Matchlib Latency and Capacity Back-Annotation
Stuart Swan, Platform Architect, Siemens EDA
stuart.swan@siemens.com
9 December 2022

This document introduces the Matchlib latency and capacity back-annotation feature and provides some guidelines on usage.

Matchlib enables "throughput accurate" simulation of pre-HLS models. Matchlib throughput accuracy is high for most common design patterns, including pipelines, arbitration/contention, explicit synchronization between processes, etc. For large, real-world designs, Matchlib simulation accuracy is often just a few percent different from RTL simulation accuracy. For example, see:

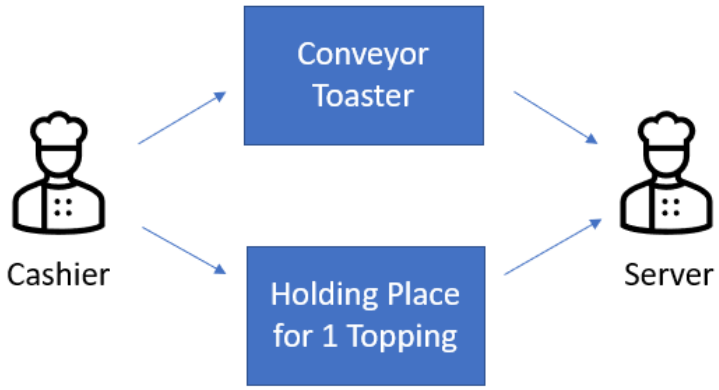
https://research.nvidia.com/sites/default/files/pubs/2018-06_A-Modular-Digital//dac2018.submitted.pdf

There are some cases, however, where the default accuracy of the pre-HLS simulation is lower since Matchlib does not have enough information about how HLS transforms the design. HLS typically adds latency and capacity when it pipelines processes. By default, Matchlib is unaware of the latency and capacity that may be added by HLS, so the pre-HLS simulation, while throughput accurate in a general sense, will not account for this latency and capacity added by HLS.

In some cases, this can have noticeable impact on the accuracy of the pre-HLS simulation. An example where accuracy is lost is when design stuttering occurs in the post-HLS design due to diverging and reconverging transaction streams with unbalanced latency and capacities. Let's first look at a design example which exhibits this stuttering behavior, and then we will review some general guidelines on using the Matchlib back-annotation feature.

The Bagel Shop Example

Imagine you are in line at a bagel shop and all the customers are purchasing toasted bagels with their selection of a particular topping. There is a cashier that takes your order for your bagel and topping. The cashier places the bagel into the toaster, and places your selected topping into a holding area that has space for one topping. The conveyor belt toaster then toasts your bagel, and there is a second person that takes your bagel out of the toaster, spreads your topping on it, and serves it to you.



The conveyor toaster takes 60 seconds to toast bagels, but a new bagel can be put in every 10 seconds by the cashier as the internal conveyor built moves along, and the cashier is able to service a new customer every 10 seconds.

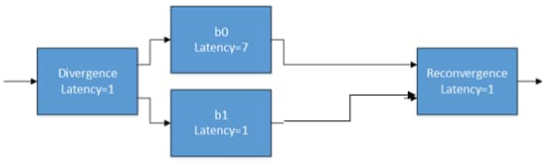
What happens in this situation, however, is that after the cashier services the first customer, that customer's topping is sitting in the holding spot for 60 seconds while the bagel toasts. The cashier cannot finish servicing the second customer since he has no place to put the second customer's topping.

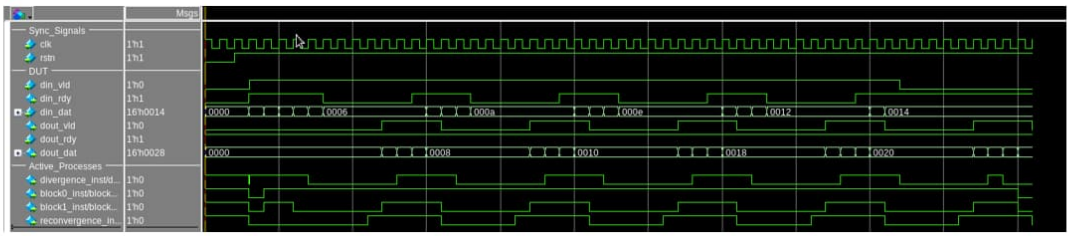
After 60 seconds, the server takes the bagel from the toaster and the first customer's topping, and then the cashier finally has a place to put the second customer's topping.

This system performance problem is referred to as "stuttering". In this case it arises because of the diverging and reconverging transaction streams, where the different pathways have unbalanced latencies and capacities. Once these sorts of problems have been identified, they are easily fixed – simply add additional capacity to the path that has insufficient capacity. (In this case, that means adding 5 more holding places for toppings).

The corresponding Matchlib example to the scenario above is example 72* in the Catapult HLS Matchlib examples. The block diagram and post-HLS waveforms are shown below. Note the stuttering behavior.

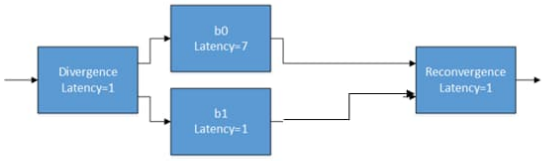
- II=1 for all blocks
- Latency of b0 is 1 pre-HLS (by default)
- Latency of b0 is 7 post-HLS
- Post-HLS waveforms below show stuttering due to reconverging data streams with unbalanced latencies and capacities.

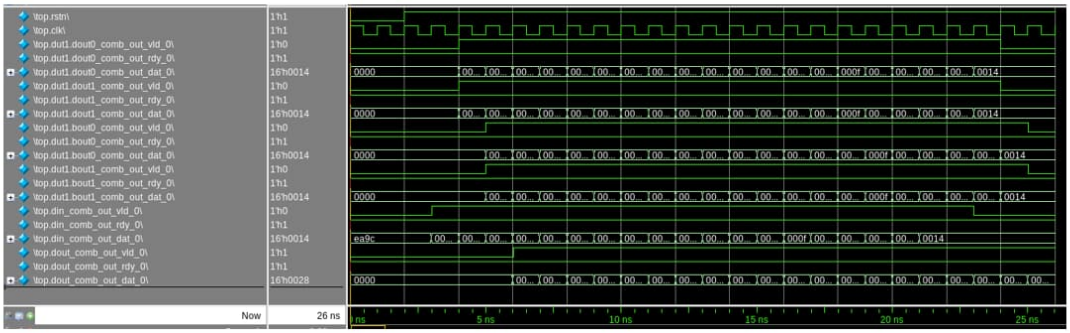




In the pre-HLS simulation, by default the latency and capacity of all blocks are one, so the stuttering behavior does not appear:

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





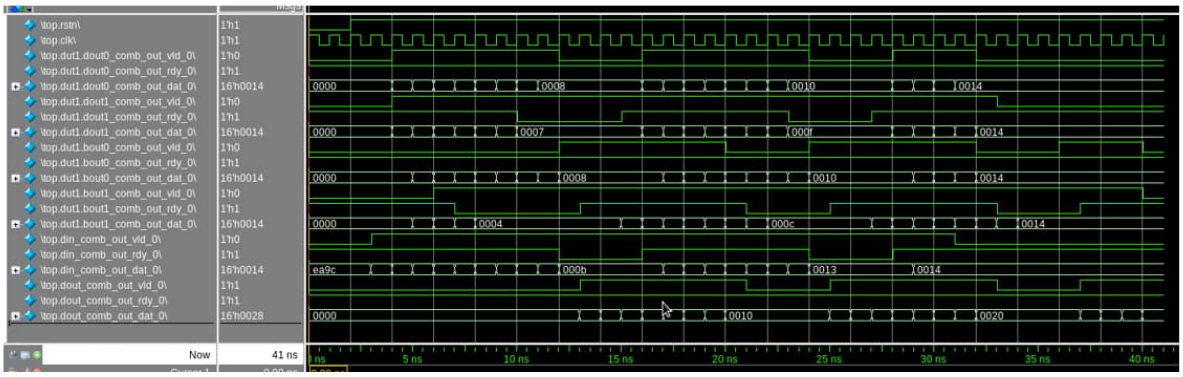
We can account for the latency and capacity that HLS introduces into the post-HLS model by using the Matchlib back-annotation feature. This lets you specify the latency and capacity numbers via a "json" text file. The latency and capacity for block0_inst is specified as follows:

```
"dut1.bout0_comb_BA": {
        "latency": 8,
        "capacity": 8,
        "src_name": "dut1.block0_inst.dout_vld",
        "dest_name": "dut1.reconvergence_inst.din0_vld"
    },
```

With this back-annotation, the stuttering behavior now also appears in the pre-HLS simulation:

- 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"
},
```



The most accurate way to observe the stuttering is in the log that the TB prints out. This log is printed out in both the pre-HLS and post-HLS sims so it is a good comparison for the overall accuracy of the two simulations. This is the pre-HLS log. The post-HLS log looks quite similar.

```
8 ns testbench STIM Pushed: 6
9 ns testbench STIM Pushed: 7
10 ns testbench STIM Pushed: 8
15 ns testbench RESP Popped: 0x2
16 ns testbench RESP Popped: 0x4
17 ns testbench RESP Popped: 0x6
17 ns testbench STIM Pushed: 9
18 ns testbench RESP Popped: 0x08
18 ns testbench STIM Pushed: a
19 ns testbench RESP Popped: 0x0A
19 ns testbench STIM Pushed: b
20 ns testbench RESP Popped: 0x0C
20 ns testbench STIM Pushed: c
21 ns testbench STIM Pushed: d
22 ns testbench STIM Pushed: e
27 ns testbench RESP Popped: 0x0E
28 ns testbench RESP Popped: 0x10
29 ns testbench RESP Popped: 0x12
29 ns testbench STIM Pushed: f
30 ns testbench RESP Popped: 0x14
30 ns testbench STIM Pushed: 10
31 ns testbench RESP Popped: 0x16
31 ns testbench STIM Pushed: 11
32 ns testbench RESP Popped: 0x18
32 ns testbench STIM Pushed: 12
33 ns testbench STIM Pushed: 13
39 ns testbench RESP Popped: 0x1A
40 ns testbench RESP Popped: 0x1C
41 ns testbench RESP Popped: 0x1E
42 ns testbench RESP Popped: 0x20
43 ns testbench RESP Popped: 0x22
44 ns testbench RESP Popped: 0x24
51 ns testbench RESP Popped: 0x26
```

Once we see the stuttering behavior exhibited in the pre-HLS model, we can now use the capacity annotation feature to test out the fix. In this case we need to add additional capacity to the path that goes through block1_inst. This is done by adding the following to the testbench.input.json file:

```
"dut1.bout1_comb_BA": {
    "latency": 1,
    "capacity": 8,
    "src_name": "dut1.block1_inst.dout_vld",
    "dest_name": "dut1.reconvergence_inst.din1_vld"
},
```

With this additional capacity along the b1 path, we now see the stuttering mostly gone:

```
9 ns testbench STIM Pushed: 7
```

```
10 ns testbench STIM Pushed: 8
11 ns testbench STIM Pushed: 9
12 ns testbench STIM Pushed: a
13 ns testbench STIM Pushed: b
14 ns testbench STIM Pushed: c
15 ns testbench RESP Popped: 0x2
15 ns testbench STIM Pushed: d
16 ns testbench RESP Popped: 0x4
16 ns testbench STIM Pushed: e
17 ns testbench RESP Popped: 0x6
17 ns testbench STIM Pushed: f
18 ns testbench RESP Popped: 0x08
18 ns testbench STIM Pushed: 10
19 ns testbench RESP Popped: 0x0A
19 ns testbench STIM Pushed: 11
20 ns testbench RESP Popped: 0x0C
20 ns testbench STIM Pushed: 12
21 ns testbench RESP Popped: 0x0E
21 ns testbench STIM Pushed: 13
22 ns testbench RESP Popped: 0x10
24 ns testbench RESP Popped: 0x12
25 ns testbench RESP Popped: 0x14
26 ns testbench RESP Popped: 0x16
27 ns testbench RESP Popped: 0x18
28 ns testbench RESP Popped: 0x1A
29 ns testbench RESP Popped: 0x1C
30 ns testbench RESP Popped: 0x1E
31 ns testbench RESP Popped: 0x20
```

With the fix tested out in the pre-HLS model, the final synthesizable fix might be to add a FifoModule<> along the b1 path with the needed capacity so that it is synthesized into the HW.

In this Matchlib example 72*, the latency for block b0 is specified via the go_hls.tcl script using a cycle constraint:

```
go assembly
go architect
cycle set dout.Push() -from din.Pop() -equal 7
```

This constraint only applies to block b0 and not to any other blocks in the design. All of the blocks in the design are pipelined with an II=1 and flushing enabled. To accurately reflect these circumstances in the testbench.input.json file, we should leave all values for latency and capacity as zero except for the Connections::Combinational<> channel which is attached to the output port of block0_inst:

```
"dut1.bout0_comb_BA": {
    "latency": 8,
    "capacity": 8,
    "src_name": "dut1.block0_inst.dout_vld",
    "dest_name": "dut1.reconvergence_inst.din0_vld"
},
```

Note that dut1 is the name of the DUT as instantiated in the testbench, and that bout0 is the name of the specific Connections::Combinational<> channel noted above.

In this case we set both the latency and capacity values to 8. The value is 8 because the "cycle set" command above sets the distance between the Push and Pop operations as 7 cycles, but we need to add an additional cycle for the Pop operation itself to the overall latency. The reason that the latency and capacity values are the same is that the loop is pipelined with an II=1, which means that the storage capacity of the pipeline is the same as its latency.

When you run the pre-HLS and post-HLS sims with back-annotation enabled as above, the stuttering behavior will quite accurately be reflected in the pre-HLS simulation, but the pre-HLS and post-HLS simulations won't exactly match cycle by cycle. You should not expect them to match exactly cycle by cycle.

Note that when backannotation is used there currently is a bug in the pre-HLS waveforms produced by Matchlib (into the ".vcd" file), so you should not rely on the waveform file being fully accurate. However, the overall simulation will be accurate as described in this document, and the simulation times that are seen in your TB and DUT will be accurate.

Setting Up Your Design for Back-Annotation

To enable back-annotation in your design add the following code to at the top your testbench file:

```
#ifndef __SYNTHESIS__
#ifndef CCS_SYSC
#include <connections/annotate.h>
#endif
#endif
```

Then, before calling the start() method that starts simulation in your testbench, call the following code:

```
#ifndef __SYNTHESIS__
#ifndef CCS_SYSC
Connections::annotate_design(testbench);
#endif
#endif
```

In the code above "testbench" is the top level sc_module in the overall system.

Before running your simulation, create an empty "json" file for backannotion, in this case named "testbench.input.json" :

```
{
    "channels": {
    }
}
```

Now, run your pre-HLS simulation. This will read your empty json file, and it will produce a "testbench.output.json" file that is fully populated with the names of all of the channels in your design that can be back-annotated. By default all of the latency and capacity values will be zero in this file. You can then copy the output file to "testbench.input.json" and modify any latency and capacity values you

want to change. There are many tools and libraries available for editing json files if you would like to automate the process.

Each time you run your pre-HLS simulation with back-annotation enabled it will read the “input” json file and produce a new “output” json file. All latency and capacity values (for channels still present in the design) will be copied from the input to output file, and any new channels that are added to the design will be added to the output file with default values of zero for the latency and capacity.

Guidelines for Usage of Back-Annotation

1. By default, leave all latency and capacity values as zero in your back-annotation file. This will have the same effect as not doing any back-annotation at all. If, instead, you back-annotated all latency and capacity values as one, this would have a different simulation behavior as compared to not performing any back-annotation at all. In particular, note that block b1 in example 72* above has an “overall” latency and capacity of one. However, in both the pre-HLS and post-HLS simulations, Matchlib and Catapult will give all sequential processes this default latency and capacity of one with no additional back-annotation required. Thus, back-annotating b1’s Combinational::Out<> port with a value of 1 or greater would give it an inaccurate simulation result as compared to what is desired.

2. Focus on annotating the latency and capacity numbers for Connections::Combinational<> channels that are driven by Connections::Out<> ports of interest. The goal is to determine the appropriate latency and capacity numbers for those particular Connections::Out<> ports and annotate those values onto their associated Connections::Combinational<> channels. For example, with example 72* above, we only annotate the long latency path representing the block0_inst, and we do this by annotating the channel on its Connections::Out port which is named bout0.

```
"dut1.bout0_comb_BA": {
        "latency": 8,
        "capacity": 8,
        "src_name": "dut1.block0_inst.dout_vld",
        "dest_name": "dut1.reconvergence_inst.din0_vld"
    },
```

3. In general, for pipelined designs with II=1, latency and capacity annotation should be the same. For II=2, capacity should be 1/2 the latency, for II=4 capacity should be 1/4 the latency, etc. (“II” means “initiation interval”.)

4. If the main loop of a process is not pipelined at all, then capacity should be set to one and the latency annotation should reflect the latency time to output the transaction from the process after receipt of its inputs.

5. Even with the use of Matchlib back-annotation with accurate latency and capacity values, you should not expect an exact match between the cycle level behavior of the pre-HLS and post-HLS models. Instead, you should expect that the throughput of all the various transaction

streams in the system will be accurate when comparing the pre-HLS and post-HLS models, even when complicating factors such as divergence and reconvergence are present.

6. Latency numbers reported by Catapult HLS in its reports may not be accurate in all cases. For example, if loop iterations may take a varying number of cycles depending on the input data to the process, then latency numbers reported by Catapult are likely to be inaccurate. However, typically loops are pipelined with an II=1, and have fixed latency between the corresponding Pop() and Push() operations in the post-HLS model, so the latency numbers in these typical cases can be relied on to be used as the back-annotated values.

7. Catapult reports latency and throughput numbers within its text reports. This information is also available by using command line scripting capabilities in Catapult. For example, the "cycle.rpt" file for example 72* shown above contains the following:

```
Processes/Blocks in Design
    Process                Real Operation(s) count Latency Throughput Reset Length II
    ---------------------- ----------------------- ------- ---------- ------------ --
    /top/reconvergence/run                       4       1          1            1  0
    /top/block1/run                              2       1          1            1  0
    /top/block0/run                              2       7          1            1  0
    /top/divergence/run                          3       1          1            1  0
    Design Total:                               11      10          1            1  0
```

   This information can be used to either manually or automatically update the "input.json" file described previously. Note that the hierarchical paths reported by Catapult do not exactly match the hierarchical paths used by the Matchlib back-annotation mechanism (the latter use the SystemC hierarchical instance names). When using these numbers, note the caveats about the latency numbers reported by Catapult mentioned earlier.

8. Note that currently the pre-HLS waveforms and "channel logs" generated by Matchlib when using the back-annotation feature are not accurate, and they should not be relied on. The overall simulation works as expected and is accurate as described in this document, and in particular explicitly added logging statements in the TB and DUT will accurately reflect timing behavior even when back-annotation is used.

9. The current Matchlib back-annotation mechanism allows both latency and capacity to be set to zero, but if either is non-zero then the other must also be non-zero.

10. If you have additional guidelines you think should be added to this document, or if you have additional examples that demonstrate the benefits of the back-annotation feature, feel free to contact me at the email address at the top of this document.