Catapult SystemC and Matchlib Training

Stuart Swan
Platform Architect
Siemens EDA
10 December 2024

SIEMENS

Agenda

1. Simple Introductory Matchlib Example
2. Introduction to SystemC Coding for HLS
3. What is a “Throughput Accurate” Model?
4. Introduction to Matchlib
5. Catapult SystemC Coding Styles
6. Automatic Generation of Transaction Field Methods
7. Matchlib AXI4 Interfaces
8. Matchlib Accuracy
9. Using Questa, VCS, and Xcelium with Matchlib
10. Custom Protocols in Catapult SystemC & Matchlib
11. Matchlib Memory Modeling Methodology
12. Leveraging C++ SW Development Best Practices
13. Matchlib Verification Methodology
14. Introduction to Matchlib Channel Logs
15. Introduction to Matchlib Memory Logs
16. Catapult HLS Scheduling Rules

Simple Introductory Matchlib Example

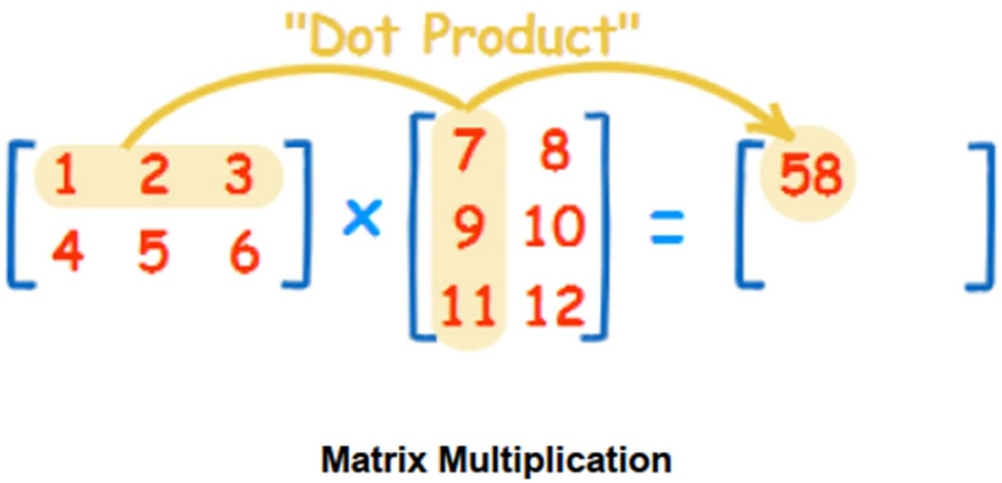
Refer to Matchlib example 38_dot_product

Stuart Swan
Platform Architect
Siemens EDA
7 Oct 2024

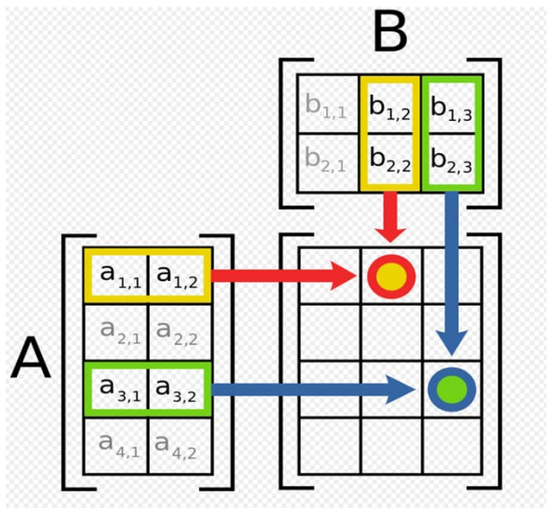
SIEMENS

Matrix Multiplication Example – 38_dot_product

- Simple example shows matrix multiplication design modeled in SystemC using Matchlib and synthesized to RTL using Catapult HLS.
- The goals of this design are to demonstrate how Matchlib pre-HLS models are throughput accurate, and to show how memory architecture can be tailored to achieve desired design goals.
- During matrix multiplication of the A * B matrices, the rows of matrix A are multiplied with the columns of matrix B and then added. This is repeated across all rows and columns.



Matrix Multiplication



Matrix Multiplication Example

- In this design example, the input and output matrices are streamed into and out of the design in row-by-row order.
- Within the design, the multiplication operation needs to occur on a row of the A matrix and a column of the B matrix.
- To do this, we form a transpose matrix of B and then perform the matrix multiplication on the A matrix and the transposed B matrix.
- In this example we will explore three different architectures for forming the transpose matrix and performing the matrix multiplication.
- Note that the testbench for all the designs in this example uses a 2 ns clock.

Design #1 – Single Process

- In the first design for the matrix multiplication, we use a single process that performs the transpose operation and the multiply accumulate operations.
- If you view the file mat_mul_single_process.h, you will see:

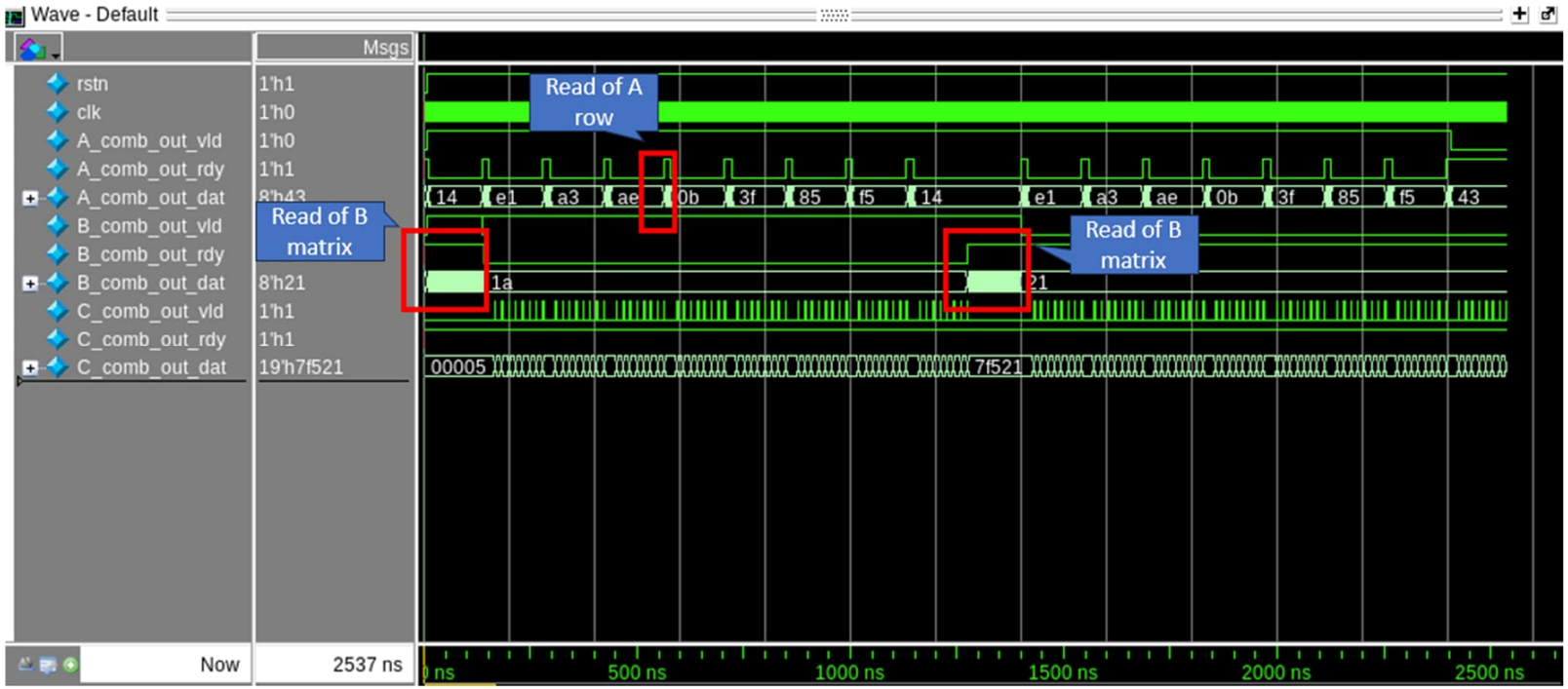
```
 7 #pragma hls_design top
 8 class matrixMultiply  : public sc_module {
 9  public:
10   sc_in<bool> CCS_INIT_S1(clk);
11   sc_in<bool> CCS_INIT_S1(rstn);
12
13   Connections::In <ac_int<8>> CCS_INIT_S1(A);
14   Connections::In <ac_int<8>> CCS_INIT_S1(B);
15   Connections::Out<ac_int<8+8+3>> CCS_INIT_S1(C);
16
17   SC_CTOR(matrixMultiply) {
18     SC_THREAD(run);
19     sensitive << clk.pos();
20     async_reset_signal_is(rstn, false);
21   }
22
```

Design #1 – Single Process (continued)

- On line 36 we read in the B matrix and form the B_transpose matrix. Note the reversal of the I and J indices for B_transpose.
- On line 41 we store an entire row of the A matrix into A_row. Once that is done, we can perform the matrix multiplication on line 48, and push out the sum on line 53.
- Note that we place a “wait()” statement on line 50 that causes each loop iteration to consume a clock cycle in the pre-HLS simulation. If this were not done, the pre-HLS simulation would complete all the iterations of the loop in zero time.

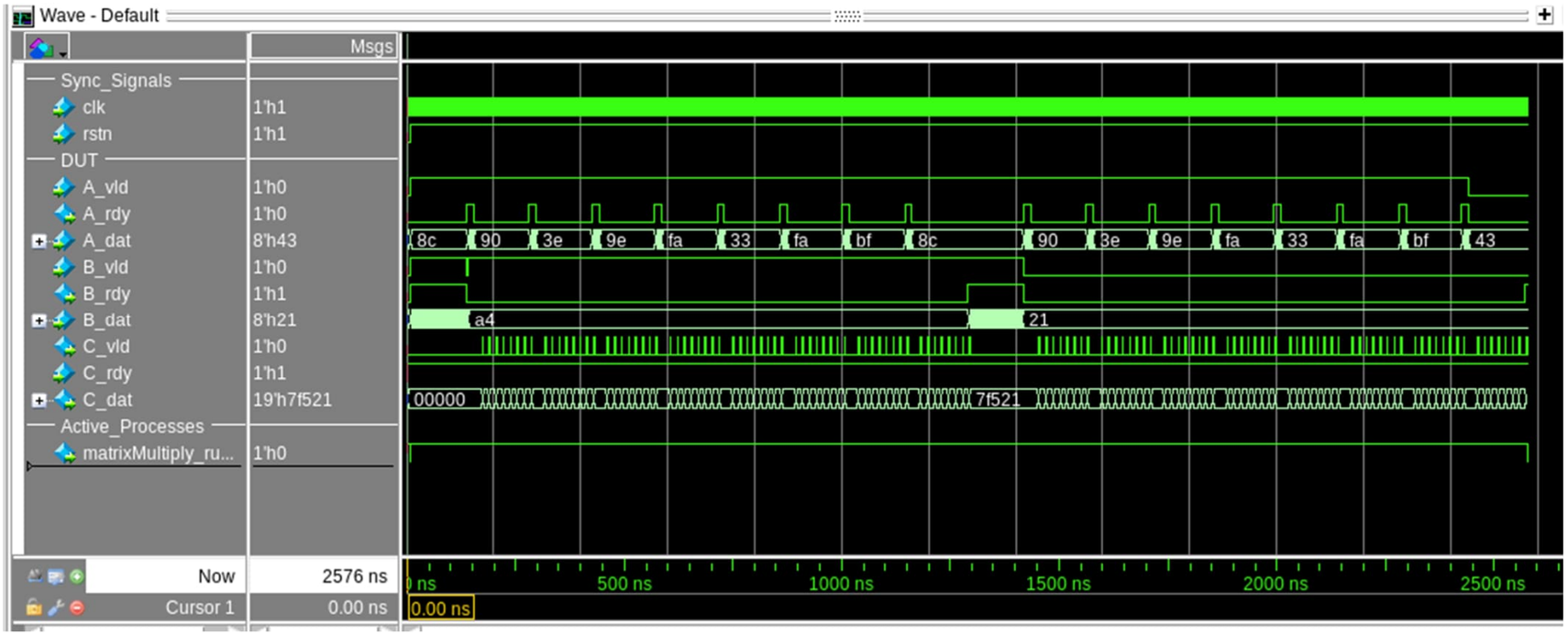
```
23  void run() {
24    A.Reset();
25    B.Reset();
26    C.Reset();
27    ac_int<8> A_row[8];
28    ac_int<8> B_transpose[8][8];
29    wait();
30    #pragma hls_pipeline_init_interval 1
31    #pragma pipeline_stall_mode flush
32    while (1) {
33      ac_int<8+8+3> acc = 0;
34      TRANSPOSEB_ROW:for (int i=0; i<8; i++) { // Transpose operation must complete first
35        TRANSPOSEB_COL:for (int j=0; j<8; j++) {
36          B_transpose[j][i] = B.Pop();
37        }
38      }
39      ROW:for (int i = 0; i < 8; i++) {
40        CPY_A:for (int c=0; c<8; c++){ //Copy one row from A
41          A_row[c] = A.Pop();
42        }
43        COL:for (int j = 0; j < 8; j++) {//Multiply row of A against all cols of B
44          acc = 0;
45          // Cannot unroll MAC loop since it would result in memory port contention..
46          #pragma hls_unroll no
47          MAC:for (int k = 0; k < 8; k++) {
48            acc += A_row[k] * B_transpose[j][k];
49            #ifndef __SYNTHESIS__
50            wait();//wait used to simulate not unrolling the loop in hardware
51            #endif
52          }
53          C.Push(acc);
54        }
55      }
56    }
57  }
58 };
59
```

Design #1 – Single Process (continued)



- Running the pre-HLS simulation gives the waveforms above.
- This design has the limitation that there is no overlapping of reading of input data with the matrix multiplication operation.
- It also has the limitation that the matrix multiplication loop cannot be unrolled during HLS because it would result in a requirement for too many ports to access the B_transpose matrix.
- Note that this design takes about 2500 ns to execute.

Design #1 – Single Process (continued)



- Running the post-HLS RTL simulation gives the waveforms above.
- *Note the very close correspondence and timing of the post-HLS and pre-HLS waveforms.*
- This correspondence exists because Matchlib simulations are “throughput accurate”.

Design #2 – Multiple Processes

- In the second design for the matrix multiplication, we use a multiple process design that performs the transpose operation, packing of the A row data, and the multiply accumulate operations in three separate processes.
- If you view the file mat_mul_multi_process.h, you will see:

```
 9 #pragma hls_design top
10 class matrixMultiply  : public sc_module {
11  public:
12   sc_in<bool> CCS_INIT_S1(clk);
13   sc_in<bool> CCS_INIT_S1(rstn);
14
15   Connections::In <ac_int<8>> CCS_INIT_S1(A);
16   Connections::In <ac_int<8>> CCS_INIT_S1(B);
17   Connections::Out <ac_int<8+8+3>> CCS_INIT_S1(C);
18
19   ac_shared<ac_int<8> [64*2]> B_transpose;
20   Connections::Combinational <array_t<ac_int<8>,8>> CCS_INIT_S1(A_row);
21   Connections::SyncChannel CCS_INIT_S1(sync); // memory synchronization between threads
22
23   SC_CTOR(matrixMultiply) {
24     SC_THREAD(transpose);
25     sensitive << clk.pos();
26     async_reset_signal_is(rstn, false);
27
28     SC_THREAD(mac);
29     sensitive << clk.pos();
30     async_reset_signal_is(rstn, false);
31
32     SC_THREAD(pack_A);
33     sensitive << clk.pos();
34     async_reset_signal_is(rstn, false);
35   }
```

- On line 19, we declare B_transpose to be a shared memory that stores 8 bit elements. The size of the array is 64*2, since we use a ping-pong synchronization scheme to enable B data to be read while the mat_mul operation also executes.
- On line 21 we declare the sync channel to coordinate access to the ping-pong memory.
- On line 20 we declare data channel to transmit the A_row data, so that packing of the A_row items can occur in parallel with the mat_mul operations.

Design #2 – Multiple Processes (continued)

```
37  void transpose() {
38    B.Reset();
39    sync.reset_sync_out();
40    bool ping_pong = false;
41    wait();
42
43    while (1) {
44      #pragma hls_pipeline_init_interval 1
45      #pragma pipeline_stall_mode flush
46      TRANSPOSEB_ROW0:for (int i=0; i<8; i++) { // Transpose operation must complete first
47        TRANSPOSEB_COL0:for (int j=0; j<8; j++) {
48          B_transpose[j*8 + i + 64*ping_pong] = B.Pop();
49        }
50      }
51      ping_pong = !ping_pong;
52      sync.sync_out();
53    }
54  }
55
56  void pack_A() {
57    A.Reset();
58    A_row.ResetWrite();
59    wait();
60
61    #pragma hls_pipeline_init_interval 1
62    #pragma pipeline_stall_mode flush
63    while (1) {
64      array_t<ac_int<8>,8> A_dat;
65      for (int i=0; i<8; i++) {
66        A_dat.data[i] = A.Pop();
67      }
68      A_row.Push(A_dat);
69    }
70  }
```

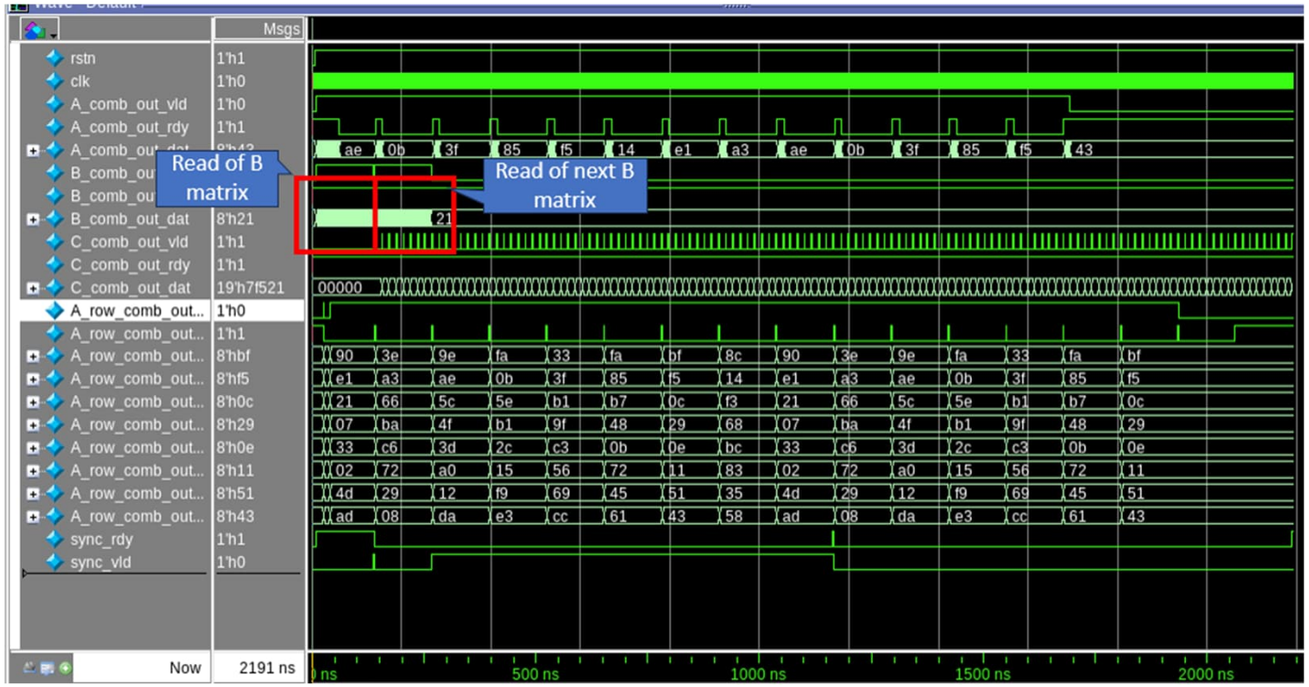
- On line 48 we read the B matrix to form the B_transpose matrix.
- On lines 51 we switch the ping_pong variable to control which half of the memory we write into, and on line 52 we notify the mac thread when new data is ready.
- On line 66 we read the A matrix data, and when we have a full row we push it out on line 68.

Design #2 – Multiple Processes (continued)

```
72  void mac() {
73    C.Reset();
74    sync.reset_sync_in();
75    A_row.ResetRead();
76    array_t<ac_int<8>,8> A_dat;
77    ac_int<8> B_dat;
78    bool ping_pong = false;
79    wait();
80
81    while (1) {
82      ac_int<8+8+3> acc = 0;
83      sync.sync_in();
84      #pragma hls_pipeline_init_interval 1
85      #pragma pipeline_stall_mode flush
86      ROW:for (int i = 0; i < 8; i++) {
87        A_dat = A_row.Pop();
88        COL:for (int j = 0; j < 8; j++) {
89          acc = 0;
90          MAC:for (int k = 0; k < 8; k++) { // Cannot unroll - would result in mem port contention
91            B_dat = B_transpose[j*8 + k + 64*ping_pong];
92            acc += A_dat.data[k] * B_dat;
93            #ifndef __SYNTHESIS__
94            wait();//need a wait for simulation if loop not unrolled for accurate timing
95            #endif
96          }
97          C.Push(acc);
98        }
99      }
100     ping_pong = !ping_pong;
101   }
102 }
```

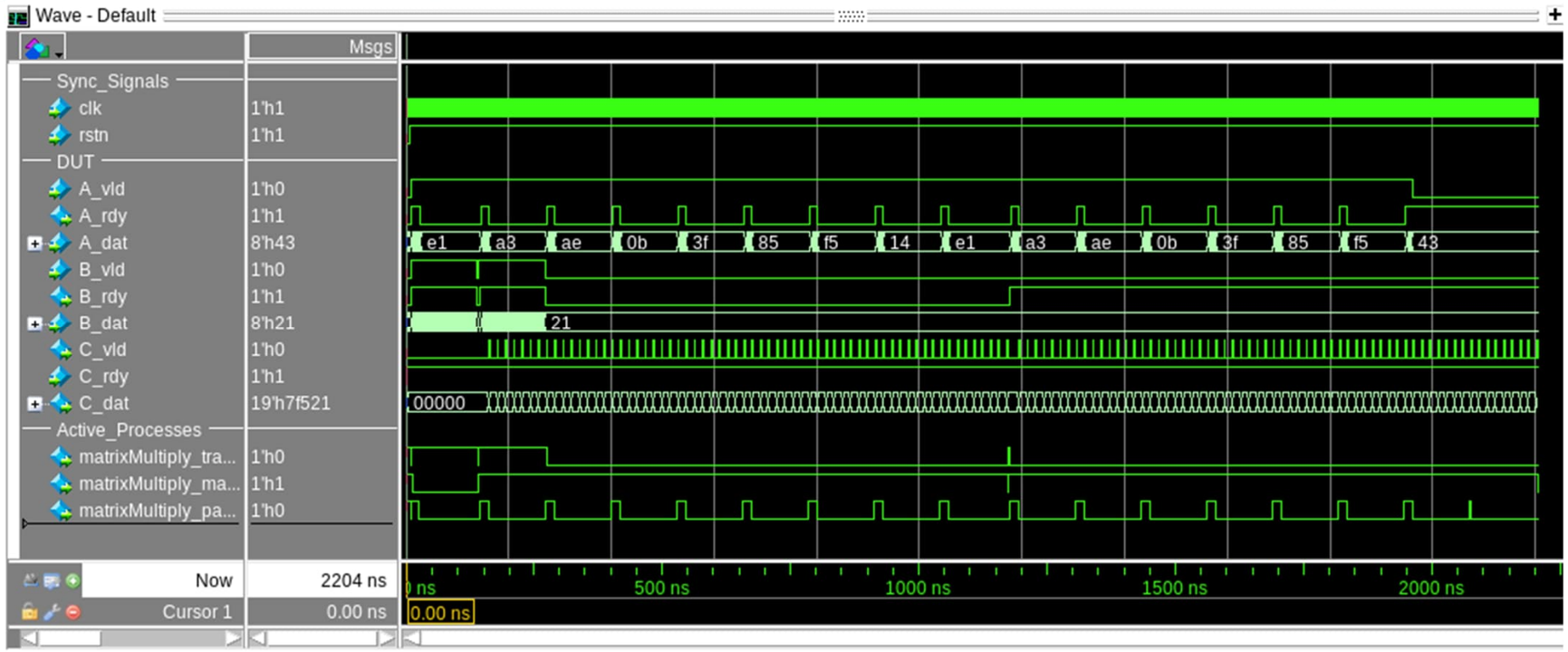
- On line 83 we synchronize with the transpose process so that the B_transpose data is ready in memory.
- On line 87 we read an entire A_row of data, and then on line 92 we perform the multiply accumulate operation.
- Note that we still cannot unroll the MAC loop on line 90 since it would require too many ports on the B_transpose memory, which is still mapped to a 1R1W memory.

Design #2 – Multiple Processes (continued)



- Running the pre-HLS simulation gives the waveforms above.
- Because of the ping-pong memory for the B_transpose matrix and the separate process for reading the B data, we see that both B matrices are fully read early in the simulation.
- The mat_mul operations and the output of the C data begins right after the first B matrix is read.
- The reading of the A matrix occurs in parallel with the mat_mul operations.
- Note, however, that the MAC loop is still rolled, so the overall execution time for this design is only a little better than the first design, despite the additional parallelism enabled by the multiple processes.

Design #2 – Multiple Processes (continued)



- Running the post-HLS RTL simulation gives the waveforms above.
- *Note the very close correspondence and timing of the post-HLS and pre-HLS waveforms.*
- This correspondence exists because Matchlib simulations are “throughput accurate”.

Design #3 – Banked Memory and Multiple Processes

- In the third design for the matrix multiplication, we use a multiple process design that performs the transpose operation, packing of the A row data, and the multiply accumulate operations in three separate processes.
- In addition, we use a banked memory for the B_transpose memory, which will enable the MAC loop to be unrolled fully to increase performance.
- If you view the file mat_mul_banked_multi_process.h, you will see:

```
11 #pragma hls_design top
12 class matrixMultiply  : public sc_module {
13  public:
14   sc_in<bool> CCS_INIT_S1(clk);
15   sc_in<bool> CCS_INIT_S1(rstn);
16
17   Connections::In <ac_int<8>> CCS_INIT_S1(A);
18   Connections::In <ac_int<8>> CCS_INIT_S1(B);
19   Connections::Out<ac_int<8+8+3>> CCS_INIT_S1(C);
20
21   ac_shared_bank_array_2D<ac_int<8>, 8, 8*2> B_transpose;
22   Connections::SyncChannel sync; // memory synchronization between threads
23   Connections::Combinational <array_t<ac_int<8>,8>> CCS_INIT_S1(A_row);
24
25   SC_CTOR(matrixMultiply) {
26     SC_THREAD(pack_A);
27     sensitive << clk.pos();
28     async_reset_signal_is(rstn, false);
29
30     SC_THREAD(transpose);
31     sensitive << clk.pos();
32     async_reset_signal_is(rstn, false);
33
34     SC_THREAD(mac);
35     sensitive << clk.pos();
36     async_reset_signal_is(rstn, false);
37   }
```

- This code is the same as in the second design except for line 21. On line 21 we declare a 2-dimensional banked array that stores 8 bit elements. There are 8 banks, and each bank stores 8*2 elements since we are still using a ping-pong memory organization to enable concurrent reading and processing of data.

Design #3 – Banked Memory and Multiple Processes (continued)

- The pack_A function is the same as in the previous design.
- The transpose function now is:

```
59  void transpose() {
60    B.Reset();
61    sync.reset_sync_out();
62    bool ping_pong = false;
63    wait();
64
65    while (1) {
66      #pragma hls_pipeline_init_interval 1
67      #pragma pipeline_stall_mode flush
68      TRANSPOSEB_ROW0:for (int i=0; i<8; i++) { // Transpose operation must complete first
69        TRANSPOSEB_COL0:for (int j=0; j<8; j++) {
70          B_transpose[i][j + 8*ping_pong] = B.Pop();
71        }
72      }
73      ping_pong = !ping_pong;
74      sync.sync_out();
75    }
76  }
```

- We see on line 70 that we read one item from the B matrix and store it in the proper position in the banked memory, considering the current ping_pong settings.
- These transpose loops could be unrolled during HLS, but there would be no benefit in either performance or area, so we leave them rolled.

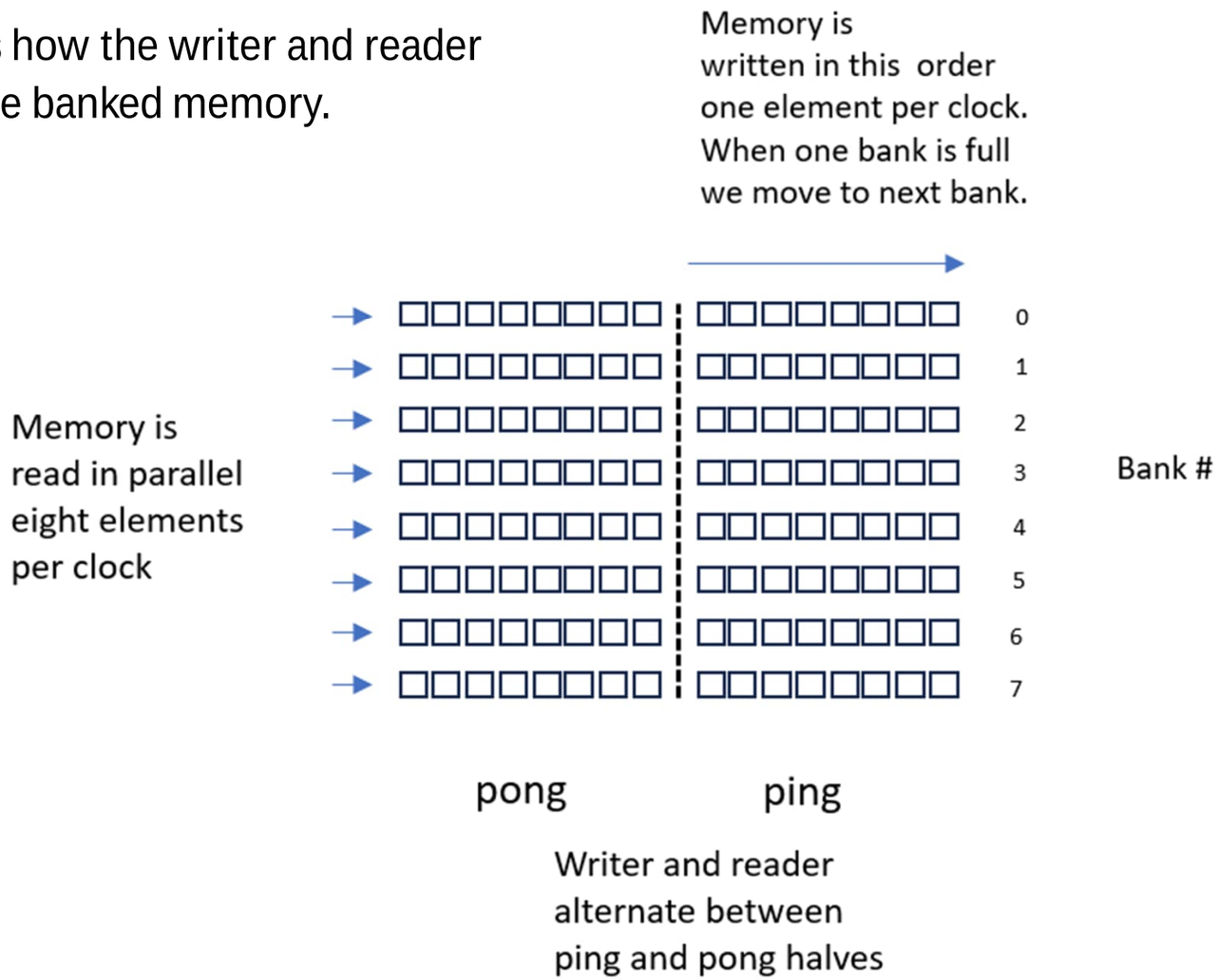
Design #3 – Banked Memory and Multiple Processes (continued)

```
78   void mac() {
79     C.Reset();
80     A_row.ResetRead();
81     sync.reset_sync_in();
82     array_t<ac_int<8>,8> A_dat;
83     ac_int<8> B_dat;
84     bool ping_pong = false;
85     wait();
86
87     while (1) {
88       ac_int<8+8+3> acc = 0;
89       sync.sync_in();
90       #pragma hls_pipeline_init_interval 1
91       #pragma pipeline_stall_mode flush
92       ROW:for (int i = 0; i < 8; i++) {
93         A_dat = A_row.Pop();
94         COL:for (int j = 0; j < 8; j++) {
95           acc = 0;
96           #pragma hls_unroll yes
97           MAC:for (int k = 0; k < 8; k++) { // loop can be unrolled without mem port contention..
98             B_dat = B_transpose[k][j + 8*ping_pong];
99             acc += A_dat.data[k] * B_dat;
100          }
101          C.Push(acc);
102        }
103      }
104      ping_pong = !ping_pong;
105    }
106  }
107 };
```

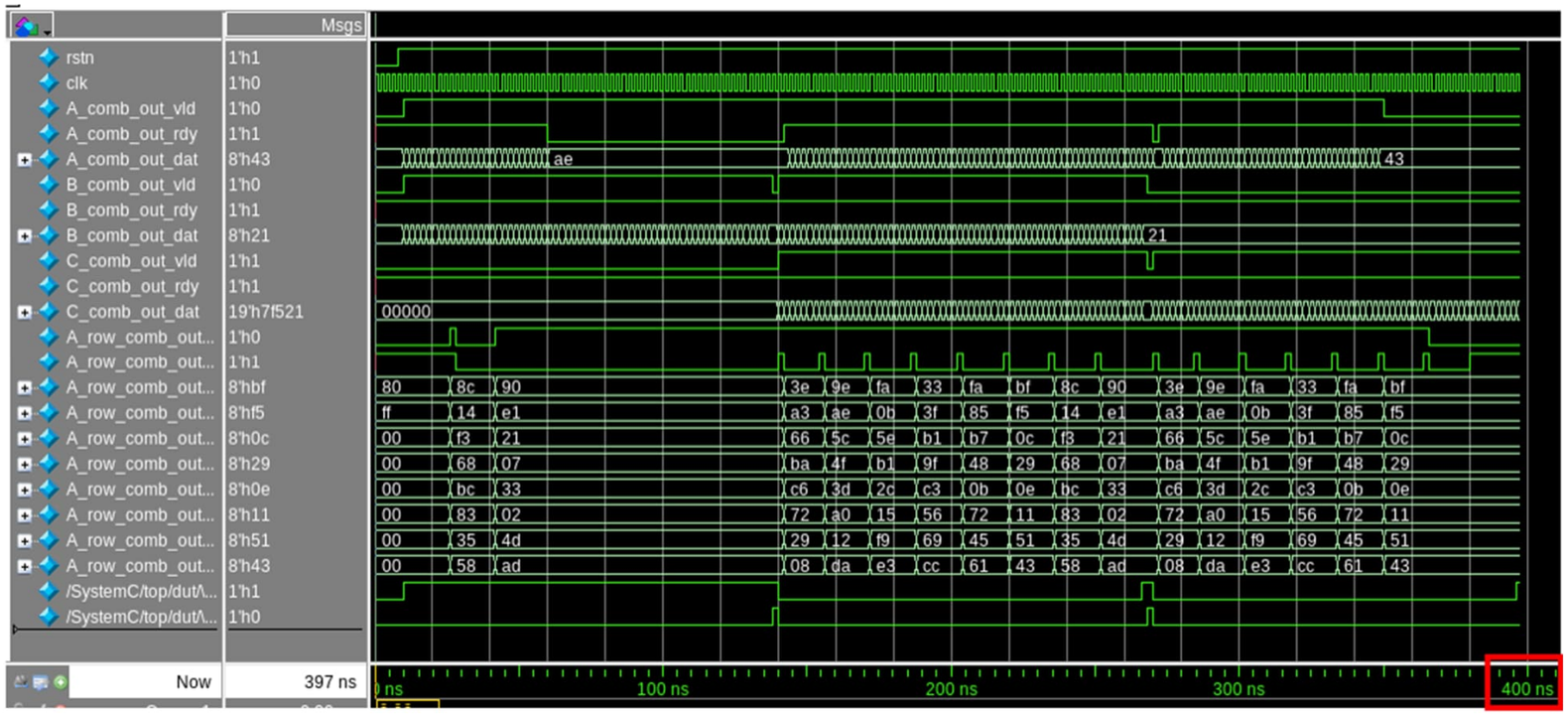
- On line 97 is the MAC loop, which we now unroll fully during HLS. When this is done, the k index to the B_transpose memory becomes a constant during HLS, and Catapult can see that there is no memory port contention. This allows a new COL loop iteration to start on every clock cycle (note however that now the design will require 8 multipliers in HW).
- On line 101 we push out a new dot product result, which now will occur about every clock cycle.

Design #3 – Banked Memory and Multiple Processes (continued)

- This diagram shows how the writer and reader processes access the banked memory.

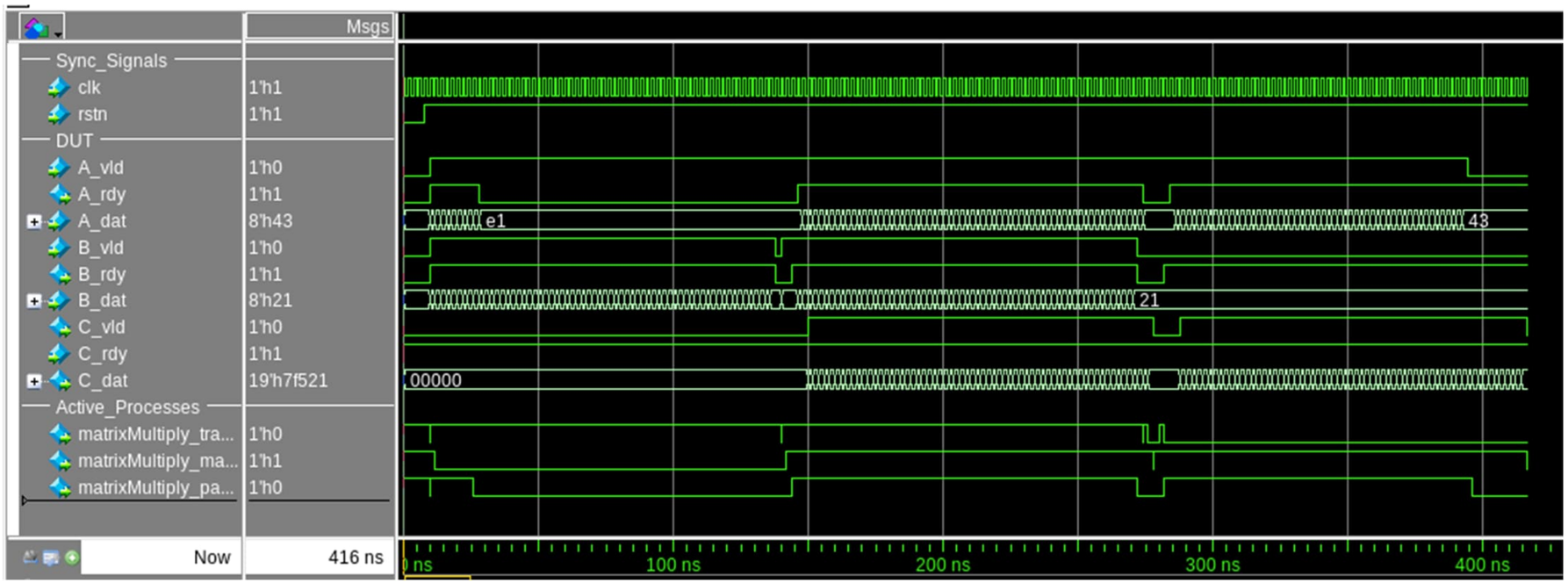


Design #3 – Banked Memory and Multiple Processes (continued)



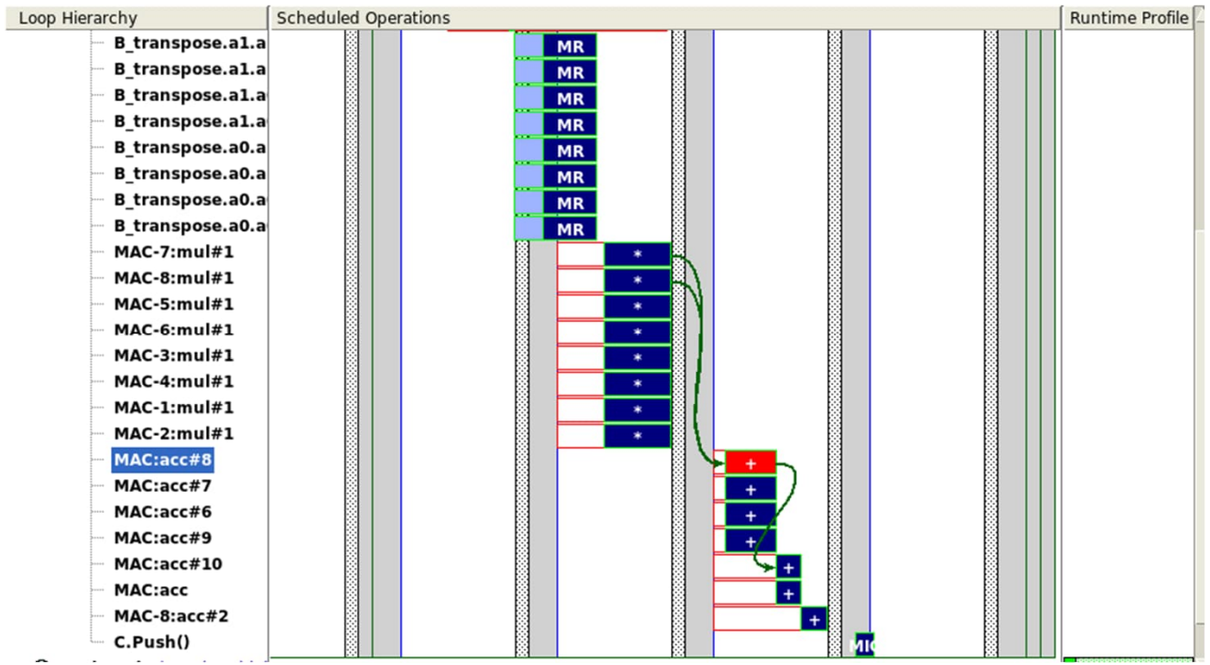
- The waveforms for the pre-HLS simulation are above.
- Note the reading of the A and B matrices happens concurrently with the mat_mul operation and the output of the C matrix.
- Note the output of the C matrix is completely overlapping with the reading of the second B matrix.
- The simulation completes in about 400 ns, which is much faster than the > 2000 ns for the previous two designs. This is because the MAC loop is now able to be completely unrolled due to the use of the banked B_transpose memory.

Design #3 – Banked Memory and Multiple Processes (continued)



- Running the post-HLS RTL simulation gives the waveforms above.
- *Note the very close correspondence and timing of the post-HLS and pre-HLS waveforms.*
- This correspondence exists because Matchlib simulations are “throughput accurate”.

Design #3 – Banked Memory and Multiple Processes (continued)



- If we view the schedule in Catapult for the operations in the MAC loop, we see the picture above.
- The overall pipeline for the COL loop body has several clock cycles of latency, but on each cycle 8 memory read operations occur, and 8 multiplications and corresponding additions occur.
- In the post-HLS RTL, all these details are present. In the pre-HLS model, the memory read operations and the 8 multiplications and corresponding additions occur in zero time.
- Despite these differences between the two models, the overall performance accuracy of the pre-HLS model compared to the post-HLS model is very high.

Introduction to SystemC Coding for HLS

Stuart Swan
Platform Architect
Siemens EDA
3 April 2024

SIEMENS

Why use HLS?

- Automatic scheduling, resource sharing, pipelining
- Easier retargeting to different silicon technologies
- More abstract design models
- Enable more DV and debugging to occur at higher abstraction level
- Make architectural exploration feasible

Why use C++ for HLS?

- Templates, classes, inheritance, …
- Easy integration of existing C/C++ models
- Simulation speed
- …

Why use SystemC + Matchlib for HLS?

- SystemC provides HDL semantics on top of C++ language
  - Time, module structure, hierarchy, channels, HW semantics, resets, signals
  - Think of SystemC as an HDL that uses C++
- 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

SC Combinational Process (example O1_and_gate)

```
 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

SC Sequential Process (example 02_flop)

```
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

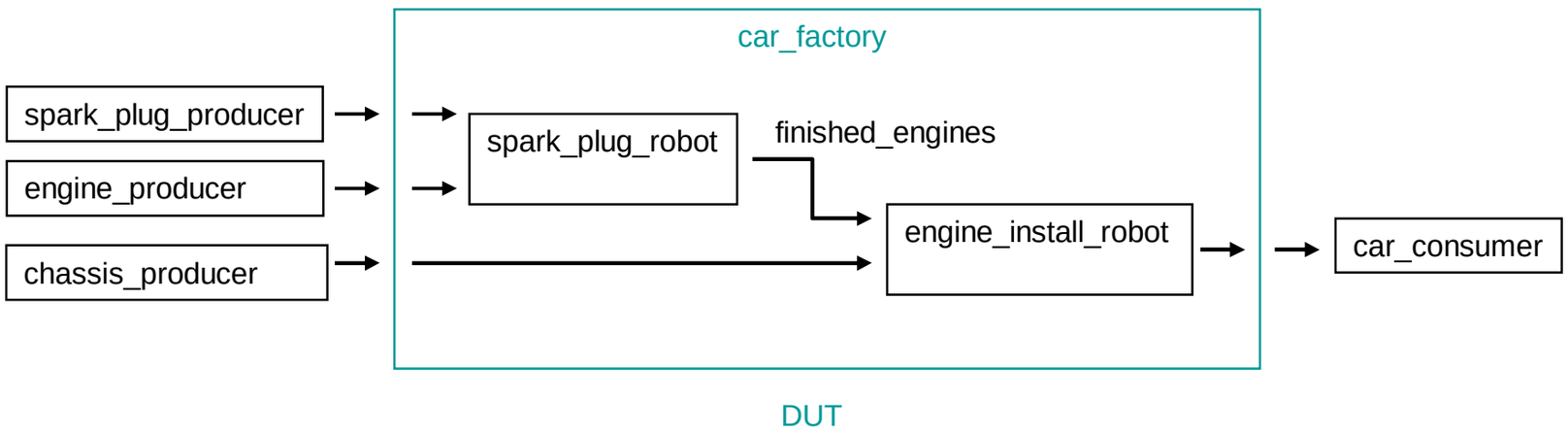
Important to understand how HLS sees your SC model:

- Set of processes communicating **only** thru signals (sc_signal<>)
  - Only exception: Memories shared between processes are modeled as shared arrays with explicit synchronization
- 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 throughput 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

Simple Example of HW Architectural Model using SC + Matchlib (ex 07*)



spark_plug_producer

```
15 class spark_plug_producer : public sc_module {
16 public:
17   sc_in<bool>                    INIT_S1(clk);
18   Connections::Out<spark_plug_t> INIT_S1(spark_plugs);
19
20   SC_CTOR(spark_plug_producer) {
21     SC_THREAD(main);
22     sensitive << clk.pos();
23   }
24
25   void main() {
26     int count=0;
27
28     while (1) {
29       spark_plug_t spark_plug;
30       spark_plug.spark_plug = count++;
31       spark_plugs.Push(spark_plug);
32       wait(3);
33       if (rand() & 1)
34         wait(3);
35     }
36   }
37 };
```

produces new spark_plug every 3-6 seconds

engine_producer

```
39 class engine_producer : public sc_module {
40 public:
41   sc_in<bool>                 INIT_S1(clk);
42   Connections::Out<engine_t> INIT_S1(engines);
43
44   SC_CTOR(engine_producer) {
45     SC_THREAD(main);
46     sensitive << clk.pos();
47   }
48
49   void main() {
50     int count=0;
51
52     while (1) {
53       engine_t engine;
54       engine.engine = count++;
55       engines.Push(engine);
56       wait(20);
57     }
58   }
59 };
```

produces new engine every 20 seconds

chassis_producer

```
61 class chassis_producer : public sc_module {
62 public:
63   sc_in<bool>                   INIT_S1(clk);
64   Connections::Out<chassis_t>   INIT_S1(chassis_out);
65
66   SC_CTOR(chassis_producer) {
67     SC_THREAD(main);
68     sensitive << clk.pos();
69   }
70
71   void main() {
72     int count=0;
73
74     while (1) {
75       chassis_t chassis;
76       chassis.chassis = count++;
77       chassis_out.Push(chassis);
78       wait(25);
79     }
80   }
81 };
```

produces new chassis every 25 seconds

car_consumer

```
83 class car_consumer : public sc_module {
84 public:
85   sc_in<bool>             INIT_S1(clk);
86   Connections::In<car_t>  INIT_S1(cars);
87
88   SC_CTOR(car_consumer) {
89     SC_THREAD(main);
90     sensitive << clk.pos();
91   }
92
93   void main() {
94     int count = 0;
95     while (1) {
96       cars.Pop();
97       ++count;
98       LOG("got car # " << count);
99       if (count == 10)
100      {
101        LOG("total cars produced: " << count);
102        LOG("time per car: " << sc_time_stamp() / count);
103        sc_stop();
104      }
105    }
106  }
107 };
```

consumes cars as quickly as possible

Simple car_factory

```
139 #if defined(SIMPLE)
140 class car_factory : public sc_module {
141 public:
142   sc_in<bool>                   INIT_S1(clk);
143   Connections::In<spark_plug_t> INIT_S1(spark_plugs);
144   Connections::In<engine_t>     INIT_S1(engines);
145   Connections::In<chassis_t>    INIT_S1(chassis);
146   Connections::Out<car_t>       INIT_S1(cars);
147
148   Connections::Combinational<engine_t> INIT_S1(finished_engines);
149
150   spark_plug_robot      INIT_S1(spark_plug_robot1);
151   engine_install_robot  INIT_S1(engine_install_robot1);
152
153   SC_CTOR(car_factory)
154   {
155     spark_plug_robot1.clk(clk);
156     spark_plug_robot1.spark_plugs(spark_plugs);
157     spark_plug_robot1.engines_in(engines);
158     spark_plug_robot1.engines_out(finished_engines);
159
160     engine_install_robot1.clk(clk);
161     engine_install_robot1.chassis(chassis);
162     engine_install_robot1.engines(finished_engines);
163     engine_install_robot1.cars(cars);
164   }
165 };
```

spark_plug_robot

```
71 class spark_plug_robot : public sc_module {
72 public:
73   sc_in<bool>                     INIT_S1(clk);
74   Connections::In<spark_plug_t>   INIT_S1(spark_plugs);
75   Connections::In<engine_t>       INIT_S1(engines_in);
76   Connections::Out<engine_t>      INIT_S1(engines_out);
77   SC_SIG(bool, busy);
78   SC_SIG(bool, maintenance);
79
80   SC_CTOR(spark_plug_robot)
81   {
82     SC_THREAD(main);
83     sensitive << clk.pos();
84   }
85
86   void main() {
87     int count = 0;
88     while (1) {
89        engine_t engine_in = engines_in.Pop();
90        for (int i=0; i < engine_t::plugs; i++)
91           engine_in.spark_plugs[i] = spark_plugs.Pop();
92        busy = 1;
93        wait(60);
94        busy = 0;
95        engines_out.Push(engine_in);
96        if ((count++ & 1) && (rand() & 3))
97        {
98          maintenance = 1;
99          wait(60);
100         maintenance = 0;
101       }
102    }
103  }
104 };
```

Consumes 4 spark_plugs and 1 unfinished engine
Produces finished_engine after 60 seconds
After every other engine, 75% of time
needs 60 seconds of maintenance (ie idle time)

engine_install_robot

```
106 class engine_install_robot : public sc_module {
107 public:
108   sc_in<bool>                  INIT_S1(clk);
109   Connections::In<chassis_t>   INIT_S1(chassis);
110   Connections::In<engine_t>    INIT_S1(engines);
111   Connections::Out<car_t>      INIT_S1(cars);
112   SC_SIG(bool, busy);
113
114   SC_CTOR(engine_install_robot)
115   {
116     SC_THREAD(main);
117     sensitive << clk.pos();
118   }
119
120   void main() {
121     while (1) {
122       car_t car;
123       car.chassis = chassis.Pop();
124       car.engine = engines.Pop();
125       busy = 1;
126       wait(30);
127       busy = 0;
128       cars.Push(car);
129     }
130   }
131 };
```

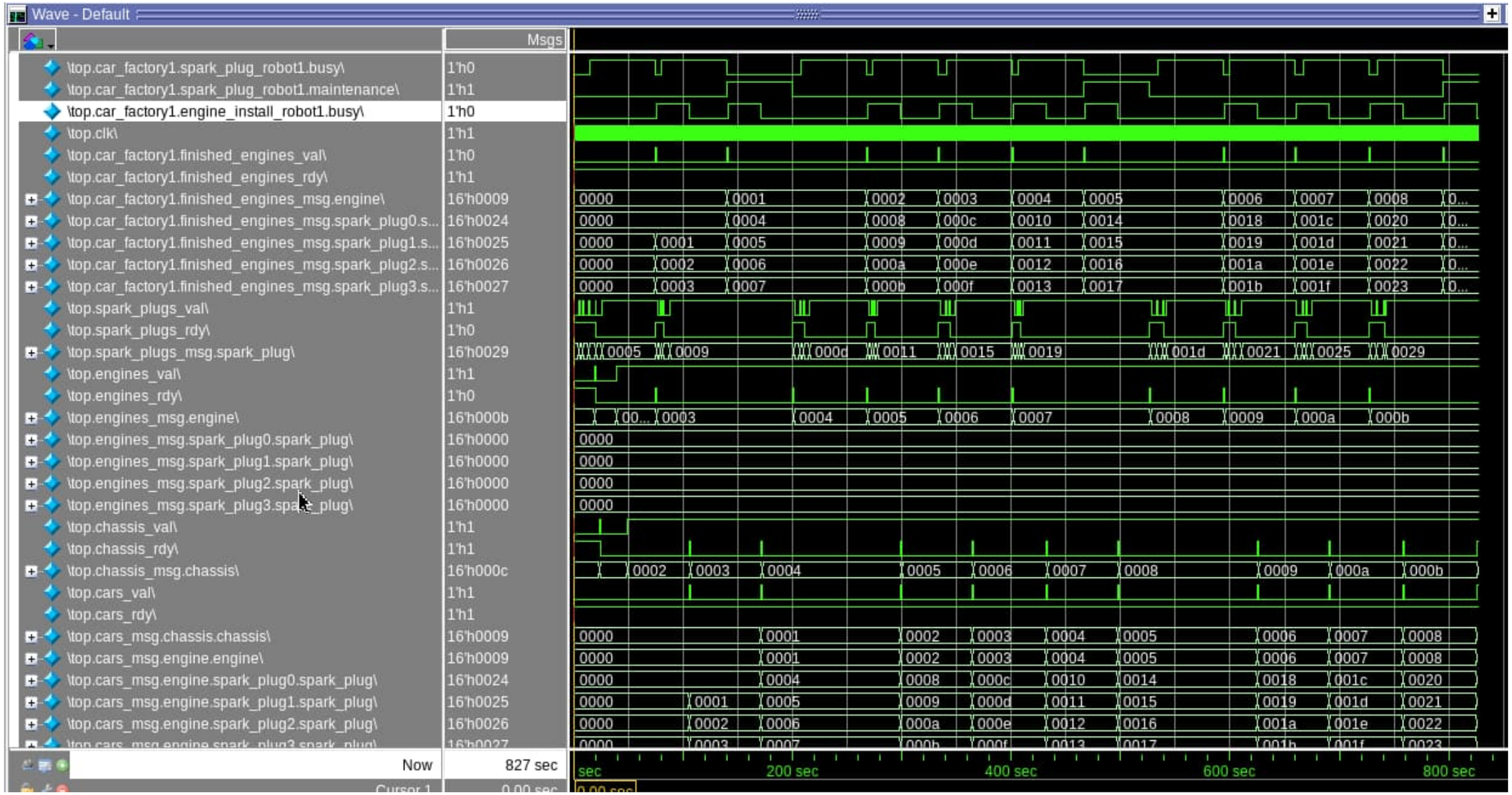
Consumes 1 chassis and 1 finished_engine
Produces car after 30 seconds

Running simple car_factory

```
107 s top.car_consumer1 got car # 1
172 s top.car_consumer1 got car # 2
300 s top.car_consumer1 got car # 3
365 s top.car_consumer1 got car # 4
433 s top.car_consumer1 got car # 5
498 s top.car_consumer1 got car # 6
626 s top.car_consumer1 got car # 7
691 s top.car_consumer1 got car # 8
759 s top.car_consumer1 got car # 9
827 s top.car_consumer1 got car # 10
827 s top.car_consumer1 total cars produced: 10
827 s top.car_consumer1 time per car: 82700 ms

Info: /OSCI/SystemC: Simulation stopped by user.
```

Goal is to produce each car in smallest amount of time

Running simple car_factory



Sequential car_factory

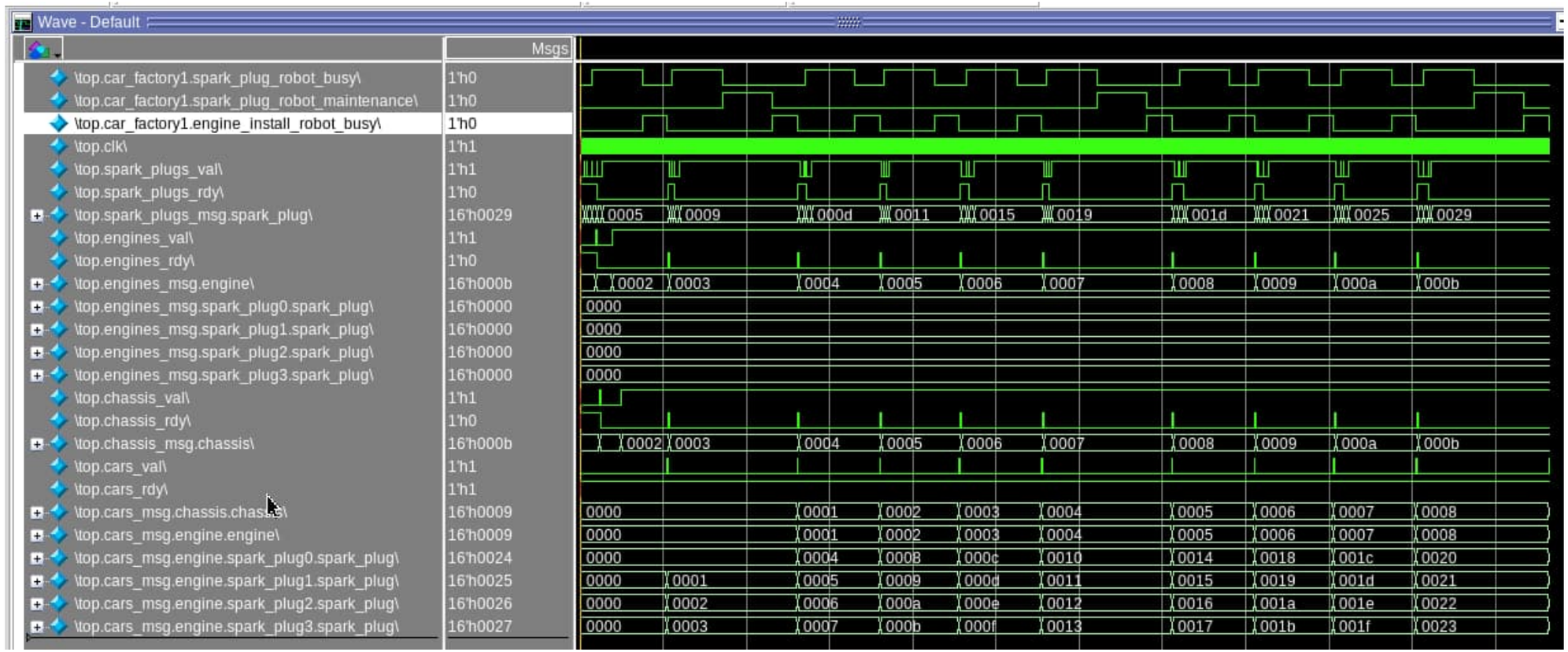
```
#elif defined(SEQUENTIAL)
class car_factory : public sc_module {
public:
  sc_in<bool>                   INIT_S1(clk);
  Connections::In<spark_plug_t> INIT_S1(spark_plugs);
  Connections::In<engine_t>     INIT_S1(engines);
  Connections::In<chassis_t>    INIT_S1(chassis);
  Connections::Out<car_t>       INIT_S1(cars);

  SC_CTOR(car_factory)
  {
    SC_THREAD(main);
    sensitive << clk.pos();
  }

  SC_SIG(bool, spark_plug_robot_busy);
  SC_SIG(bool, spark_plug_robot_maintenance);
  SC_SIG(bool, engine_install_robot_busy);

  void main() {
    spark_plug_t plugs[engine_t::plugs];
    int plug_count = 0;

    engine_t unfinished_engine;
    int unfinished_engine_count = 0;
    engine_t finished_engine;
    int finished_engine_count = 0;
    chassis_t chassis_inst;
    int chassis_count = 0;
    int spark_plug_robot_count = 0;

    while (1) {
      if (plug_count < engine_t::plugs)
        if (spark_plugs.PopNB(plugs[plug_count]))
          ++plug_count;

      if (unfinished_engine_count == 0)
        if (engines.PopNB(unfinished_engine))
          ++unfinished_engine_count;

      if (chassis_count == 0)
        if (chassis.PopNB(chassis_inst))
          ++chassis_count;

      if ((unfinished_engine_count == 1) && (plug_count == engine_t::plugs)
          && (finished_engine_count == 0))
      {
       finished_engine = unfinished_engine;
       for (int i=0; i < engine_t::plugs; i++)
          finished_engine.spark_plugs[i] = plugs[i];
       spark_plug_robot_busy = 1;
       wait(60);
       spark_plug_robot_busy = 0;
       if ((spark_plug_robot_count++ & 1) && (rand() & 3))
       {
         spark_plug_robot_maintenance = 1;
         wait(60);
         spark_plug_robot_maintenance = 0;
       }
       finished_engine_count = 1;
       plug_count = 0;
       unfinished_engine_count = 0;
      }

      if ((finished_engine_count == 1) && (chassis_count == 1))
      {
        car_t car;
        car.chassis = chassis_inst;
        car.engine = finished_engine;
        engine_install_robot_busy = 1;
        wait(30);
        engine_install_robot_busy = 0;
        cars.Push(car);
        finished_engine_count = 0;
        chassis_count = 0;
      }

      wait();
    }
  }
};
```

Running sequential car_factory

```
106 s top.car_consumer1 got car # 1
262 s top.car_consumer1 got car # 2
361 s top.car_consumer1 got car # 3
457 s top.car_consumer1 got car # 4
556 s top.car_consumer1 got car # 5
712 s top.car_consumer1 got car # 6
811 s top.car_consumer1 got car # 7
907 s top.car_consumer1 got car # 8
1006 s top.car_consumer1 got car # 9
1165 s top.car_consumer1 got car # 10
1165 s top.car_consumer1 total cars produced: 10
1165 s top.car_consumer1 time per car: 116500 ms

Info: /OSCI/SystemC: Simulation stopped by user.
```

Car production time got worse!

Running sequential car_factory



← Worse!

← Worse!

Will HLS fix sequential car_factory?

- Can HLS split the single sequential process into 2 different processes to improve the utilization / QOR?
  - No. HLS always generates a single FSM+Datapath per SC process in the input model.

- OK, then can HLS generate single FSM that is the product of the 2 simpler state machines?
  - No. Even if it did, you would have "state explosion", leading to bad QOR due to a huge FSM.

The “state explosion” problem

5.1.1 State Explosion

A flat FSM suffers from *state explosion*, which occurs when multiple independent activities interfere in a single model. Assume that an FSM has to capture two independent activities, each of which can be in one of three states. The resulting FSM,

P.R. Schaumont, *A Practical Introduction to Hardware/Software Codesign*, 133
DOI 10.1007/978-1-4419-6000-9_5, © Springer Science+Business Media, LLC 2010

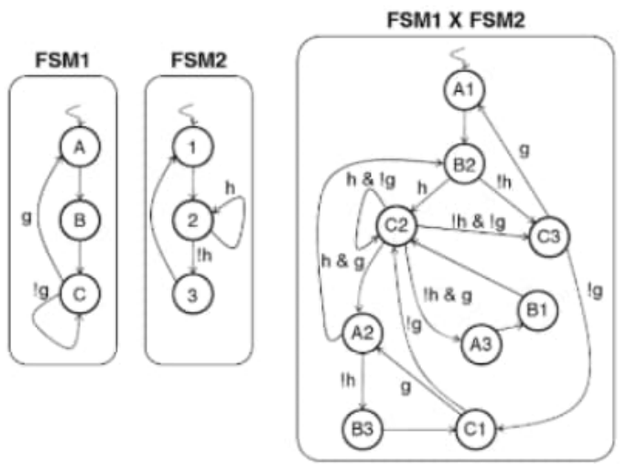


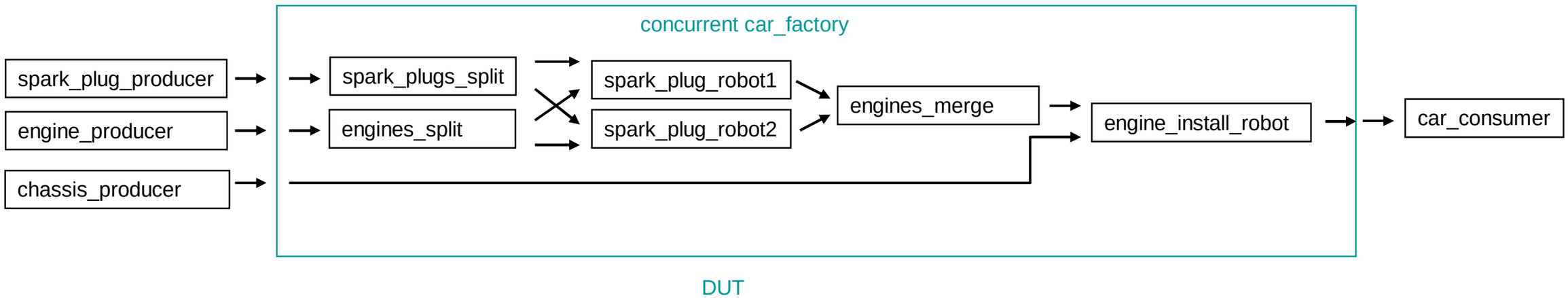
Fig. 5.1 State explosion in FSM when creating a product state-machine

called a *product state-machine*, needs nine states to represent the overall model. The product state-machine needs to keep track of the current state from two independent state machines at the same time. Due to conditional state transitions, one state machine can remain in a single state while the other state machine proceeds to the next state. This results in multiple intermediate states such as `A1`, `A2`, and `A3`. Figure 5.1 illustrates the effect of state explosion in a product state-machine. Two state machines, `FSM1` and `FSM2`, need to be merged into a single product state-machine `FSM1xFSM2`. In order to represent all individual states, 9 states are needed in total. The resulting number of state transitions (and state transition conditions) is even higher. Indeed, if we have n independent state transition conditions in the individual state machines, the resulting product state-machine can have up to 2^n state transition conditions.

How do we fix the car_factory architecture?

- Primary problem in “simple” car_factory is overutilization of spark_plug_robot
- Obvious solution: add another spark_plug_robot

concurrent car_factory



spark_plugs_split and engines_split

```
class spark_plugs_split : public sc_module {
public:
  sc_in<bool>                       INIT_S1(clk);
  Connections::In<spark_plug_t>     INIT_S1(spark_plugs_in);
  Connections::Out<spark_plug_t>    INIT_S1(spark_plugs_out1);
  Connections::Out<spark_plug_t>    INIT_S1(spark_plugs_out2);

  SC_CTOR(spark_plugs_split)
  {
    SC_THREAD(main);
    sensitive << clk.pos();
  }

  void main() {
    while (1) {
      spark_plug_t spark_plug = spark_plugs_in.Pop();
      while (1) {
        if (spark_plugs_out1.PushNB(spark_plug))
          break;
        if (spark_plugs_out2.PushNB(spark_plug))
          break;
        wait();
      }
    }
  }
};
```

```
class engines_split : public sc_module {
public:
  sc_in<bool>                   INIT_S1(clk);
  Connections::In<engine_t>     INIT_S1(engines_in);
  Connections::Out<engine_t>    INIT_S1(engines_out1);
  Connections::Out<engine_t>    INIT_S1(engines_out2);

  SC_CTOR(engines_split)
  {
    SC_THREAD(main);
    sensitive << clk.pos();
  }

  void main() {
    while (1) {
      engine_t engine = engines_in.Pop();
      while (1) {
        if (engines_out1.PushNB(engine))
          break;
        if (engines_out2.PushNB(engine))
          break;
        wait();
      }
    }
  }
};
```

engines_merge

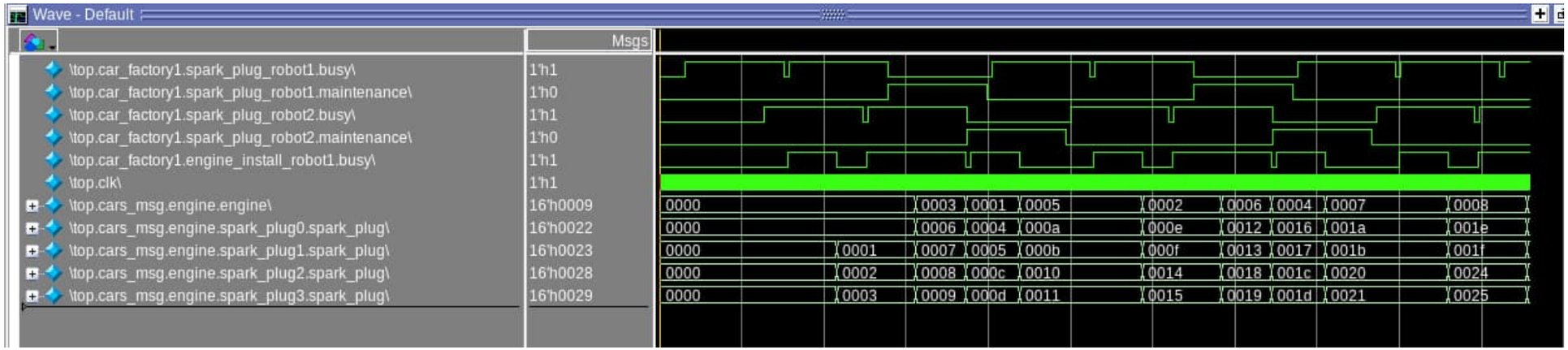
```
class engines_merge : public sc_module {
public:
  sc_in<bool>                    INIT_S1(clk);
  Connections::In<engine_t>      INIT_S1(engines_in1);
  Connections::In<engine_t>      INIT_S1(engines_in2);
  Connections::Out<engine_t>     INIT_S1(engines_out);

  SC_CTOR(engines_merge)
  {
    SC_THREAD(main);
    sensitive << clk.pos();
  }

  void main() {
    while (1) {
      engine_t engine;
      while (1) {
        if (engines_in1.PopNB(engine))
          break;
        if (engines_in2.PopNB(engine))
          break;
        wait();
      }
      engines_out.Push(engine);
    }
  }
};
```

Running concurrent car_factory

```
109 s top.car_consumer1 got car # 1
157 s top.car_consumer1 got car # 2
187 s top.car_consumer1 got car # 3
220 s top.car_consumer1 got car # 4
295 s top.car_consumer1 got car # 5
343 s top.car_consumer1 got car # 6
373 s top.car_consumer1 got car # 7
406 s top.car_consumer1 got car # 8
481 s top.car_consumer1 got car # 9
529 s top.car_consumer1 got car # 10
529 s top.car_consumer1 total cars produced: 10
529 s top.car_consumer1 time per car: 52900 ms

Info: /OSCI/SystemC: Simulation stopped by user.
```

Big improvement in car production time!

Running concurrent car_factory



Both spark_plug_robots busy at same time

Better utilization

Output order of engines and plugs no longer matches input order

Summing it up…

- In practice, SC modeling for HLS is a series of stepwise refinements that get to a good architectural model.
- You can do a lot of refinement & architectural analysis in pure SC model by analyzing the pre-HLS simulations.
- Use HLS to find power/performance/area (PPA) issues and iterate with changes to the SC model source code.

SystemC HLS Process Structure and Blocking/Non-Blocking IO

- A process (SC_THREAD) is like an “always” block in Verilog that is sensitive to the clock.
  - Usually it is “bigger” than a Verilog always block because it is an implicit state machine, may be pipelined by catapult, etc.
  - Usually every "block" in the HW architecture diagram will imply at least one independent SC_THREAD in the HLS model.
- Process structure has a huge impact on HLS runtime and QOR
  - Big processes usually mean "big" FSMs with lots of states and transitions **== bad QOR**
  - So, knowing how to arrive at a "good" process structure is extremely important
  - We saw in the car_factory example how breaking the design down into simpler concurrent activities led to a more efficient implementation

Guidelines for arriving at a good process structure

- Usually IO is the biggest consideration in how to structure SC_THREADs.
- The ideal structure is usually an SC_THREAD with at least one blocking input (ie one Pop operation) and/or at least one blocking output (ie Push operation).
- At least one of the blocking inputs or outputs should be called unconditionally on every iteration of the main loop.
- If functionality can easily be expressed as independent SC_THREADs then it is generally better to do so – they will run independently, have smaller control FSMs, etc.
- But, resources cannot be shared by Catapult across different SC_THREADs, and any communication between 2 SC_THREADs will take at least 1 clock cycle since they are clocked processes.

Unconditional blocking inputs or outputs..

- Why is it important to have at least one unconditional blocking input or output?
  - Usually the blocking IO is tied to each iteration of the main loop (which may be pipelined with for example II=1).
  - The blocking IO semantics and the clear II semantics clearly tie down the expected throughput requirements between the user and the HLS tool.
  - In contrast, if the IO is non-blocking, then even if the II=1 then Catapult may generate a state machine where on some iterations of a pipelined loop no actual IO occurs (since it is non-blocking)
  - Also, with blocking IO, the FSM will be in a clear "blocked" state when no IO can occur, and this makes idle state modeling easier (e.g. for power saving)

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.
  - With all non-blocking IO, typically you do not want to use PushNB/PopNB, but instead use disable_spawn and write the code using the “RTL in SystemC” style. See the 52_apb example discussed later in these slides for further details.

Summary: 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.

Where to use wait() in HLS models

- Use 1 wait() statement at top of main loop to model the reset state.
- Only use wait() in HLS models within loops that have exclusively non-blocking IO.
- If you are modeling low-level protocols using sc_signal, you may need to use wait() (but you should try to avoid doing this!)
- You should not be using wait() anywhere else.

Always try to think first about the HW architecture of the model

- Put your HW architects in front of a white board and get them to talk!
- What are primary blocks, transaction streams and required throughputs?
- What activities can naturally be expressed as separate concurrent processes?
- Where does arbitration need to occur?
- How can the architecture be modified to reduce undesirable contention?
- How can bottlenecks to RAM ports be avoided?
- How can data storage and lifetime be minimized?
- What are expensive resources that should be shared ?
  - Either add into same process,
  - Or add arbiter to allow multiple processes to access

What should be left for Catapult to do?

- Automated microarchitecture generation:
  - Detailed scheduling
  - resource allocation and sharing
  - loop unrolling
  - loop pipelining
  - memory access scheduling and interfacing
  - FSM generation
  - meeting timing of target silicon
  - PPA (power, performance, area) optimization

Conclusion

- Modeling a "good" process structure is key to getting good QOR thru HLS
- HLS tools such as Catapult can provide analysis feedback to help you identify "bad" process structures, but they will not automatically find a "good" one for you.
- Properly modeling the HW architecture and a good process structure is the responsibility of the HLS modeling engineers and HW architects

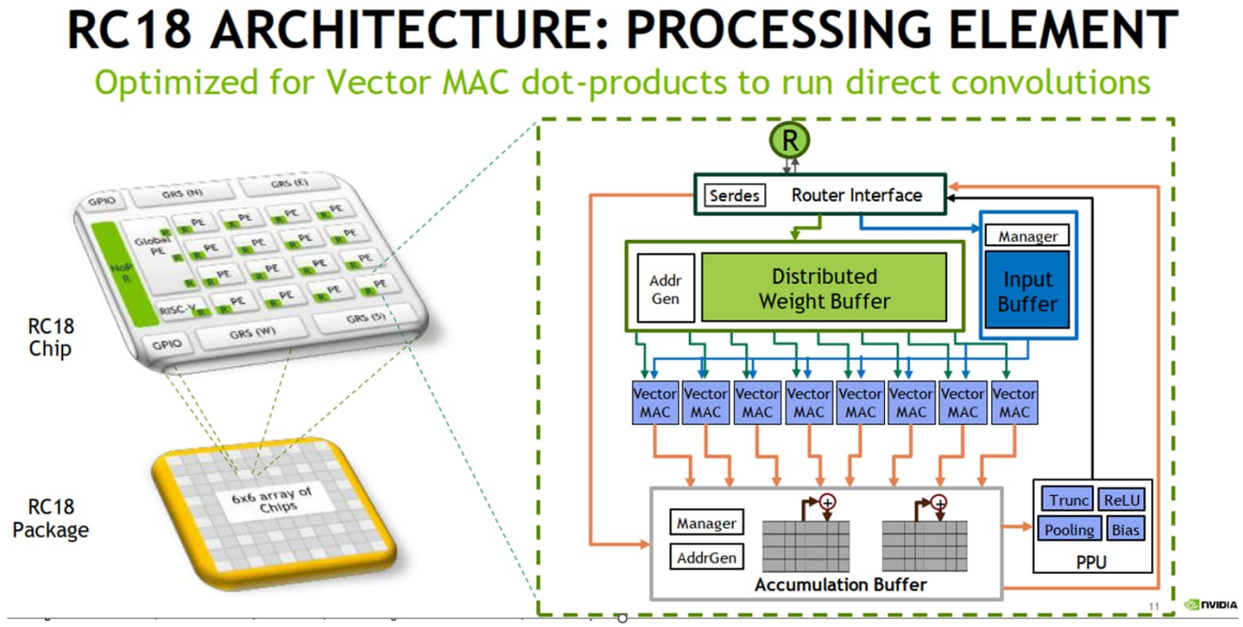
What is a “Throughput Accurate” Model?

Stuart Swan
Platform Architect
Siemens EDA
18 January 2024

SIEMENS

What is a “Throughput Accurate” Model?

- Consider an example SOC design shown here.
  - Entire chip is modeled in SystemC and implemented with HLS on a particular ASIC technology.
  - Blocks in the chip will have clocks, resets, transaction interfaces, signal interfaces, etc.
  - Clocks will have specific clock frequencies that account for target technology.
- In real chip, silicon will have gate delays.
  - HLS will implement pipelines and RTL synthesis will implement gate level logic to account for gate delays.
  - These delays will be accounted for in the RTL in a cycle accurate manner.



SIEMENS

Now, Assume Infinitely Fast Silicon…

- Consider the original SystemC model, but let's assume that the silicon is infinitely fast.
  - Keep exactly same clocks and clock frequencies as in RTL.
  - Keep exactly same transaction and signal interfaces as in RTL.
  - By default, we do not model any latency in pipelines, since the silicon is infinitely fast.
  - By default, we do not model any latency for memory accesses modeled as arrays.
  - Note that transactions will still stream through interfaces at the same rate as the real silicon.
  - This model is **throughput accurate** with respect to the real silicon.

Matchlib Enables Throughput Accurate Models

- Matchlib enables easy construction of throughput accurate models for pre-HLS simulation.
- This is very valuable because:
  - Performance-accurate model is available very early in design flow.
  - Much easier to debug than RTL.
  - Very high simulation performance (>30x RTL)
  - Pin level compatible with RTL.
  - Enables focus of verification and debug effort to move from RTL to pre-HLS model.
  - Post-HLS RTL model will closely match performance characteristics of pre-HLS model.

Is Matchlib Required for Throughput Accurate Models?

- Can you create a throughput accurate model in SystemC without using Matchlib?
  - Yes, but you need to code each process such that all the wait statements are manually merged.
  - Doing this is much harder, especially as designs becomes larger.
  - Model code will become complex and difficult to maintain.

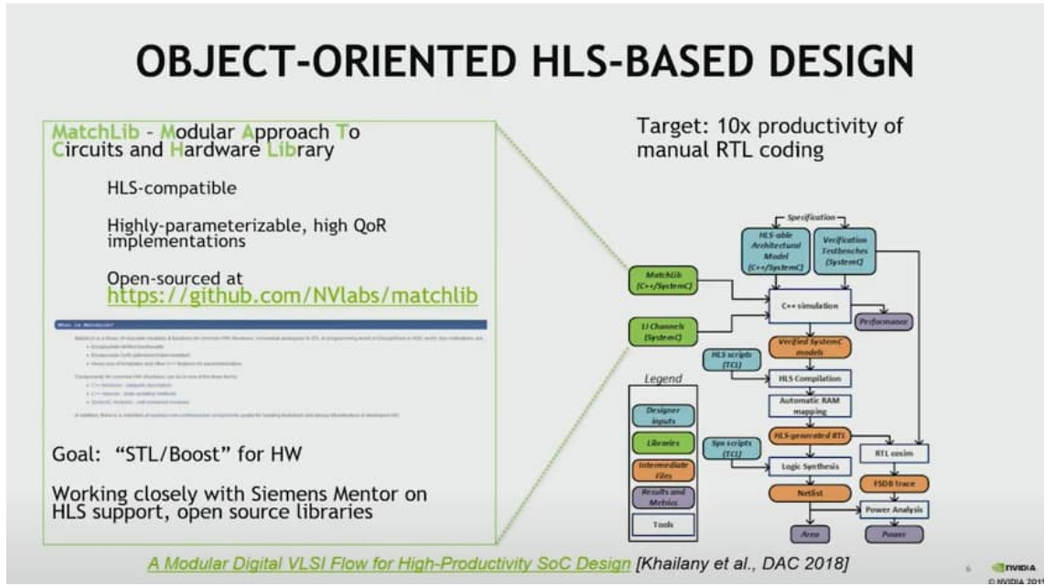
Introduction to Matchlib

Stuart Swan
Platform Architect
Siemens EDA
18 January 2024

SIEMENS

What is NVIDIA MatchLib?

- Good 20 minute intro video here:
  - https://www.youtube.com/watch?v=n8_G-CaSSPU



Key Parts of MatchLib

- “Connections”
  - Synthesizeable Message Passing Framework
  - SystemC/C++ used to accurately model concurrent IO that synthesized HW will have
  - Automatic stall injection enables interconnect to be stress tested in SystemC
  - Supports message latency and capacity back-annotation into pre-HLS model
- Parameterized AXI4 Fabric Components
  - Router/Splitter
  - Arbiter
  - AXI4 <-> AXI4Lite
  - Automatic burst segmentation and last bit generation
- Parameterized Banked Memories, Crossbar, Reorder Buffer
- Parameterized NOC components

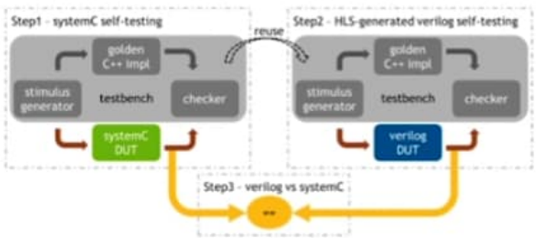
MatchLib SystemC Model Characteristics

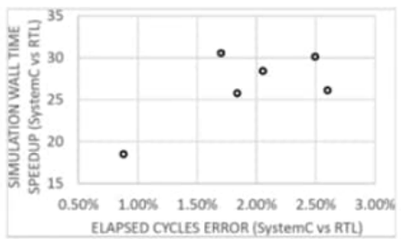
- Small
  - Typically 1/10 or less than the size of comparable RTL models
- Fast
  - Simulates ~30 times faster than RTL models in timing accurate mode
  - Simulates ~300 times faster than RTL models in blocking TLM mode (via compile time flag)
- Accurate
  - Not exactly RTL cycle accurate, but pretty close
  - Concurrent transactions in HW are modeled very accurately
- Fully automated path to placed gates via SystemC HLS
- Enables SW/FW models to be integrated via C++ host-code or CPU models
- Enables single-source model for HW and FW for full flow

MatchLib Results using HLS

RC17 SYSTEMC-BASED VERIFICATION

Functional and Performance Verification on SystemC models





FUNCTIONAL VERIFICATION

- Most verification run on SystemC/C++, signed off using C++ coverage tools
- Reuse of SystemC testbenches on HLS-generated RTL DUTs
- Automated stall injection and in-design assertions for improved coverage

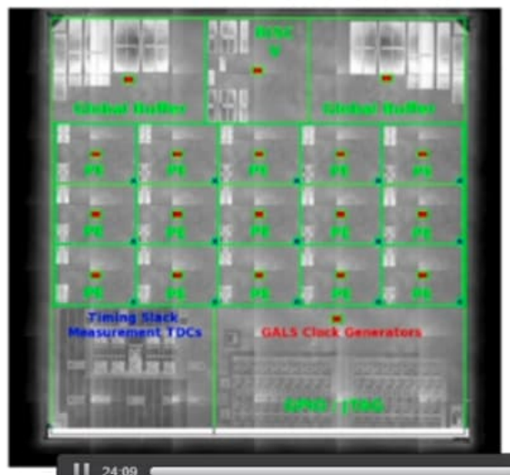
PERFORMANCE VERIFICATION

- Sim-accurate SystemC models for Latency-Insensitive Channels
- Up to 30x speedup vs. RTL
- Less than 2.6% error in cycle count

17 NVIDIA

RC17 SOC PHYSICAL DESIGN

87M Transistor SoC in TSMC 16nm FinFET



	RC17 Stats
Die Size	4 mm^2
Partitions	19 (5 unique)
Frequency range	510 MHz - 1.96 GHz
Voltage range	0.55-1.2 Volts
Performance (16b GMACS)	61.2-235.2
Max GMACS/W	192.1
Programmability	ML workloads (NN inference, K-means)

NVIDIA

Complexity/Risk in Modern Designs has Shifted…

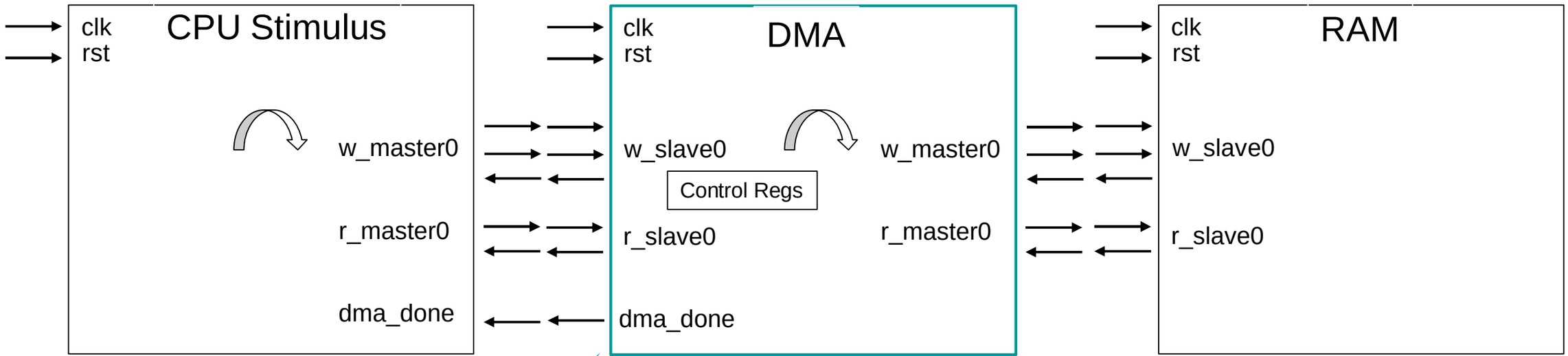
- As an example, performance of ML / Vision chips is often in terms of trillions of MACs per second
- But, design and verification of MACs is not the hard part
- Hard part is often managing the movement of data in the chip across all scenarios
- Today's HW designs often process huge sets of data, with large intermediate results.
  - Machine Learning, Computer Vision, 5G Wireless
- The design of the memory/interconnect architecture and the management of data movement in the system often has more impact on power/performance than the design of the computation units themselves.

MatchLib + SystemC HLS Addresses Complexity / Risk in Modern Designs

- Evaluating and verifying memory/interconnect architecture at RTL level is often not feasible:
  - Too late in design cycle.
  - Too much work to evaluate multiple candidate architectures.

- The most difficult/costly HW (& HW/SW) problems are found during system integration.
  - If integration first occurs in RTL, it is very late and problems are very costly.
  - MatchLib + SystemC HLS lets integration occur early when fixing problems is much cheaper.

Simple Example: AXI4 DMA using MatchLib (example 08*)



The CPU Stimulus programs DMA control registers

The design to be synthesized with HLS is a DMA

```
/**
 *  * \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);
```

The DMA reads and writes to the RAM

= top level of design

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 }
```

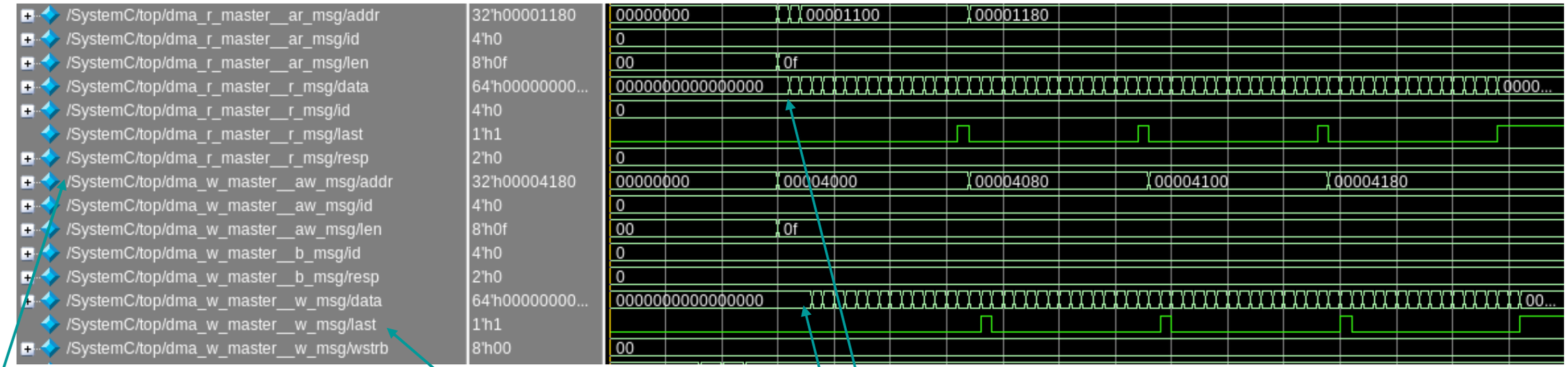
The only clock/wait is for reset state

This IO is in parallel

Main compute loop gets pipelined in HLS

This IO is in parallel

AXI4 DMA Waveforms Before HLS (SystemC simulation)



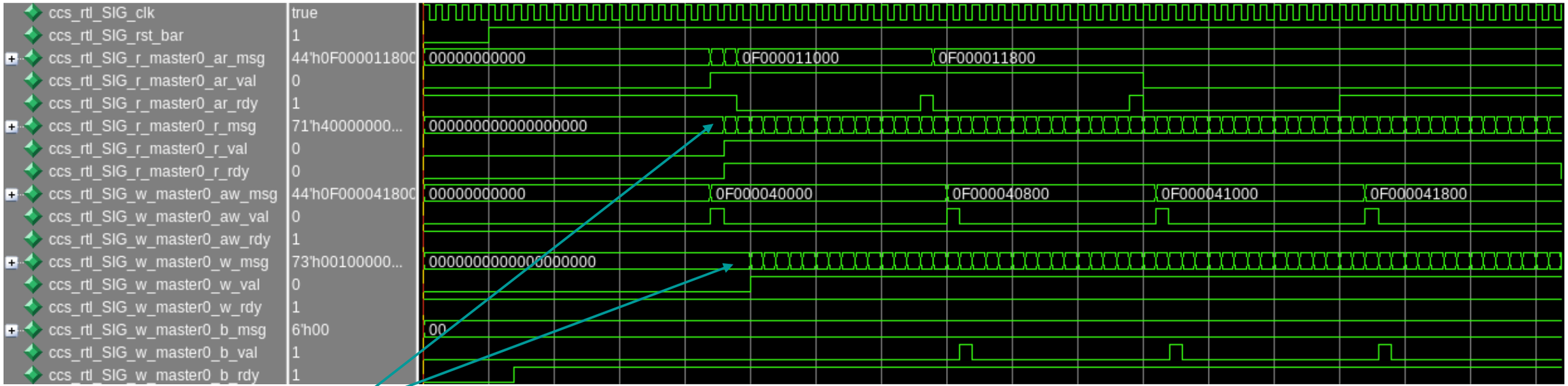
Automatic AXI4 last bit generation

Automatic AXI4 burst address segmentation

Read and write burst streams are concurrent.

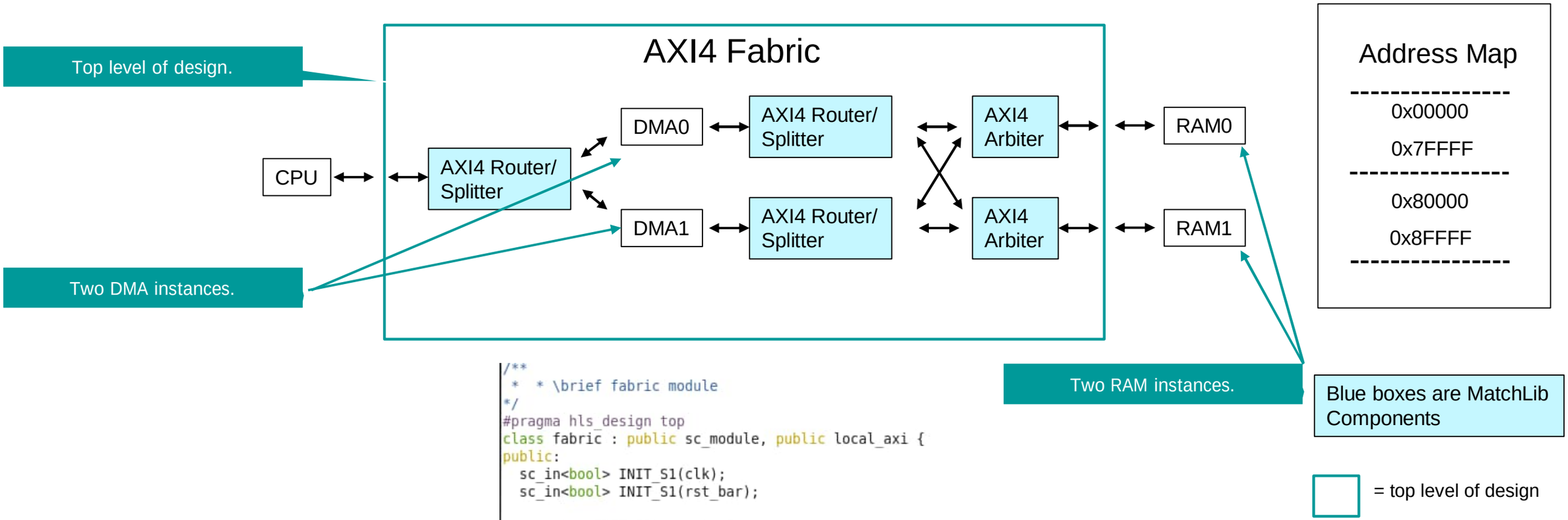
Read/write bus utilization is 100%

AXI4 DMA Waveforms After HLS (Verilog Sim)



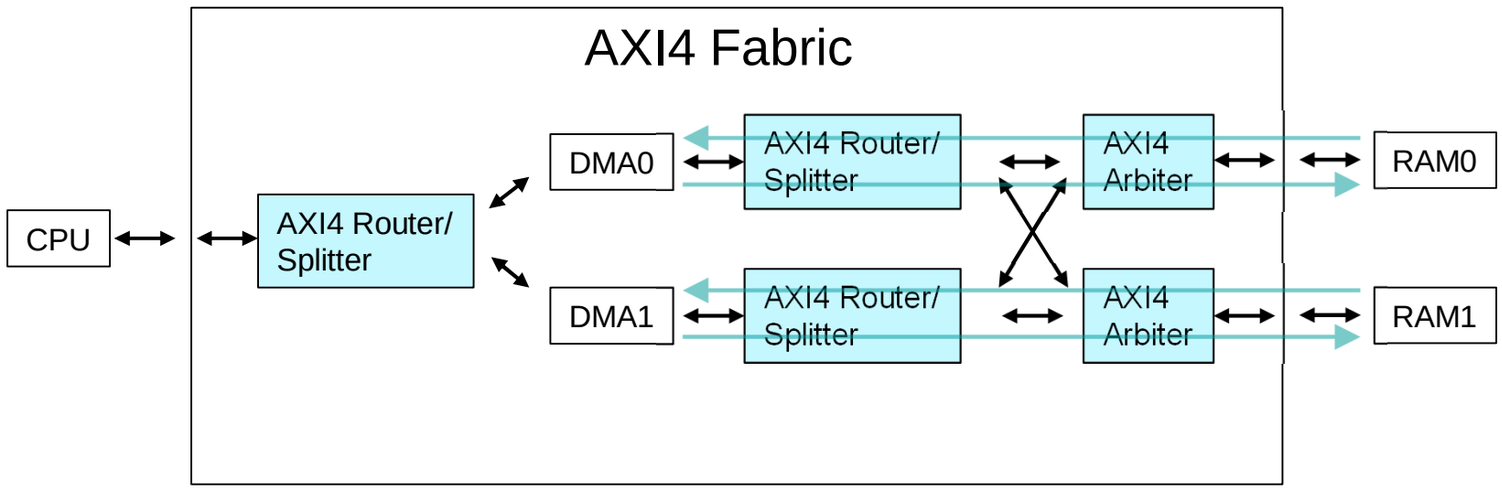
RTL waveforms are almost same as SystemC waveforms. Throughput is the same.

Larger Example: AXI4 Bus Fabric using MatchLib (example 09*)




```
/**
 *  * \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);
```

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

AXI4 Bus Fabric Test #0 simulation logs

BEFORE HLS (SystemC simulation)

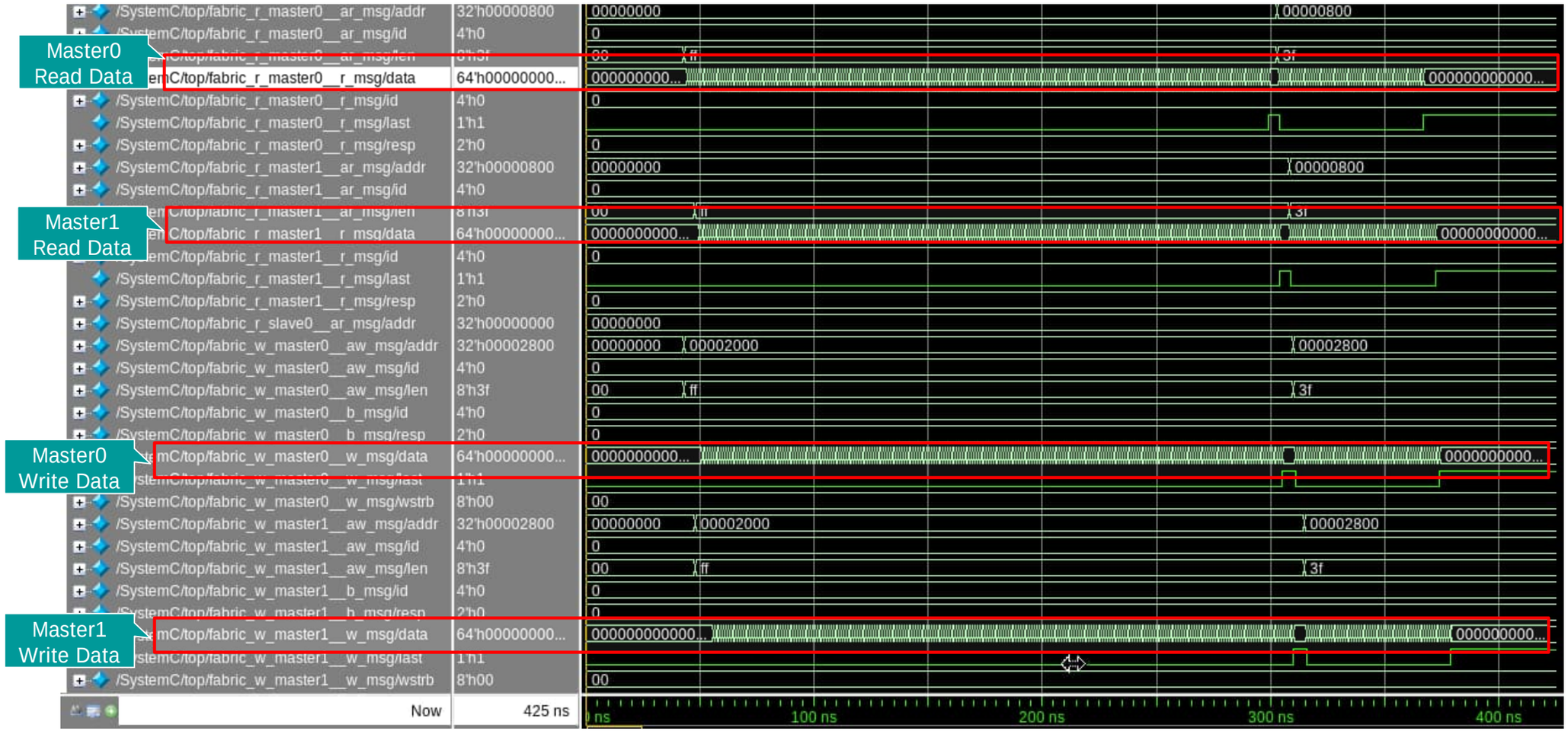
```
0 s top Stimulus started
6 ns top Running FABRIC_TEST # : 0
44 ns top.ram0 ram read  addr: 000000000 len: 0ff
44 ns top.ram0 ram write addr: 000002000 len: 0ff
49 ns top.ram1 ram write addr: 000002000 len: 0ff
49 ns top.ram1 ram read  addr: 000000000 len: 0ff
304 ns top.ram0 ram read  addr: 000000800 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)

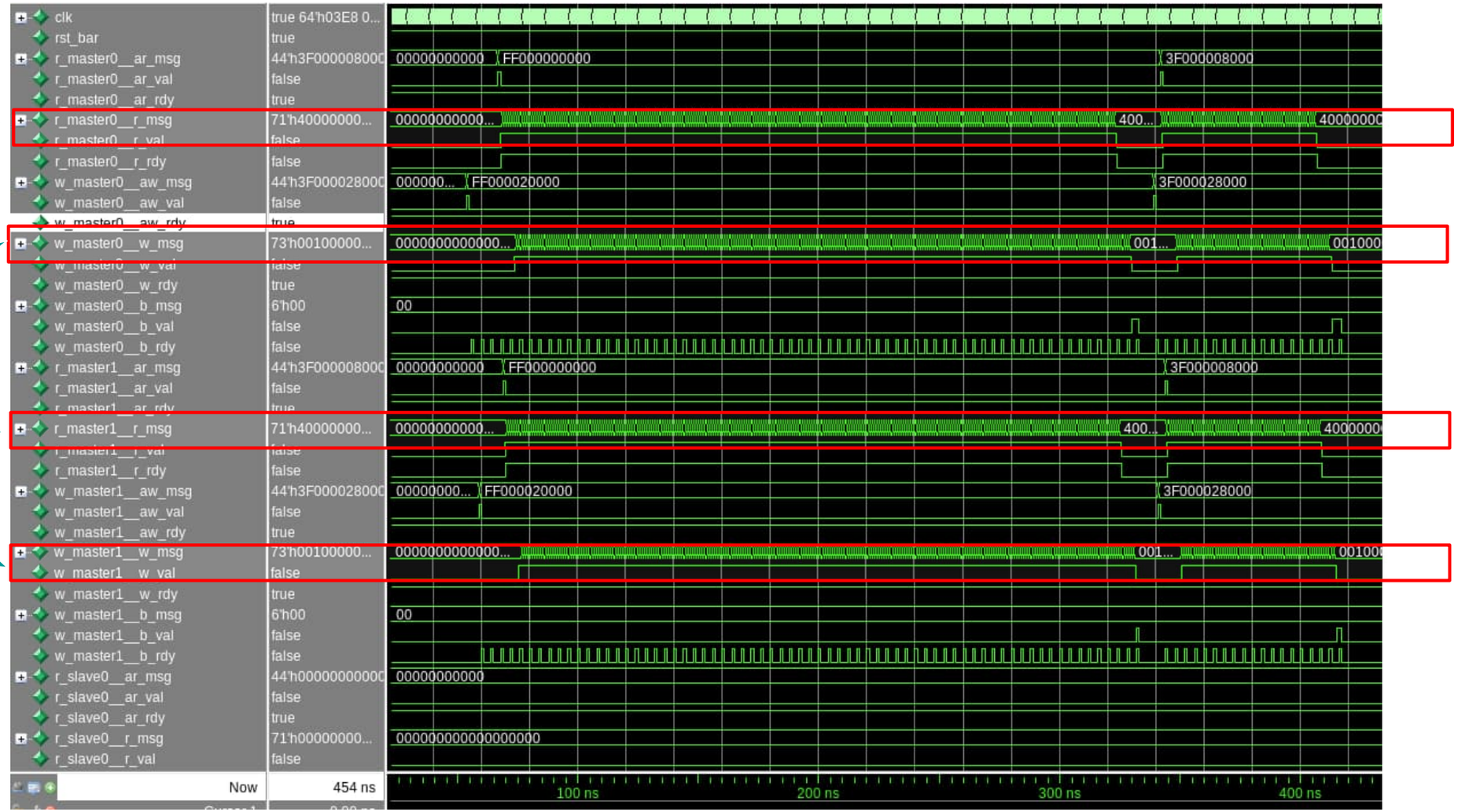
```
# 0 s top Stimulus started
# 6 ns top Running FABRIC_TEST # : 0
# 55 ns top/ram0 ram write addr: 000002000 len: 0ff
# 60 ns top/ram1 ram write addr: 000002000 len: 0ff
# 68 ns top/ram0 ram read  addr: 000000000 len: 0ff
# 70 ns top/ram1 ram read  addr: 000000000 len: 0ff
# 340 ns top/ram0 ram write 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 HLS–Test #0 (SystemC)

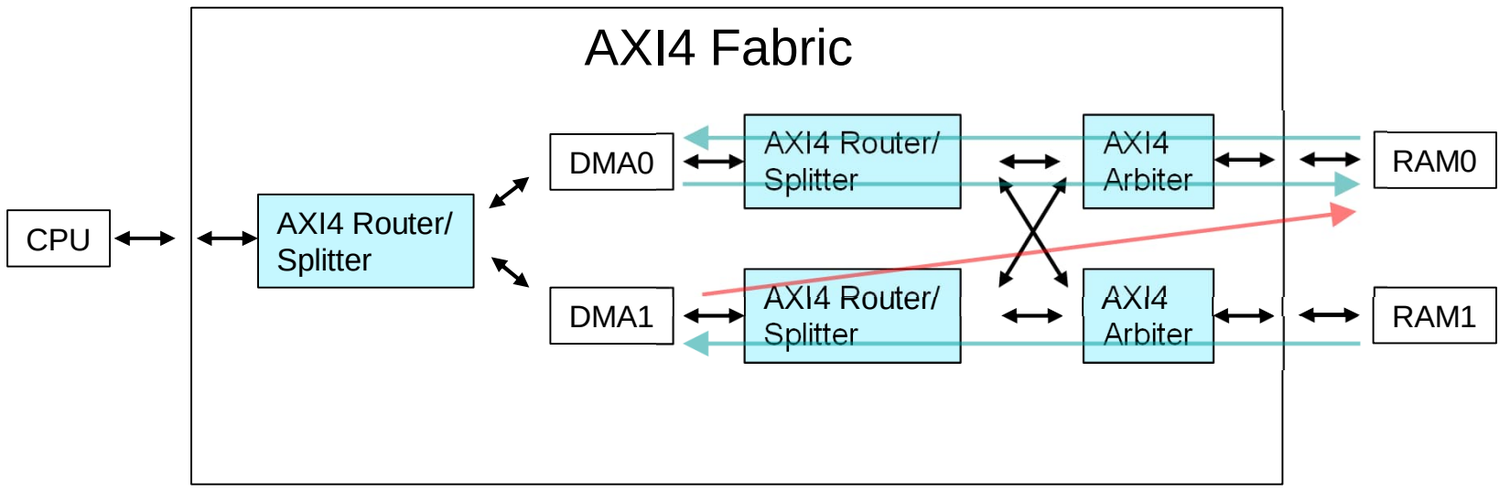


AXI4 Fabric Waveforms After HLS – Test #0 (Verilog)



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)

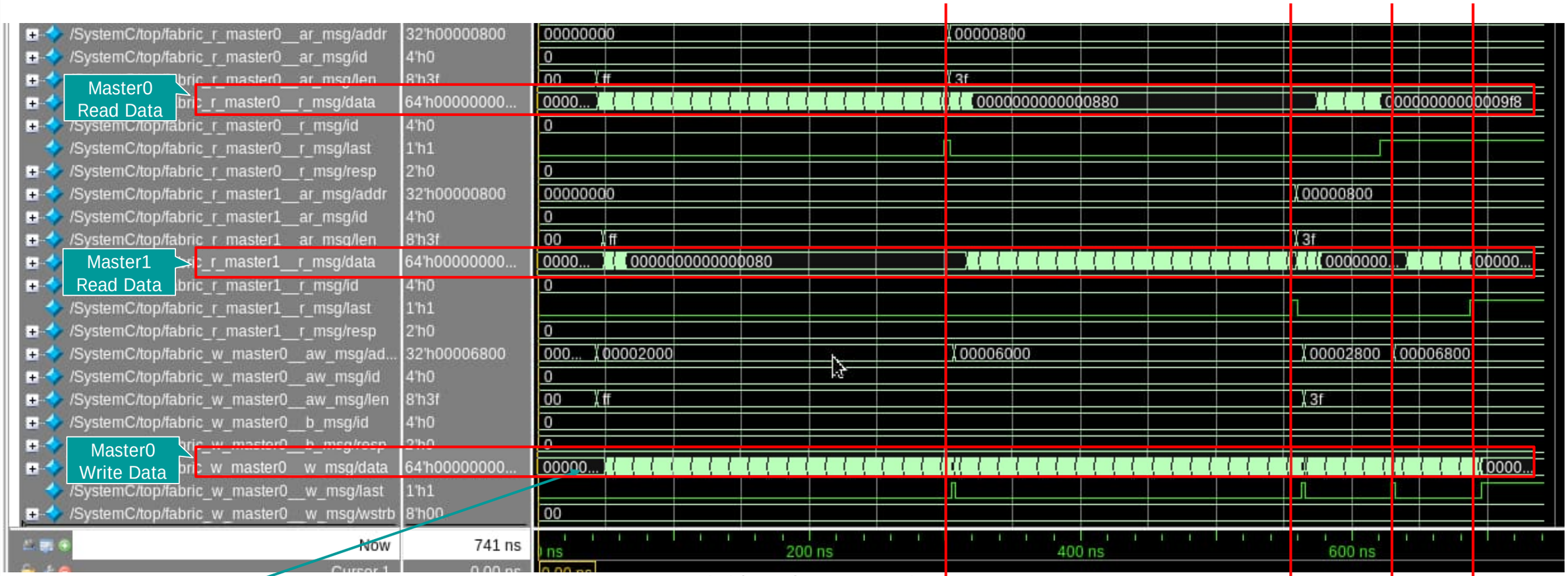
```
0 s top Stimulus started
6 ns top Running FABRIC_TEST # : 1
44 ns top.ram0 ram read  addr: 000000000 len: 0ff
44 ns top.ram0 ram write addr: 000002000 len: 0ff
49 ns top.ram1 ram read  addr: 000000000 len: 0ff
304 ns top.ram0 ram read  addr: 000000800 len: 03f
308 ns top.ram0 ram write addr: 000006000 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 HLS –Test#1 (SystemC)



256 beats from r_master0

256 beats from r_master1

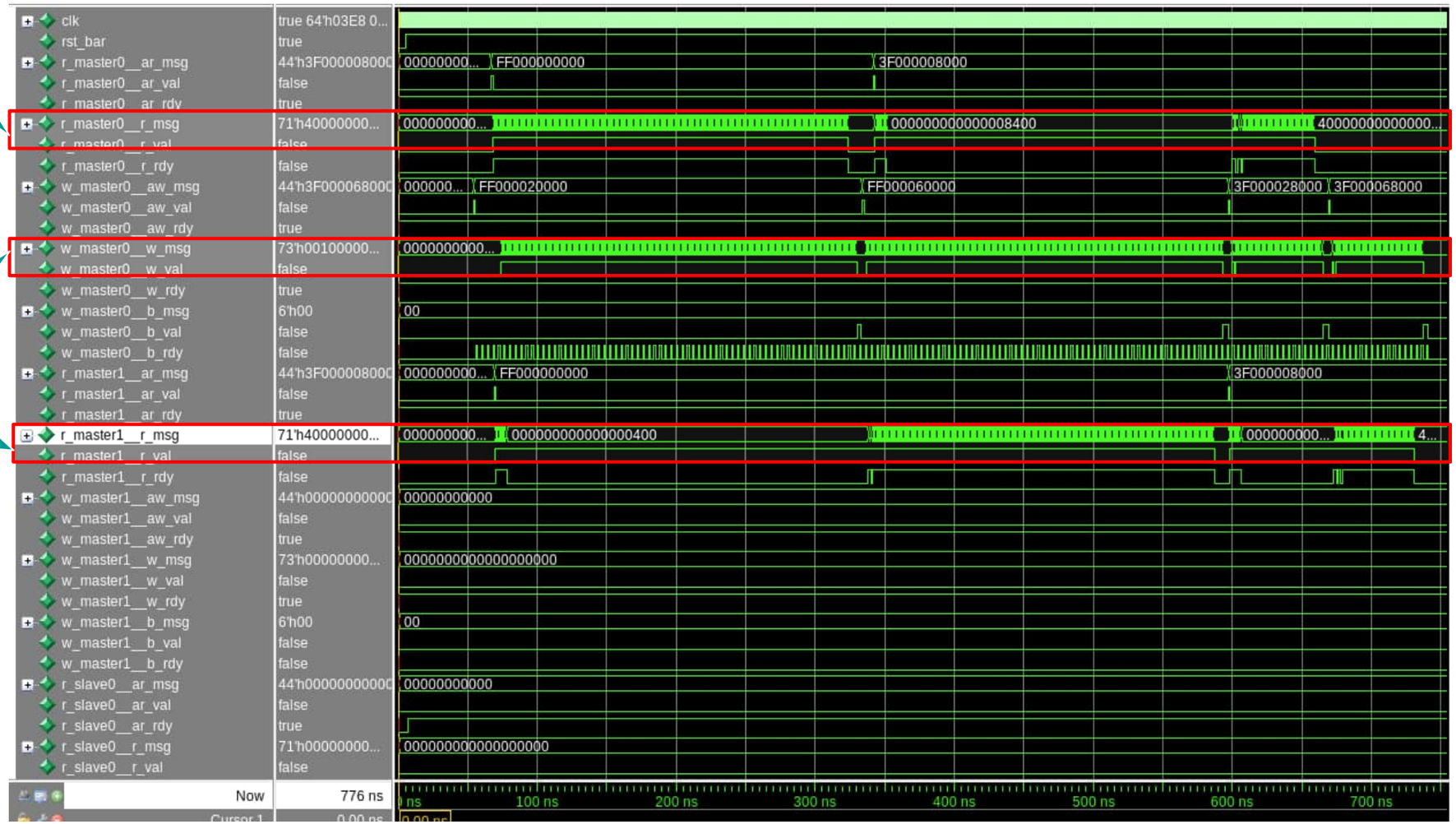
64 beats from r_master0

64 beats from r_master1

w_master0 fully utilized over 700 ns due to write contention

r_master0 and r_master1 underutilized due to write contention

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

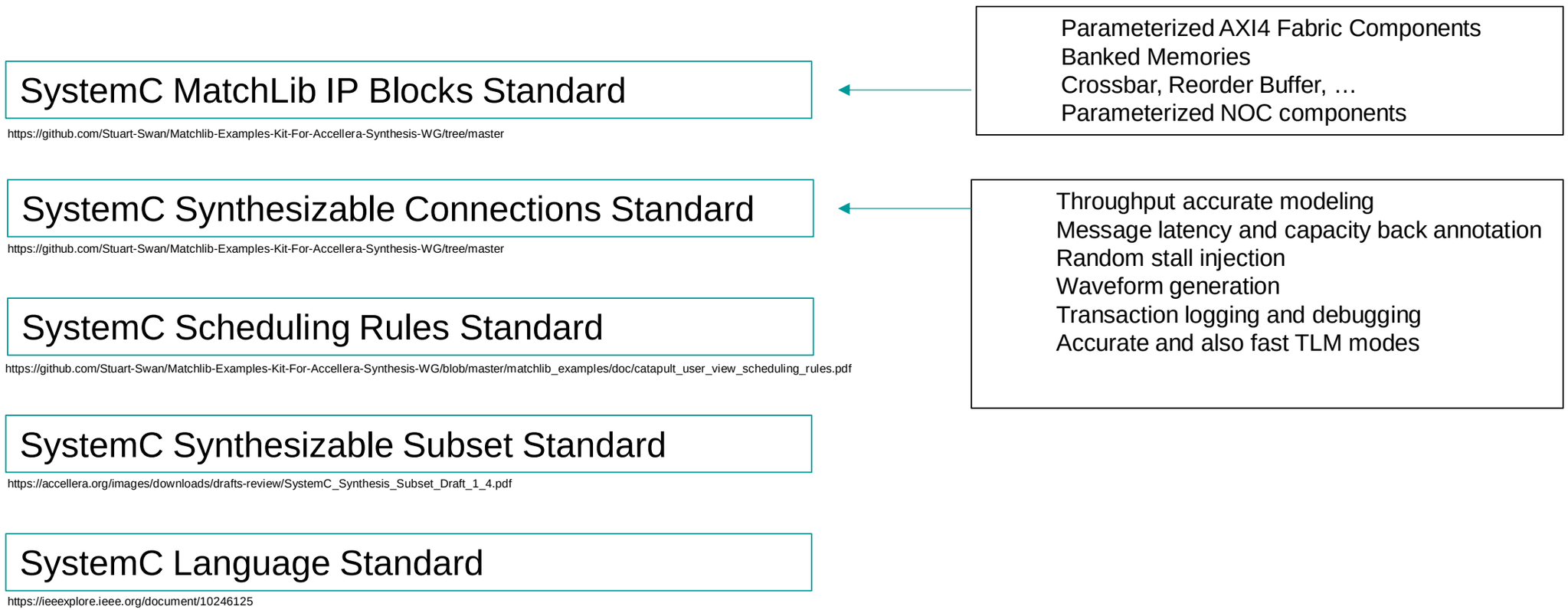


Throughput in RTL Matches SystemC

Recap: MatchLib and HLS Enable Modern D/V Flow

- Designer focuses on chip architecture, functionality, and throughput analysis/verification.
  - HLS adds pipelining, optimizes microarchitecture, provides fully automated flow to placed gates.
- Focus of verification effort moves to C++/SystemC level, enabling much greater efficiency.
- Additional introductory material on MatchLib is publicly available on web:
  - https://www.youtube.com/watch?v=n8_G-CaSSPU
  - https://forums.accellera.org/files/file/127-matchlib-examples-kit-for-accellera-synthesis-working-group/

Proposed Accellera SystemC HLS Standards Layers



Catapult SystemC Coding Styles

Stuart Swan
Platform Architect
Siemens EDA
18 January 2024

SIEMENS

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

“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   }
```

“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 implicit_fsm=true 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

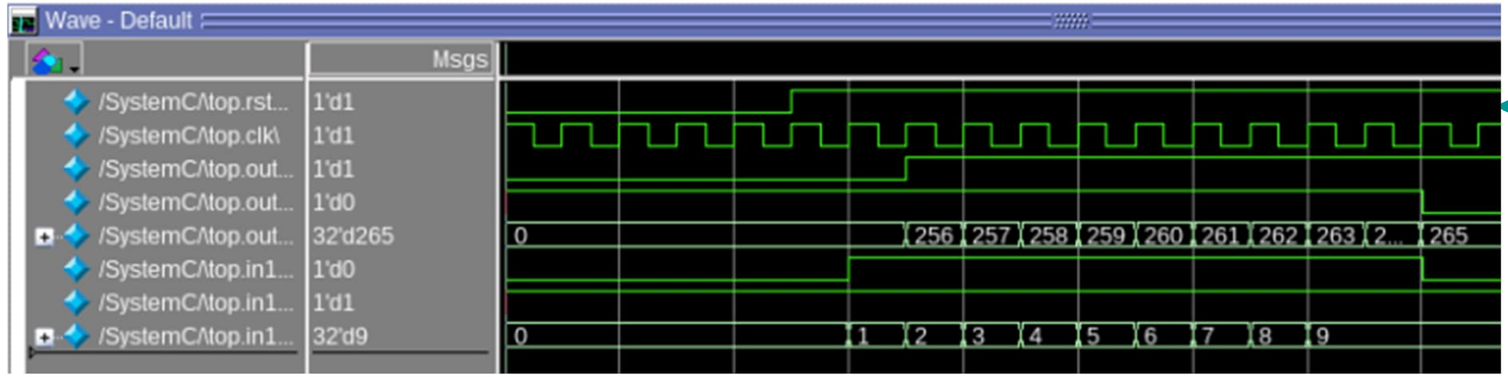
- Advantages:
  - Useful if RTL pin level protocols are “fixed”
  - Useful for integrating with existing Verilog/VHDL RTL designs
  - Sometimes (if II=1) useful for dat/vld protocol if (i.e. no backpressure)
    - common in time-domain signal processing
    - See example 32*
  - May be a better fit for “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 };
```

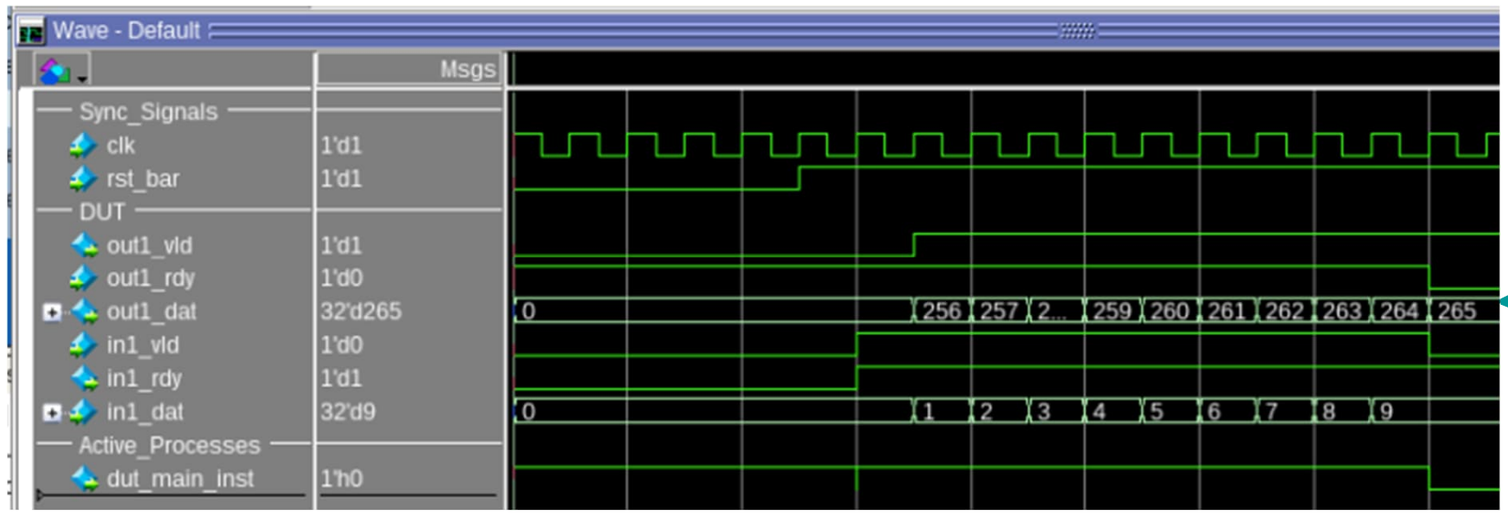
2nd Pop and 1st Push are concurrent

05 Waveforms



Pre-HLS

Question:
What's the throughput?



Post-HLS

SIEMENS

“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
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 };
```

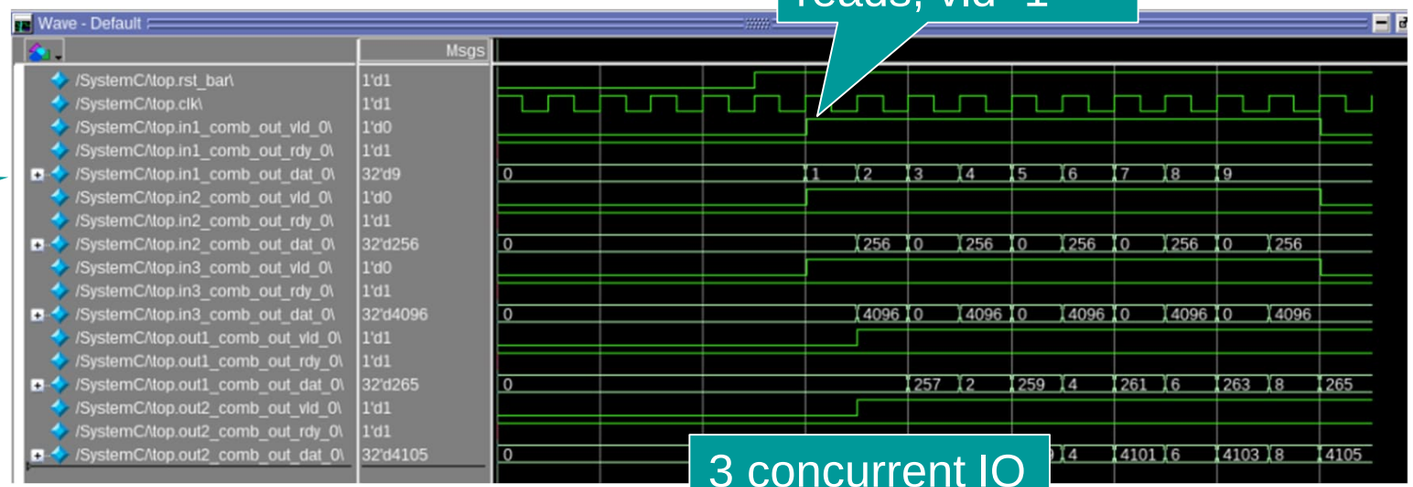
3 concurrent IO reads

2 concurrent IO writes

SIEMENS

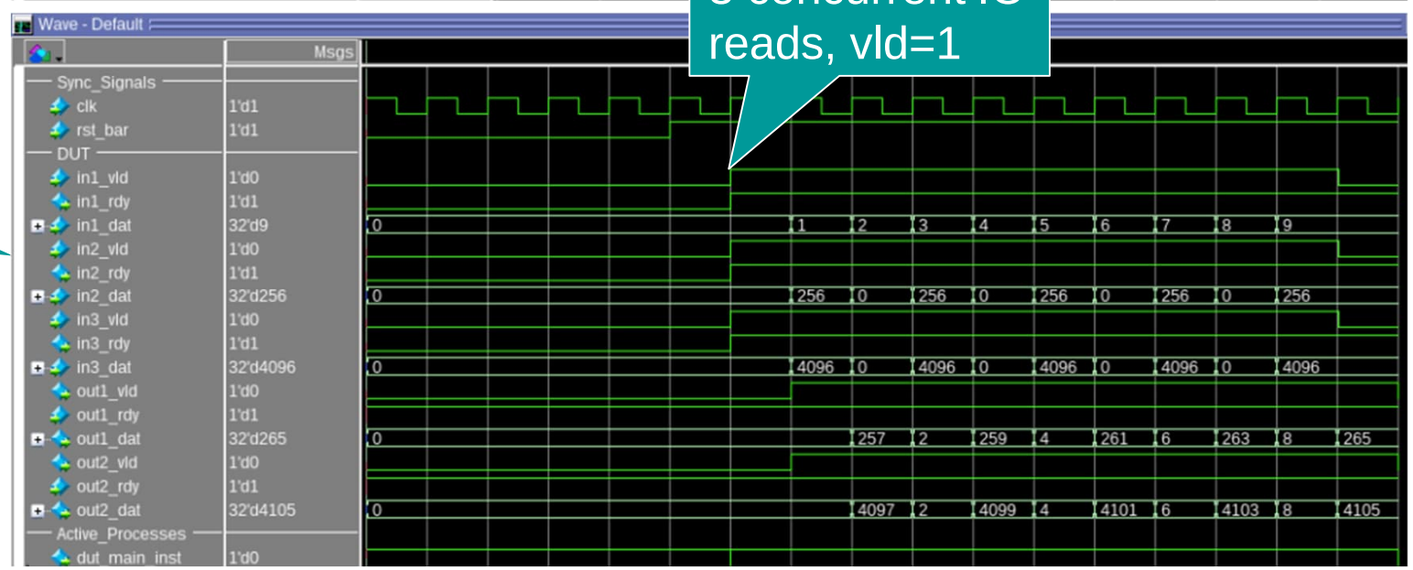
06 Waveforms

Pre-HLS



Question:
What's the throughput?

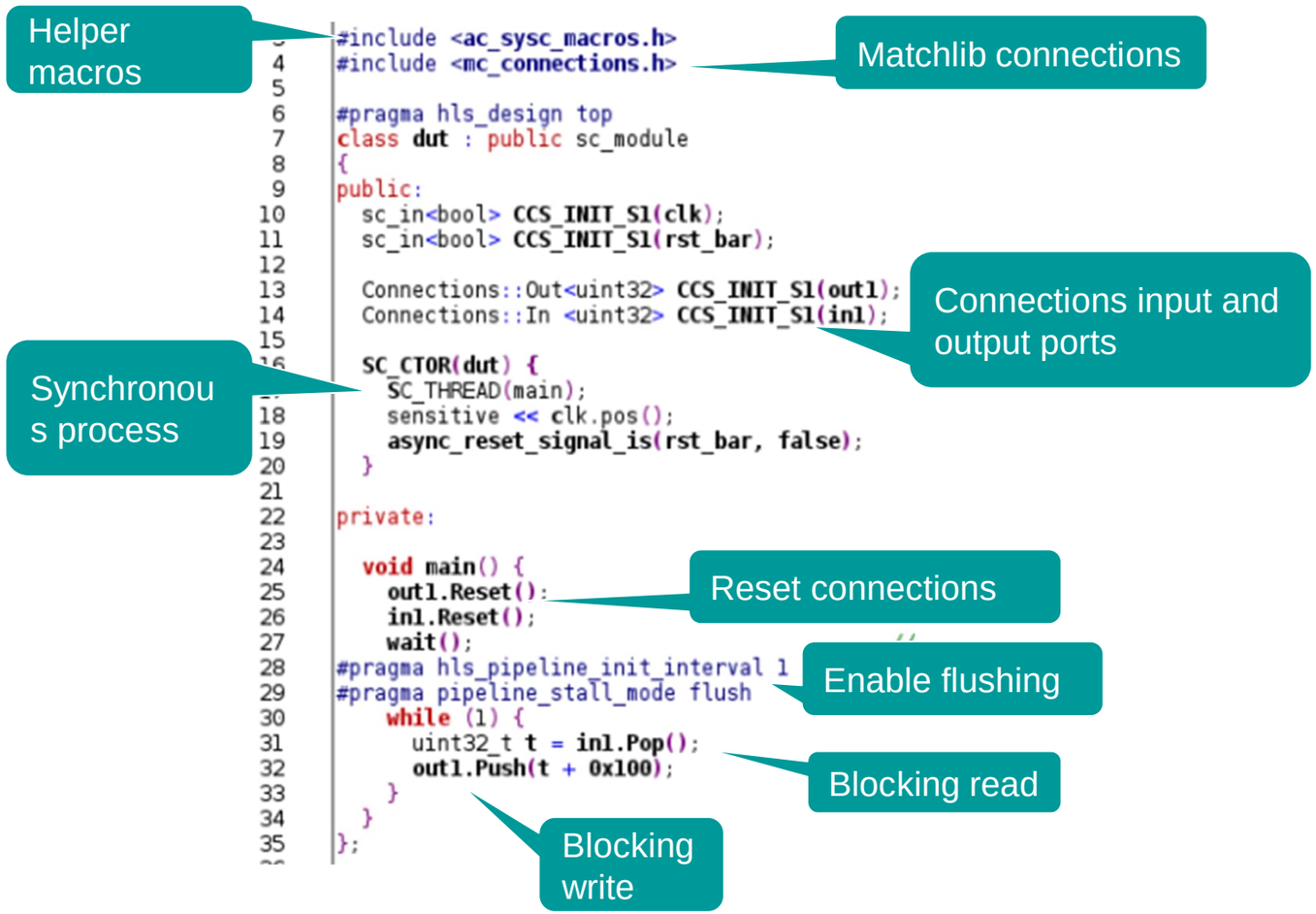
Post-HLS



SIEMENS

Using Matchlib Connections - example 05: Pop/Push

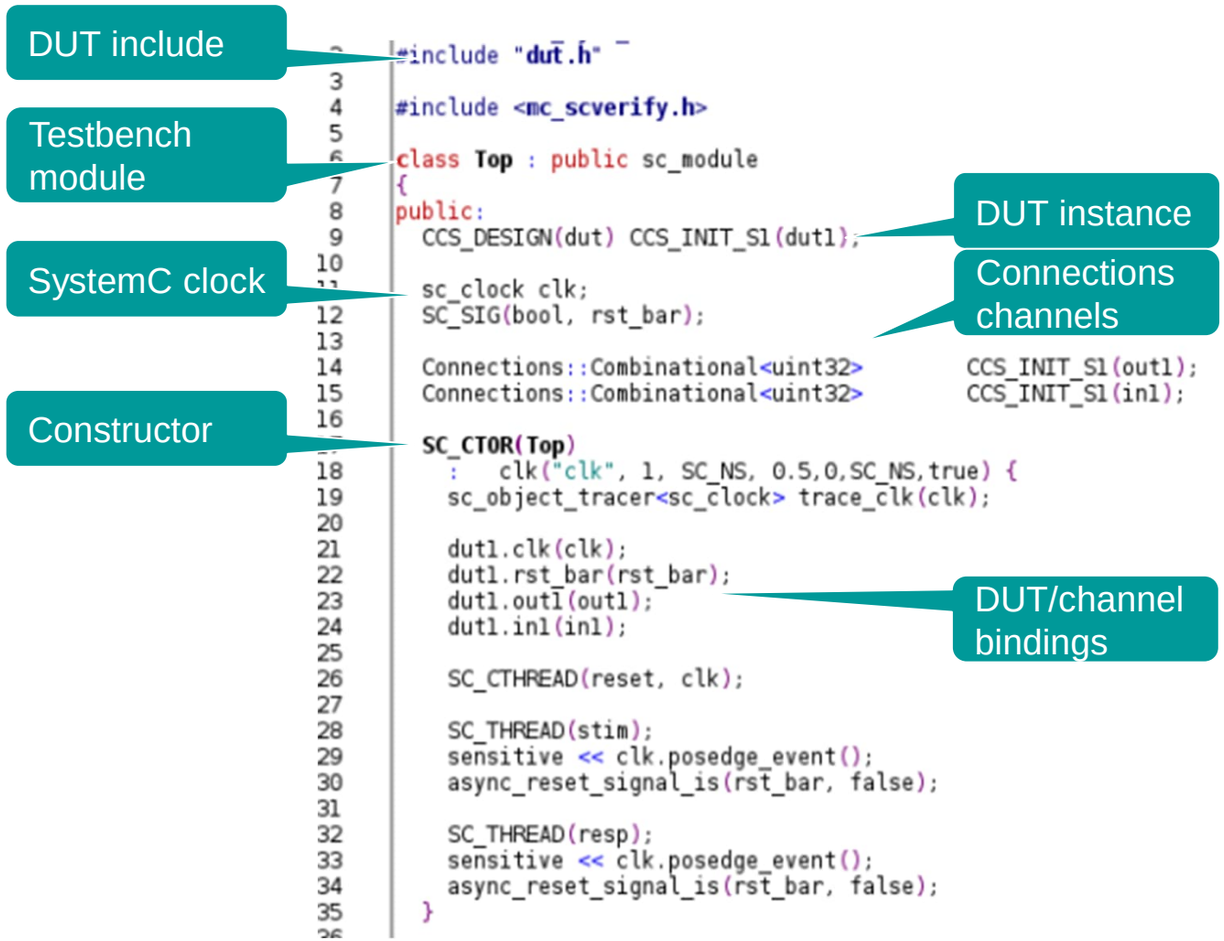
- Include the connections library
  - #include <mc_connections.h>
- Include helper macros
  - CCS_INIT_S1 for port/module constructor naming
- Consider enabling pipeline flushing to allow outputs to flush when there are no more inputs
  - This only affects synthesis
- Add In and Out connections ports for the design
- Connections **must** be reset in the reset state of the thread
  - Call *.Reset(); method
  - Error generated in pre-hls simulation if reset is not set



SIEMENS

Writing SystemC Testbenches for Matchlib Connections - example 05: Pop/Push

- Include the DUT *.h files
- Include Catapult verification macros (Covered later)
- Declare a testbench module
- Instantiate the DUT in the testbench module
- Declare the systemC clock
- Declare reset signals
- Declare connections channels for connecting to the DUT
- Declare a class constructor for the testbench module
  - Bind the connections simulation clock to systemC clock
  - Bind the connections channels to the DUT
  - Declare threads for stimulus, response, and resets



SIEMENS

Writing SystemC Testbenches for Matchlib Connections - example 05: Pop/Push

- Code thread implementations to Push/Pop data to/from the DUT using Matchlib channels
  - Use separate threads for stimulus and response
  - May need multiple stimulus threads or FIFOs to buffer data
- Use CCS_LOG macro to log systemC transactions to the std::cout
  - #include<ac_sysc_macros.h>
  - Logs data with systemC time stamp
- Declare an sc_main
  - Set up a trace file for capturing systemC waveforms
  - Declare an instance of the testbench module
  - Start the simulation

```
37  void stim() {
38    CCS_LOG("Stimulus started");
39    in1.ResetWrite();
40    wait();
41
42    for (int i = 0; i < 10; i++) {
43      in1.Push(i);
44    }
45
46    sc_stop();
47    wait();
48  }
49
50  void resp() {
51    out1.ResetRead();
52    wait();
53
54    while (1) {
55      CCS_LOG("TB resp sees: " << std::hex << out1.Pop());
56    }
57  }
58
59  void reset() {
60    rst_bar.write(0);
61    wait(5);
62    rst_bar.write(1);
63    wait();
64  }
65};
66
67int sc_main(int argc, char **argv)
68{
69  sc_trace_file *trace_file_ptr = sc_trace_static::setup_trace_file("trace");
70
71  Top top("top");
72  trace_hierarchy(&top, trace_file_ptr);
73  sc_start();
74  return 0;
75}
```

Push data into channel connected to the DUT

Log data

Pop data from channel connected to the DUT

SystemC main

Instantiate the test bench

Start the simulation

Create trace file

SIEMENS

“Matchlib SystemC” Coding Style Rules

- Clock and reset ports use sc_in<bool>
- Code up to first wait() in process models the reset state.
  - Reset() methods on all ports should be called (both In and Out ports).
- 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 (covered later)
  - Non-synthesizable (#ifndef __SYNTHESIS__) for performance modeling of inferred memories and rolled loops
- Use default Catapult scheduling modes only
- Only use coding style documented in Siemens EDA Matchlib examples and Labs (!)

```
3   #include <ac_sysc_macros.h>
4   #include <mc_connections.h>
5
6   #pragma hls_design top
7   class dut : public sc_module
8   {
9   public:
10    sc_in<bool> CCS_INIT_S1(clk);
11    sc_in<bool> CCS_INIT_S1(rst_bar);
12
13    Connections::Out<uint32> CCS_INIT_S1(out1);
14    Connections::In <uint32> CCS_INIT_S1(in1);
15
16    SC_CTOR(dut) {
17      SC_THREAD(main);
18      sensitive << clk.pos();
19      async_reset_signal_is(rst_bar, false);
20    }
21
22  private:
23
24    void main() {
25      out1.Reset();
26      in1.Reset();
27      wait();
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  };
```

sc_in for clock and reset

Reset wait

Blocking read and blocking write, no waits in processing loop

SIEMENS

“Matchlib SystemC” Coding Style Rules (continued)

Only use sc_in, sc_out, sc_signal for data 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 SystemC” Coding Style Rules (continued)

- Recommended to use blocking IO over non-blocking 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

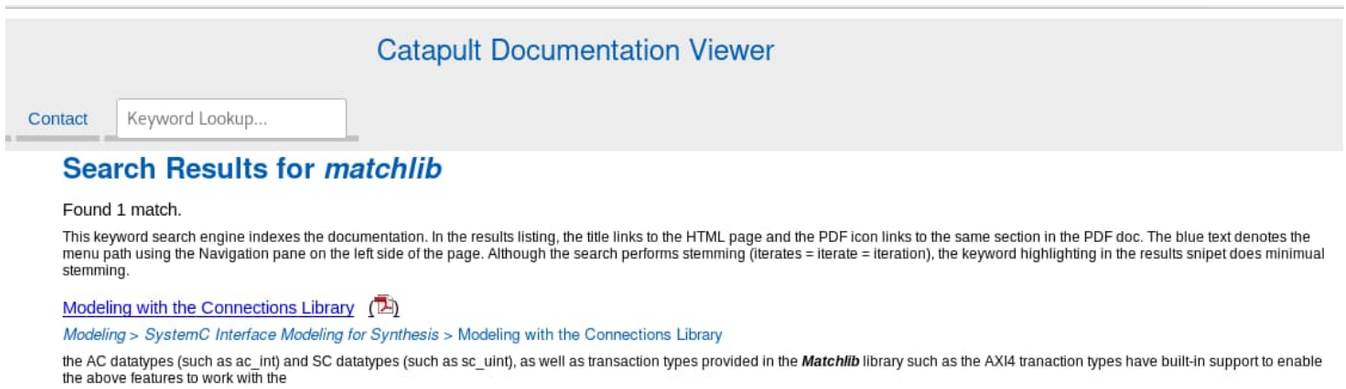
- Non-blocking IO is sometimes needed to model arbitration
  - Introduces timing dependent behavior
  - Can make verification more difficult in some cases (Covered later)

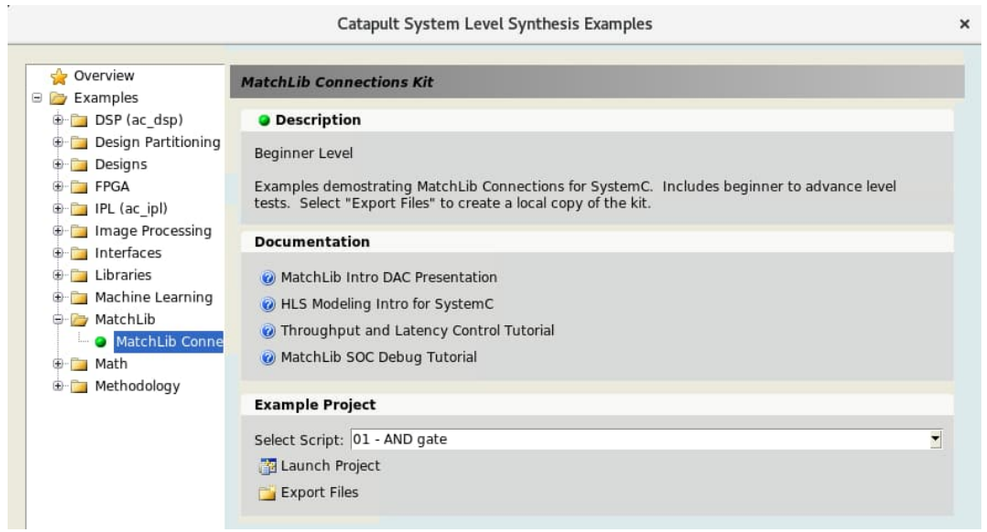
“Matchlib Coding Style” Advantages/Disadvantages

- 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
    - May be helpful to use an IDE such as Vscode, Eclipse
  - 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*

MatchLib Provides the Ability to Synchronize a Process

- Synchronization is often needed
  - To allow external configuration data to be set-up before reading
  - For exchanging data between processes
- MatchLib Sync channels and interfaces can be thought of as a barrier
  - Cannot be pipelined
  - Allows the pipeline to flush before the next sync

Connections::	Methods
SyncIn	SyncPop(), SyncPopNB(), Reset()
SyncOut	SyncPush(), SyncPushNB(), Reset()
SyncChannel	SyncPush(), SyncPop(),SyncPushNB(),SyncPopNB(), ResetWrite(), ResetRead()

Using Connections::SyncIn for Configuration Data

- Configuration data is written before SyncIn is driven

```
class mult_add : public sc_module {
 public:
  sc_in<bool> CCS_INIT_S1(clk);
  sc_in<bool> CCS_INIT_S1(rstn);

  Connections::In <ac_int<16,false>> CCS_INIT_S1(a);
  Connections::In <ac_int<16,false>> CCS_INIT_S1(b);
  Connections::SyncIn                CCS_INIT_S1(sync);
  #pragma hls_direct_input
  sc_in<ac_int<6,false>>             CCS_INIT_S1(num_madds);
  Connections::Out<ac_int<38,false>> CCS_INIT_S1(result);

  SC_CTOR(mult_add) {
    SC_THREAD(main);
    sensitive << clk.pos();
    async_reset_signal_is(rstn, false);
  }

 private:
```

SyncIn I/F

Configuration I/F

Reset SyncIn I/F

```
private:
 void main() {
   a.Reset();
   b.Reset();
   sync.Reset();
   result.Reset();
   wait();
   while (1) {
     ac_int<38,false> acc = 0;
#pragma hls_direct_input_sync all
     sync.SyncPop();
     #pragma hls_pipeline_init_interval 1
     #pragma pipeline_stall_mode flush
     for(int i=0; i<32; i++){
       ac_int<16> ai = a.Pop();
       ac_int<16,false> bi = b.Pop();
       acc += ai*bi;
       if(i==num_madds.read()-1)
         break;
     }
     result.Push(acc);
   }
 }
};
```

Configuration data is set-up and available before sync

Does Siemens support everything in Matchlib?

- Siemens supports all constructs documented in Matchlib Reference Manual:
  - https://github.com/Stuart-Swan/Matchlib-Examples-Kit-For-Accellera-Synthesis-WG/blob/master/matchlib_examples/doc/matchlib_reference_manual.pdf
- Seimens supports all constructs used in Matchlib toolkit examples included in Catapult.

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

Automatic Generation of Transaction Field Methods

Stuart Swan
Platform Architect
Siemens EDA
1 December 2022

SIEMENS

Introduction

- Matchlib and SystemC require that user-defined transactions implement several utility methods within the class/struct which defines the transaction.
- These methods are:

1. `template <unsigned int Size> void Marshall(Marshaller<Size>& m);`

   - This function is needed for Matchlib connections.

2. `inline friend void sc_trace(sc_trace_file *tf, const this_type &v, const std::string &NAME );`

   - This function is needed for SystemC standard waveform tracing support.

3. `inline friend std::ostream &operator<<(ostream &os, const this_type &rhs);`

   - This function is needed for SystemC standard transaction streaming/printing support.

4. `static const unsigned int width = ... ;`

   - This constant is needed for Matchlib connections.

5. `bool operator==(const this_type & rhs) const ;`

   - This function is needed for transactions that are used with SystemC sc_signal<T>

Example of Manual Coding of Field methods (example 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

Problems with Manual Coding of Field Methods

- All field methods need to be updated when transaction fields are updated.
- Tedious and error-prone, especially for transactions with many fields.
- Manual coding of field methods often leads to inconsistent printing and naming behavior across different transactions within a large design.
- To solve these problems, we have provided a feature which automatically generates all these needed utility functions and parameters.

Automatic generation of field methods (example 07*)

```
#include "auto_gen_fields.h"

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

  AUTO_GEN_FIELD_METHODS(engine_t, ( \
      engine \
    , spark_plugs \
  ) )
  //
};
```

Required include file

Open parenthesis here is required

List all the fields, separated by commas

Two closing parenthesis here are required

- For further information see examples 07* and 23* in the Matchlib examples kit.

Matchlib AXI4 Interfaces

Refer to Matchlib example 08*

Stuart Swan
Platform Architect
Siemens EDA
18 January 2024

SIEMENS

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

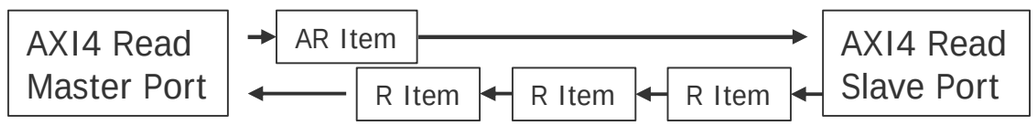
- Message Passing is very common, goes by many names:
  - Kahn Process Networks (KPN)
  - “Latency Insensitive”
- Matchlib Connections is also a message passing framework
  - So is ac_channel
- Message passing is good because:
  - Deterministic
  - Scalable
  - Latency insensitive
  - Can add pipeline 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

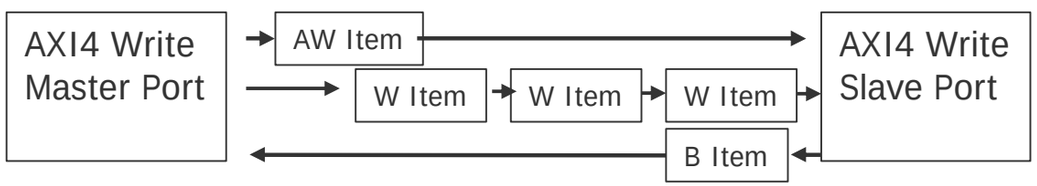
AXI4 Read Channels

- 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

```
 5 #include "stdlib.h"
 6 #include "axi4_segment.h"
 7
 8
 9 typedef axi::axi4_segment<axi::cfg::standard> local_axi;
10
11 /**
12  *  \brief A simple RAM module with 1 axi4 read slave and 1 axi4 write slave
13 */
14
15 class ram : public sc_module, public local_axi {
16 public:
17   sc_in<bool> INIT_S1(clk);
18   sc_in<bool> INIT_S1(rst_bar);
19   r_slave<AUTO_PORT>     INIT_S1(r_slave0);
20   w_slave<AUTO_PORT>     INIT_S1(w_slave0);
21
22   static const int sz = 0x10000; // size in axi_cfg::dataWidth words
23
24   typedef NVUINTW(axi_cfg::dataWidth) arr_t;
25   arr_t* array {0};
26
27   SC_CTOR(ram)
28   {
29     array = new arr_t[sz];
30
31     SC_THREAD(slave_r_process);
32     sensitive << clk.pos();
33     async_reset_signal_is(rst_bar, false);
34
35     SC_THREAD(slave_w_process);
36     sensitive << clk.pos();
37     async_reset_signal_is(rst_bar, false);
38
39     for (int i=0; i < sz; i++)
40      array[i] = i * bytesPerBeat;
41   }
```

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

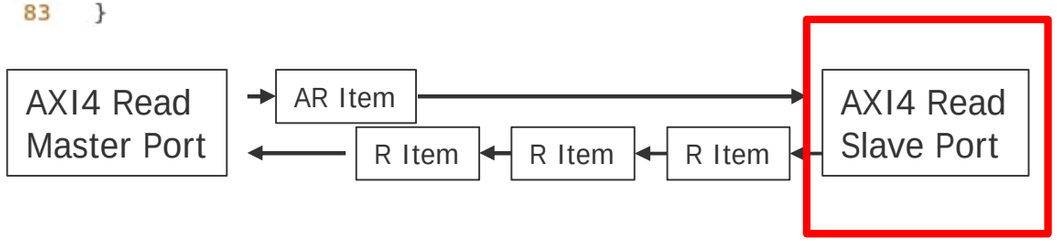
start_multi_read() is a convenience method implemented in the AXI4 read slave port.
Start waiting for any length burst read (“multi”).
Result is returned in “ar”

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”

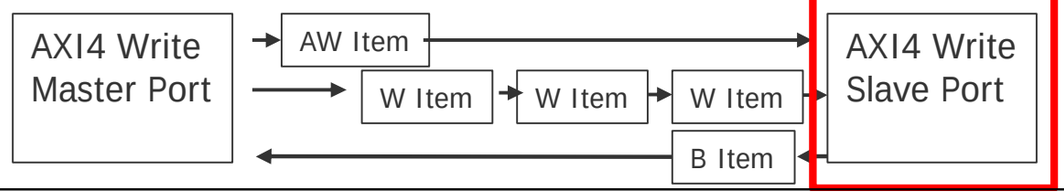
Pop a w item

On invalid address set write response to an error

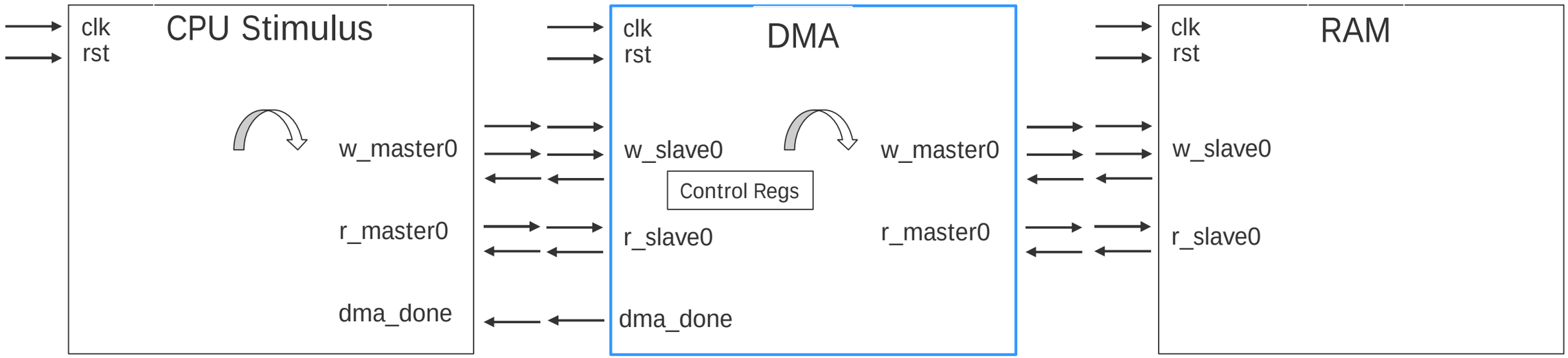
Write the data into the ram array

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

Push write response



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

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 }
```

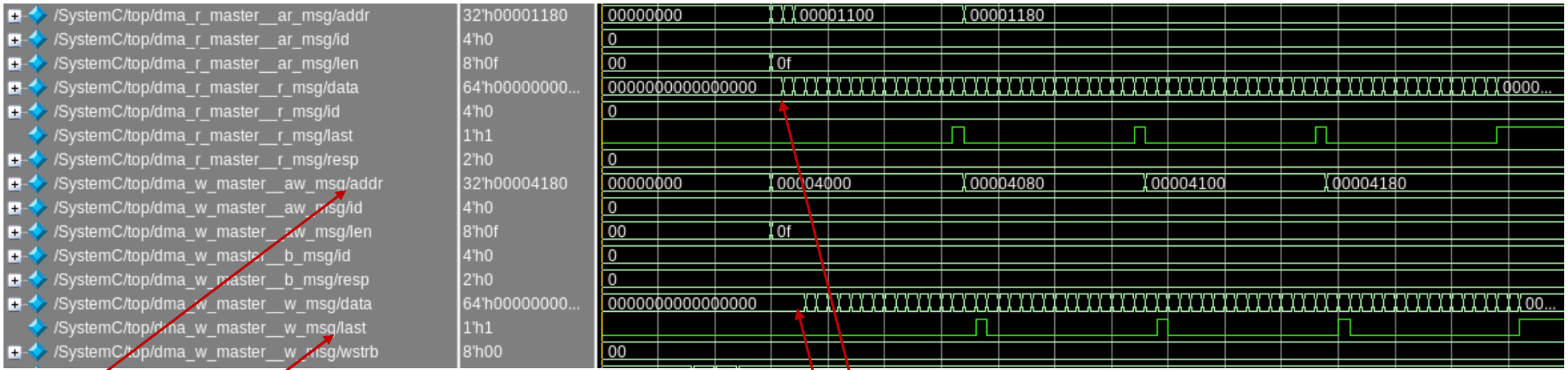
The only clock/wait is for reset state

This IO is in parallel

Main compute loop gets pipelined in HLS

This IO is in parallel

AXI4 DMA Waveforms Before HLS (ex 08* SystemC simulation)



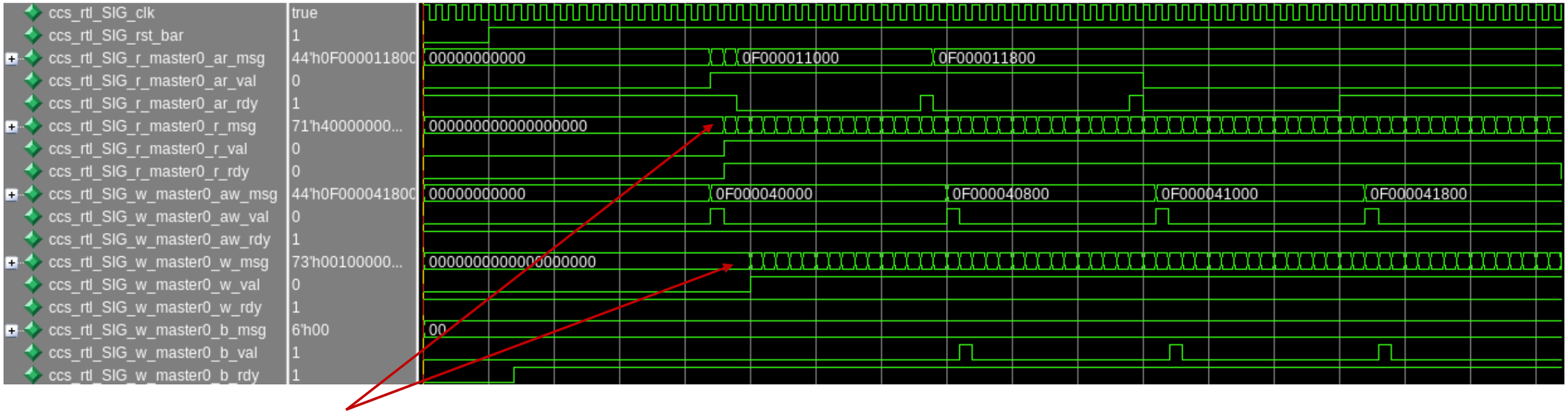
Automatic AXI4 last bit generation

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 (ex 08* 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  }
```

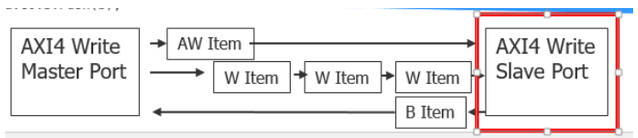
Reset signals that slave process drives

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

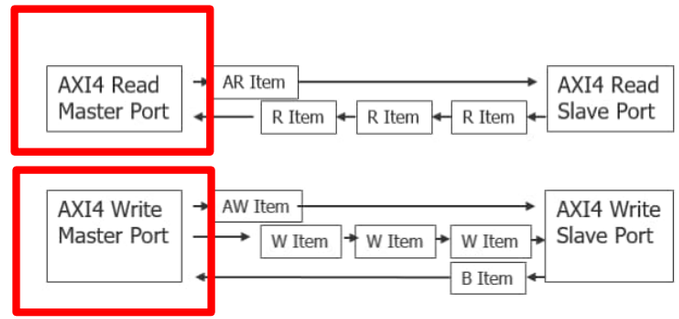
Push out AW and AR addresses to RAM

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

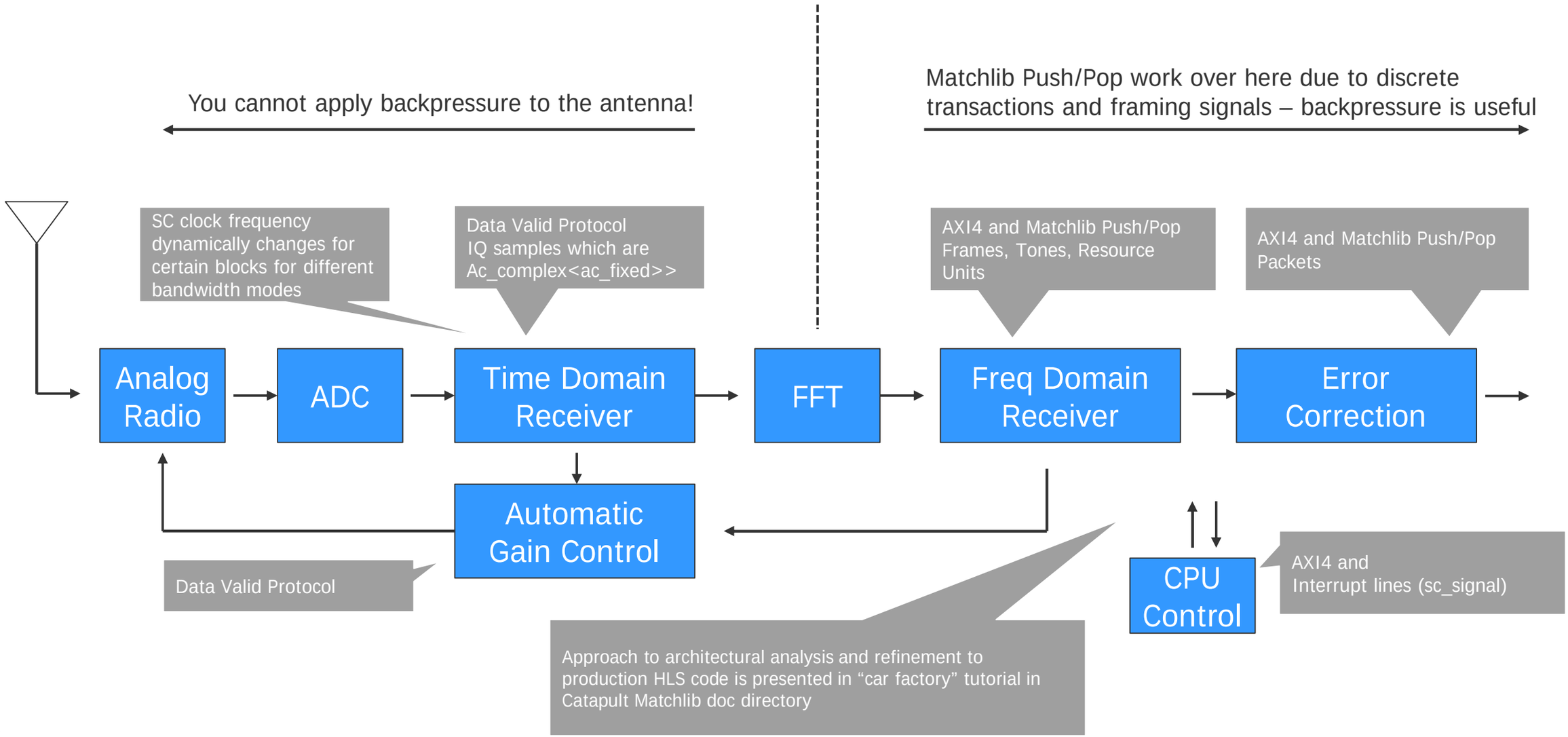
Consume write response and send interrupt to CPU



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

Real World Design Example – 5G Receiver



Matchlib Accuracy

Stuart Swan
Platform Architect
Siemens EDA
23 July 2024

SIEMENS

Matchlib Accuracy

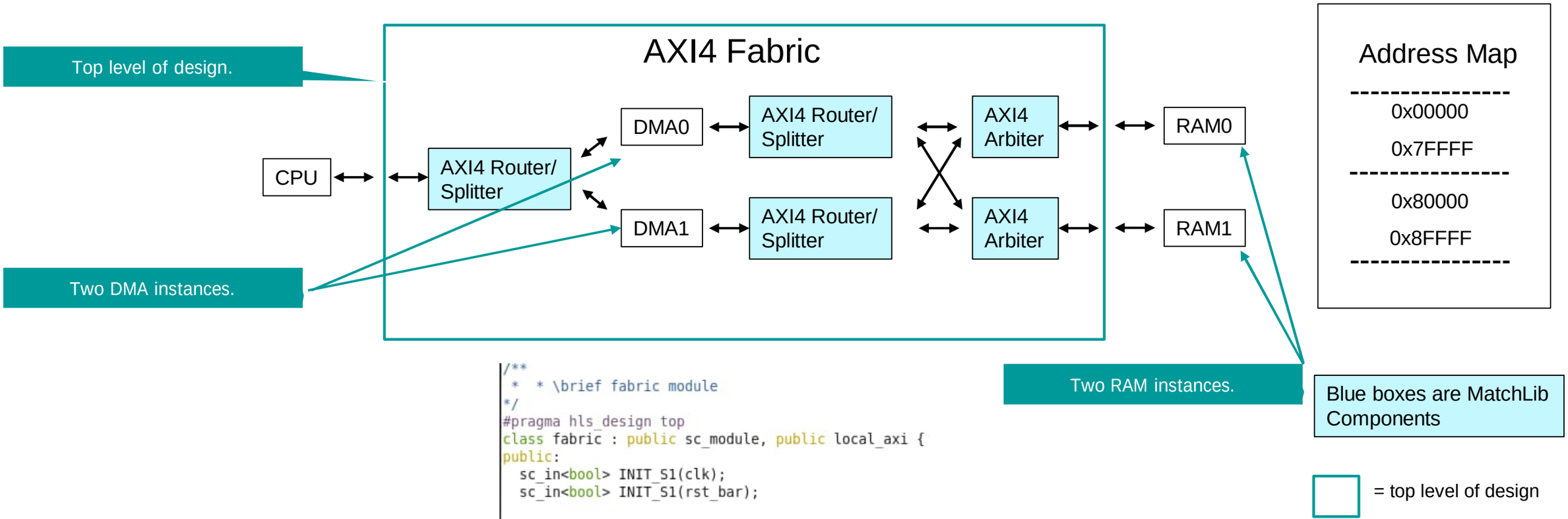
- Goal of Matchlib is to be “throughput accurate” in pre-HLS models, even as you scale up size of systems.
- Goal is NOT to be precisely cycle accurate wrt RTL
- Even with some differences in accuracy wrt RTL, there is great value in being able to model time/performance in the pre-HLS models.
- It is 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.

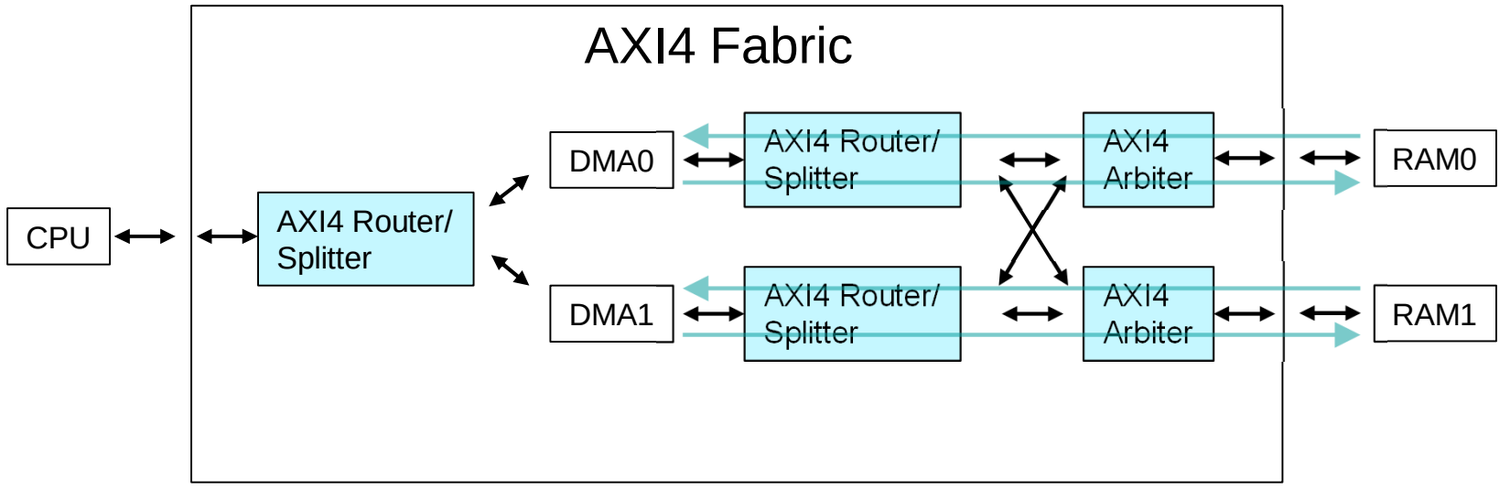
```
12   Connections::Out<uint32> CCS_INIT_S1(out1);
13   Connections::In <uint32> CCS_INIT_S1(in1);
14
15   SC_CTOR(dut) {
16     SC_THREAD(main);
17     sensitive << clk.pos();
18     async_reset_signal_is(rst_bar, false);
19   }
20
21 private:
22
23   void main() {
24     out1.Reset();
25     in1.Reset();
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 (ex 09*)




```
/**
 *  * \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);
```

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

AXI4 Bus Fabric Test #0 simulation logs

BEFORE HLS (SystemC simulation)

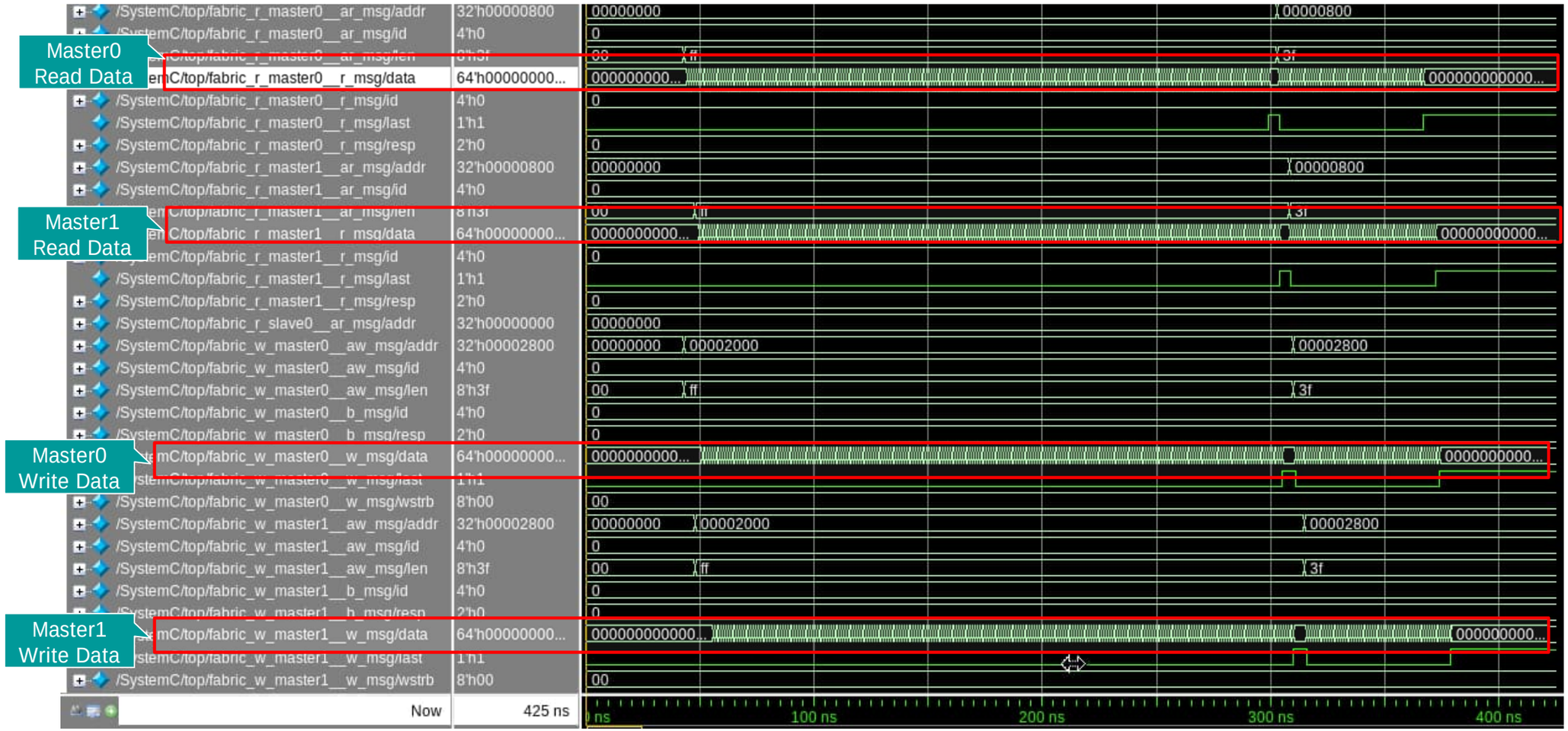
```
0 s top Stimulus started
6 ns top Running FABRIC_TEST # : 0
44 ns top.ram0 ram read  addr: 000000000 len: 0ff
44 ns top.ram0 ram write addr: 000002000 len: 0ff
49 ns top.ram1 ram write addr: 000002000 len: 0ff
49 ns top.ram1 ram read  addr: 000000000 len: 0ff
304 ns top.ram0 ram read  addr: 000000800 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)

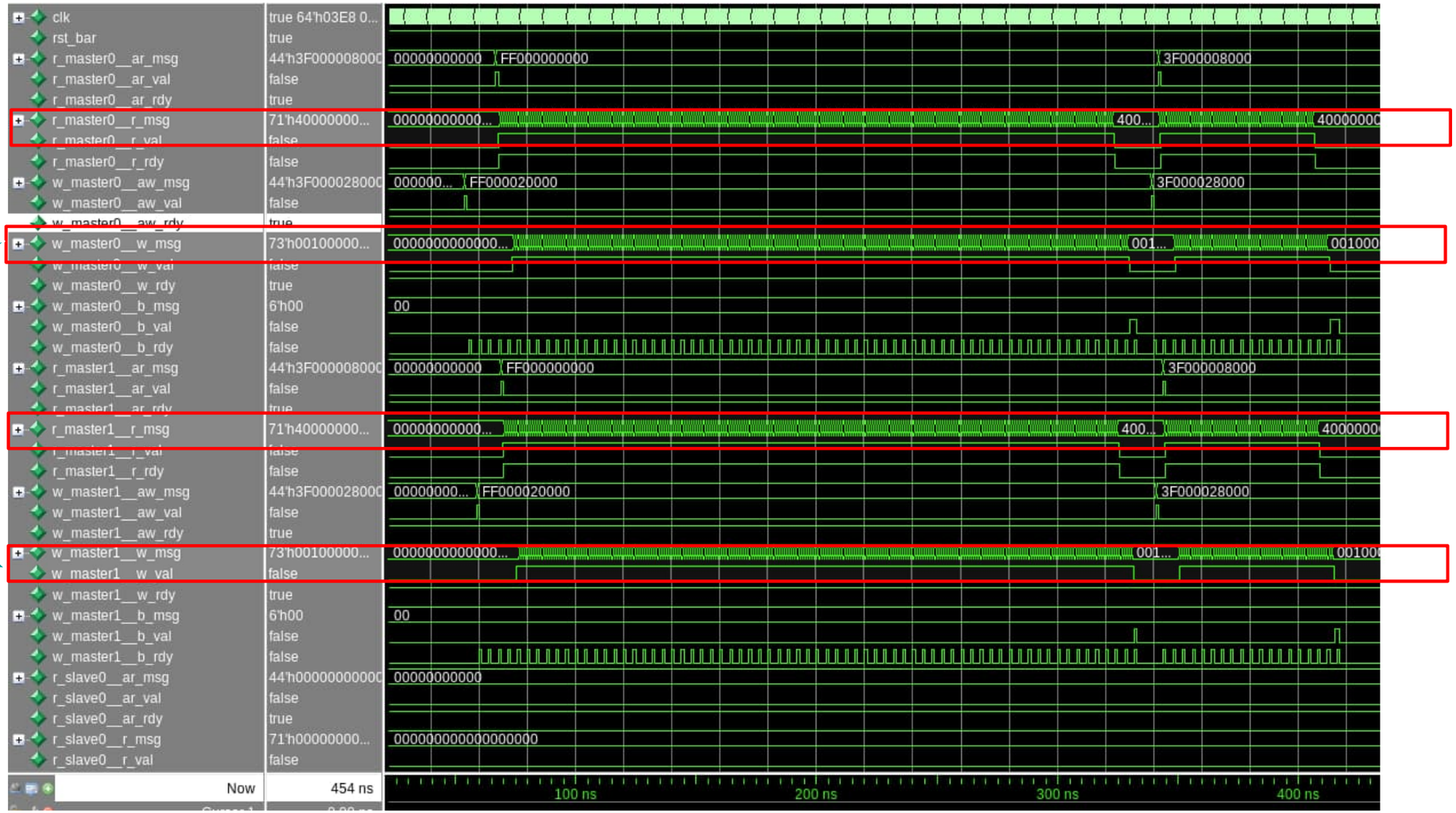
```
# 0 s top Stimulus started
# 6 ns top Running FABRIC_TEST # : 0
# 55 ns top/ram0 ram write addr: 000002000 len: 0ff
# 60 ns top/ram1 ram write addr: 000002000 len: 0ff
# 68 ns top/ram0 ram read  addr: 000000000 len: 0ff
# 70 ns top/ram1 ram read  addr: 000000000 len: 0ff
# 340 ns top/ram0 ram write 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

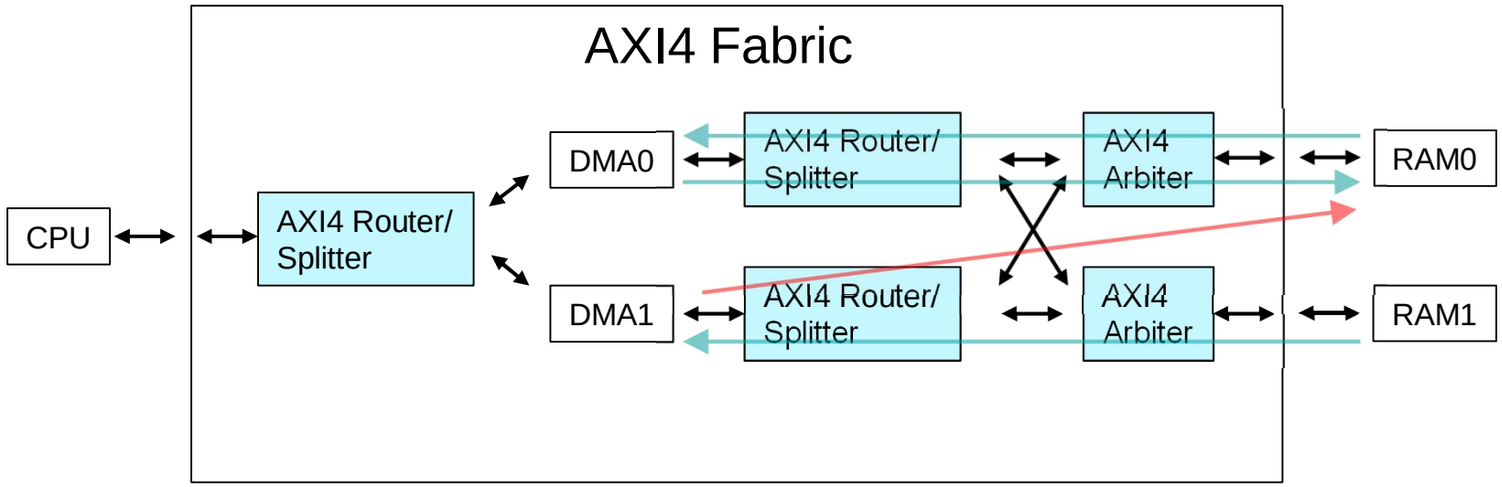
AXI4 Fabric Waveforms Before HLS–Test #0 (SystemC)



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



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)

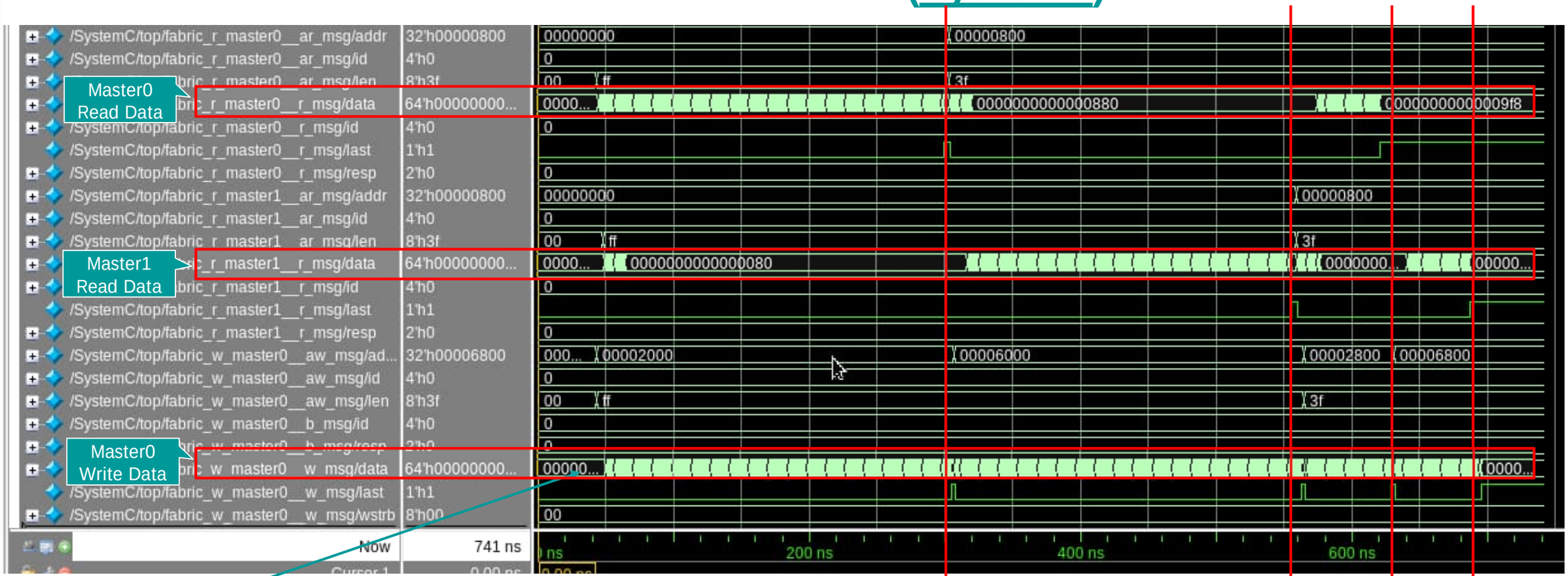
```
0 s top Stimulus started
6 ns top Running FABRIC_TEST # : 1
44 ns top.ram0 ram read  addr: 000000000 len: 0ff
44 ns top.ram0 ram write addr: 000002000 len: 0ff
49 ns top.ram1 ram read  addr: 000000000 len: 0ff
304 ns top.ram0 ram read  addr: 000000800 len: 03f
308 ns top.ram0 ram write addr: 000006000 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 HLS –Test#1 (SystemC)



256 beats from r_master0

256 beats from r_master1

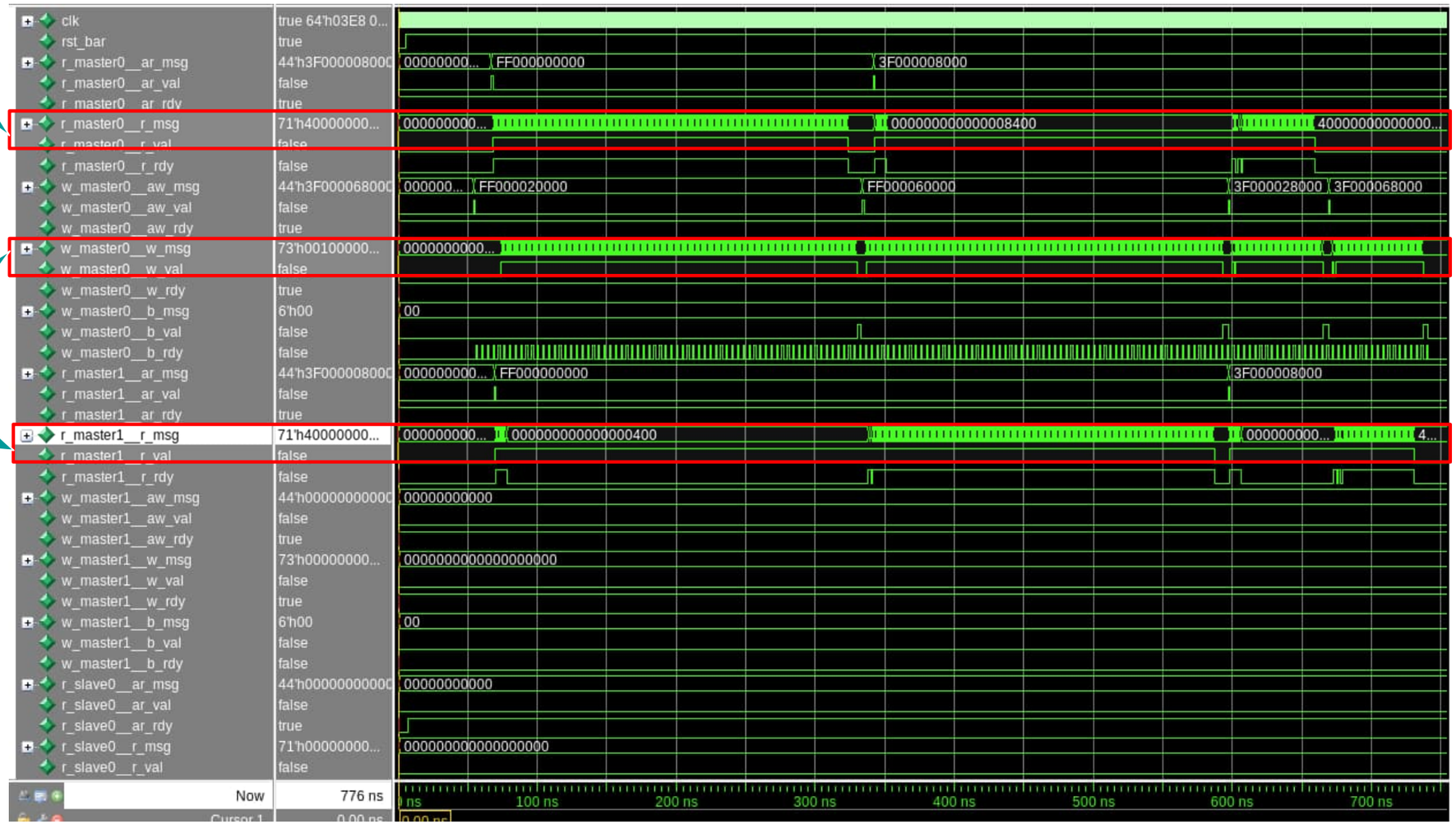
64 beats from r_master0

64 beats from r_master1

w_master0 fully utilized over 700 ns due to write contention

r_master0 and r_master1 underutilized due to write contention

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



Throughput in RTL Matches SystemC

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.
- The Matchlib memory methodology strongly recommends that any memory port contention and arbitration is present in the pre-HLS model, so that bottlenecks are visible before RTL.
  - See “Matchlib Memory Modeling Methodology”
- By default, latency for memory accesses modeled as arrays is zero.
  - If array memory port access is a performance bottleneck in your design, add explicit wait statements into your pre-HLS model to account for bottleneck.
  - See Matchlib example 38*.
- 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.

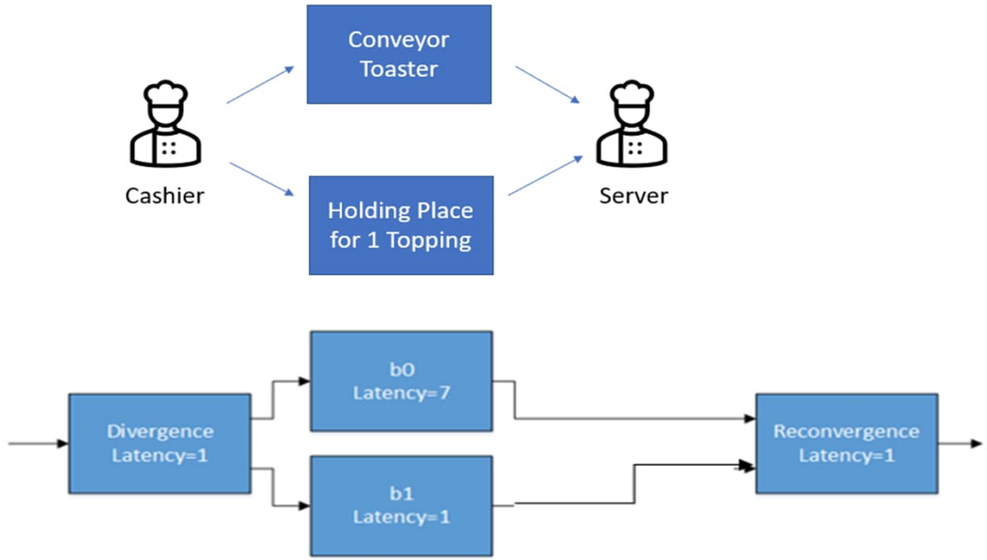
```
  void main() {
    out1.Reset();
    in1.Reset();
    wait();
#pragma hls_pipeline_init_interval 4
#pragma pipeline_stall_mode flush
    while (1) {
#ifndef __SYNTHESIS__
      wait();
      wait();
      wait();
#endif
      uint32_t t = in1.Pop();
      out1.Push(t + 0x100);
    }
  }
};
```

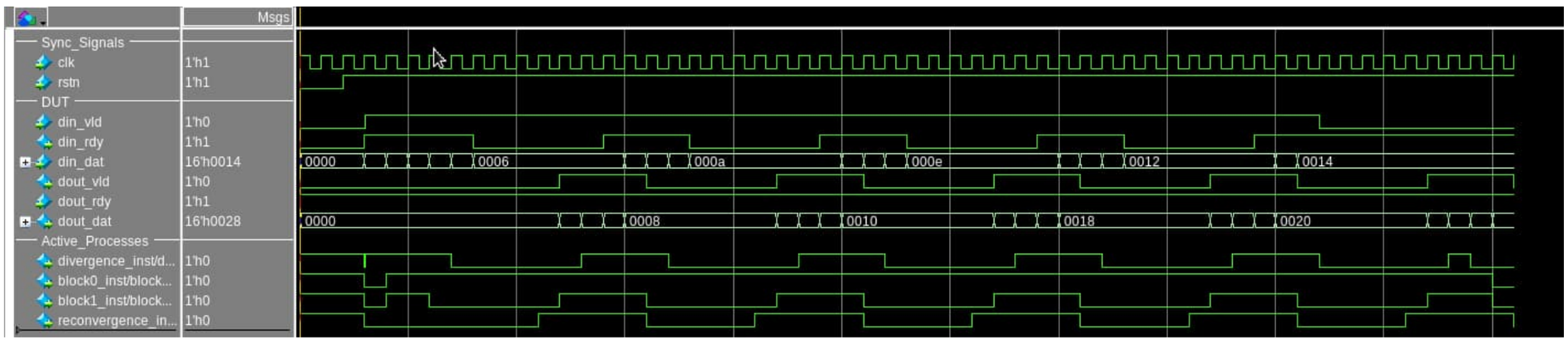
How Catapult changes latency and capacity of design…

- 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.

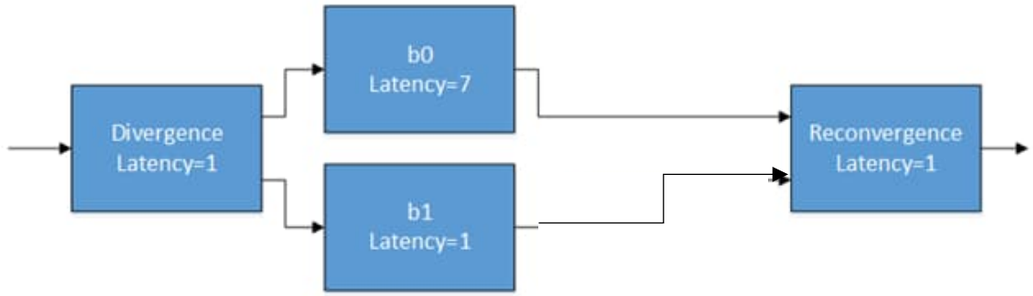


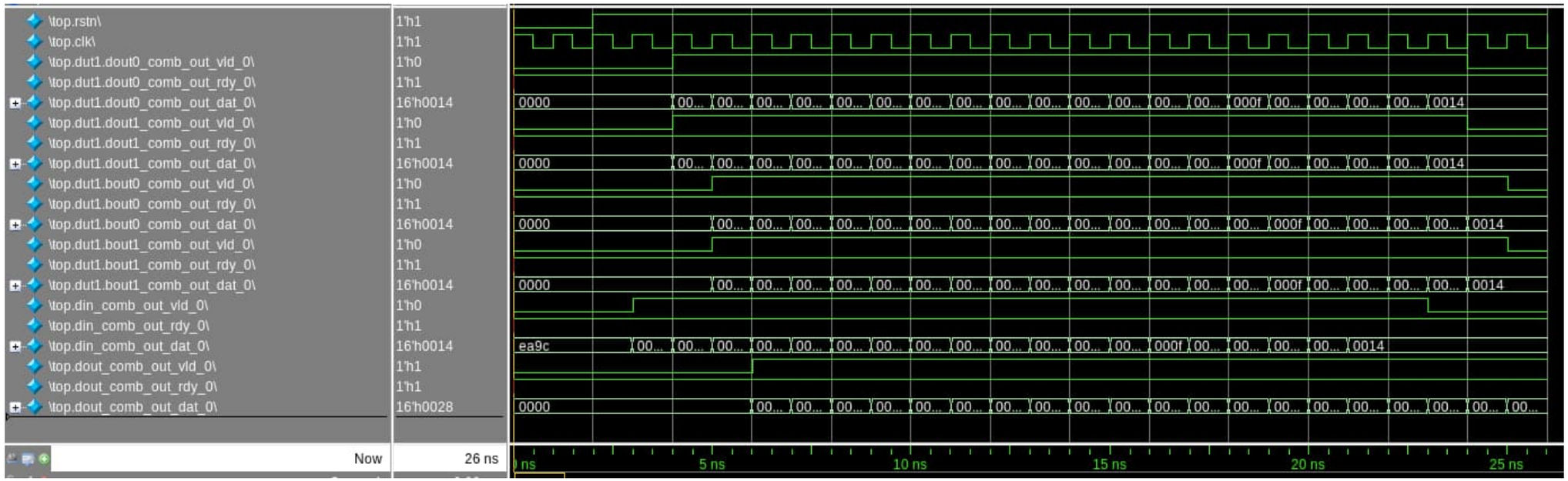


SIEMENS

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.



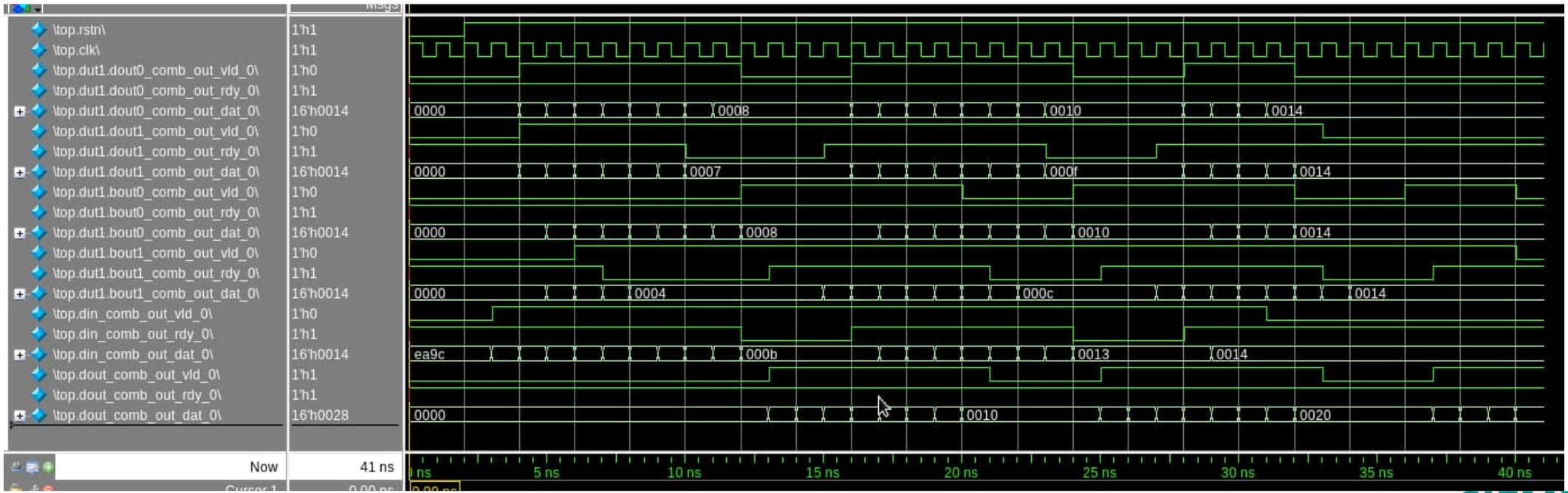


SIEMENS

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).

Using Questa, VCS, and Xcelium with Matchlib

Stuart Swan
Platform Architect
Siemens EDA
18 January 2024

SIEMENS

Using Questa, VCS, and Xcelium with Matchlib

- All major EDA HDL simulators have native support for SystemC models
  - Matchlib models work in all simulators
- First use case: SystemC-only hierarchy
  - Useful if designers are comfortable with HDL debuggers and want a HW-centric view of TB + DUT
  - Dynamic waveform viewing is more interactive and has more features than post-run waveform viewing
  - C++ source code debugging may have better capabilities than gdb/ddd
- Second use case: Verilog or SV TB + SC DUT (example 45*)
  - Very useful (and recommended) way to perform pre-HLS verification with Verilog/SV TB
  - Enables use of SV UVM testbenches to stress test the pre-HLS model (and reuse on RTL)

Verilog TB with SC DUT

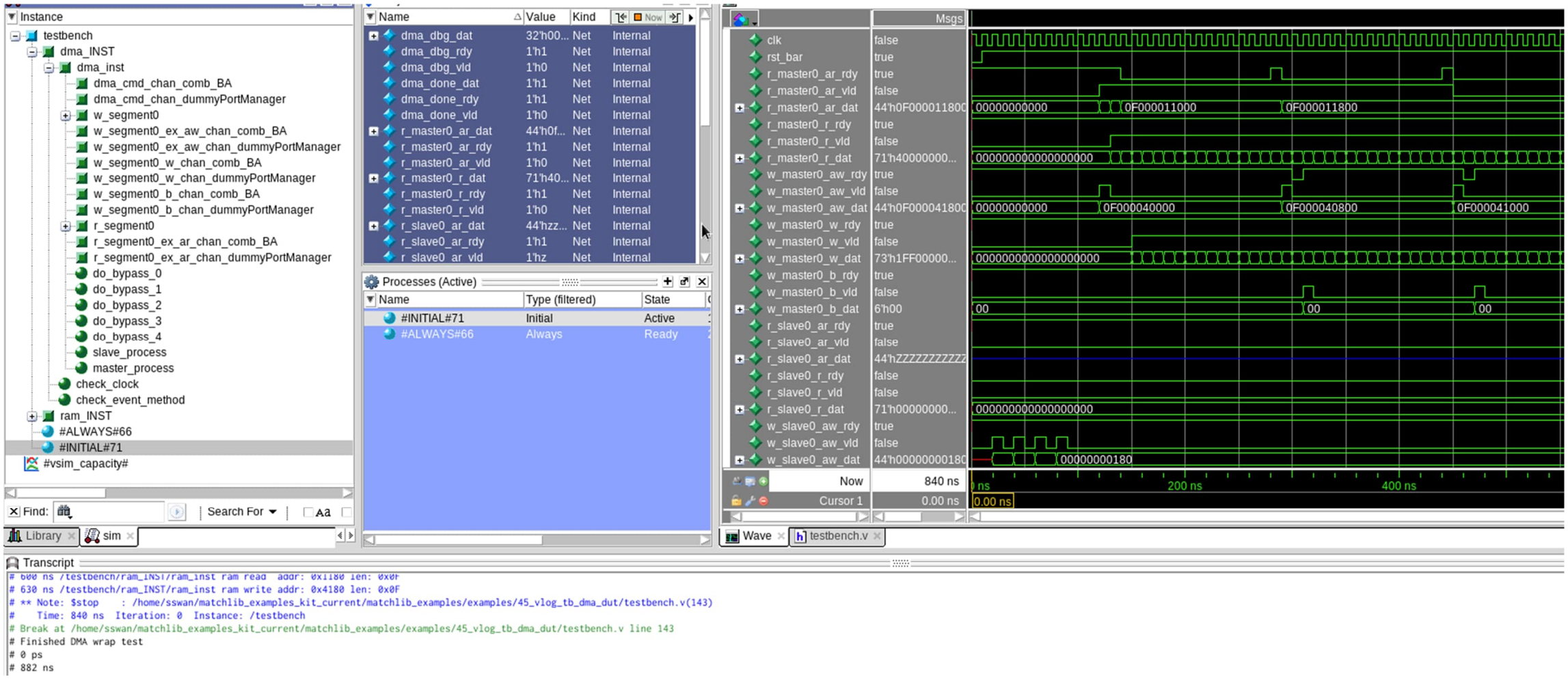
- Matchlib supports automated generation of wrapper files to enable SC DUT to be instantiated in Verilog/SV TB. (examples 08*, 45*)

```
44 class dma : public sc_module, public local_axi
45 {
46 public:
47   sc_in<bool> CCS_INIT_S1(clk);
48   sc_in<bool> CCS_INIT_S1(rst_bar);
49
50   r_master<> CCS_INIT_S1(r_master0);
51   w_master<> CCS_INIT_S1(w_master0);
52   r_slave<>  CCS_INIT_S1(r_slave0);
53   w_slave<>  CCS_INIT_S1(w_slave0);
54   Connections::Out<bool> CCS_INIT_S1(dma_done);
55   Connections::Out<sc_uint<32>> CCS_INIT_S1(dma_dbg);
56
57   AUTO_GEN_PORT_INFO(dma, ( \
58     clk \
59   , rst_bar \
60   , r_master0 \
61   , w_master0 \
62   , r_slave0 \
63   , w_slave0 \
64   , dma_done \
65   , dma_dbg \
66   ) )
67   //
```

```
1  // Auto generated on: Wed Mar  6 15:36:50 2024
2
3  // This file is an SC wrapper of the pre-HLS model to an HDL simulator
4
5  #include "dma.h"
6
7  extern sc_trace_file* trace_file_ptr;
8
9  class dma_wrap : public sc_module {
10 public:
11   dma CCS_INIT_S1(dma_inst);
12
13   decltype(dma_inst.clk)   CCS_INIT_S1(clk);
14   decltype(dma_inst.rst_bar)   CCS_INIT_S1(rst_bar);
15   decltype(dma_inst.r_master0.ar.rdy)   CCS_INIT_S1(r_master0_ar_rdy);
16   decltype(dma_inst.r_master0.ar.vld)   CCS_INIT_S1(r_master0_ar_vld);
17   decltype(dma_inst.r_master0.ar.dat)   CCS_INIT_S1(r_master0_ar_dat);
18
19   // some lines omitted..
20
21   SC_CTOR(dma_wrap)
22   : connections_clk("connections_clk", 10, SC_NS, 0.5,0,SC_NS,true)
23   {
24     SC_METHOD(check_clock);
25     sensitive << connections_clk << clk;
26
27     SC_METHOD(check_event_method);
28     sensitive << check_event;
29
30     trace_file_ptr = sc_create_vcd_trace_file("trace");
31     trace_hierarchy(this, trace_file_ptr);
32
33     dma_inst.clk(clk);
34     dma_inst.rst_bar(rst_bar);
35     dma_inst.r_master0.ar.rdy(r_master0_ar_rdy);
36     dma_inst.r_master0.ar.vld(r_master0_ar_vld);
37     dma_inst.r_master0.ar.dat(r_master0_ar_dat);
38     dma_inst.r_master0.r.rdy(r_master0_r_rdy);
39     dma_inst.r_master0.r.vld(r_master0_r_vld);
40     dma_inst.r_master0.r.dat(r_master0_r_dat);
41     dma_inst.w_master0.aw.rdy(w_master0_aw_rdy);
42     dma_inst.w_master0.aw.vld(w_master0_aw_vld);
43     dma_inst.w_master0.aw.dat(w_master0_aw_dat);
```

```
1  // Auto generated on: Wed Mar  6 15:36:50 2024
2
3
4  // This file shows the Verilog input/output decl
5  // This file is only for documentation purposes.
6
7  module dma(
8    clk
9  , rst_bar
10 , r_master0_ar_rdy
11 , r_master0_ar_vld
12 , r_master0_ar_dat
13 , r_master0_r_rdy
14 , r_master0_r_vld
15 , r_master0_r_dat
16 , w_master0_aw_rdy
17 , w_master0_aw_vld
18 , w_master0_aw_dat
19 , w_master0_w_rdy
20 , w_master0_w_vld
21 , w_master0_w_dat
22 , w_master0_b_rdy
23 , w_master0_b_vld
24 , w_master0_b_dat
25 , r_slave0_ar_rdy
26 , r_slave0_ar_vld
27 , r_slave0_ar_dat
28 , r_slave0_r_rdy
29 , r_slave0_r_vld
30 , r_slave0_r_dat
31 , w_slave0_aw_rdy
32 , w_slave0_aw_vld
33 , w_slave0_aw_dat
34 , w_slave0_w_rdy
35 , w_slave0_w_vld
```

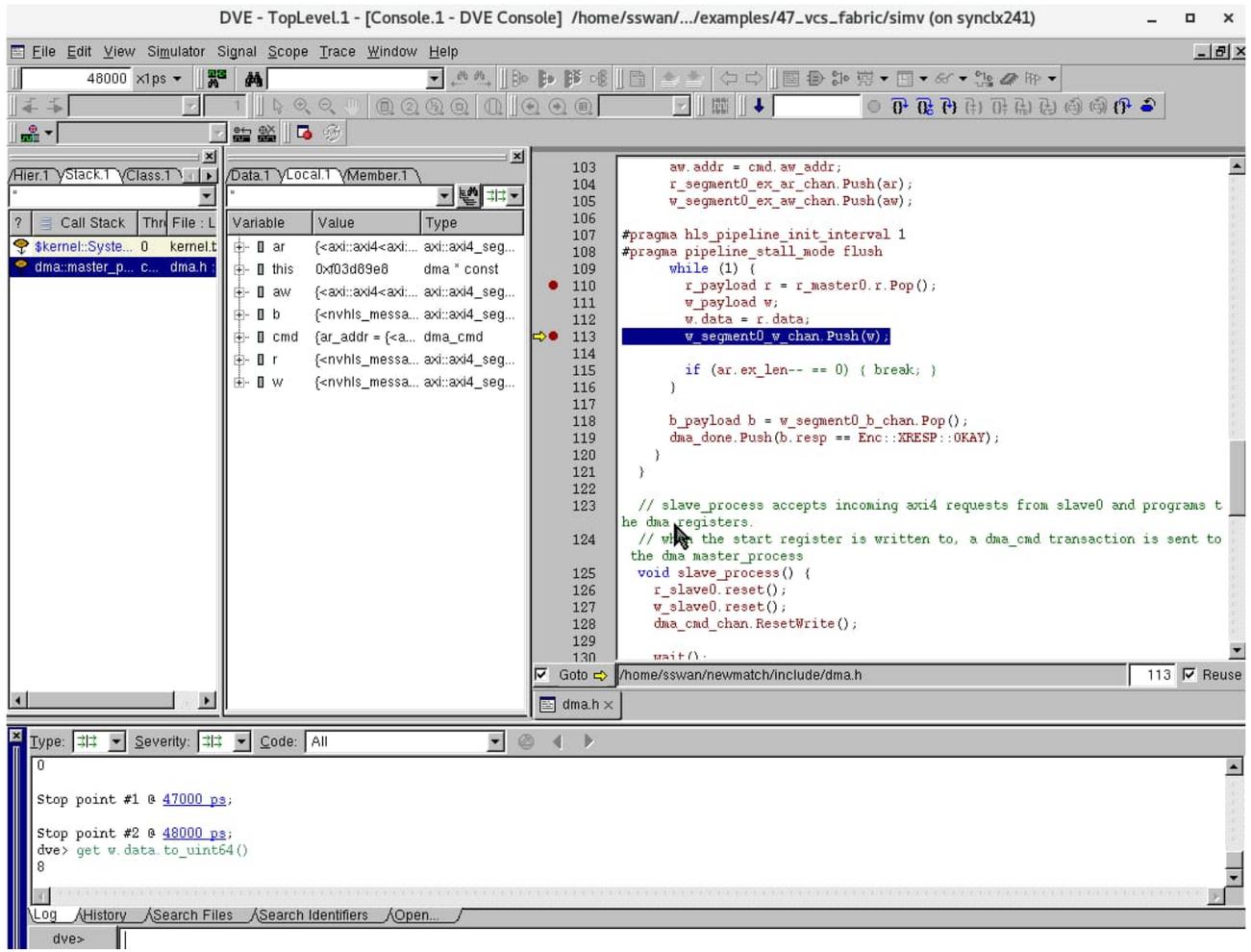
Pre-HLS Debugging in Questa with Verilog TB + SC DUT



Refer to example 45*

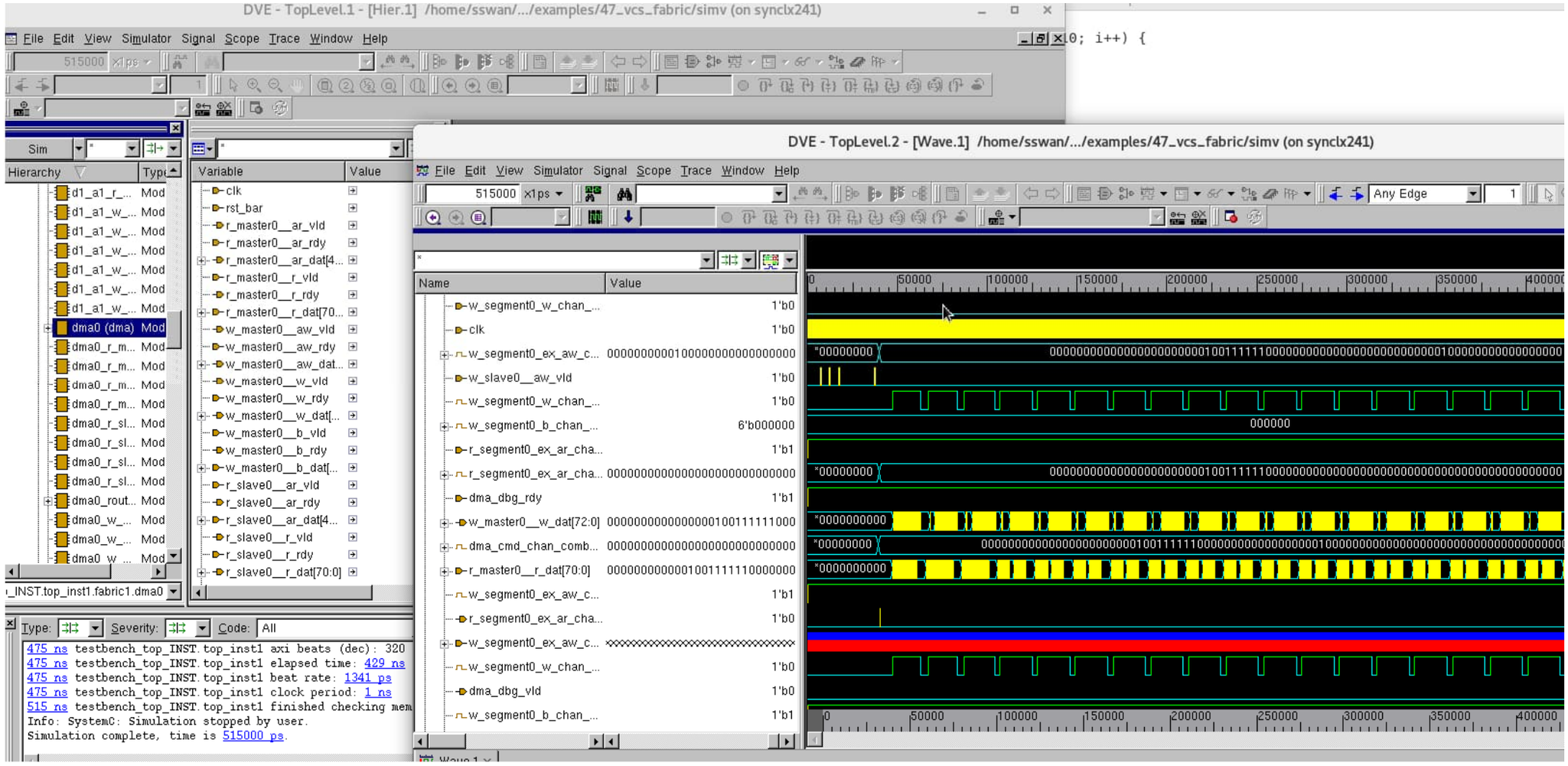
Pre-HLS Debugging in VCS

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



Refer to example 45*

Pre-HLS Debugging in VCS (SystemC-only hierarchy)



Custom Protocols in Catapult SystemC & Matchlib

Stuart Swan
Platform Architect
Siemens EDA
18 January 2024

SIEMENS

Supporting Custom Protocols in Catapult SystemC & Matchlib

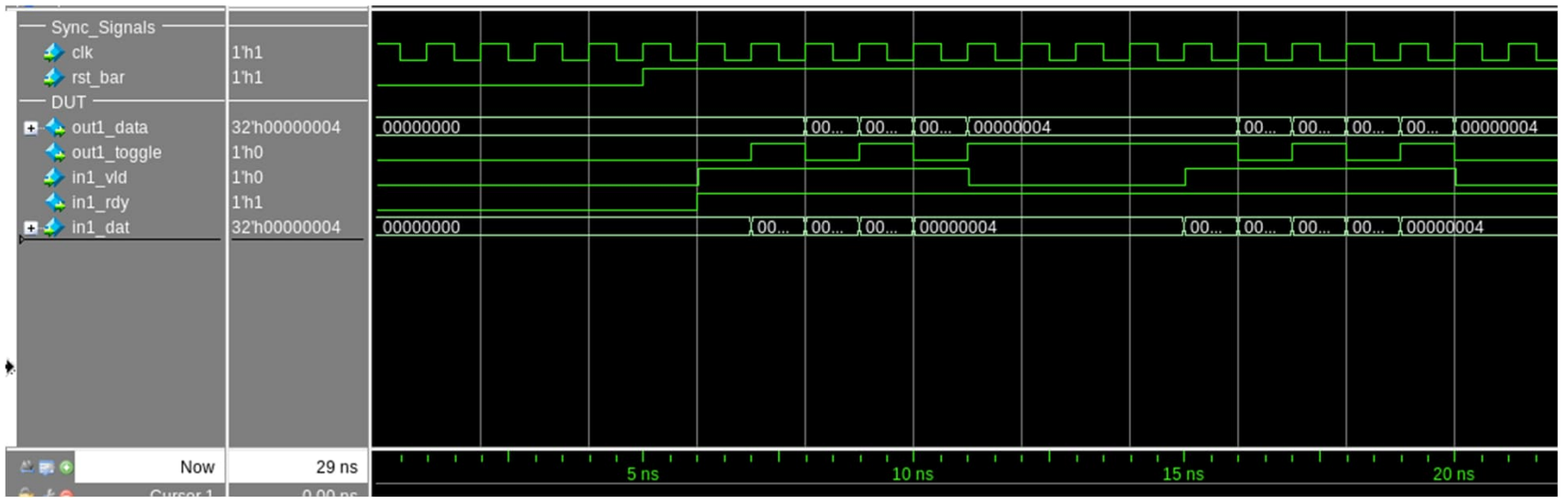
- First question: Do you really need a new custom protocol?
  - Most modern on-chip bus protocols use message passing (e.g. AXI3/4, NOC protocols, etc).
  - Message passing means "ready/valid/data" signaling. (Actual signal names may vary.)
  - Message passing protocols are easily layered on top of Matchlib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

- Second question: Is it the “dat/vld” protocol?
  - “dat/vld” protocol is same as dat/rdy/vld protocol (i.e. message passing), but rdy is assumed to be true always.
  - This is easily modeled by layering on top of Matchlib Connections Push/Pop.
  - See Matchlib Examples kit example 32_dat_vld

- Third 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

Simple “Toggle” Protocol (example 13*)

```
 8 class dut : public sc_module
 9 {
10 public:
11   sc_in<bool> CCS_INIT_S1(clk);
12   sc_in<bool> CCS_INIT_S1(rst_bar);
13
14   sc_out<sc_uint<32>>  CCS_INIT_S1(out1_data);
15   sc_out<bool>         CCS_INIT_S1(out1_toggle);
16
17   Connections::In <sc_uint<32>> CCS_INIT_S1(in1);
18
19   SC_CTOR(dut) {
20     SC_THREAD(main);
21     sensitive << clk.pos();
22     async_reset_signal_is(rst_bar, false);
23   }
24
25 private:
26
27   void main() {
28     out1_data = 0;
29     out1_toggle = false;
30     in1.Reset();
31     bool toggle = false;
32     wait();                                  // WA
33 #pragma hls_pipeline_init_interval 1
34 #pragma pipeline_stall_mode flush
35     while (1) {
36       uint32_t t = in1.Pop();
37       out1_data = t;
38       toggle = !toggle;
39       out1_toggle = toggle;
40     }
41   }
42 };
```



out1_toggle changes value for each new output

SIEMENS

Non-trival Protocols in Catapult SystemC & Matchlib

- For non-trivial protocols, there are a few different modeling and synthesis approaches possible.
  - Each approach is demonstrated in example 52_apb
  - Each approach has some advantages and disadvantages in area, latency, etc.
- Within this presentation, we present the most general approach
  - This approach supports throughput accurate pre-HLS simulation and supports any protocol type
  - It may have a small area & latency disadvantage (in current Catapult release) compared to other approaches in some cases.
  - See example 52_apb for discussion and details on all approaches

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.

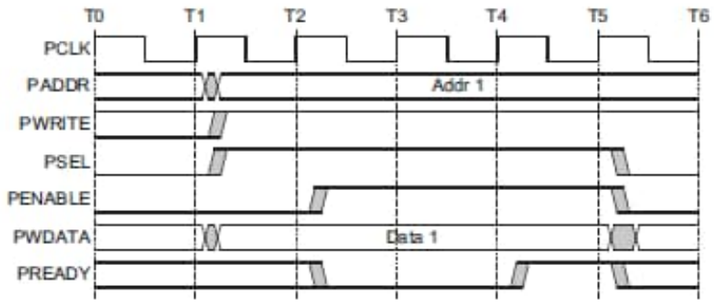


Figure 3-2 Write transfer with wait states

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

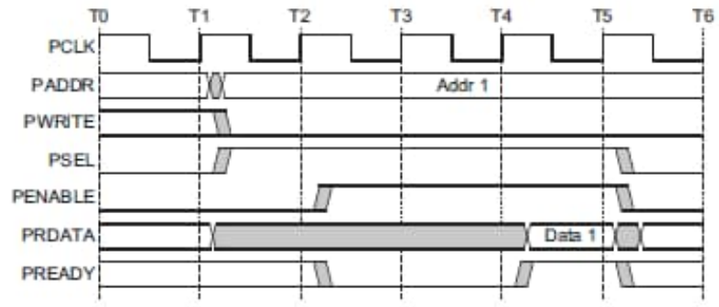


Figure 3-5 Read transfer with wait states

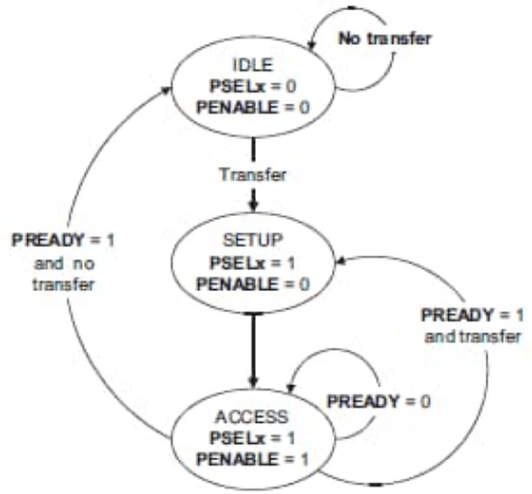


Figure 4-1 State diagram

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:
  - Use signal level protocol to send and receive messages with user’s model
  - While simultaneously managing the signals and detailed timing of the desired protocol
- 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);
```

Non-Trivial Custom Protocol Example: Step 4

- Define transaction message classes

```
48      struct apb_req : public nvhls_message {
49        bool         is_write { false };
50        addr_payload addr;
51        w_payload    w;
52
53        AUTO_GEN_FIELD_METHODS(apb_req, ( \
54           is_write \
55         , addr \
56         , w \
57        ) )
58        //
59      };
```

```
62      struct apb_rsp : public nvhls_message {
63        r_payload    r; // if req was a write, then write resp
64
65        AUTO_GEN_FIELD_METHODS(apb_rsp, ( \
66           r \
67        ) )
68        //
69      };
```

Non-Trivial Custom Protocol Example: Step 5

- Create the code for master transactor
- Key points:
  - Use disable_spawn() so we can use signals to directly send/receive messages
  - Only use a single wait() statement in transactor (for HLS QOR)
  - We are effectively writing RTL here..

```
198     class apb_master_xactor : public sc_module
199     {
200     public:
201       sc_in<bool> CCS_INIT_S1(clk);
202       sc_in<bool> CCS_INIT_S1(rst_bar);
203
204       Connections::In<apb_req, PortType>   CCS_INIT_S1(req_port);
205       Connections::Out<apb_rsp, PortType>  CCS_INIT_S1(rsp_port);
206       sc_out<Addr>     CCS_INIT_S1(PADDR);
207       sc_out<bool>     CCS_INIT_S1(PWRITE);
208       sc_out<bool>     CCS_INIT_S1(PENABLE);
209       sc_out<bool>     CCS_INIT_S1(PSEL);
210       sc_out<Data>     CCS_INIT_S1(PWDATA);
211       sc_out<Wstrb>    CCS_INIT_S1(PSTRB);
212       sc_out<Prot_t>   CCS_INIT_S1(PPROT);
213       sc_in<Data>      CCS_INIT_S1(PRDATA);
214       sc_in<bool>      CCS_INIT_S1(PSLVERR);
215       sc_in<bool>      CCS_INIT_S1(PREADY);
216
217       SC_CTOR(apb_master_xactor) {
218         SC_THREAD(main);
219         sensitive << clk.pos();
220         async_reset_signal_is(rst_bar, false);
221
222 #ifdef CONNECTIONS_SIM_ONLY
223         req_port.disable_spawn();
224         rsp_port.disable_spawn();
225 #endif
226       }
```

Non-Trivial Custom Protocol Example: Step 5 (continued)

```
254         apb_req req;
255         apb_rsp rsp;
256
257         typedef enum {APB_IDLE, APB_GET_RSP} apb_state_t;
258         apb_state_t state = APB_IDLE;
259
260         req_port.rdy = 1;
261
262         while (1) {
263          wait();
264
265          switch (state) {
266           case APB_IDLE:
267             state = APB_IDLE;
268             PSEL = 0;
269             PENABLE = 0;
270             if (rsp_port.rdy) {
271              rsp_port.vld = 0;
272             }
273             if (req_port.vld) {
274               req_port.rdy = 0;
275               state = APB_GET_RSP;
276               bits_to_type_if_needed(req, req_port.dat);
277               // SETUP state
278               PSEL = 1;
279               PENABLE = 0;
280               PADDR = req.addr.addr.to_uint64();
281               PWRITE = req.is_write;
282               if (req.is_write) {
283                 PWDATA = req.w.data.to_uint64();
284                 PSTRB = req.w.wstrb.to_uint64();
285               } else {
286                 PWDATA = 0;
287                 PSTRB = 0;
288               }
289             }
290             break;
```

```
292           case APB_GET_RSP:
293             state = APB_GET_RSP;
294             PENABLE = 1;
295
296             if (PREADY == 1) {
297               state = APB_IDLE;
298               rsp.r.data = PRDATA.read().to_uint64();
299               rsp.r.resp = PSLVERR.read();
300               PSEL = 0;
301               PENABLE = 0;
302               rsp_port.vld = 1;
303               type_to_bits_if_needed(rsp_port.dat, rsp);
304               req_port.rdy = 1;
305             } else {
306               state = APB_GET_RSP;
307             }
308          }
309         }
310       }
311     };
```


SIEMENS

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

- Create the code for slave transactor
- Key points:
  - Use disable_spawn() so we can use signals to directly send/receive messages
  - Only use a single wait() statement in transactor (for HLS QOR)
  - We are effectively writing RTL here..

```
317     class apb_slave_xactor : public sc_module
318     {
319     public:
320       sc_in<bool> CCS_INIT_S1(clk);
321       sc_in<bool> CCS_INIT_S1(rst_bar);
322
323       Connections::Out<apb_req, PortType> CCS_INIT_S1(req_port);
324       Connections::In<apb_rsp, PortType>  CCS_INIT_S1(rsp_port);
325       sc_in<Addr>       CCS_INIT_S1(PADDR);
326       sc_in<bool>       CCS_INIT_S1(PWRITE);
327       sc_in<bool>       CCS_INIT_S1(PENABLE);
328       sc_in<bool>       CCS_INIT_S1(PSEL);
329       sc_in<Data>       CCS_INIT_S1(PWDATA);
330       sc_in<Wstrb >     CCS_INIT_S1(PSTRB);
331       sc_in<Prot_t >    CCS_INIT_S1(PPROT);
332       sc_out<Data>      CCS_INIT_S1(PRDATA);
333       sc_out<bool>      CCS_INIT_S1(PSLVERR);
334       sc_out<bool>      CCS_INIT_S1(PREADY);
335
336       template <class C>
337       void operator()(C &c) {
338         PADDR(c.PADDR);
339         PWRITE(c.PWRITE);
340         PENABLE(c.PENABLE);
341         PSEL(c.PSEL);
342         PWDATA(c.PWDATA);
343         PSTRB(c.PSTRB);
344         PPROT(c.PPROT);
345         PRDATA(c.PRDATA);
346         PSLVERR(c.PSLVERR);
347         PREADY(c.PREADY);
348       }
```

Non-Trivial Custom Protocol Example: Step 5 (continued)

```
368         bool pending_read = false;
369         apb_req req;
370         apb_rsp rsp;
371
372         wait();
373
374         typedef enum {APB_IDLE, APB_GET_RSP, APB_DONE} apb_state_t;
375         apb_state_t state = APB_IDLE;
376
377         while (1) {
378           wait();
379
380           switch (state) {
381           case APB_IDLE:
382            if (PSEL == 0)
383              state = APB_IDLE;
384            else {
385              req.is_write = PWRITE.read();
386              req.w.data = PWDATA.read().to_uint64();
387              req.w.wstrb = PSTRB.read().to_uint64();
388              req.addr.addr = PADDR.read().to_uint64();
389              if (req.is_write) {
390               pending_read = false;
391              } else {
392               pending_read = true;
393              }
394              type_to_bits_if_needed(req_port.dat, req);
395              req_port.vld = 1;
396              rsp_port.rdy = 1;
397              state = APB_GET_RSP;
398            }
399            break;
400
```

```
401           case APB_GET_RSP:
402             if (req_port.rdy)
403               req_port.vld = 0;
404
405             if (rsp_port.vld) {
406               rsp_port.rdy = 0;
407               bits_to_type_if_needed(rsp, rsp_port.dat);
408             }
409
410             if ((!rsp_port.vld) || (PENABLE == 0)) {
411               state = APB_GET_RSP;
412             } else {
413               state = APB_DONE;
414               PREADY = 1;
415               if (pending_read) {
416                 PRDATA = rsp.r.data.to_uint64();
417                 pending_read = false;
418               }
419               PSLVERR = rsp.r.resp;  // works for both reads and writes..
420             }
421             break;
422
423           case APB_DONE:
424            PREADY = 0;
425            PSLVERR = 0;
426            state = APB_IDLE;
427            break;
428         }
429       }
430     }
431   };
```


SIEMENS

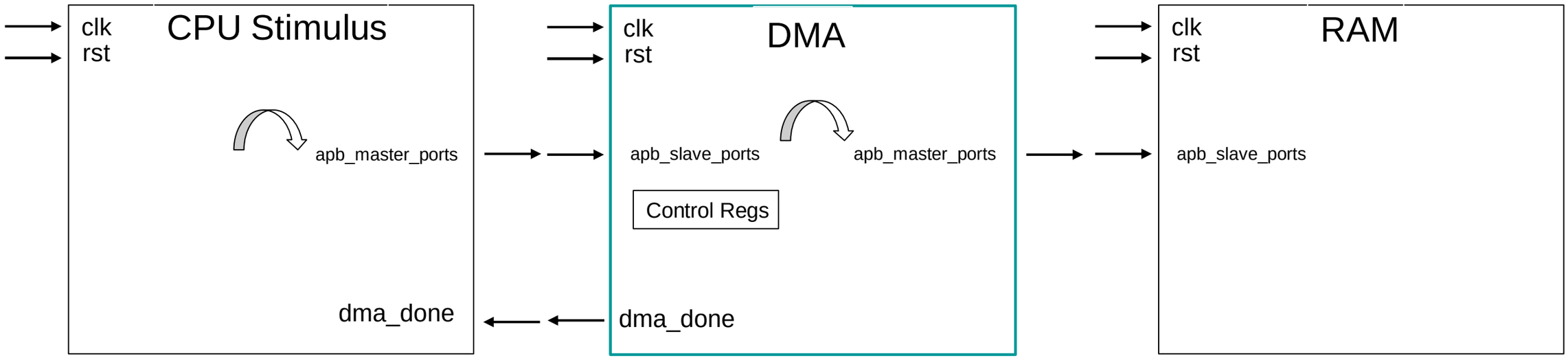
Non-Trivial Custom Protocol Example: Step 6

- Instantiate transactor in user's HLS model

```
50   apb_master_ports<>                    CCS_INIT_S1(master0_ports);
51 // #define MASTER_XACTOR 1
52 #ifdef MASTER_XACTOR
53   apb_master_xactor<>                   CCS_INIT_S1(master0_xactor);
54   Connections::Combinational<apb_req>   CCS_INIT_S1(master0_req_chan);
55   Connections::Combinational<apb_rsp>   CCS_INIT_S1(master0_rsp_chan);
56 #endif
57
58   apb_slave_xactor<>                    CCS_INIT_S1(slave0_xactor);
59   apb_slave_ports<>                     CCS_INIT_S1(slave0_ports);
60   Connections::Combinational<apb_req>   CCS_INIT_S1(slave0_req_chan); // Connec
61   Connections::Combinational<apb_rsp>   CCS_INIT_S1(slave0_rsp_chan); // Connec
62
63   SC_CTOR(dma) {
64     SC_THREAD(slave_process);
65     sensitive << clk.pos();
66     async_reset_signal_is(rst_bar, false);
67
68     SC_THREAD(master_process);
69     sensitive << clk.pos();
70     async_reset_signal_is(rst_bar, false);
71
72     slave0_xactor.clk(clk);
73     slave0_xactor.rst_bar(rst_bar);
74     slave0_xactor.req_port(slave0_req_chan);
75     slave0_xactor.rsp_port(slave0_rsp_chan);
76     slave0_xactor(slave0_ports);
77
78 #ifdef MASTER_XACTOR
79     master0_xactor.clk(clk);
80     master0_xactor.rst_bar(rst_bar);
81     master0_xactor.req_port(master0_req_chan);
82     master0_xactor.rsp_port(master0_rsp_chan);
83     master0_xactor(master0_ports);
84 #endif
85   }
86
```

```
 99 #ifdef MASTER_XACTOR
100     master0_req_chan.ResetWrite();
101     master0_rsp_chan.ResetRead();
102 #else
103     apb_master_rw_reset(master0_ports);
104 #endif
105
106     wait();
107
108     while (1) {
109       dma_cmd cmd = dma_cmd_chan.Pop(); // Blocking read - waitir
110       bool status = Enc::XRESP::OKAY;
111 // #pragma hls_pipeline_init_interval 2
112 // #pragma pipeline_stall_mode stall
113       while (1) {
114         apb_req req;
115         apb_rsp rsp;
116
117         req.is_write = false;
118         req.addr.addr = cmd.ar_addr;
119 #ifdef MASTER_XACTOR
120         master0_req_chan.Push(req);
121         rsp = master0_rsp_chan.Pop();
122 #else
123         apb_master_rw(master0_ports, req, rsp);
124 #endif
125
126         req.is_write = true;
127         req.addr.addr = cmd.aw_addr;
128         req.w.data = rsp.r.data;
129 #ifdef MASTER_XACTOR
130         master0_req_chan.Push(req);
131         rsp = master0_rsp_chan.Pop();
132 #else
133         apb_master_rw(master0_ports, req, rsp);
134 #endif
135
136         if (rsp.r.resp != Enc::XRESP::OKAY) { status = 0; }
137
138         if (cmd.len-- == 0) { break; } // DMA transfer done
139
140         cmd.aw_addr += bytesPerBeat;
141         cmd.ar_addr += bytesPerBeat;
142       }
143       dma_done.Push(status);
```

52_apb Design Example



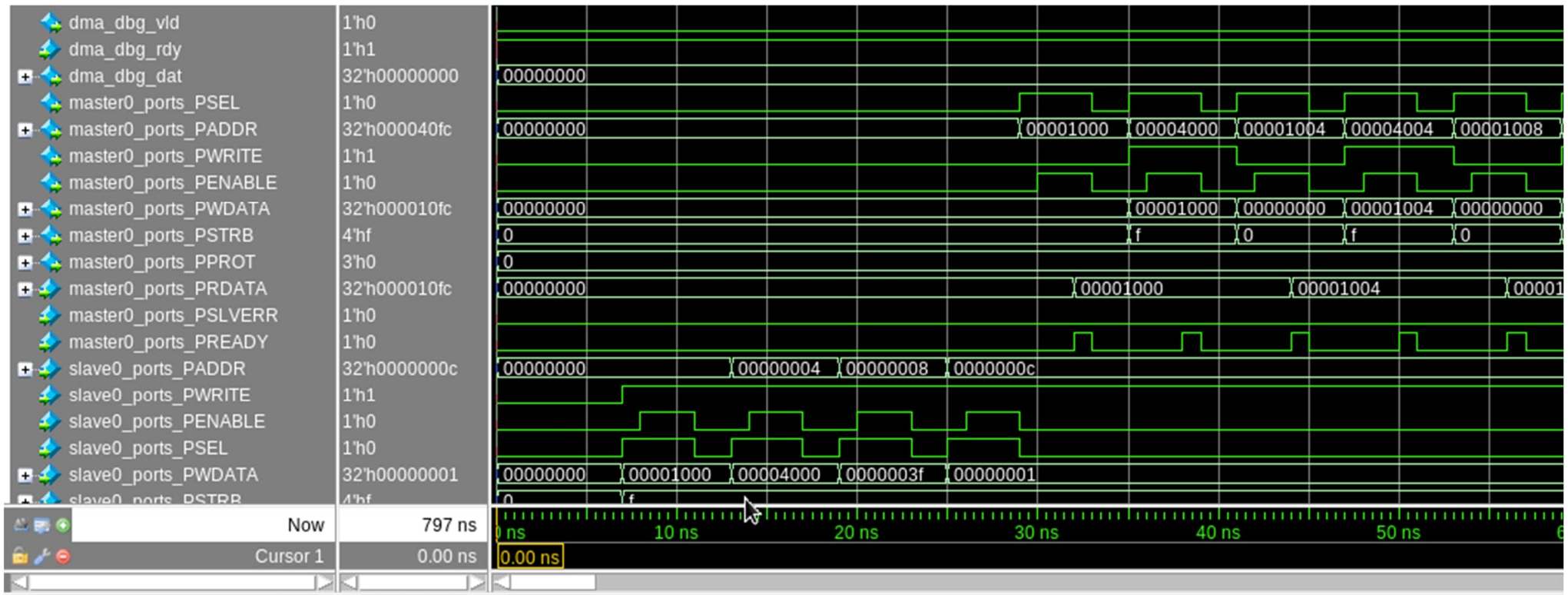

```
42 class dma : public sc_module, public local_apb
43 {
44 public:
45   sc_in<bool>                          CCS_INIT_S1(clk);
46   sc_in<bool>                          CCS_INIT_S1(rst_bar);
47   Connections::Out<bool>               CCS_INIT_S1(dma_done);
48   Connections::Out<uint32_t>           CCS_INIT_S1(dma_dbg);
49
50   apb_master_ports<>                   CCS_INIT_S1(master0_ports);
51 // #define MASTER_XACTOR 1
52 #ifdef MASTER_XACTOR
53   apb_master_xactor<>                  CCS_INIT_S1(master0_xactor);
54   Connections::Combinational<apb_req>  CCS_INIT_S1(master0_req_chan);
55   Connections::Combinational<apb_rsp>  CCS_INIT_S1(master0_rsp_chan);
56 #endif
57
58   apb_slave_xactor<>                   CCS_INIT_S1(slave0_xactor);
59   apb_slave_ports<>                    CCS_INIT_S1(slave0_ports);
```

= 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 and test the transactors in Verilog RTL first and then just translate to SC Matchlib afterwards.
  - After all, we are effectively writing RTL in the transactors.

- If area or latency of transactors are a concern, see discussion in 52_apb example.

Matchlib Memory Modeling Methodology

Refer to doc/matchlib_memory_modeling_methodology.pdf

Stuart Swan
Platform Architect
Siemens EDA
15 April 2024

SIEMENS

Memories in HLS Designs

- In pre-HLS models, RAMs appear as normal C arrays (“implicit memories”)
  - Usually these C arrays are “wrapped” in a C++ class to provide additional features.
    - Important feature: automatically check in pre-HLS and post-HLS sims for invalid indexes.
  - RAMs modeled as sc_modules are called “explicit memories” and are deprecated.
- RAMs and associated logic must be carefully designed to meet PPA targets.
- Often, have maximum of 1 read port and 1 write port per RAM (due to area costs of > 1 port)
- Often, need multiple RAM banks to meet performance goals
- Often, want to pipeline with II=1

Goals of the Matchlib Memory Modeling Methodology

1. Meet PPA (power, performance, area) goals without any compromises.
2. Discover and resolve all functional and performance issues related to memories in the pre-HLS model, not in the post-HLS model.
3. The pre-HLS model code should be clean and easy to understand.
4. Emit error message for any invalid indexes in both pre-HLS and post-HLS simulations.

Overview of the Matchlib Memory Models

- In almost all cases these classes are implemented as RAMs/ROMs in the RTL generated by HLS.
- They are also sometimes mapped to register arrays in the RTL.
- These classes are not intended to be used for other purposes (e.g. modeling arrays within transaction payloads).
- These classes provide optimal QOR even for “non-power of 2” memories.
- Because the intent is to map to RAMs/ROMs, these classes should not be read or written to in the reset state of processes which access them.
  - With this restriction, Catapult can generate PSL or SVA assertions into the RTL to perform checking for index violations for these models.
- All classes work in both the Catapult SystemC flow as well as the C++ flow.
- Normal C arrays should not be used to model RAMs/ROMs in newly developed HLS models because they do not provide the benefits listed above.

Overview of the Matchlib Memory Models

ac_array_1D

- Simple non-shared 1D array implemented as a RAM/ROM.
- Provides built-in checking for index violations for all array dimensions in both pre-HLS model and in RTL (via PSL or SVA assertions in RTL).
- Usage is same as normal C array, aside from declaration.
- See Matchlib toolkit example 35*.

ac_shared_array_1D

- Simple shared 1D array implemented as a RAM/ROM.
- Provides built-in checking for index violations for all array dimensions in both pre-HLS model and in RTL (via PSL or SVA assertions in RTL).
- Usage is same as normal C array, aside from declaration.
- See Matchlib toolkit example 12*.

Overview of the Matchlib Memory Models

ac_bank_array_2D, ac_bank_array_3D, ac_bank_array_vary

- Banked memory where each bank is a separate instance.
- During HLS, right-most index implemented as RAM port, all other indexes become either constants after loop unrolling or a mux tree.
- Should only be used in situations where bank conflicts are clearly not possible.
- Provides best achievable QOR in all cases, including for "non-power of 2" arrays.
- Provides built-in checking for index violations for all array dimensions in both pre-HLS model and in RTL (via PSL or SVA assertions in RTL).
- Usage is same as normal C array, aside from declaration.
- See Matchlib toolkit example 36*.

ac_shared_bank_array_2D, ac_shared_bank_array_3D, ac_shared_bank_array_vary

- Shared banked memory where each bank is a separate instance.
- During HLS, right-most index implemented as RAM port, all other indexes become either constants after loop unrolling or a mux tree.
- Should only be used in situations where bank conflicts are clearly not possible.
- Provides best achievable QOR in all cases, including for "non-power of 2" arrays.
- Provides built-in checking for index violations for all array dimensions in both pre-HLS model and in RTL (via PSL or SVA assertions in RTL).
- Usage is same as normal C array, aside from declaration.
- See Matchlib toolkit example 38*.

Overview of the Matchlib Memory Models

Scratchpad

- Banked memory where low order bits of address select bank (like Catapult interleave directive).
- Requests routed thru crossbar to proper bank, responses routed thru crossbar from proper bank.
- Emits error message on any bank conflicts.
- No arbitration or queueing, backpressure is not possible.
- Supports memory transaction log file generation.
- See Matchlib toolkit example 41* and later discussion in this section.

ArbitratedScratchpad

- Banked memory where low order bits of address select bank (like Catapult interleave directive).
- Requests routed thru crossbar to proper bank, responses routed thru crossbar from proper bank.
- Bank conflicts are allowed.
- Has arbitration and queues to handle potential bank conflicts.
- Incoming requests subject to backpressure due to potential bank conflicts.
- See Matchlib toolkit example 42* and later discussion in this section.

Overview of the Matchlib Memory Models

extended_array

- Provides memory transaction log file generation for debugging.
- Inherits from ac_array_1D and can be used in place of it.
- Can also be used in place of ac_shared_array_1D for pre-HLS sim only.
- Usage is same as normal C array, aside from declaration.
- Emits error message on uninitialized memory reads (UMR) in pre-HLS simulation only
- See Matchlib example 12* and “memory_logging_and_debug.pdf” in the Matchlib toolkit doc directory.

ac_wr_mask_array_1D

- Simple 1D array implemented as a RAM/ROM with configurable write mask.
- Each element can have 1 or more slices, any or all slices can be written as per the write mask.
- Provides built-in checking for index violations for all array dimensions in both pre-HLS model and in RTL (via PSL or SVA assertions in RTL).
- Usage is same as normal C array, aside from declaration.
- See Matchlib toolkit example 33*.

Avoid using these classes for modeling memories

ac_interleave_array

- Should never be used, use Scratchpad or ArbitratedScratchpad instead.
- Scratchpad provides predictable usage model during HLS and best QOR.
- Scratchpad emits error messages for bank conflicts in pre-HLS simulation.
- ArbitratedScratchpad handles more complex cases that would result in scheduling failures if ac_interleave_array was instead used.

ac_array

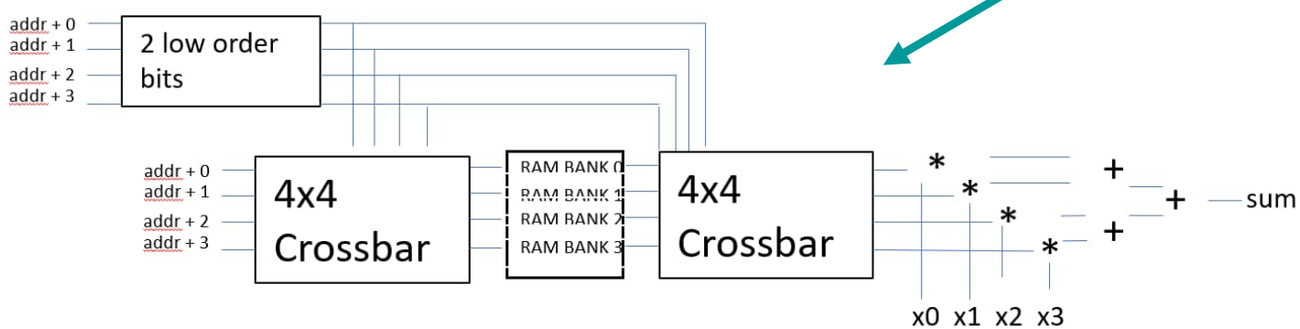
- Should never be used to model RAMs/ROMs.
- Supports from 1 to 3 dimensions.
- By default, does not provide index bounds checking.
- Does not provide good QOR for "non-power of 2" memories.
- ac_bank_array* provides better QOR for cases where number of dimensions is 2 or more.

Discussion of Scratchpad and ArbitratedScratchpad

- The rest of this section provides an in-depth discussion of Scratchpad and ArbitratedScratchpad.
- We begin by introducing three simple design examples to frame the discussion.

Example Design #1

- Similar to Matchlib example 41*
- Require II=1, so new “sum” produced on every clock
- Note that “addr” may not be evenly divisible by 4
- HW must look like design below.




```
uint16 coeffs[1024];

 void main() {
    wait();

#pragma hls_pipeline_init_interval 1
#pragma pipeline_stall_mode flush
  while (1) {
    uint16 addr = addr_in.Pop();
    uint16 x0 = in0.Pop();
    uint16 x1 = in1.Pop();
    uint16 x2 = in2.Pop();
    uint16 x3 = in3.Pop();
    uint16 sum = (x0 * coeffs[addr + 0]) +
                 (x1 * coeffs[addr + 1]) +
                 (x2 * coeffs[addr + 2]) +
                 (x3 * coeffs[addr + 3]);
    out1.Push(sum);
  }
 }
```

- Bank conflicts are not possible, no arbitration needed.
- Note that HW design never needs to apply backpressure to upstream requests.

Example Design #2

- coeffs indexes now depend on func0, func1, etc.
- If it can be proven that never any bank conflicts, can use same HW as in design #1
- Otherwise, more complex HW needed:
  - Arbiters to handle bank conflicts
  - Queuing of stalled requests
  - Backpressure to requests coming into block
  - Can no longer guarantee a new “sum” is produced on every clock

```
uint16 coeffs[1024];

 void main() {
   wait();

#pragma hls_pipeline_init_interval 1
#pragma pipeline_stall_mode flush
  while (1) {
    uint16 addr = addr_in.Pop();
    uint16 x0 = in0.Pop();
    uint16 x1 = in1.Pop();
    uint16 x2 = in2.Pop();
    uint16 x3 = in3.Pop();
    uint16 sum = (x0 * coeffs[func0(addr)]) +
                 (x1 * coeffs[func1(addr)]) +
                 (x2 * coeffs[func2(addr)]) +
                 (x3 * coeffs[func3(addr)]);
    out1.Push(sum);
  }
 }
```

Example Design #3

- coeffs indexes now come from outside block
- HLS can never prove that never any bank conflicts
- But designer may **know** there can never be bank conflicts
  - If so, then simpler HW as in design #1 can be used.
- Otherwise, more complex HW needed:
  - Arbiters to handle bank conflicts
  - Queuing of stalled requests
  - Backpressure to requests coming into block
  - Can no longer guarantee a new “sum” is produced on every clock

```
uint16 coeffs[1024];

 void main() {
   wait();

#pragma hls_pipeline_init_interval 1
#pragma pipeline_stall_mode flush
  while (1) {
    uint16 addr0 = addr_in0.Pop();
    uint16 addr1 = addr_in1.Pop();
    uint16 addr2 = addr_in2.Pop();
    uint16 addr3 = addr_in3.Pop();
    uint16 x0 = in0.Pop();
    uint16 x1 = in1.Pop();
    uint16 x2 = in2.Pop();
    uint16 x3 = in3.Pop();
    uint16 sum = (x0 * coeffs[addr0]) +
                 (x1 * coeffs[addr1]) +
                 (x2 * coeffs[addr2]) +
                 (x3 * coeffs[addr3]);
    out1.Push(sum);
  }
 }
```

Catapult Memory Banking Directives

- For design #1, using Catapult –INTERLEAVE 4 directive on coeffs array:
  - Catapult will do index analysis and prove there are no bank conflicts
  - Catapult will produce HW similar to diagram for design #1

- For design #2, Catapult may or may not succeed in proving no bank conflicts.
  - If it succeeds, will generate HW similar to design #1
  - If it fails, it will not generate arbitration and queuing logic.
    - Instead, it will give up on building a banked memory
    - II=1 goal will not be met

- For design #3, Catapult will not succeed in proving no bank conflicts.
  - It will not generate arbitration and queuing logic.
    - Instead, it will give up on building a banked memory
    - II=1 goal will not be met

Catapult Memory Banking Directives Don’t Meet Our Goals

- We see that Catapult INTERLEAVE directive for Examples #2 and #3 does not meet our top two goals for the memory modeling methodology:
  1. Meet PPA (power, performance, area) goals without any compromises.
  2. Discover and resolve all functional and performance issues related to memories in the pre-HLS model, not in the post-HLS model.

- Because of this, we recommend never using INTERLEAVE, BLOCK_SIZE, or WORD_WIDTH directives within Matchlib models.
  - BLOCK_SIZE and WORD_WIDTH are similar to INTERLEAVE and have similar problems.

- Instead, we use memory models in the pre-HLS design that explicitly specify the desired memory architecture
  - Guarantees that HLS will build what designer wants
  - Enables all performance and functional problems related to memory architecture to be seen in pre-HLS simulation

Scratchpad

- Implements 2 crossbars with banked memory structure shown in design #1
- In pre-HLS simulation, if there are bank conflicts, emits an error message
- Safe to use for design scenarios #2 and #3 since it emits an error message if any bank conflicts.

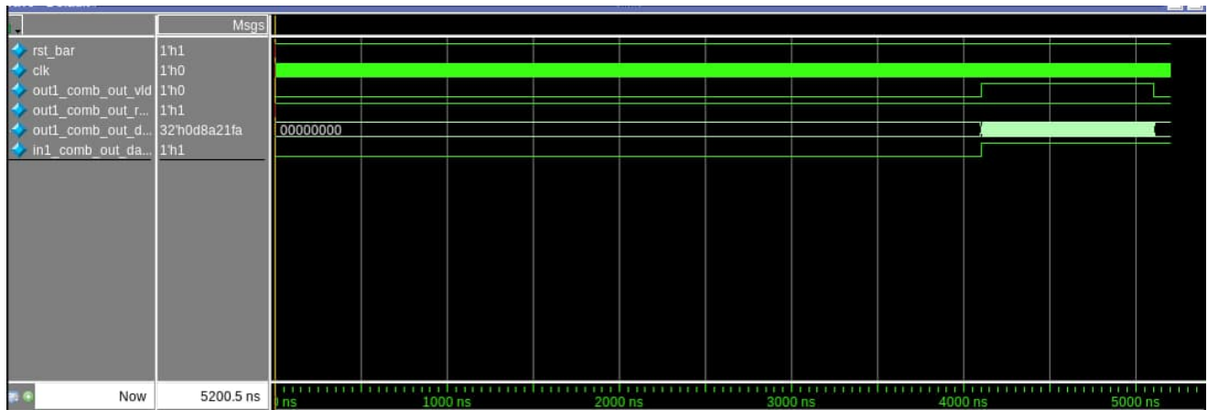
- The Scratchpad has these characteristics:
  1. The template parameters are the word_type (ie the element type that the overall array stores), the number of banks, and the total capacity of the banked memory in words.
  2. Each request to the banked memory includes multiple inputs. The number of inputs is equal to the number of banks. Each input is on its own "lane".
  3. Similarly, each response includes multiple outputs.
  4. One new request is consumed on each invocation.
  5. One new response is produced on each invocation if the requesting operation is a load (i.e. a read).
  6. The model can be pipelined with II=1 or any other desired II.
  7. There are no variations in delay since there is no arbitration or backpressure.

Using Scratchpad – Matchlib Example 41*

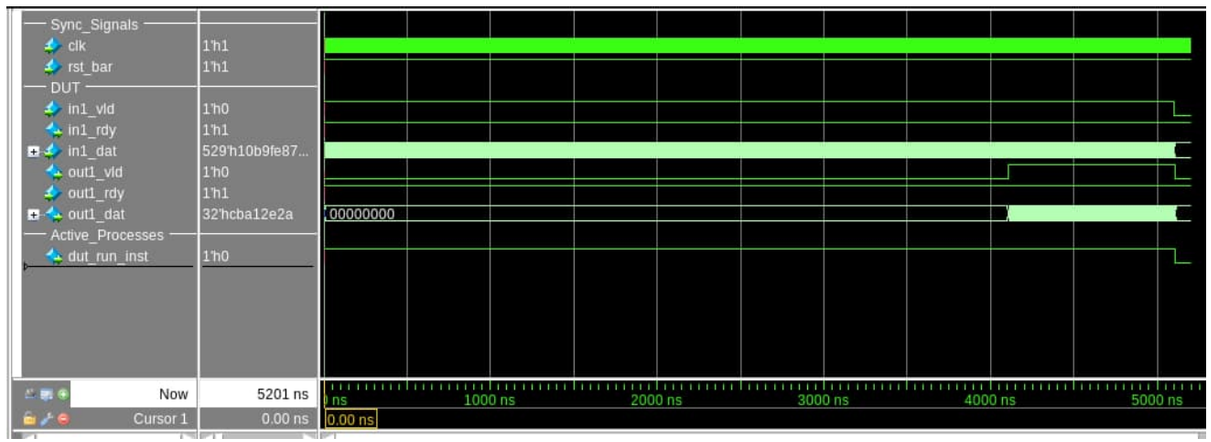
```
54   void run() {
55     in1.Reset();
56     out1.Reset();
57     wait();
58
59 #pragma hls_pipeline_init_interval 1
60 #pragma pipeline_stall_mode flush
61     while (1) {
62       // get the input request from the testbench
63       dut_in_t req1 = in1.Pop();
64
65       local_mem::scratchpad_req_t sp_req;  // local scratchpad request type
66
67       // copy incoming request to scratchpad request
68 #pragma hls_unroll yes
69       for (int i=0 ; i < local_mem::num_inputs; i++)
70         sp_req.set(i, req1.addr + i, req1.data[i]);
71
72       if (req1.is_load)
73       {
74         // if it is a load (i.e. read) operation, get the read data from the RAM
75         local_mem::base_rsp_t rsp = scratchpad1.load(sp_req);
76
77         // compute MAC
78         local_mem::word_type sum=0;
79 #pragma hls_unroll yes
80         for (int i=0; i < local_mem::num_inputs; i++) {
81           sum += rsp.data[i] * req1.data[i];
82         }
83
84         // Push out the sum
85         out1.Push(sum);
86       }
87       else
88       {
89         // if it is a store (i.e. write) operation, write the data to the RAM
90         scratchpad1.store(sp_req);
91       }
92     }
93   }
94 };
```

Scratchpad Simulation

Pre-HLS

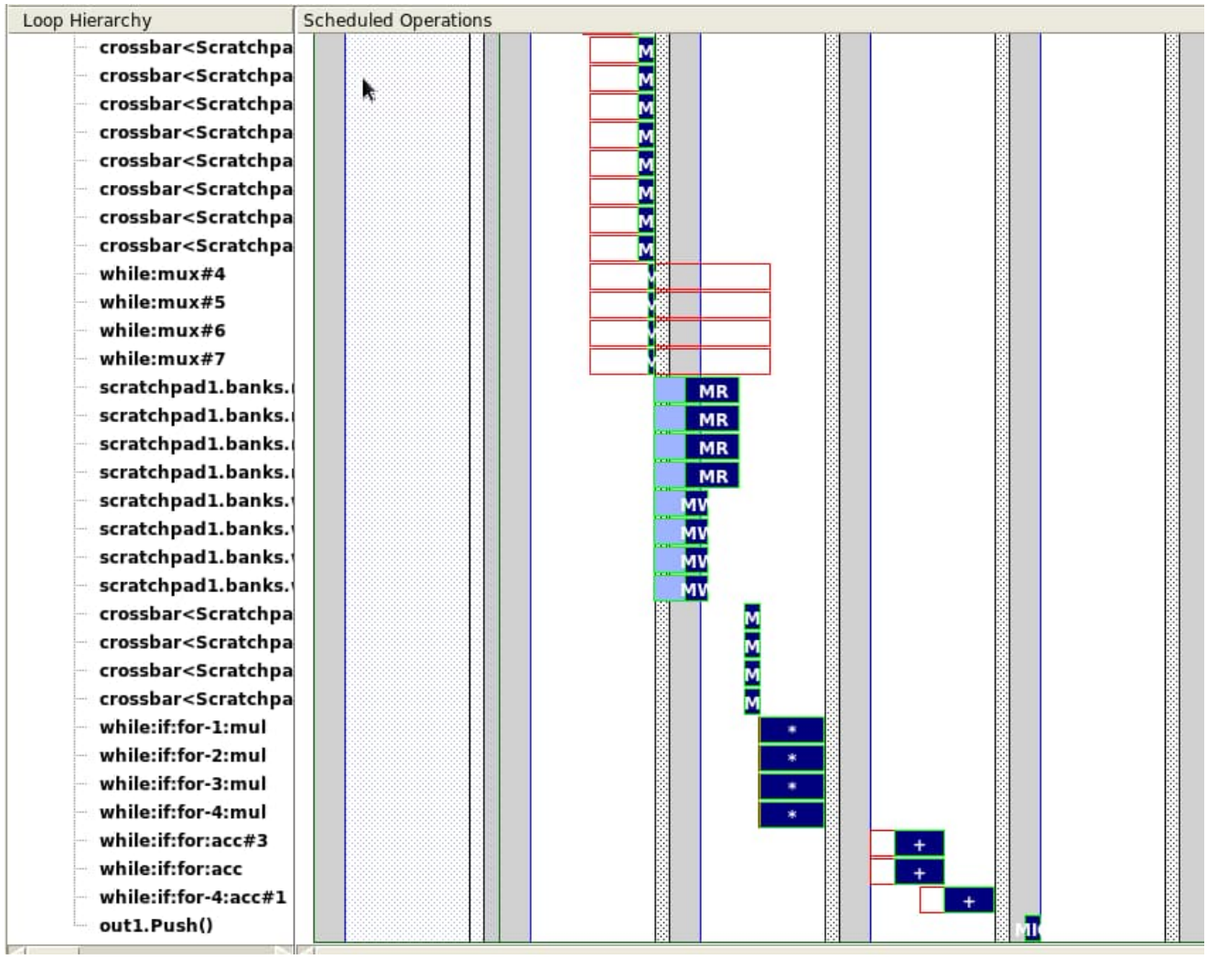


Post-HLS



SIEMENS

Scratchpad HW Schedule (Ex 41*, num_banks set to 4)



Note close correspondence to the schematic diagram shown for Example #1.

SIEMENS

ArbitratedScratchpad (Ex 42*) – When you really need bank arbitration

- Implements 2 crossbars with banked memory structure shown in design #1
- Has round-robin arbiters for requests for each bank
- Has queues for delayed requests

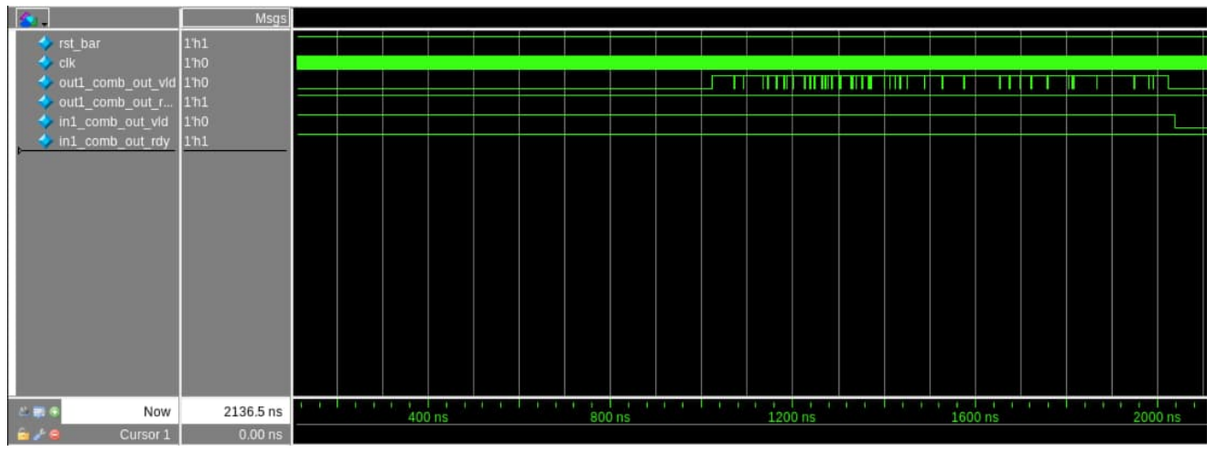
- The ArbitratedScratchpad has these characteristics:
  - Generally, any desired thruput can be achieved simply by increasing the number of banks in the memory and the number of inputs in each request
  - Generally, you want the number of banks to be greater (e.g. 4-16x) than number of inputs in each request. This reduces possible bank contention at minimal additional HW cost.
  - Bank contention is OK, so no error or warning message is emitted when it occurs.
  - Note the "was_consumed" flags in dut.h. These indicate if requests were accepted in previous call, if they were not consumed they need to be presented again to the memory.

ArbitratedScratchpad Usage

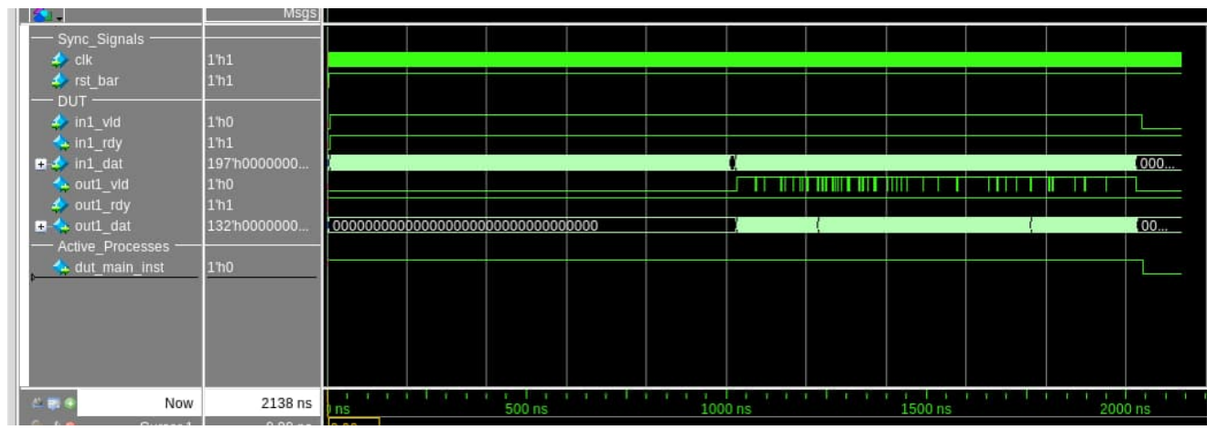
```
64   void main() {
65     out1.Reset();
66     in1.Reset();
67
68     bool was_consumed[local_mem::num_inputs];
69     bool all_consumed = 1;
70
71     wait();
72
73     local_mem::req_t req;
74     local_mem::rsp_t rsp;
75
76 #pragma hls_pipeline_init_interval 1
77 #pragma pipeline_stall_mode flush
78     while (1) {
79       // if there are remaining requests in some lanes that were not yet consumed
80       // then dont Pop a new input request
81       if (all_consumed)
82         req = in1.Pop();
83
84       mem.load_store(req, rsp, was_consumed);
85
86       bool any_valid = 0;
87       all_consumed = 1;
88 #pragma hls_unroll yes
89       for (unsigned i=0; i < local_mem::num_inputs; i++) {
90         // check if some of the requests were not consumed..
91         if (was_consumed[i] == 0)
92           all_consumed = 0;
93
94         // if a particular request was consumed, then it is no longer valid
95         if (was_consumed[i] == 1)
96           req.valids[i] = 0;
97
98         // check if we have any read (aka "load") responses
99         if (rsp.valids[i] == 1)
100          any_valid = 1;
101      }
102
103      if (any_valid)
104        out1.Push(rsp);
105
106      wait();
107    }
108  }
```

ArbitratedScratchpad Simulation

Pre-HLS



Post-HLS



Writes and reads are to random addresses, so bank conflicts occur at random times.

SIEMENS

ArbitratedScratchpad – Ordering Concerns

You should be aware of these ordering concerns when using ArbitratedScratchpad:

- Write requests to the same address on different lanes may not occur in the order that they are presented to the model due to the arbitration and queuing inside the model.
- Read and write requests to the same address on different lanes may not occur in the order that they are presented to the model due to the arbitration and queuing inside the model.
- In the two scenarios above, in all cases, the pre-HLS and post-HLS ArbitratedScratchpad designs will always behave identically (i.e. there will not be any differences).

These ordering issues are inherent to banked memories like this which support arbitration and queueing.

- They can be properly handled by appropriate synchronization and coordination at higher levels in the design.
- In example 42*, the testbench performs all writes to independent addresses, then flushes the writes (by sending several empty write requests), then proceeds to only issue read requests.
- This is one possible strategy for insuring expected behavior with this model.

What about the Catapult BLOCK_SIZE directive?

Scratchpad and ArbitratedScratchpad effectively implement the Catapult INTERLEAVE directive.

- If you instead want the higher order address bits to determine bank selection:
  - use ac_int::get_slc<> and ac_int::set_slc<> to form a new address with the bank selection bits located as the low order bits in a transformed address
  - Present this transformed to Scratchpad and ArbitratedScratchpad.

What about the Catapult WORD_WIDTH directive?

We do not recommend using WORD_WIDTH directive in Matchlib models.

Instead, use Scratchpad where word_type should be the same as the data type used for the original array accesses in the pre-HLS model.

- The Scratchpad num_banks parameter should be: WORD_WIDTH / number_of_bits(word_type)
- Here, WORD_WIDTH is the number of bits that would have been used in the Catapult WORD_WIDTH directive.

Leveraging C++ SW Development Best Practices

Stuart Swan
Platform Architect
Siemens EDA
18 January 2024

SIEMENS

Leveraging C++ SW Development Best Practices

- HLS/HLV ASIC projects are large-scale SW development projects
- Important to leverage best practices from C++ SW development world.
- General principles
  - Integrate early and often
  - Stress test models, including at highest level of abstraction possible
  - Find and fix bugs/issues at highest level of abstraction possible
  - Take advantage of C++ SW development and testing tools
  - Testbenches are a critical part of project, rigorous checking needs to include them.
  - Code defensively (e.g. use assert(x))

C++ Tools to Consider Using

- Vscode, Eclipse IDEs
  - Helpful for understanding C++ code base
  - Helpful for coding tips
  - May be some work to set up for a specific project
- ddd/gdb
  - Obviously useful for debugging
  - But, also useful for understanding C++ code base using commands such as **whatis, ptype, break, run, next**
  - Requires no work to set up, all you need to do is compile your pre-HLS model with g++ –g option
- Chatgpt, MS copilot, github copilot, etc.
  - Given code snippets and compiler error messages, etc., can provide very helpful guidance.

Best Practices at Start of Project

- Source code control + automated regression testing + bug tracking
- Static checks:
  - g++ -Wall
  - CDesignChecker
- Dynamic checks:
  - assert(x) – i.e. the standard C++ assertion statement
    - Assert statements in SC DUT will be automatically generated into RTL if you include “ac_assert.h”
  - Always use array classes that check index violations in both pre-HLS sims and in RTL
    - e.g. ac_array_1D<>, ac_bank_array_2D, ac_shared_bank_array_2D, Scratchpad, ArbitratedScratchpad
  - self-checking TBs
  - CCOV and code coverage tools
  - AddressSanitizer (built into g++, very easy to enable, needs SystemC to use pthreads)
  - MemorySanitizer (requires use of clang++, some effort to set up, all source must be recompiled)
- Note that everything above also catches errors in TB (except CDesignChecker and CCOV)

Best Practices Soon after Start of Project

- Run Catapult on all blocks ASAP, and frequently thereafter.
  - Goal is not QOR, instead goal is to catch coding problems
  - If needed, can use any tech library (e.g. example nangate-45nm in Catapult release)
  - If needed, can use slower clock frequency
  - Verify that the generated RTL passes same tests as pre-HLS model
- Start full runs to placed gates when feasible
  - Goal is to enable multiple runs thru full flow prior to final tape-out
  - Helps identify issues early.
- Track QOR metrics for each block throughout project

Matchlib Verification Methodology

Stuart Swan
Platform Architect
Siemens EDA
18 January 2024

SIEMENS

Matchlib Verification Methodology

- General strategy:
  - Stress test the pre-HLS model using SystemC TB and/or SV UVM
  - Measure and meet functional coverage and code coverage metrics on pre-HLS model
  - Reuse pre-HLS tests on RTL
    - Measure and meet functional coverage metrics on RTL
    - Measure and meet RTL code coverage metrics
  - Focus of verification effort should be on the pre-HLS model
    - Much easier to debug than RTL model
      - True even if RTL were instead hand-written
    - Enables quick edit and debug turnaround.
    - If you are spending significant amounts of time analyzing and debugging issues in the RTL, something is wrong.
      - Verification engineers accustomed to RTL analysis and debug may need to be trained to focus on pre-HLS model.
      - Analogy: When RTL synthesis was adopted, analyzing and debugging the gate level netlist needed to be avoided.

Matchlib Pre-HLS Verification Methodology

- Pre-HLS testbench environments:
  - Matchlib models can use SystemC testbenches, and easily integrate C++ libraries and models (ex 42*)
  - Matchlib models can use SV and SV UVM testbenches (ex 45*)
  - Matchlib models can integrate with python and MATLAB to use them as test environments (ex 70*)

- Pre-HLS Verification techniques:
  - Use Matchlib throughput accurate modeling to assess system performance
  - Use random stimulus and Matchlib random stall injection to easily “stress test” the model
  - Use Matchlib channel logs to observe and debug channel IO
  - Use Matchlib memory logs to observe and debug memory accesses
  - Use Matchlib waveform generation to debug detailed timing issues
  - Use Matchlib latency and capacity annotation if needed to more closely match RTL performance (ex. 72*)
  - Use C++ debuggers such as vscode, eclipse, and gdb/ddd to do detailed debugging of Matchlib models
  - See Matchlib toolkit example 60*

Matchlib Post-HLS Verification Methodology

- Post-HLS testbench environments:
  - RTL models can use SystemC testbenches, and easily integrate C++ libraries and models (ex 42*)
  - RTL models can use SV and SV UVM testbenches

- Post-HLS Verification techniques:
  - Always use preserve_struct=true and –DFORCE_AUTO_PORT=Connections::DIRECT_PORT during HLS
    - Enables much easier debugging in RTL
  - Use EDA simulators and debuggers to analyze the RTL
  - Reuse pre-HLS tests on RTL
    - Measure and meet functional coverage metrics on RTL
    - Measure and meet RTL code coverage metrics
    - Use formal tools such as covercheck to exclude unreachable RTL from code coverage analysis
  - Can use `STALL_FLAG_SV` to automatically perform random stall injection in the RTL DUT
  - Can use Matchlib channel logs to observe and debug channel IO & compare to pre-HLS logs
  - Can use Matchlib memory logs to observe and debug memory accesses in RTL (ex 80*)

Introduction to Matchlib Channel Logs

References:
“Matchlib SOC Debug Tutorial”
Example 60*

Stuart Swan
Platform Architect
Siemens EDA
18 January 2024

SIEMENS

Introduction to Matchlib Channel Logs

- Matchlib supports automatic generation of channel logs
  - This is a simple and versatile way to analyze and debug pre-HLS simulations
  - Here we use example 60* to demonstrate.

```
230   channel_logs logs;
231 #ifdef CONN_RAND_STALL
232   logs.enable("stall_log");
233 #else
234   logs.enable("no_stall_log");
235 #endif
236   logs.log_hierarchy(top);
```

- This code is placed in your sc_main or top module constructor.
- The enable() call is provided the name of the logs to generate
  - Optional “true” second arg can be used for unbuffered output, useful for debugging sim crashes or hangs.
- The log_hierarchy() call specifies the module hierararchy you want to log (does not need to be top).

Two Log Files Generated:

no_stall_log_names.txt:

```
1 top.fabric1.axi_arbiter0.comb_ba_0
2 top.fabric1.axi_arbiter0.comb_ba_1
3 top.fabric1.axi_arbiter0.comb_ba_2
4 top.fabric1.axi_arbiter0.comb_ba_3
5 top.fabric1.axi_arbiter1.comb_ba_0
6 top.fabric1.axi_arbiter1.comb_ba_1
7 top.fabric1.axi_arbiter1.comb_ba_2
8 top.fabric1.axi_arbiter1.comb_ba_3
9 top.fabric1.dma0_r_master0_out__ar_comb_out_dat
10 top.fabric1.dma0_r_master0_out__r_comb_out_dat
11 top.fabric1.dma0_w_master0_out__aw_comb_out_dat
12 top.fabric1.dma0_w_master0_out__w_comb_out_dat
13 top.fabric1.dma0_w_master0_out__b_comb_out_dat
14 top.fabric1.dma0_r_slave0_in__ar_comb_out_dat
15 top.fabric1.dma0_r_slave0_in__r_comb_out_dat
16 top.fabric1.dma0_w_slave0_in__aw_comb_out_dat
17 top.fabric1.dma0_w_slave0_in__w_comb_out_dat
18 top.fabric1.dma0_w_slave0_in__b_comb_out_dat
19 top.fabric1.dma1_r_master0_out__ar_comb_out_dat
20 top.fabric1.dma1_r_master0_out__r_comb_out_dat
21 top.fabric1.dma1_w_master0_out__aw_comb_out_dat
22 top.fabric1.dma1_w_master0_out__w_comb_out_dat
23 top.fabric1.dma1_w_master0_out__b_comb_out_dat
24 top.fabric1.dma1_r_slave0_in__ar_comb_out_dat
25 top.fabric1.dma1_r_slave0_in__r_comb_out_dat
26 top.fabric1.dma1_w_slave0_in__aw_comb_out_dat
27 top.fabric1.dma1_w_slave0_in__w_comb_out_dat
```

Channel Name Index

Channel Name

no_stall_log_data.txt:

```
63 | id{0x0} addr{0x0} burst{} len{0x0} size{} cache{} auser{}  | 6 ns
64 | data{0x0} last{0x1} wstrb{0x0FF} wuser{}  | 6 ns
16 | id{0x0} addr{0x0} burst{} len{0x0} size{} cache{} auser{}  | 7 ns
17 | data{0x0} last{0x1} wstrb{0x0FF} wuser{}  | 8 ns
18 | id{0x0} resp{0x0} buser{}  | 9 ns
65 | id{0x0} resp{0x0} buser{}  | 10 ns
63 | id{0x0} addr{0x08} burst{} len{0x0} size{} cache{} auser{}  | 11 ns
64 | data{0x2000} last{0x1} wstrb{0x0FF} wuser{}  | 11 ns
16 | id{0x0} addr{0x08} burst{} len{0x0} size{} cache{} auser{}  | 12 ns
17 | data{0x2000} last{0x1} wstrb{0x0FF} wuser{}  | 13 ns
18 | id{0x0} resp{0x0} buser{}  | 14 ns
65 | id{0x0} resp{0x0} buser{}  | 15 ns
63 | id{0x0} addr{0x10} burst{} len{0x0} size{} cache{} auser{}  | 16 ns
64 | data{0x0F} last{0x1} wstrb{0x0FF} wuser{}  | 16 ns
16 | id{0x0} addr{0x10} burst{} len{0x0} size{} cache{} auser{}  | 17 ns
17 | data{0x0F} last{0x1} wstrb{0x0FF} wuser{}  | 18 ns
18 | id{0x0} resp{0x0} buser{}  | 19 ns
65 | id{0x0} resp{0x0} buser{}  | 20 ns
63 | id{0x0} addr{0x080000} burst{} len{0x0} size{} cache{} auser{}  | 21 ns
64 | data{0x2000} last{0x1} wstrb{0x0FF} wuser{}  | 21 ns
26 | id{0x0} addr{0x0} burst{} len{0x0} size{} cache{} auser{}  | 22 ns
27 | data{0x2000} last{0x1} wstrb{0x0FF} wuser{}  | 23 ns
28 | id{0x0} resp{0x0} buser{}  | 24 ns
65 | id{0x0} resp{0x0} buser{}  | 25 ns
63 | id{0x0} addr{0x080008} burst{} len{0x0} size{} cache{} auser{}  | 26 ns
64 | data{0x082000} last{0x1} wstrb{0x0FF} wuser{}  | 26 ns
26 | id{0x0} addr{0x08} burst{} len{0x0} size{} cache{} auser{}  | 27 ns
27 | data{0x082000} last{0x1} wstrb{0x0FF} wuser{}  | 28 ns
28 | id{0x0} resp{0x0} buser{}  | 29 ns
65 | id{0x0} resp{0x0} buser{}  | 30 ns
63 | id{0x0} addr{0x080010} burst{} len{0x0} size{} cache{} auser{}  | 31 ns
64 | data{0x0F} last{0x1} wstrb{0x0FF} wuser{}  | 31 ns
26 | id{0x0} addr{0x10} burst{} len{0x0} size{} cache{} auser{}  | 32 ns
27 | data{0x0F} last{0x1} wstrb{0x0FF} wuser{}  | 33 ns
28 | id{0x0} resp{0x0} buser{}  | 34 ns
65 | id{0x0} resp{0x0} buser{}  | 35 ns
63 | id{0x0} addr{0x18} burst{} len{0x0} size{} cache{} auser{}  | 36 ns
```

Channel Name Index

Transaction Data

Simulation Time

Using gen_logs.sh script

- Use gen_logs.sh script to generate separate log files for each channel:

```
[sswan@orw-sswan7-rh 60_rand_stall]$ ../bin/gen_logs.sh no_stall_log
Processing top.fabric1.axi_arbiter0.comb_ba_0
Processing top.fabric1.axi_arbiter0.comb_ba_1
Processing top.fabric1.axi_arbiter0.comb_ba_2
Processing top.fabric1.axi_arbiter0.comb_ba_3
Processing top.fabric1.axi_arbiter1.comb_ba_0
Processing top.fabric1.axi_arbiter1.comb_ba_1
Processing top.fabric1.axi_arbiter1.comb_ba_2
Processing top.fabric1.axi_arbiter1.comb_ba_3
Processing top.fabric1.dma0_r_master0_out__ar_comb_out_dat
Processing top.fabric1.dma0_r_master0_out__r_comb_out_dat
Processing top.fabric1.dma0_w_master0_out__aw_comb_out_dat
```

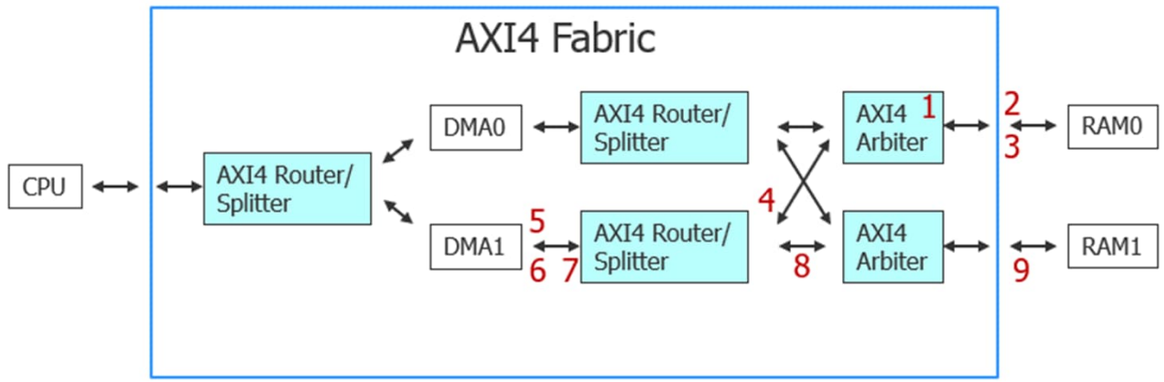
- This generates a “data” and “times” file for each channel:

```
id{0x0} addr{0x2000} burst{} len{0x1} size{} cache{} auser{}
id{0x0} addr{0x2010} burst{} len{0x1} size{} cache{} auser{}
id{0x0} addr{0x2020} burst{} len{0x1} size{} cache{} auser{}
id{0x0} addr{0x2030} burst{} len{0x1} size{} cache{} auser{}
id{0x0} addr{0x2040} burst{} len{0x1} size{} cache{} auser{}
id{0x0} addr{0x2050} burst{} len{0x1} size{} cache{} auser{}
id{0x0} addr{0x2060} burst{} len{0x1} size{} cache{} auser{}
id{0x0} addr{0x2070} burst{} len{0x1} size{} cache{} auser{}
~
~
"top.fabric1.d0_a0_w__aw_comb_out_dat.txt"
```

```
42 ns
55 ns
62 ns
69 ns
76 ns
83 ns
90 ns
97 ns
~
"top.fabric1.d0_a0_w__aw_comb_out_dat.txt.times"
```

Channel Logs Are Versatile Verification Tool

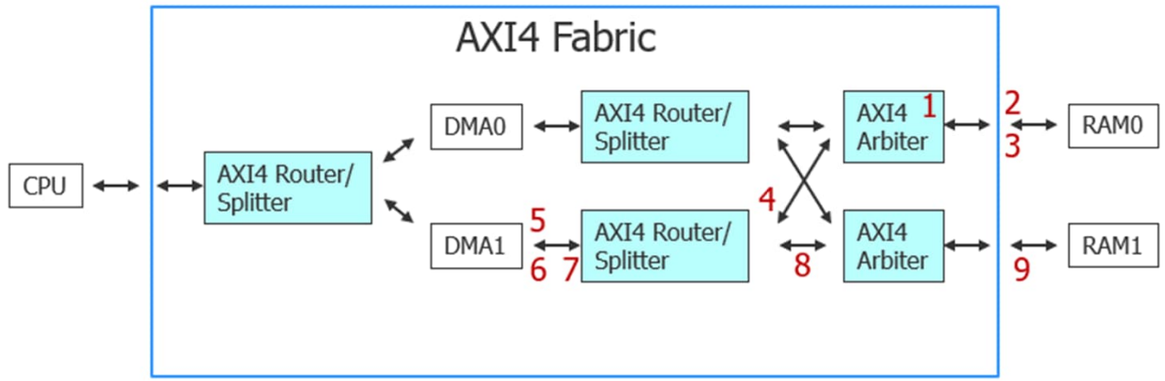
- Example 60* has an intentional bug in testbench.
- The bug is a “race” involving writes and reads from shared RAM0.
- The bug only occurs when the system is run with random stall injection.
- Scenario is described in detail in “matchlib_soc_debug_tutorial.pdf”.



Hunting for Bugs with Channel Logs

- With random stall injection, transaction times differ between the no_stall and stall scenarios.
- But transaction order and data should not differ in general between two scenarios.
- We can use earliest_diff.sh script to hunt for true differences:

```
./bin/earliest_diff.sh no_stall_log stall_log

-----------------
First 10 channels with differences, format is:
no_stall_log time | stall_log time | file_name

55 98 top.fabric1.axi_arbiter0.comb_ba_0.txt
55 98 top.fabric_r_master0__ar_comb_out_dat.txt
56 102 top.fabric_r_master0__r_comb_out_dat.txt
62 113 top.fabric1.d1_a0_r__r_comb_out_dat.txt
63 114 top.fabric1.dma1_r_master0_out__r_comb_out_dat.txt
64 120 top.fabric1.dma1.w_segment0_w_chan_comb_BA.txt
65 121 top.fabric1.dma1_w_master0_out__w_comb_out_dat.txt
66 123 top.fabric1.d1_a1_w__w_comb_out_dat.txt
67 128 top.fabric_w_master1__w_comb_out_dat.txt

-----------------

Earliest differences :
< is no_stall_log/top.fabric1.axi_arbiter0.comb_ba_0.txt
> is stall_log/top.fabric1.axi_arbiter0.comb_ba_0.txt

3d2
< 0x0
10d8
< 0x1
16a15,16
> 0x1
> 0x0
-----------------

Finished difference summary
```



Customize Your Usage of Matchlib Channel Logs

- Channel logs provide easy and comprehensive observability of your simulations
- Develop your own scripts to aid in debug, analytics, etc.
- Simple text file formats make it easy to process log information in bash, python, etc.

Introduction to Matchlib Memory Logs

References:
doc/memory_logging_and_debug.pdf
Example 60*

Stuart Swan
Platform Architect
Siemens EDA
18 January 2024

SIEMENS

Introduction to Matchlib Memory Logs

- Matchlib supports automatic generation of memory logs
  - This is a simple and versatile way to analyze and debug memories in pre-HLS simulations
  - The memory logs track the write and read accesses to RAMs
  - Here we use example 60* to demonstrate.
  - Detailed information is in “memory_logging_and_debug.pdf”

Memory Logs

- Memory logs are enabled by using the extended_array class
- This class overloads operator[] to log all writes and reads to a file.
- The format of the log files is:
  - `Address  write_count  value`
- For example, a log file of read accesses might have the line:
  - `0000040e 2 0000000000000070`
  - This indicates that address 0x40e was read and the read data was 0x070, and this data was a result of the 2nd write operation to that location

- Array accesses can sometimes be reordered. Sometimes this OK, sometimes it is a bug.
  - OK: 2 threads read from the same location in a shared memory, but the order of their reads may vary due to arbitration delays in the system. In all cases they read consistent values.
  - Bug: Thread 1 writes values 1,2,3 to location 0xf. Thread 2 sometimes reads values 1,2,3 from same location, but sometimes reads values 1,2,2
  - OK: HLS reorders memory read accesses when it pipelines a loop to enable better throughput

- Memory logs can be sorted based on the address without losing information. (sort –k ram0_read.log)
- This enables powerful debugging even in presence of various reordering scenarios.

Using Memory Logs to Debug Example 60*

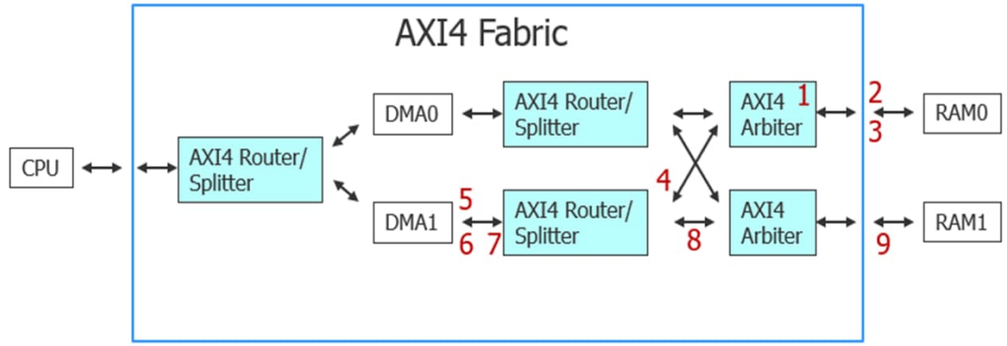
top.ram0_no_stall_read.log

```
1 00000000 1 0000000000000000
2 00000001 1 0000000000000008
3 00000002 1 0000000000000010
4 00000003 1 0000000000000018
5 00000004 1 0000000000000020
6 00000005 1 0000000000000028
7 00000400 2 0000000000000000
8 00000401 2 0000000000000008
9 00000006 1 0000000000000030
10 00000007 1 0000000000000038
11 00000402 2 0000000000000010
12 00000403 2 0000000000000018
13 00000008 1 0000000000000040
14 00000009 1 0000000000000048
15 00000404 2 0000000000000020
16 00000405 2 0000000000000028
17 0000000a 1 0000000000000050
18 0000000b 1 0000000000000058
19 00000406 2 0000000000000030
20 00000407 2 0000000000000038
21 0000000c 1 0000000000000060
22 0000000d 1 0000000000000068
23 00000408 2 0000000000000040
24 00000409 2 0000000000000048
25 0000000e 1 0000000000000070
26 0000000f 1 0000000000000078
27 0000040a 2 0000000000000050
28 0000040b 2 0000000000000058
29 0000040c 2 0000000000000060
30 0000040d 2 0000000000000068
31 0000040e 2 0000000000000070
32 0000040f 2 0000000000000078
33 00000000 1 0000000000000000
34 00000400 2 0000000000000000
35 00000001 1 0000000000000008
36 00000401 2 0000000000000008
37 00000002 1 0000000000000010
38 00000402 2 0000000000000010
```

top.ram0_stall_read.log

```
1 00000000 1 0000000000000000
2 00000001 1 0000000000000008
3 00000002 1 0000000000000010
4 00000003 1 0000000000000018
5 00000400 1 0000000000002000
6 00000401 1 0000000000002008
7 00000004 1 0000000000000020
8 00000005 1 0000000000000028
9 00000402 1 0000000000002010
10 00000403 1 0000000000002018
11 00000006 1 0000000000000030
12 00000007 1 0000000000000038
13 00000404 1 0000000000002020
14 00000405 1 0000000000002028
15 00000008 1 0000000000000040
16 00000009 1 0000000000000048
17 0000000a 1 0000000000000050
18 0000000b 1 0000000000000058
19 00000406 1 0000000000002030
20 00000407 1 0000000000002038
21 0000000c 1 0000000000000060
22 0000000d 1 0000000000000068
23 00000408 1 0000000000002040
24 00000409 1 0000000000002048
25 0000040a 1 0000000000002050
26 0000040b 1 0000000000002058
27 0000040c 1 0000000000002060
28 0000040d 1 0000000000002068
29 0000040e 1 0000000000002070
30 0000040f 1 0000000000002078
31 0000000e 1 0000000000000070
32 0000000f 1 0000000000000078
33 00000000 1 0000000000000000
34 00000400 2 0000000000000000
35 00000001 1 0000000000000008
36 00000401 2 0000000000000008
37 00000002 1 0000000000000010
38 00000402 2 0000000000000010
```

Comparing file file:///home/sswan/matchlib_examples_kit_..._examples/examples/60_rand_stall/top.ram0_stall_read.log 1 of 7 differences. 0 applied 1 of 1 file



The left side is the correct behavior, and the right side is the "stall" incorrect behavior.

All the reads starting at indexes 0x400 and higher represent the second copy operation in the test scenario. We see on the left that those read operations all read data with "write_counts" that are consistently 2. However, on the right side, we see the incorrect behavior where those reads are occurring where some write_counts are still 1. This is the root cause of the bug - it is a clear indication that the reads from the second copy operation are occurring too soon in the stall scenario.

Catapult HLS Scheduling Rules

Stuart Swan
Platform Architect
Siemens EDA
31 Oct 2024

SIEMENS

Catapult HLS Scheduling Rules

- Catapult synthesizes all processes in a design independently.
- The IO operations that processes have need to be scheduled precisely so the overall design works as expected in the post-HLS model.
- The types of IO operations are:
  1. Synchronization interface calls (wait(), SyncChannel calls)
  2. Signal IO (sc_signal read and write operations)
  3. Message passing operations (Push/Pop/PushNB/PopNB)
- The basic conceptual model is:
  1. Synchronization interface calls always remain in source code order.
  2. Signal read operations occur at closest preceding synchronization interface call.
  3. Signal write operations occur at closest succeeding synchronization interface call.
  4. All message passing calls after a synchronization interface will not start until the synchronization call has completed.
  5. All message passing calls before a synchronization interface call will be completed when the synchronization interface call has completed.

Catapult HLS Scheduling Rules (continued)

- The spec below specifies the rules in detail.
  - https://github.com/Stuart-Swan/Matchlib-Examples-Kit-For-Accellera-Synthesis-WG/blob/master/matchlib_examples/doc/catapult_user_view_scheduling_rules.pdf
- This spec has been proposed to Accellera as part of the SystemC synthesis standard.

Catapult HLS User View Scheduling Rules

Stuart Swan, Platform Architect, Siemens EDA

4 October 2024

Introduction

This document describes the user view scheduling rules for C++ and SystemC models that are synthesized through Catapult HLS. The goals of this document are:

1. To provide easy-to-understand rules to HLS users.
2. To ensure precise and consistent rules in both SystemC and C++.
3. To offer rules that are effectively compatible with how Catapult currently operates.
4. To enable the best possible *quality of results* (QOR) in Catapult synthesis.
5. To cover all known user requirements and scenarios.
6. To serve as a suitable starting point for a standardization proposal (e.g., in Accellera SWG).

To illustrate the goals of this document more specifically, consider what an engineer writing a testbench for an HLS model needs to understand about how HLS tools operate. This engineer may be using SystemVerilog UVM or may be writing a testbench in C++/SystemC. They likely are not an expert in any specific HLS tool (and may not want to be), but their testbench needs to work for both the pre-HLS model as well as the post-HLS model. Thus, it is crucial for the DV engineer to have a precise understanding of how the HLS tool will transform the design while still enabling it to be fully verified. This document describes what transformations the HLS tool is allowed to perform so that the pre-HLS and post-HLS models can be effectively verified with the same testbench. The overarching philosophy of the scheduling rules is to present “no surprises” to such a DV engineer, while still giving the HLS tool ample freedom to optimize the design.

Scheduling Rules: Modeling Guidelines Summary

1. Prefer to use Connections::In/Out + SyncChannel over signal IO and wait().
2. Prefer to use Pop()/Push() over PopNB()/PushNB().
3. When pipelining a loop, prefer to use `hls_stall_mode flush`.

When using signal IO:

1. If modeling a cycle-accurate process, follow example 52_apb style and do not use Push/Pop in the process.
2. If modeling a *direct input*, use `hls_direct_input` and possibly `hls_direct_input_sync`, and only change the signal at allowed times.
3. If combining signal IO with Connections::In/Out in same process, use proper signal IO handshake that does not rely on In/Out ports for process synchronization. Place signal IO operations very close to their wait() statements.
4. Unless you are modeling a cycle-accurate process, you should expect that latency will change between the pre-HLS and post-HLS models.