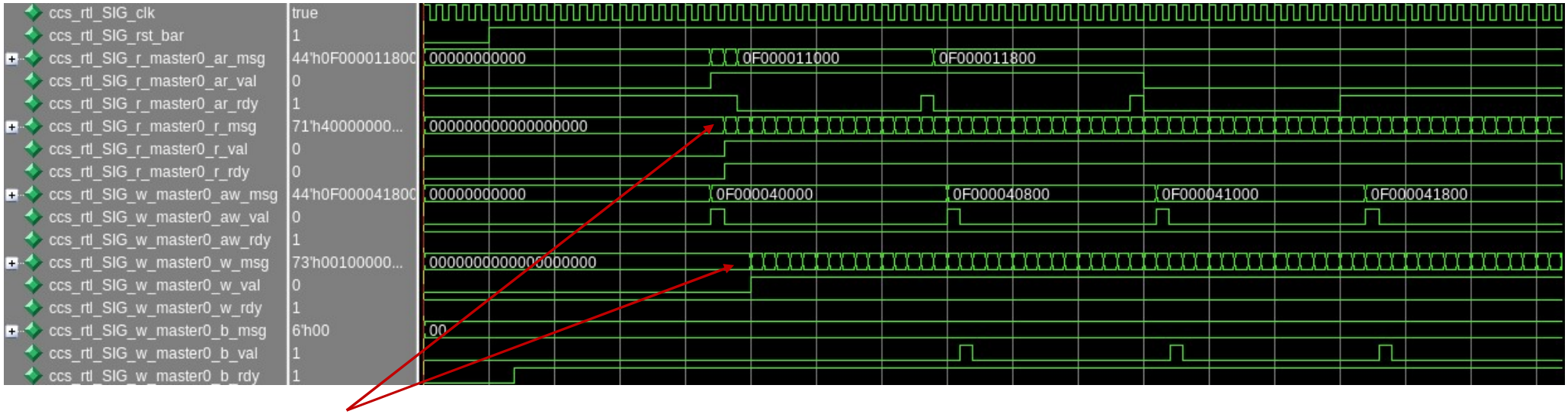
Automatic AXI4 last bit generation

Automatic AXI4 burst address segmentation

Read and write burst streams are concurrent.
R/W bus utilization is 100%
(1 read and 1 write beat per clock cycle)

Makefile sets segmentation size to 16 rather than 256 so easier to see in waveforms

AXI4 DMA Waveforms After Catapult HLS (Verilog Sim)



RTL waveforms are almost the same as SystemC waveforms:

- Throughput is same
- Bus utilization is the same
- HLS may have added pipeline stages (under user control)
- HLS may have increased latency (under user control)

$MGC_HOME/shared/examples/matchlib/toolkit/include/dma.h

```
 7 #include "axi4_segment.h"
 8
 9 /**
10  *  * \brief dma command sent to the DMA engine
11 */
12 struct dma_cmd
13 {
14   NVUINTW(32) ar_addr {0};
15   NVUINTW(32) aw_addr {0};
16   NVUINTW(32) len {0};  // For 08_dma we use AXI4 beat length encoding !
17
18   // Marshalling, tracing methods omitted..
19
20 };
21
22 /**
23  *  * \brief dma address map as seen by the CPU
24 */
25 struct dma_address_map
26 {
27   uint64_t  ar_addr;
28   uint64_t  aw_addr;
29   uint64_t  len;      // For 08_dma we use AXI4 beat length encoding !
30   uint64_t  start;
31 };
32
33
34 #pragma hls_design top
35 class dma : public sc_module, public local_axi {
36 public:
37   sc_in<bool> INIT_S1(clk);
38   sc_in<bool> INIT_S1(rst_bar);
39
40   r_master<> INIT_S1(r_master0);
41   w_master<> INIT_S1(w_master0);
42   r_slave<>  INIT_S1(r_slave0);
43   w_slave<>  INIT_S1(w_slave0);
44   Connections::Out<bool> INIT_S1(dma_done);
45   Connections::Out<sc_uint<32>> INIT_S1(dma_dbg);
46
```

DMA command sent from slave to master process

DMA address map as seen by CPU

AXI4 read and write master ports

AXI4 read and write slave ports

DMA done interrupt

$MGC_HOME/shared/examples/matchlib/toolkit/include/dma.h(cont)

```
112  // slave_process accepts incoming axi4 requests from slave0 and programs the dma registers.
113  // when the start register is written to, a dma_cmd transaction is sent to the dma master_proce
114  void slave_process() {
115    r_slave0.reset();
116    w_slave0.reset();
117    dma_cmd_chan.ResetWrite();
118
119    wait();
120
121    dma_cmd cmd1;
122
123    while(1) {
124      aw_payload aw;
125      w_payload w;
126      b_payload b;
127
128      if (w_slave0.get_single_write(aw, w, b))
129      {
130        b.resp = Enc::XRESP::SLVERR;
131        switch (aw.addr)
132        {
133        case offsetof(dma_address_map, ar_addr):
134        cmd1.ar_addr = w.data;
135        b.resp = Enc::XRESP::OKAY;
136        break;
137        case offsetof(dma_address_map, aw_addr):
138        cmd1.aw_addr = w.data;
139        b.resp = Enc::XRESP::OKAY;
140        break;
141        case offsetof(dma_address_map, len):
142        cmd1.len = w.data;
143        b.resp = Enc::XRESP::OKAY;
144        break;
145        case offsetof(dma_address_map, start):
146        dma_cmd_chan.Push(cmd1);
147        b.resp = Enc::XRESP::OKAY;
148        break;
149        }
150        w_slave0.b.Push(b);
151      }
152    }
153  }
```

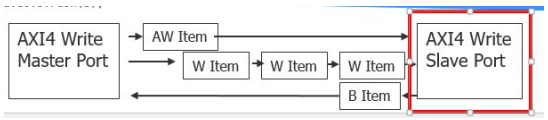
Reset signals that slave process drives

Get an AXI4 write where there must be only a single beat. If there is more than one beat, single_write() will automatically deal with the AXI4 protocol error requirements

Decode the CSR write and update status fields

When start register is written issue new DMA command

Every aw.Pop needs a b.Push



$MGC_HOME/shared/examples/matchlib/toolkit/include/dma.h(cont)

```
54 #pragma hls_design top
55 class dma : public sc_module, public local_axi {
56 public:
57   sc_in<bool> INIT_S1(clk);
58   sc_in<bool> INIT_S1(rst_bar);
59
60   r_master<> INIT_S1(r_master0);
61   w_master<> INIT_S1(w_master0);
62   r_slave<>  INIT_S1(r_slave0);
63   w_slave<>  INIT_S1(w_slave0);
64   Connections::Out<bool> INIT_S1(dma_done);
65   Connections::Out<sc_uint<32>> INIT_S1(dma_dbg);
66
67   SC_CTOR(dma)
68   {
69     SC_THREAD(slave_process);
70     sensitive << clk.pos();
71     async_reset_signal_is(rst_bar, false);
72
73     SC_THREAD(master_process);
74     sensitive << clk.pos();
75     async_reset_signal_is(rst_bar, false);
76
77     AXI4_W_SEGMENT_BIND(w_segment0, clk, rst_bar, w_master0);
78     AXI4_R_SEGMENT_BIND(r_segment0, clk, rst_bar, r_master0);
79   }
80
81 private:
82
83   Connections::Combinational<dma_cmd> INIT_S1(dma_cmd_chan);
84
85   // write and read segmenters segment long bursts to conform to AXI4 protocol (which
   ts maximum).
86   AXI4_W_SEGMENT(w_segment0);
87   AXI4_R_SEGMENT(r_segment0);
88 ▉
```

Bind W and R segmenters to native AXI4 master ports

Instantiate DMA command channel Between slave and master processes

Instantiate W and R Segmenters

$MGC_HOME/shared/examples/matchlib/toolkit/include/dma.h(cont)

```
90  // the master_process performs the dma operations via the master0 axi port,
91  // and then sends a done signal to the requester via the dma_done transaction.
92  void master_process() {
93    AXI4_W_SEGMENT_RESET(w_segment0, w_master0);
94    AXI4_R_SEGMENT_RESET(r_segment0, r_master0);
95
96    dma_cmd_chan.ResetRead();
97    dma_dbg.Reset();
98    dma_done.Reset();
99
100   wait();
101
102   while(1) {
103     ex_ar_payload ar;
104     ex_aw_payload aw;
105
106     dma_cmd cmd = dma_cmd_chan.Pop();
107     ar.ex_len = cmd.len;
108     aw.ex_len = cmd.len;
109     ar.addr = cmd.ar_addr;
110     aw.addr = cmd.aw_addr;
111     r_segment0_ex_ar_chan.Push(ar);
112     w_segment0_ex_aw_chan.Push(aw);
113
114     #pragma hls_pipeline_init_interval 1
115     #pragma pipeline_stall_mode flush
116     while (1) {
117      r_payload r = r_master0.r.Pop();
118      w_payload w;
119      w.data = r.data;
120      w_segment0_w_chan.Push(w);
121
122      if (ar.ex_len-- == 0)
123       break;
124     }
125
126     b_payload b;
127     b = w_segment0_b_chan.Pop();
128     dma_done.Push(true);
129   }
130 }
```

Reset all signals this process drives

Extended AR and AW payloads work with segmenters and allow > 256 beats

Pop a new DMA command from slave process

Copy length, and AW and AR addresses

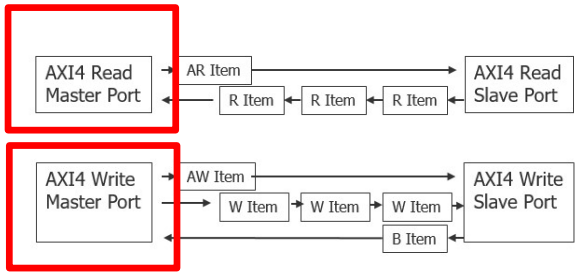
Push out AW and AR addresses to RAM

Insure full thruput of R and W channels

Copy R data to W data and push out

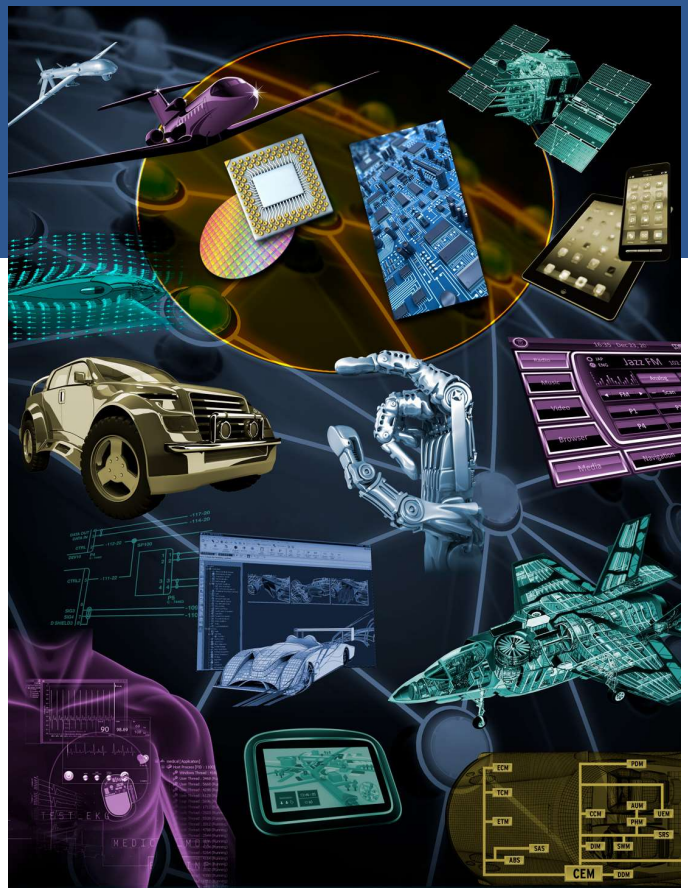
If ex_len **is currently** 0 we are done

Consume write response and send interrupt to CPU



Miscellaneous AXI4 Stuff

- Write strobes / "aka byte enables"
  - W item has one bit per byte in data payload indicating whether a particular byte is to be written.
  - Allows for precise control of which bytes are written, even if data width of beats is large
  - Default constructor for W item sets wstrb bits to all 1 so default behavior is to write all bytes
  - Works fine with write segmenter too.
- "AXI4 Narrow transfers"
  - This is when the actual data width read or written is less than the bus data width
  - Supported in native AXI4 master and slave interfaces
  - There are complex AXI4 rules about data alignment for narrow transfers!
  - Not supported in R and W segmenters



Scatter Gather DMA

Stuart Swan

HLS IP/Platform Architect

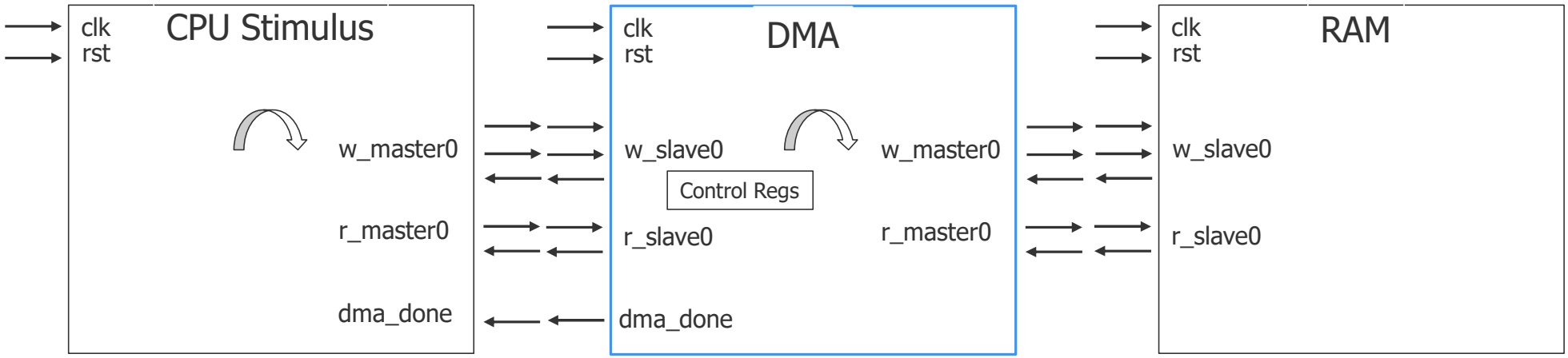
June 2020

Mentor®
A Siemens Business

Scatter Gather DMA Introduction

- DMA modules are common in SOCs to offload data movement tasks from other blocks
- The Scatter Gather DMA is a simple but representative example of common SOC blocks that use bus fabrics and which are not only datapath blocks but also include control
  - Also represents typical DV aspects for SOC blocks
- The DMA fully uses the AXI4 features explained previously such as automatic burst segmentation
- The scatter gather DMA adds features to the simple DMA block presented previously
  - Same top level interfaces as simple DMA
  - Same RAM block

Scatter Gather DMA block diagram




```
/**
 *  * \brief dma module
*/
#pragma hls_design top
class dma : 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_slave  INIT_S1(r_slave0);
  w_slave  INIT_S1(w_slave0);
  Connections::Out<bool> INIT_S1(dma_done);
```

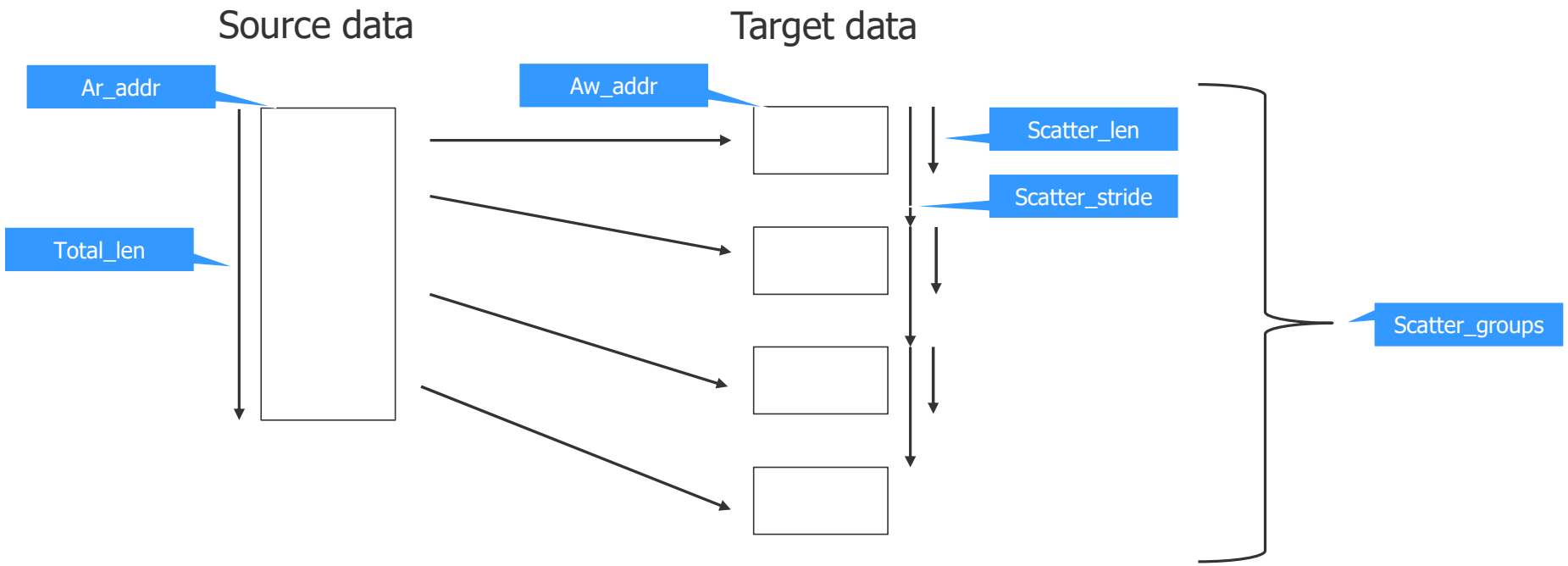
= top level of design

Scatter Gather Slave Address Map

```
 6
 7 enum dma_mode_t {COPY=0, SCATTER=1, GATHER=2};
 8
 9 /**
10  *  * \brief dma address map as seen by the CPU
11 */
12 struct dma_address_map
13 {
14   uint64_t  ar_addr;        // source address (byte address as per AXI)
15   uint64_t  aw_addr;        // target address (byte address as per AXI)
16   uint64_t  total_len;      // total length to be copied in bytes
17   uint64_t  scatter_stride; // stride between each scatter group, in bytes
18   uint64_t  scatter_len;    // length of each scatter group, in bytes
19   uint64_t  scatter_groups; // number of scatter groups
20   uint64_t  dma_mode;       // COPY, SCATTER, GATHER
21   uint64_t  start;          // DMA command is complete, cause it to be queued to start
22 };
```

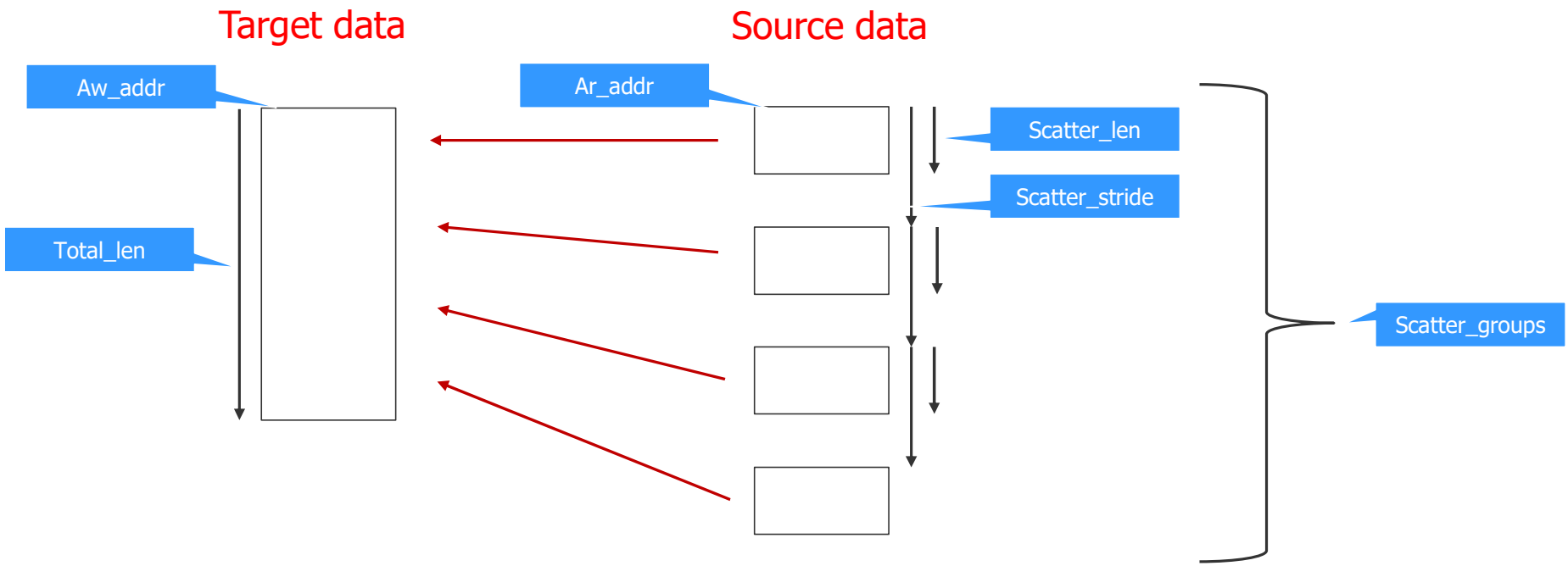
Note: In previous DMA (example 08*), total_len was in beats, but now it is a byte length

Scatter Operation



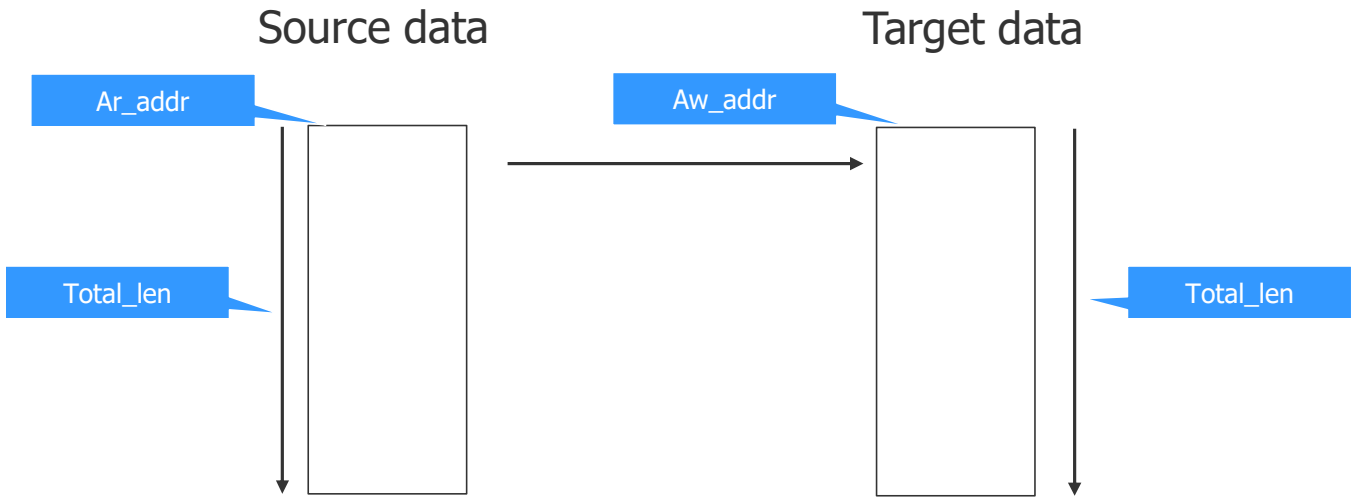
All addresses are byte addresses
All lengths are byte lengths
total_len = scatter_groups * scatter_len

Gather Operation



All addresses are byte addresses
All lengths are byte lengths
total_len = scatter_groups * scatter_len

Copy Operation



All addresses are byte addresses
All lengths are byte lengths
For copy operation, scatter_len, scatter_stride, scatter_groups are ignored

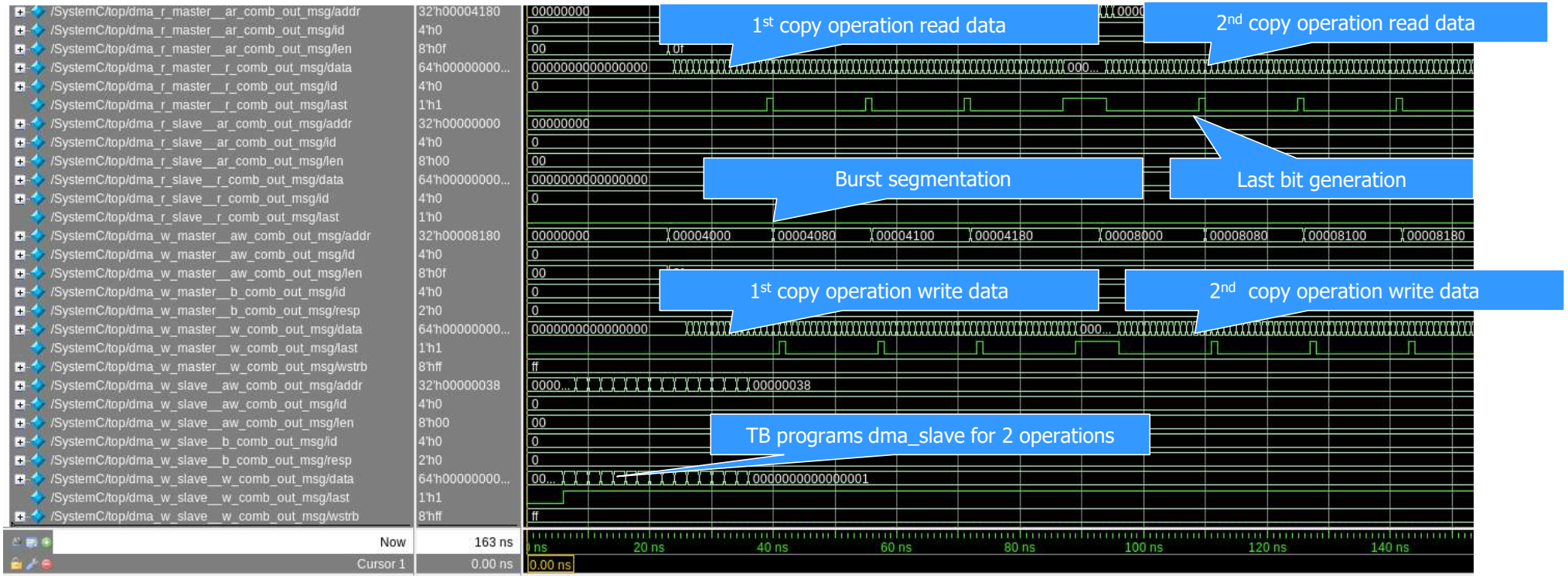
SC Testbench

```
62  int test_iterations = 1;
63  bool copy_mode = true;
64  int total_len = 64 * bytesPerBeat;
65  int scatter_groups = 4;
66  int scatter_len = total_len / scatter_groups;
67  int scatter_stride = scatter_len * 2;
68  int source_addr = 0x1000;
69  int target1_addr = 0x4000;
70  int target2_addr = 0x8000;
71  sc_time start_time, end_time;
72
```

If true, each iteration is a copy then a copy, else it is a scatter then a gather

Target address for first operation, source address for second operation

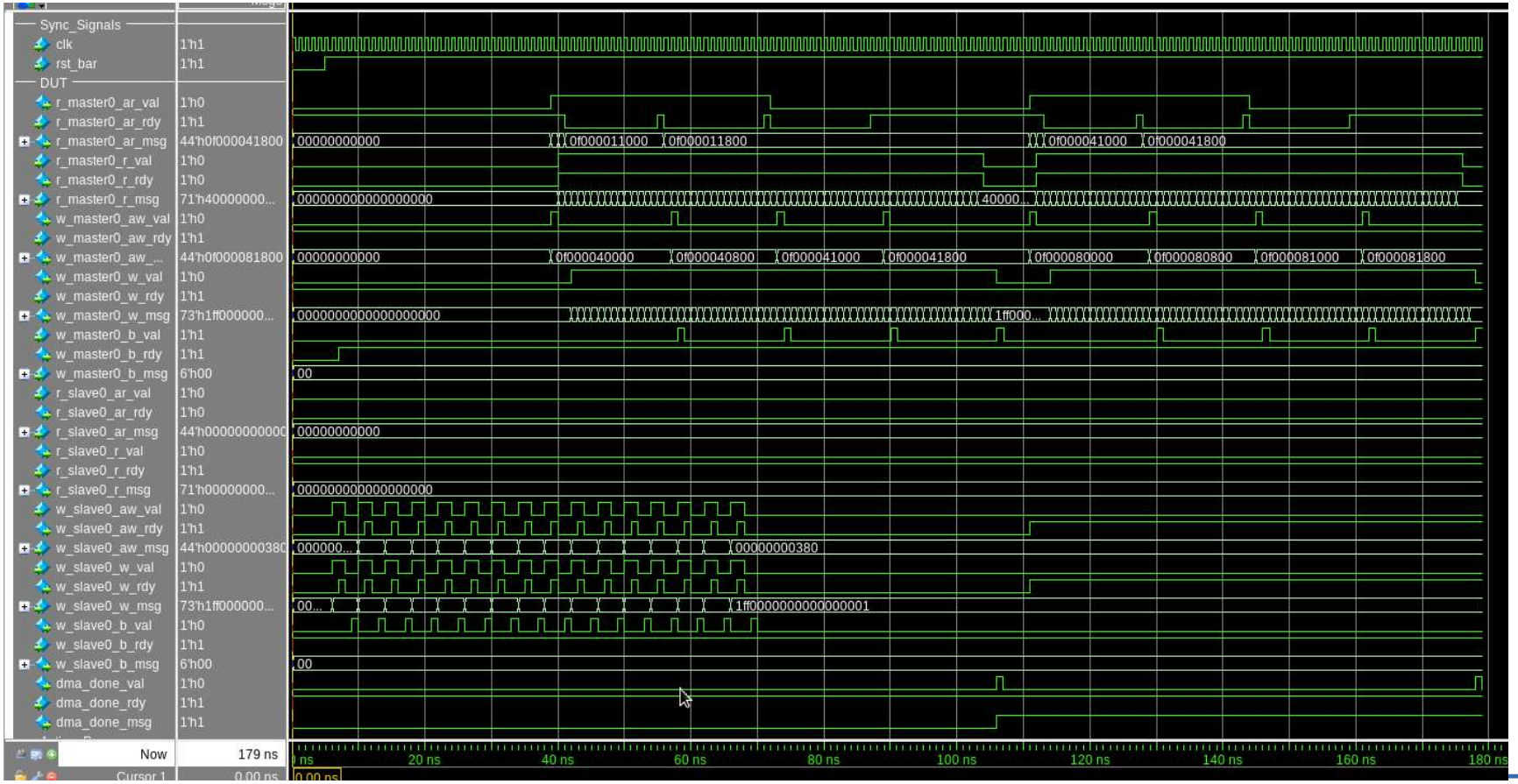
Copy_mode = true, test_iterations = 1



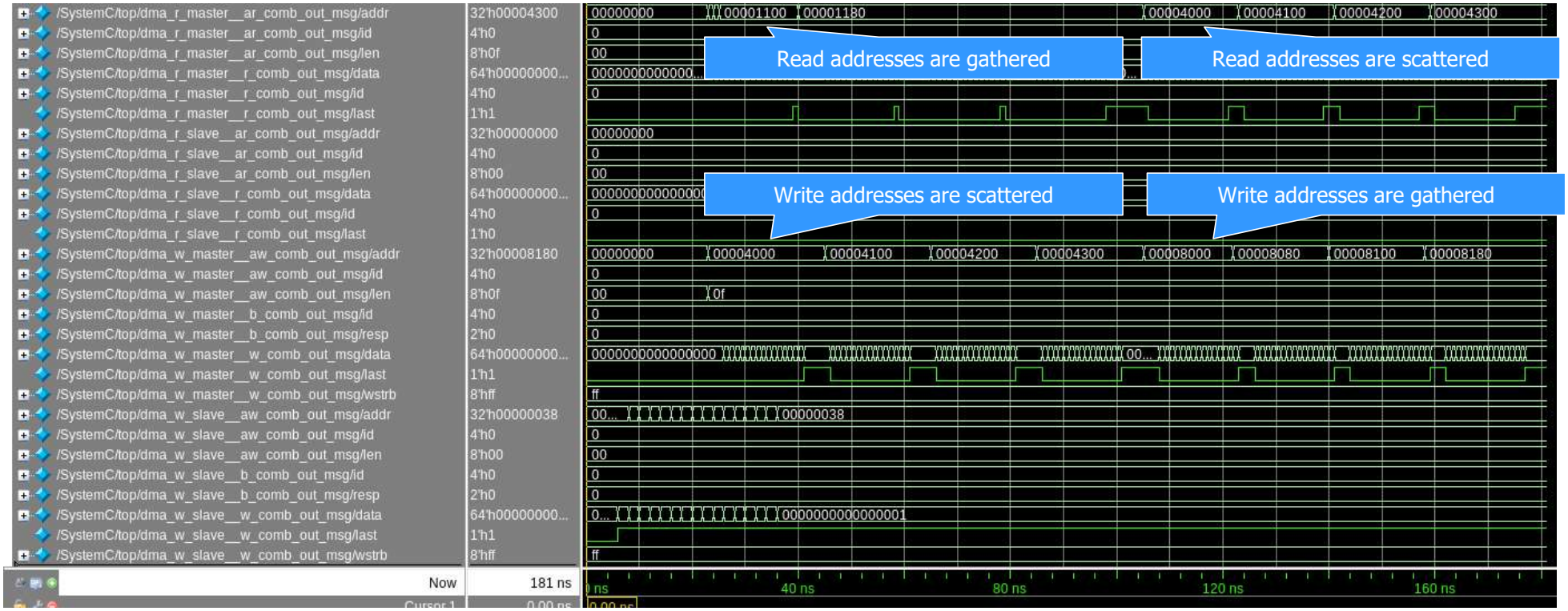
Note: SystemC Waveforms, (not RTL)

Makefile sets segmentation size to 16 rather than 256 so easier to see in waveforms

Same scenario, but in RTL

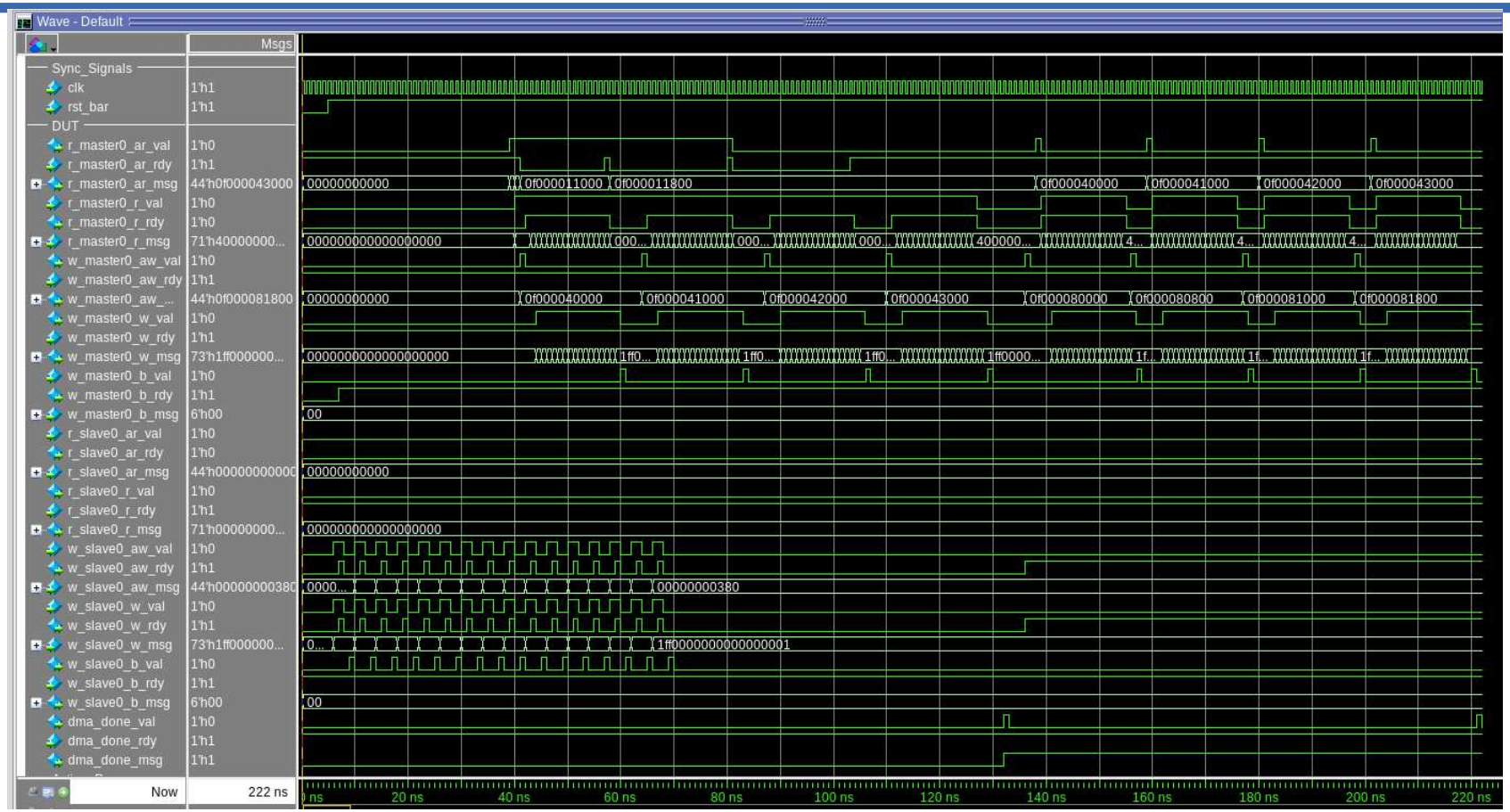


Copy_mode = false, test_iterations = 1



Note: SystemC Waveforms, (not RTL)

Same scenario, but in RTL



Mentor A Siemens Business

Keeping things simple..

- To simplify both the DUT and the TB, the DUT enforces that all addresses and lengths are aligned to bus datawidth boundaries
- When TB writes to "start" CSR, checks are enforced and a AXI4 error response if returned if there any errors.

```
295         case offsetof(dma_address_map, start):
296         if ((cmd1.ar_addr & (bytesPerBeat-1)) ||
297             (cmd1.aw_addr & (bytesPerBeat-1)) ||
298             (cmd1.total_len & (bytesPerBeat-1)))
299         {
300           LOG("discarding invalid DMA command");
301           break;
302         }
303         if (cmd1.dma_mode != dma_mode_t::COPY)
304          if((cmd1.scatter_len & (bytesPerBeat-1)) ||
305             (cmd1.scatter_stride & (bytesPerBeat-1)) ||
306             (cmd1.scatter_len * cmd1.scatter_groups != cmd1.total_len))
307         {
308           LOG("discarding invalid DMA command");
309           break;
310         }
311         dma_cmd_chan.Push(cmd1);
312         b.resp = Enc::XRESP::OKAY;
313         break;
314
315         default:
316         break;
317         }
318
319         w_slave0.b.Push(b);
```

When TB writes to "start" CSR

Make sure addresses and lengths are aligned

Do additional checks if not in COPY mode

Send AXI4 error code to TB if any checks fail

COPY implementation in DMA

```
122     while(1) {
123      dma_cmd cmd = dma_cmd_chan.Pop();
124
125      switch (cmd.dma_mode) {
126      case dma_mode_t::COPY: {
127       ex_ar_payload ar;
128       ex_aw_payload aw;
129       ar.ex_len = (cmd.total_len / bytesPerBeat) - 1;
130       aw.ex_len = (cmd.total_len / bytesPerBeat) - 1;
131       ar.addr = cmd.ar_addr;
132       aw.addr = cmd.aw_addr;
133       r_segment0_ex_ar_chan.Push(ar);
134       w_segment0_ex_aw_chan.Push(aw);
135
136       #pragma hls_pipeline_init_interval 1
137       #pragma pipeline_stall_mode flush
138       while (1) {
139        r_payload r = r_master0.r.Pop();
140        w_payload w;
141        w.data = r.data;
142        w_segment0_w_chan.Push(w);
143
144        if (ar.ex_len-- == 0)
145         break;
146       }
147
148       b_payload b;
149       b = w_segment0_b_chan.Pop();
150       dma_done.Push(true);
151       break;
152       }
```

Pop next DMA command

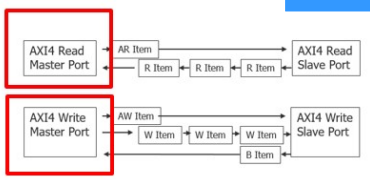
Convert byte length to beat length (AXI4 beat length encoding)

Push out AR and AW bursts

Copy data from R to W

We are done when length **is currently** 0 (AXI4 beat encoding)

Pop write response from b channel



Mentor A Siemens Business

SCATTER implementation in DMA

```
154     case dma_mode_t::SCATTER: {
155      ex_ar_payload ar;
156      ex_aw_payload aw;
157      ar.ex_len = (cmd.total_len / bytesPerBeat) - 1;
158      aw.ex_len = (cmd.scatter_len / bytesPerBeat) - 1;
159      ar.addr = cmd.ar_addr;
160      aw.addr = cmd.aw_addr;
161      r_segment0_ex_ar_chan.Push(ar);
162
163      while (1) {
164        w_segment0_ex_aw_chan.Push(aw);
165
166        #pragma hls_pipeline_init_interval 1
167        #pragma pipeline_stall_mode flush
168        while (1) {
169         r_payload r = r_master0.r.Pop();
170         w_payload w;
171         w.data = r.data;
172         w_segment0_w_chan.Push(w);
173
174         if (aw.ex_len-- == 0)
175          break;
176        }
177
178        w_segment0_b_chan.Pop();
179        aw.addr += cmd.scatter_stride;
180        aw.ex_len = (cmd.scatter_len / bytesPerBeat) - 1;
181
182        cmd.total_len -= cmd.scatter_len;
183        if (cmd.total_len == 0)
184          break;
185      }
186
187      dma_done.Push(true);
188      break;
189      }
```

Convert byte length to beat length (AXI4 beat length encoding)

aw.ex_len is computed based on scatter_len, not total_len

Start the AR burst, but not the AW yet

Start the AW burst for current region in the scatter groups

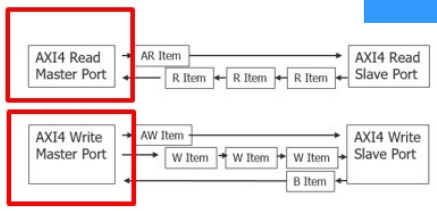
Copy data from R to W

We are done when length **is currently** 0 (AXI4 beat encoding)

Need to do b.Pop for every Push(aw)

Increment aw.addr for next scatter group region

We are done when the remaining total_len is zero



Mentor A Siemens Business

Homework #1 - GATHER implementation in DMA

- Steps:
  1. Edit testbench.cpp and change copy_mode to be false
  2. Edit scatter_gather.h and write the code for GATHER
     - This is strictly confined to the GATHER branch of the case statement.
     - If you are stuck for more than 15 minutes, ask for help.
  3. Compile and run your SC code and make sure the TB self-check passes
  4. View SC and RTL waveforms – see README file in same dir.

Homework #2 – Optimize the beat rate

- For copy_mode = false and test_iterations = 10, make small modifications to model and synthesis directives to optimize throughput (as reported in log output as "beat rate").
- Measure in both SC sim and RTL sim.
- Remember your best results and approach to report to group.

Homework #3 – Compare SC and RTL sim performance #1

- Edit Makefile and add –O3 to CFLAGS
- Edit testbench.cpp:
  1. Set copy_mode = false and test_iterations = 1000
  2. Disable SC tracing and transaction logging as shown below
  3. Disable LOG messages in ram.h by modifying code in testbench.h as below
- Measure simulation CPU time in SC and RTL simulation as reported in log output

```
178 sc_trace_file* trace_file_ptr;
179
180 int sc_main(int argc, char** argv) {
181  // trace_file_ptr = sc_create_vcd_trace_file("trace");
182
183   Top top("top");
184 //  trace_hierarchy(&top, trace_file_ptr);
185
186 /*
187   channel_logs logs;
188   logs.enable("chan_log");
189   logs.log_hierarchy(top);
190 */
191
192   sc_start();
193   return 0;
194 }
```

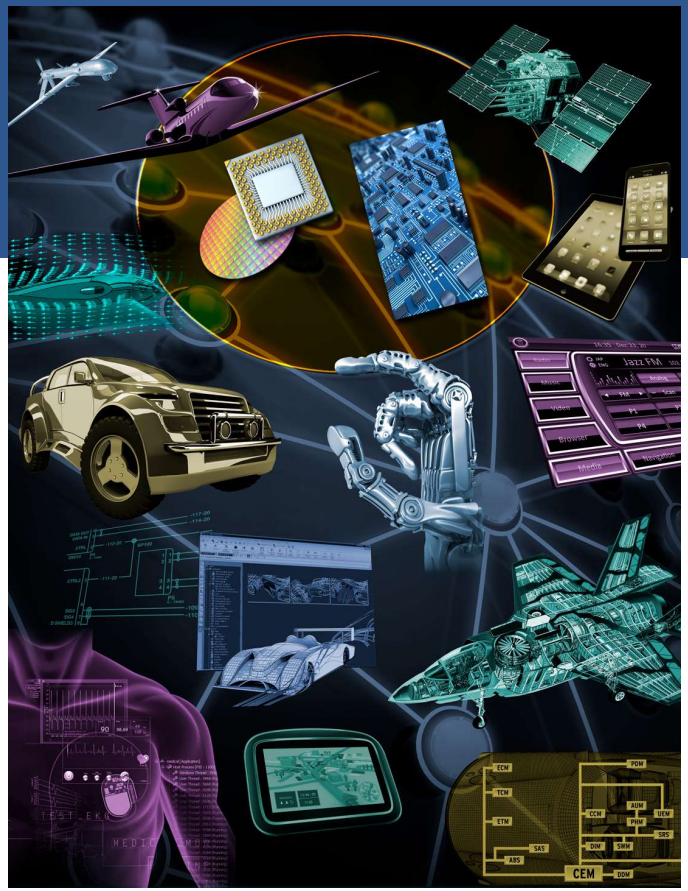
```
1 #include "scatter_gather_dma.h"
2 #undef LOG
3 #define LOG(x) // empty
4 #include "ram.h"
5 #undef LOG
6 #define LOG(x)  std::cout << sc_time_stamp() << " " << name() << " " << x << std::endl
7 #include <mc_scverify.h>
8
```

Homework #4 – Compare SC and RTL sim performance #2

- Perform same sim performance comparison as previous slide, but use Catapult Matchlib example 20_DCT_sysc
- Use the same approaches as previous slide
  - Note that DCT test runs long enough that a single iteration is adequate
- Try to explain any performance differences from previous test

Homework #5 – Matchlib SOC Verification and Debug Tutorial

- Read and follow the steps in the document:
  - $MGC_HOME/shared/examples/matchlib/toolkit/doc/matchlib_soc_debug_tutorial.pdf



Matchlib Homework Wrap-Up and Miscellaneous Topics

Stuart Swan

HLS IP/Platform Architect

July 2020

Mentor®
A Siemens Business

Homework Wrap Up and Q&A

- Discuss Homework assignments 1-5
- Q&A

Verilog Wrapper Generation

- Since Matchlib models are "protocol-accurate" and support pin level interfaces, it is easy to generate Verilog RTL wrappers for them.
- This makes it easy to embed Matchlib and "RTL in SystemC" models in SV UVM testbenches
  - Or even in "regular old Verilog" testbenches
- Wrapped SC model has **identical** Verilog pins as actual RTL
  - So same testbench can be used for SC model as well as RTL just by swapping the DUT model.

Verilog Wrapper Generation for Scatter Gather DMA

- Do the following in scatter_gather dir:

```
chmod +x ../bin/wrapper*py
../bin/wrapper_gen.py --clock_name clk --clock_period 1 sim_sc top.dma1
```

- This generates two files:
  - scatter_gather_dma_wrap.cpp : this wraps Matchlib ports to sc_in/sc_out and exports it to Verilog
  - scatter_gather_dma_wrap.v : this is the Verilog module interface declaration. It is not actually used in the mixed language simulation, it is only for documentation.

Verilog Wrapper Generation for Scatter Gather DMA

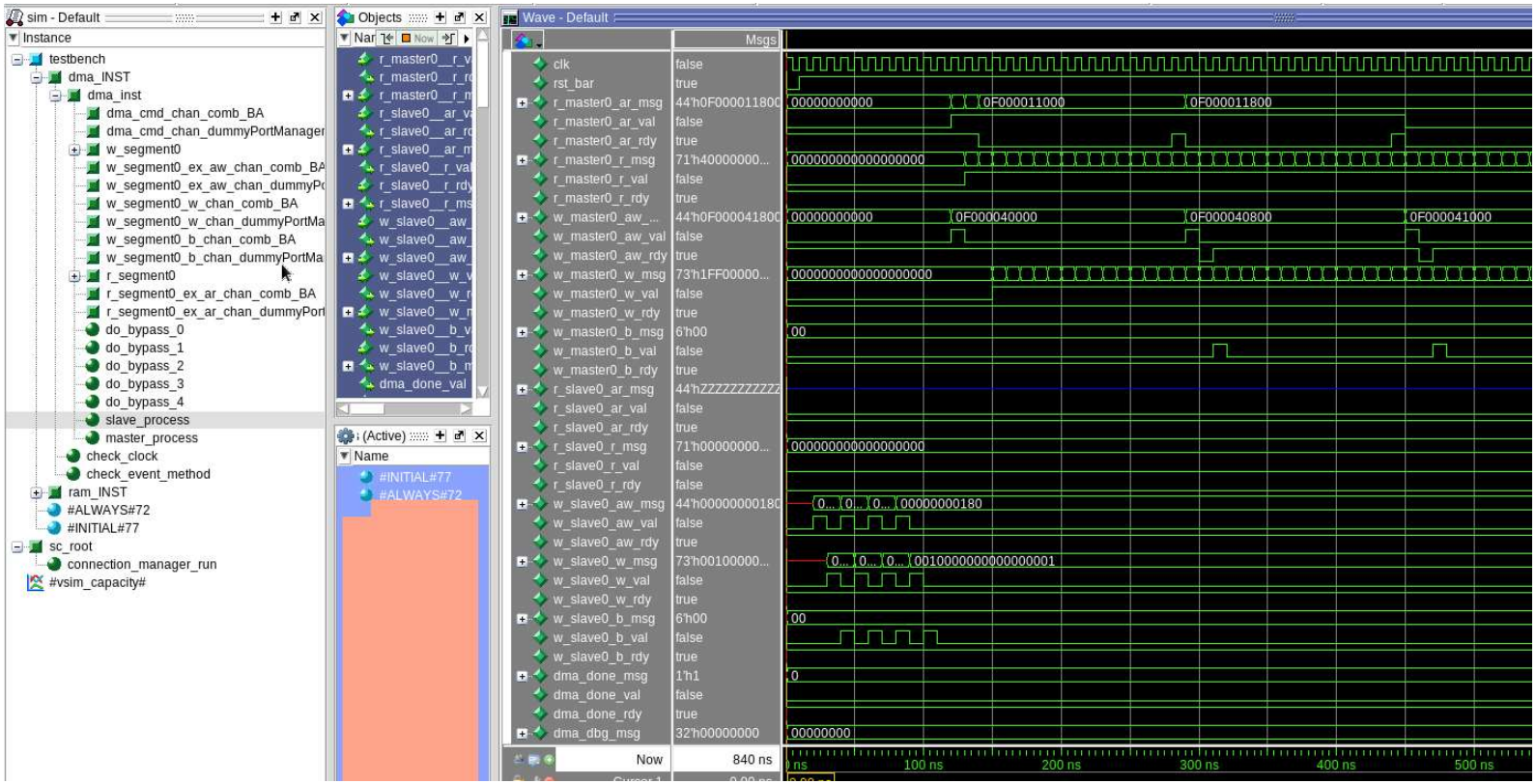
```
// Autogenerated from wrapper_gen.py on Wed, 27 May 2020 23:53:51 +0000
// Arguments: Namespace(clock_name='clk', clock_period='1', exec_name='sim_sc', not_top_level=None,
bj_name='top.dma1')

#include "scatter_gather_dma.h"

sc_trace_file* trace_file_ptr;

class scatter_gather_dma_wrap : public sc_module {
public:
  scatter_gather_dma INIT_S1(scatter_gather_dma_inst);

  sc_core::sc_in<bool>  INIT_S1(clk);
  sc_core::sc_in<bool>  INIT_S1(rst_bar);
  sc_out<sc_lv<44>> INIT_S1(r_master0_ar_msg);
  sc_out<bool>      INIT_S1(r_master0_ar_val);
  sc_in<bool>       INIT_S1(r_master0_ar_rdy);
  sc_in<sc_lv<71>> INIT_S1(r_master0_r_msg);
  sc_in<bool>       INIT_S1(r_master0_r_val);
  sc_out<bool>      INIT_S1(r_master0_r_rdy);
  sc_out<sc_lv<44>> INIT_S1(w_master0_aw_msg);
  sc_out<bool>      INIT_S1(w_master0_aw_val);
  sc_in<bool>       INIT_S1(w_master0_aw_rdy);
  sc_out<sc_lv<73>> INIT_S1(w_master0_w_msg);
  sc_out<bool>      INIT_S1(w_master0_w_val);
  sc_in<bool>       INIT_S1(w_master0_w_rdy);
  sc_in<sc_lv<6>> INIT_S1(w_master0_b_msg);
  sc_in<bool>       INIT_S1(w_master0_b_val);
  sc_out<bool>      INIT_S1(w_master0_b_rdy);
  sc_in<sc_lv<44>> INIT_S1(r_slave0_ar_msg);
  sc_in<bool>       INIT_S1(r_slave0_ar_val);
```

Verilog Wrapper Generation for Scatter Gather DMA

```
module scatter_gather_dma (
  clk
, rst_bar
, r_master0_ar_msg
, r_master0_ar_val
, r_master0_ar_rdy
, r_master0_r_msg
, r_master0_r_val
, r_master0_r_rdy
, w_master0_aw_msg
, w_master0_aw_val
, w_master0_aw_rdy
, w_master0_w_msg
, w_master0_w_val
, w_master0_w_rdy
, w_master0_b_msg
, w_master0_b_val
, w_master0_b_rdy
, r_slave0_ar_msg
, r_slave0_ar_val
, r_slave0_ar_rdy
, r_slave0_r_msg
, r_slave0_r_val
, r_slave0_r_rdy
, w_slave0_aw_msg
, w_slave0_aw_val
, w_slave0_aw_rdy
, w_slave0_w_msg
, w_slave0_w_val
, w_slave0_w_rdy
, w_slave0_b_msg
, w_slave0_b_val
, w_slave0_b_rdy
, dma_done_msg
, dma_done_val
, dma_done_rdy
);
  input clk;
  input rst_bar;
  output [43:0] r_master0_ar_msg;
  output r_master0_ar_val;
  input  r_master0_ar_rdy;
  input  [70:0] r_master0_r_msg;
  input  r_master0_r_val;
  output r_master0_r_rdy;
  output [43:0] w_master0_aw_msg;
```

Simulating SC Matchlib DMA with a Verilog Testbench

- A simple Verilog TB for similar DMA is shown in Catapult Matchlib example 45*:



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"
  - 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

Shared Memories in SystemC Designs – Approach #3

- Approach #3: Use ac_shared<>
  - This approach is currently not supported and not recommended for SC designs.

Toggle Protocol Converter – Matchlib Example 13*

- Toggle Protocol Converter is a simple (and useful) example of "RTL in SystemC" mixed with Matchlib style
- Toggle protocol indicates that a message has been sent when the toggle signal changes
  - Reliable message delivery as long as consumer process "can keep up"
  - Producer and Consumer can be running at different clock rates

```
 9 #pragma hls_design top
10 class dut : public sc_module {
11 public:
12   sc_in<bool> INIT_S1(clk);
13   sc_in<bool> INIT_S1(rst_bar);
14
15   sc_out<sc_uint<32>>  INIT_S1(out1_data);
16   sc_out<bool>         INIT_S1(out1_toggle);
17
18   Connections::In <sc_uint<32>> INIT_S1(in1);
19
20   SC_CTOR(dut)
21   {
22     SC_THREAD(main);
23     sensitive << clk.pos();
24     async_reset_signal_is(rst_bar, false);
25   }
26
```

Data output

Toggle output

Input uses Matchlib rdy/vld protocol

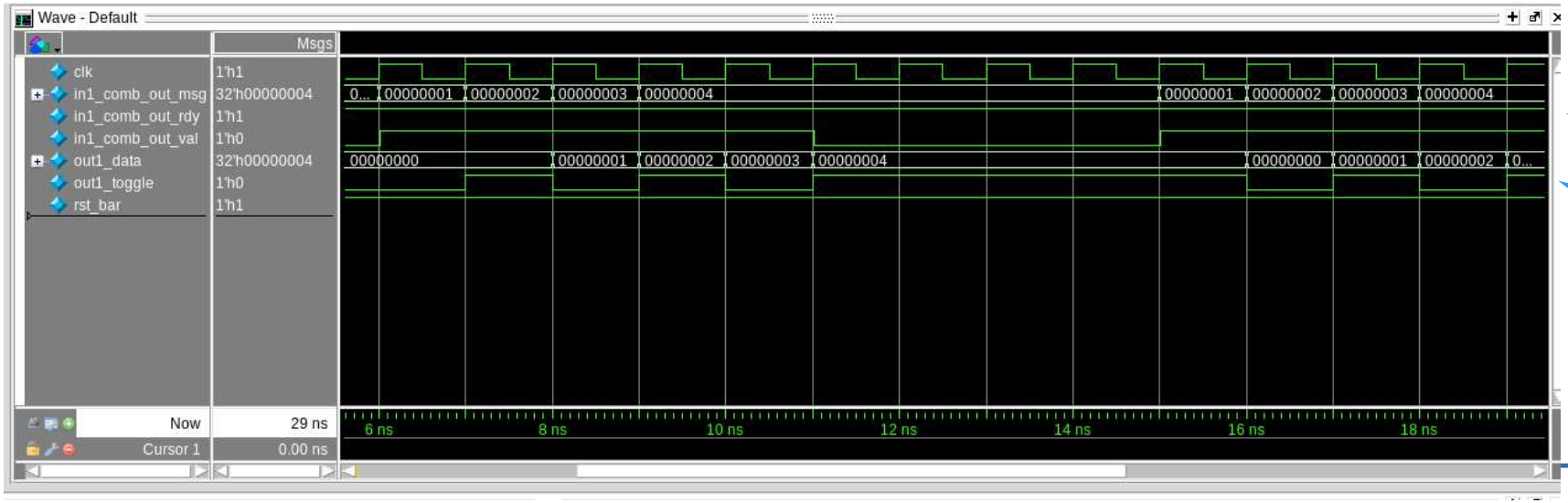
```
29   void main() {
30     out1_data = 0;
31     out1_toggle = false;
32     in1.Reset();
33     bool toggle = false;
34
35     wait();
36
37       #pragma hls_pipeline_init_interval 1
38       #pragma pipeline_stall_mode flush
39     while(1) {
40       uint32_t t = in1.Pop();
41       out1_data = t;
42       toggle = !toggle;
43       out1_toggle = toggle;
44     }
45   }
```

Toggle output when we output new data

Toggle Protocol Converter – Matchlib Example 13*

```
49  void stim() {
50    LOG("Stimulus started");
51    in1.ResetWrite();
52    wait();
53
54    for (int i = 0; i < 5; i++)
55      in1.Push(i);
56
57    wait(5);
58
59    for (int i = 0; i < 5; i++)
60      in1.Push(i);
61
62    wait(10);
63    sc_stop();
64  }
65
```

Stimulus uses Matchlib rdy/vld protocol



Input is rdy/vld protocol

Output is toggle protocol

Data Valid Protocol – Matchlib Example 15*

- Data Valid protocol is similar to rdy/vld protocol, except rdy signals are removed because either:
  1. consumers are assumed to "always be ready" to receive messages
  2. if consumers are not ready and messages are dropped, it is OK.
     - Note: In this case, if latency of design changes from SC model to RTL, then sim results may change due to this timing sensitivity
- Assumption is that if vld=true for two consecutive clock cycles, then two messages were sent.
- Data valid protocol is modeled using sc_in/sc_out (ie. "RTL in SystemC" style)
- This protocol is common in time domain signal processing, where each block is clocked at a particular sampling frequency and consumes one sample per clock. The valid bit indicates if the particular sample is valid.
  - When a chain of filter blocks come out of reset, the first sample at the beginning of the chain will be valid, and input samples further down the chain will not be valid until the first valid sample propagates thru the chain.
  - rdy/vld protocol is not suitable for such designs.

Real World Design Example – 5G Receiver