CSEE E6863: Hardware Project Report

Judicael S. E. Clair Botong Xiao

January 9, 2022

1 Introduction – ESP: The Open-Source SoC Platform [1]

The goal of this project was to formally verify various components of ESP [1], which is an open-source project that facilitates the design of heterogeneous System-on-Chips (SoCs). It is under active development by Prof Carloni's System-Level Design Group at Columbia University.

An SoC designed using ESP consists of tiles, such as memory controllers, CPUs, and custom accelerators. Tiles send data to one another through a custom network-on-chip (NoC), which is a packet switching network. That is, data is transmitted as packets that are forwarded to the intended destination tile using routers. ESP's NoC has a mesh topology so that routers are arranged as a 2D grid and each router has its own local tile as shown in figure 1.

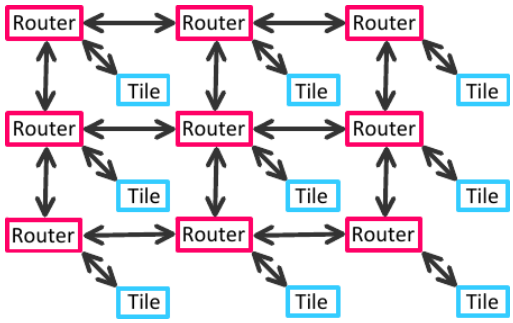

Figure 1: NoC with mesh topology.

The NoC has 6 bidirectional planes where each plane has a unique function. For example, plane #1 is used to send coherence requests (e.g. coherent memory requests from the CPU) and plane #3 deals with coherence responses. Naturally, the NoC uses a custom packet-based communication protocol. However, standard bus-based communication protocols, such as AXI4 [2], are typically used by third-party IP blocks to interface with other IP blocks. Accordingly, ESP has an AXI–NoC proxy [3], which is a bridge between the NoC and an AXI4 slave interface. This allows third-party IP to be effortlessly integrated into an ESP-based SoC.

2 Objectives

We chose to verify the AXI–NoC proxy for this project as it is a relatively small component with few dependencies. Furthermore, the main objective of this project was to explore the capabilities of the formal verification tool instead of trying to extensively verify as many components as possible. Indeed, there was plenty of opportunity to push the formal verification tool to its limits (and beyond) by exclusively verifying the AXI–NoC proxy in isolation.

3 Work Environment Setup

For this project, Mentor Questa Formal (MQF) was used. Additionally, the AXI–NoC proxy, which is the design-under-test (DUT), is written in VHDL. Nevertheless, we chose to use SystemVerilog Assertions (SVA) as we did not want to diverge too much from the workflow recommended by Dr Theobald.

Since the DUT has very few dependencies, we managed to isolate it from ESP's codebase and got it compiling and running with MQF. To this end, all constants, types, helper functions, etc, that were referenced by the DUT were copied into a new, standalone package `env_pkg`. Then, the DUT's package imports were modified so that the DUT only referenced `env_pkg` and a few standard packages, such as `ieee.numeric_std`. As a result, given only the `env_pkg`, the DUT could now be compiled standalone.

Given the standalone version of the DUT, it was binded to a SystemVerilog module that performed a dummy assertion. Specifically, the tautology `rst` || $\sim$`rst` was asserted; noting that only the `clk` and `rst` ports were connected. This was relatively straightfoward as inspiration was taken from the examples provided by MQF. That being said, VHDL code is imported entirely in lower case to SystemVerilog (e.g. constants defined in VHDL are named in lower case within SystemVerilog), which took us a while to figure out. Note also, most of the build system files given in the examples, namely the `Makefile`, were initially used. However, we eventually implemented our own build system, which was partly inspired by the original one but was now easier to use and no longer spewed build artefacts all over the place.

Several issues arose when we tried to verify the DUT itself. Firstly, generic parameters must be assigned a default value as, unlike with input ports, the tool uses a fixed, pre-defined value for generic parameters. However, there were several generic parameters that we wanted to vary, such as the address (`slv_y, slv_x`) of the I/O tile. To resolve this issue, generic parameters of interest that were not used in a constant expression context were simply made to be input ports. However, `nmst` and `mem_num` could not be converted to input ports as they are used to determine for loop upper bounds. Since `nmst` and `mem_num` were the only remaining generic parameter that must be varied, it was now feasible to instantiate the DUT multiple times, where each instance has a different combination of values for `nmst` and `mem_num`. Furthermore, since `nmst` and `mem_num` are exclusively used to define for loop upper bounds, it suffices to verify the DUT with `(nmst, mem_num)`$\in \{1, 2\}^2$. This ensures that the assertions have effectively been checked on the entirety of the state space reachable in practice; noting that these parameters are always greater than zero. Thus, only 4 instances of the DUT needed to be verified. Note, this approach would not have been feasible if there was instead several generic parameters of interest that must each be assigned a variety of values so as to fully explore the state space.

Instancing the DUT multiple times turned out to be very tricky. Firstly, at the interface between the DUT and the bounded SystemVerilog module, the tool does not support variable length arrays of non-trivial types. For instance, specifying `input axi_mosi_type [0:nmst-1] mosi` in the SystemVerilog module produced the error: "cannot create a packed array from unpacked elements." Instead, we must have `input array_of_axi_mosi_type mosi`. Consequently, if a DUT port is of a non-trivial array type, then its type must have concretely defined bounds and so a fixed size, which must be the same for all instances of the DUT. This meant modifying the DUT's interface, as well as defining aliases within the DUT so that the DUT's implementation is unaffected by these changes. Take for example the `mosi` port, which is an array of `axi_mosi_type`. Originally, its size is equal to `nmst` but it is now `max(nmst) = 2`. Accordingly, the `mosi` input port was renamed to `mosi_extended` and the `mosi` alias was defined, which restricts access to only the first `nmst` elements of `mosi_extended`; noting that the DUT's implementation uses `mosi` and not `mosi_extended`. Lastly, we had to create a top-level entity in VHDL that instantiates the DUT multiple times. Naturally, the top-level entity appropriately forwards all the signals between the DUT instances and the formal verification tool. Fortunately, the binding mechanism itself did not have to be changed as the tool automatically detects the top-level entity and its associated DUT instances.

4 Familiarisation with Code

The first step was to understand how the DUT is used in practice. Accordingly, we reverse engineered various parts of ESP and in particular, how the DUT is used by a processor, such as Ariane, which is a single-core 64-bit RISC-V processor (see `doc/axislv2noc.pdf`). Evidently, this included determining the purpose of each port and generic parameter of the DUT (see `doc/axislv2noc.pdf`). Thankfully, throughout this process, Prof Carloni and his team gave us a few pointers and confirmed our high-level understanding of the DUT and ESP in general. That being said, they did not provide any assistance with comprehending the DUT's implementation itself. Hence, it took us a considerable amount of time to understand every line of code in the DUT. In fact, we only obtained a comprehensive understanding of the DUT's internals once we analysed the counterexamples produced by many SVA assertions we erroneously thought would be proven. Lastly, to understand the DUT's implementation, it was evidently necessary for us to spend some time familiarising ourselves with the standard AXI4 protocol, which also allowed us to write meaningful assertions that ensure the DUT conforms to the aforesaid protocol.

5 General Verification Approach

After familiarising ourselves with SVA, which involved writing lots of basic assertions and assumptions, we realised that the behaviour of the NoC has to be modelled in order to verify interesting properties of the DUT. In particular, by observing the NoC signals, we should assume that if a read request is correctly sent to the NoC (i.e. conforms to the NoC specification), then a suitable reply is eventually received from the NoC. Assuming this behaviour can be modelled, then it is possible to verify that the DUT conforms to both the NoC and AXI protocols. For instance, suppose a user sends a read request using the AXI protocol to the DUT. Then, if the DUT is functioning correctly, a read request is sent to the NoC and an appropriate response is eventually received from the NoC. This response is then forwarded to the user in a manner that conforms to the AXI protocol. Thus, an interesting property to verify is that if an AXI read request is sent to the DUT, a suitable AXI reply is eventually received.

Aside from modelling the NoC and writing assertions that make use of this model, assertions that do not require such a model but are nevertheless critical to verify were also implemented. For instance, the DUT should never write to the NoC when the NoC is full, and never read from it when it is empty. Additionally, we also needed to implement several assumptions, such as assuming the bounds on generic parameters that had been converted to input ports, so as to constrain the search space and avoid bogus counterexamples. These assertions and assumptions are covered in more detail in "Overview of All Implemented Properties", section 7.

6 Modelling the NoC

Modelling the NoC involved implementing several assumptions (no assertions or cover properties). Firstly, it is reasonable to assume that the NoC will always eventually be able to service a request. For instance, as shown below, we added the assumption that the coherence request plane is always eventually not full,

```
assume property (@(posedge clk) (s_eventually ~coherence_req_full));
```

Now, as detailed in "General Verification Approach", section 5, we must assume that when a correct read request is sent to the NoC, an appropriate response is eventually sent back. Such an assumption takes the following form,

```
property prop;
    @(posedge clk) disable iff(~rst)
    (
      // sequence of events that constitute
      // a valid read request.
    )

    |->

    strong(
      // sequence of events that constitute
      // a valid response.
    )
endproperty : prop
assume_prop : assume property(prop());
```

6.1 Implication Operator

The overlapped implication operator `|->` is used here as when a match of the antecedent is found, the consequent is attempted to be matched starting from the **last** clock cycle of the matched antecedent sequence; thereby encoding the fact that the request must precede the response. Note, the `implies` operator cannot be used instead, as the consequent would be matched starting from the same clock cycle as for the antecedent sequence, which is clearly undesirable. Since SVA has so many operators that are seemingly alike but sometimes actually have very different characteristics, it made it very challenging at first to determine which operator to use and when. This confusion was exacerbated by the fact that it is very tedious to check that an assumption actually means what we think it means, and if it doesn't, it is difficult to figure out the reason why.

6.2 Strong & Weak Properties

Indeed, it took substantial troubleshooting to realise that the consequent needed to be a strong property. Specifically, without the use of the `strong()` operator, the consequent is by default a weak property. When a property is weak, its associated sequence is not required to complete, which is a very subtle but critical difference from a strong property. For example, the weak property `s[*0:$] ##0 v` would hold if `s` is high forever and `v` is always low. In this context however, we require that the consequent sequence always completes as a response must always eventually be fully sent back following a read request. It was very difficult to work out that the consequent needed to be a strong property as when it was weak, the relevant assertions were bizzarly firing or being proven.

6.3 Read Response (Assumption Consequent)

Unfortunately, when the consequent of an assumed property is strong, almost all the SVA operators cannot be used to define the consequent sequence as they are not supported by the tool in that context. Thus, it was extremely difficult to incorporate variability in the consequent. In particular, we not only wanted the response's payload length to be the same as the requested length, which is arbitrary, but also that the response's flits may be received non-consecutively, where a flit denotes an atomic chunk of data that constitutes a packet. We will first describe what didn't work and then what actually worked. Now, in its ideal form, the consequent is defined as follows,

```
strong(
    // DUT receives header flit.
    ##[1:$] (~rsp_empty)

    // DUT receives & forwards payload data.
    ##1 (~rsp_empty && axi_master_ready)[->payload_length]

    // Final flit must have its preamble set to 'preamble_tail',
    // which denotes that it is the last flit of the packet.
    ##0 (get_preamble(rsp_data) == preamble_tail)
)
```

where `axi_master_ready` indicates that the user that sent the read request is ready to read the next flit of the response, and `rsp_empty` is low when there is meaningful data in `rsp_data` (`rsp_empty` and `rsp_data` are NoC signals). Suppose a request was sent on the coherence request plane, then the response would be sent back via the coherence response plane, and so in this case we would have, `rsp_empty` $\equiv$ `coherence_rsp_rcv_empty`, `rsp_data` $\equiv$ `coherence_rsp_rcv_data_out`.

However, since we are using `[->payload_length]`, the standard requires `payload_length` to be a constant expression, which is clearly not suitable here as the payload length is arbitrary. Instead, we must use the following ingenious sequence, which is a slightly modified version of [4] as the original version implements consecutive variable repeat whereas we want it to be non-consecutive.

```
sequence dynamic_non_consecutive_repeat(s, x);
    int v = x;
    (1, v=x) ##0 first_match((1, v=v-s)[*1:$] ##0 v<=0);
endsequence
```

where `x` is the number of occurences of signal `s` being high. Then, in the consequent we would use,

```
dynamic_non_consecutive_repeat(~rsp_empty && axi_master_ready, payload_length)
```

instead of,

```
(~rsp_empty && axi_master_ready)[->payload_length]
```

Although the consequent is syntactically correct, the tool sadly does not support the following constructs in the context of the strong consequent of an assumed property:

- local variables: "Unsupported variable delay between local variable assign and read."
- `$` symbol: all operators that syntactically accept the `$` symbol failed to compile.
- non-consecutive repeat `[->n]`: "This form of fairness assumption is not supported."
- `##0`: "This usage of ##0 on the right hand of the implication is not supported."
- `first_match()`: "This usage of first_match on the right hand of an implication [..] is not supported."

Consequently, there is just no way of implementing non-consecutive variable repetition if we exclusively use the remaining SVA constructs that compile correctly; though, it may actually be possible in more recent versions of the verification tool assuming support for the necessary constructs has been added.

That being said, it is in fact possible to implement non-consecutive variable repetition by cleverly modifying the DUT implementation itself. That is, for each signal of interest, we added an auxiliary signal within the DUT that counts the number of occurences of the former signal within a desired time frame; noting that the counter is delayed by a single clock cycle with respect to its signal of interest. These counters are then used within the SVA properties as the value of a counter adequately encodes the past behaviour of the signal it is counting. This approach is somewhat analogous to how conflict driven learning adds conflict clauses to the set of clauses it is trying to satisfy. Specifically, the state space that the solver explores is modified instead of trying to make the solver itself more complicated. Accordingly, the final implementation of the consequent we used is similar to the following,

```
strong((
    // Only one of the NoC planes is used throughout this transaction.
    other_rsp_planes_not_empty_cnt == 0
) throughout (
    // Ensure that no data has yet been received from the NoC
    // since the start of the transaction (the counters are reset
    // from within the DUT at the beginning of every transaction).
    ##1 rsp_not_empty_cnt == 0

    // Within max_header_delay (predefined) clock cycles,
    // the DUT must have received the header flit.
    ##0 (~process_next_transaction &&
         get_preamble(rsp_data) == preamble_header &&
         rsp_not_empty_cnt == 0 &&
         ~axi_master_ready)
        [*1:max_header_delay]
    ##1 rsp_not_empty_cnt == 1

    // Within max_payload_delay (predefined) clock cycles, the DUT must
    // have received exactly payload_length flits (excluding the header flit).
    ##0 (~process_next_transaction &&
         get_preamble(rsp_data) == preamble_body &&
         axi_master_ready_and_rsp_not_empty_cnt < payload_length)
        [*0:max_payload_delay-1]

    // Ensure the correct number of flits have been received & acknowledged.
    ##1 axi_master_ready_and_rsp_not_empty_cnt == payload_length-1

    // Final payload flit must have its preamble set to 'preamble_tail',
    // which denotes that it is the last flit of the packet.
    ##0 (~rsp_empty &&
         axi_master_ready &&
         get_preamble(rsp_data) == preamble_tail)

    // Ensure the correct number of flits have been received & acknowledged.
    ##1 axi_master_ready_and_rsp_not_empty_cnt == payload_length
))
```

As mentioned previously, counters are used, namely `axi_master_ready_and_rsp_not_empty_cnt` and `rsp_not_empty_cnt`, which count the number of occurences of `(axi_master_ready && ~rsp_empty)` and `(~rsp_empty)` respectively. Moreover, `payload_length` is another signal that has been introduced. This signal stores the requested payload length for the duration of the transaction; noting that `payload_length` is assigned based on the values of the NoC signals. Therefore, if the DUT incorrectly specifies the payload length in the request sent to the NoC, the DUT will receive an unexpected number of flits, which would cause the assertions we have implemented to fire as desired.

It should also be pointed out that `max_header_delay` and `max_payload_delay` are predefined constants, where `max_header_delay` ≥ 1 and `max_payload_delay` $\geq \max(\texttt{payload\_length})$. In particular, if `max_header_delay` > 1 and `max_payload_delay` $> \max(\texttt{payload\_length})$, then the response's flits may be received non-consecutively. However, as these values are linearly increased, the time it takes to prove assertions that use the model grows exponentially since the reachable state space explodes in size. This was corroborated experimentally when the model and assertions were constrained enough that these assertions were proven in a few minutes, but assertions now take an unknown amount of time to be proven. Nevertheless, the `payload_length` is bounded by a small value, hence, `max_payload_delay` and `max_header_delay` do not have to be very large for the assumption to cover all possible practical scenarios; assuming the NoC is guaranteed to send a response in a timely manner.

Notice that unlike in previous definitions of the consequent, we assume that a packet cannot be received from a plane that is not the one we expect the response of the current transaction to arrive from. However, the original implementation of the DUT is not affected when a flit is received from a plane not involved in the current transaction. That being said, it does affect our modified version of the DUT. In particular, we added a counter to the DUT, which at the start of the transaction is set to the expected number of payload flits that should be received, and after the header flit is received, the counter is decremented every time a flit is received from *any* plane. If the counter is decremented to zero but the last flit was *not* a tail flit, the DUT will transition to the bad state `fv_bad_state`, which we added, and remains in this state indefinitely. If, instead, the counter is decremented to zero and the last flit *is* a tail flit, then the DUT stops listening for a response and initiates the next transaction, if any. In all other scenarios, the DUT will continue to listen for a response. Then, by asserting that the bad state is never reached and that the DUT eventually stops listening for a response, it ensures that the assumption as a whole is implemented correctly. That is, the right number of payload flits is assumed to be sent back by the NoC. Evidently, since the counter does not care from which plane a flit is received from, we must assume that the planes not involved in the current transaction are not sending any data to the DUT. Funnily enough, we actually intended for the counter to only be decremented when a flit was received from the plane used by this transaction. It was only when we formally verified the modified code that we realised that we forgot to consider the fact that other planes may be sending data to the DUT. We could have fixed the code, but we instead used this as an opportunity to use the `throughout` operator.

In actuality, the consequent is considerably more complicated but is kept simple here for illustrative purposes. In particular, since there are multiple AXI masters, all the AXI master-related signals, such as `axi_master_ready_and_rsp_not_empty_cnt`, are duplicated to form arrays, where the ith element of an array corresponds to the ith master. Accordingly, the signals corresponding to the master that performed the read request are used in the consequent. Furthermore, the above example requires `axi_master_ready` to be low when reading the header flit, however, our final implementation relaxes this requirement by using additional signal counters. In general, we strived to generalise the model of the NoC as much as possible. For instance, the DUT's internal state is inspected as little as possible and no constraints are imposed on the value of the payload data received. As a result, we can be confident that if an assertion, which leverages the model of the NoC, does not fire, then the assertion will always hold in practice. That is, the model does not overconstrain the search space too much, which would otherwise cause counterexamples that can occur in practice to not be found.

6.4 Read Request (Assumption Antecedent)

At a high-level, the antecedent, which is the sequence of events that constitute a valid read request, is defined as follows,

```
// A packet is never sent to the NoC immediately
// after sending another one. There is always
// at least a clock cycle delay in-between.
##0 req_empty

// Certain generic parameters are required to be of a
// specific value so that this assumption is only enabled
// when needed; noting that the generic parameters are
// guaranteed to always stay the same (assumed stable).
// This avoids assumptions from conflicting with each other.
##1 generic_param_1 == value_1 && generic_param_2 == value_2 && ...

// Each assumption is instantiated multiple times
// with different values for 'xindex', which is
// the index of the AXI master that the assumption
// considers. The reason there must be an assumption
// for each master is because each master has its own
// set of signals that are used in the consequent.
// Accordingly, this assumption is only enabled if
// the master that is sending the request is the
// same as the master that this assumption handles.
##0 transaction_reg.xindex == xindex

// Header flit is sent. We also require that
// the header is constructed correctly.
##0 (~req_empty &&
     get_preamble(req_data) == preamble_header &&
     get_msg_type(req_data) == expected_msg_type &&
     ...)

// Flit specifying the memory address
// of the data we want to read is sent.
##1 (~req_empty)[->1]
##0 get_preamble(req_data) == preamble_body

// Flit specifying the amount of data we
// want to read (i.e. payload length) is sent.
##1 (~req_empty)[->1]
##0 get_preamble(req_data) == preamble_tail

// Lastly, the payload length must be no more
// than what this property can assume. Without
// the following check, properties that use a
// length greater than max_payload_delay would
// be uncoverable but proven. In contrast, with
// this check, they are covered and fire as desired.
##0 req_data[7:0] <= `MAX_PAYLOAD_LENGTH
```

The in-line comments in the above code snippet provide all the details you need to know regarding the antecedent. That being said, notice that unlike in the consequent, the non-consecutive repeat operator `[->n]` can be used in the antencedent. Indeed, most, if not all the constructs that didn't work for the consequent, which are listed in section 6.3, can be used (if needed) in the antecedent.

6.5 Assumption Properties Implemented

We will now introduce every implemented assumption property that models the NoC. In practice, the DUT interacts with the coherence planes, namely plane #1 and #3, which are used to send coherence requests (e.g. coherent memory requests from the CPU) and receive coherence responses respectively. Furthermore, the DUT also interacts with plane #5, which is used to both send non-coherent requests as well as receive corresponding responses from the I/O tile. Note, the I/O tile is also known as the miscellaneous tile, which is why the shorthand `misc` is often used throughout this project's codebase.

Accordingly, we have implemented an assumption property that models the NoC sending back a response via plane #5 when the DUT sends a non-coherent read request to plane #5. This property was implemented in accordance with what has been discussed in the previous subsections, though the code itself is markedly dissimilar.

Similarly, we have implemented two assumption properties that both independently model the NoC sending back a response via plane #3 when the DUT sends a coherence read request to plane #1. Specifically, the assumption property that is enabled by default is implemented much like the previously mentioned property that deals with plane #5. In contrast, the second assumption property, which can be enabled instead of the first one by specifying an appropriate build flag (see `README.md`), is highly constrained and is consequently not a good model of the NoC. Regardless, the second implementation has been kept to illustrate the kind of stuff we initially tried.

Lastly, as previously stated at the beginning of "Modelling the NoC", section 6, it is assumed that the NoC will always eventually be able to service a request as follows,

```
assume property (@(posedge clk) (s_eventually ~remote_ahbs_snd_full));
assume property (@(posedge clk) (s_eventually ~coherence_req_full));
```

where `remote_ahbs_snd_full` is high when plane #5 is full (i.e. no request can be sent to this plane), and similarly `coherence_req_full` is high when plane #1 is full.

6.6 Model Usage

We implemented several assertions that make use of the NoC model. At a high-level, they all use the overlapped implication operator `|->`, where the antecedent is a sequence representing an AXI read request, and the consequent requires that the DUT takes appropriate actions until the response is fully received, which must eventually occur. Additionally, the DUT must never transition to the bad state (see the before last paragraph of "Read Response (Assumption Consequent)", section 6.3). In its simplest form, the consequent is defined as follows; noting that the actual consequent is considerably more complicated as it performs several additional checks but there is no use in showing them here.

```
##1 (current_state == request_header)[*1:$]
##1 (current_state == request_address)[*1:$]
##1 (current_state == request_length)[*1:$]
##1 (current_state == reply_header)[*1:$]
##1 (current_state == reply_data)[*1:$]
##1 (current_state != reply_data &&
     current_state != fv_bad_state)
```

As you can see, only the state of the DUT's finite state machine needs to be observed in the consequent, since per-state actions, such as forwarding data from the NoC to the appropriate AXI channel, can each be done in an isolated assertion. For instance, the following assertion verifies that payload data received from plane #3 is forwarded to the appropriate AXI channel when a coherent read request is performed.

```
property prop;
  @(posedge clk) disable iff(~rst)
  (transaction_reg.dst_is_mem==1'b1)
  |->
  (somi[transaction_reg.xindex].r.data==coherence_rsp_rcv_data_out[ahbdw-1:0])
endproperty : prop
assert_prop : assert property (prop());
```

Decomposing assertions into atomic assertions (i.e. assertions that cannot be broken down further) has many benefits. At the very least, it reduces user-perceived delay, since small assertions typically take seconds to verify whereas if they were combined with a complicated assertion, we would only know that everything works correctly once the combined assertion is proven, which evidently takes a much longer time. Decomposition also ensures that properties are verified over the largest possible state space since the composition of properties can only limit the reachable state space.

Surprisingly, the tool frequently failed to detect vacuous proofs, which was typically caused by an incorrect model of the NoC or assumptions that very subtly conflicted. Hence, for every assertion that leveraged the model, a corresponding coverage property was verified so that if the assertion is vacuously proven, its associated coverage property will be uncoverable, which immediately indicates that there is an issue. To this end, each assertion was duplicated and modified as follows to produce the corresponding coverage property. The `|->` operator was replaced with `##0` and the entire property was made to be strong instead of just the consequent. Alternatively, as in [7], the `|->` operator was replaced with the followed by operator `#-#`. These two approaches ensure that a vacuous proof is never covered; though the latter approach is no longer used.

Now, we eventually managed to generalise the model of the NoC so much so that the assertions that leverage the model now take a very long time (maybe even an eternity) to prove; noting that this is also due to the assertions being as general as possible. Specifically, the tool tried to prove the assertions for approximately **two weeks** to no avail, where $\texttt{max\_header\_delay} = 2$ and $\texttt{max\_payload\_delay} = 6$. Upon manually terminating the program, the minimum proof radius was 237 and the average proof radius was 1909.50. Reassuringly, their associated cover properties were all covered in just 4 seconds. Therefore, we are confident that the model is correctly implemented and the assertions do in fact hold in practice. Note, we also tried with $\texttt{max\_header\_delay} = 1$ together with $\texttt{max\_payload\_delay} = 2$, but due to the project deadline we couldn't let the tool run for more than two days. Hence, we do not know exactly how long it would take to prove the assertions when the maximum delays are minimal, but it certainly takes a very long time.

That being said, as detailed in "Assumption Properties Implemented", section 6.5, we also implemented a highly constrained assumption property that models the NoC sending back a response via plane #3 when the DUT sends a coherence read request to plane #1. Its corresponding assertions are also highly constrained. As a result, these assertions are proven within a couple of minutes; noting that their associated cover properties are covered in a matter of seconds. These properties can be enabled instead of the general ones by specifying an appropriate build flag (see `README.md`).

7 Overview of All Implemented Properties

Before implementing any complicated properties, it was necessary to constrain the environment so as to avoid bogus counterexamples. Notably, bounds were applied to the generic parameters that had been converted to input ports and these signals were also assumed to always be stable (since generic parameters are constants).

The main objective was to verify that the DUT accesses the NoC correctly. Specifically, we first ensured that the DUT never writes to the NoC when the NoC is full, and never reads from it when it is empty. Then, as detailed in “Assumption Properties Implemented”, section 6.5, we implemented assumption properties that model the NoC sending back a response when the DUT correctly sends a read request. It was then possible to verify that the DUT correctly sends read requests to the NoC and appropriately reads the response. To this end, we implemented several assertions that make use of these assumption properties (see “Model Usage”, section 6.6). Additionally, each of these assertions has a corresponding coverage property, which tremendously helped debugging as the tool typically failed to automatically detect that an assertion was vacuously proven (see “Model Usage”, section 6.6).

We then implemented assertions that verify that the DUT correctly forwards data received from the NoC to the appropriate AXI channel, and vice versa. Additionally, we asserted that AXI handshakes are performed correctly when the DUT (i.e. slave) initiates the handshake, and assumed that AXI handshakes are performed correctly when the user of the DUT (i.e. master) initiates the handshake. In particular, an AXI handshake requires that when the slave asserts the valid signal, it must remain asserted until the corresponding ready signal is asserted by the master, and vice versa. We initially thought we found a bug. Specifically, the AXI4 specification [8] requires that when the DUT receives valid data from the NoC and asserts the RVALID signal, RVALID must remain asserted until the user of the DUT (i.e. master) has read the data, which occurs when RREADY is asserted. However, the DUT asserts RVALID if and only if the NoC is not empty. Therefore, noting that the verification tool controls the master and NoC signals, the tool found that it could temporarily deassert the empty signal then reassert it whilst keeping RREADY unasserted. However, we believe that in practice (to be confirmed with the developers of ESP), once there is valid data on the NoC, the data will remain there until the DUT notifies the NoC that the data has been read, which only happens when the master asserts RREADY. If this is indeed the case, then the DUT implements AXI handshakes correctly. Nevertheless, by formally verifying the DUT in this way, we were able to discover assumptions that the DUT makes.

8 Conclusion

Most of our effort was spent trying to model the NoC and write interesting properties that leverage this model. The reason is that a key objective of this project was to push the formal verification tool to its limits (and beyond). Indeed, as detailed in "Read Response (Assumption Consequent)", section 6.3, we tried to write complicated assumptions that were syntactically correct but the tool lacked support for the SVA constructs we wanted to use. We eventually realised that we should instead implement complicated logic, such as dynamic non-consecutive repeat, directly in RTL as part of the DUT itself. This auxiliary RTL code can then be leveraged from within the SVA properties by only using simple SVA constructs. That is, we are able to assert/model complex behaviour without using complicated SVA constructs, which as previously mentioned are not well supported by the tool. Once we adopted this approach, it was then possible to generalise the model of the NoC. Accordingly, we also generalised the assertions that leverage this model. In doing so, assertions now take an unknown amount of time to be proven (at least two week with default parameters) whereas they were originally proven within one or two minutes. This is expected since the less constrained assumptions and assertions are, the greater the reachable state space, which has to be exhaustively explored.

As we have personally experienced, formal verification is a great way to gain a comprehensive understanding of the DUT. In particular, implementing assertions that you expect should pass but actually fire will inform you of scenarios you have not considered yet. In such a case, either a bug in the DUT has been found, or certain assumptions that the DUT makes have not been taken under consideration. Hence, by analysing the counterexample, it is then possible to locate the bug or determine the assumptions that the DUT is making. For example, as detailed in "Overview of All Implemented Properties", section 7, one of the assertions related to AXI handshakes fired unexpectedly, which made us realise that the DUT assumes that once valid data is received from the NoC, the data will remain there until the DUT acknowledges that it has been read. This is indeed a reasonable assumption and so there is no bug in this case. Note also, it is much harder to find such scenarios in simulation since only the ones you can think of will be tested. Hence, formal verification is very beneficial as simulation alone is unlikely to provide you with a complete understanding of the DUT.

Although we have extensively verified the AXI–NoC proxy, there is still much more that can be done. Regarding the AXI–NoC proxy itself, write requests have not been fully verified as they are essentially a boring version of read requests since no response is sent back for write requests. Furthermore, read requests where the NoC bus width is smaller than the bit width of the data being transferred (e.g. when sending 64-bit words on a 32-bit bus) has not yet been formally verified. Aside from this, there is not much else that could meaningfully be formally verified (i.e. simulation and/or proof reading the code is sufficient). Apart from the AXI-NoC proxy, an initial stretch goal was to verify the NoC as a whole, but in retrospect this obviously could not be done within the limited time frame of this project. Examples of properties we wanted to verify are as follows,

- Packets always eventually reach their intended destination. To this end, we should ensure that the NoC's queues, of which there are many, never overflow, as this is typically the reason why a packet is prematurely dropped.
- Packets are never duplicated (i.e. transmitted multiple times).
- Packets are never delivered to multiple tiles, only to the intended one.

Regardless, this project was a resounding success as we were able to implement numerous properties with some of them being so complicated that even after two weeks the tool was still trying to prove certain assertions. As a result, we have not only discovered the practical limitations of formal verification, but also meaningfully verified the AXI–NoC proxy, which is a vital component of ESP.

References

[1] System-Level Design Group at Columbia University. ESP: the open-source SoC platform. 2021. Available from: https://www.esp.cs.columbia.edu/ [Accessed 12th November 2021].

[2] Arm Limited. AMBA AXI and ACE Protocol Specification AXI3, AXI4, and AXI4-Lite ACE and ACE-Lite. 2021. Available from: https://developer.arm.com/documentation/ihi0022/e/ [Accessed 12th November 2021].

[3] System-Level Design Group at Columbia University. axislv2noc. 2021. Available from: https://github.com/sld-columbia/esp/blob/master/rtl/sockets/proxy/axislv2noc.vhd [Accessed 12th November 2021].

[4] Ben Cohen. Solving Complex Users' Assertions. 2021. Available from: http://systemverilog.us/vf/SolvingComplexUsersAssertions.pdf [Accessed 2nd December 2021].

[5] sebs. SystemVerilog: implies operator vs. |->. 2014. Available from: https://stackoverflow.com/a/25059580 [Accessed 2nd December 2021].

[6] AMIQ Consulting. SystemVerilog Assertions. Available from: https://www.amiq.com/consulting/resources/cheatsheets/ [Accessed 2nd December 2021].

[7] Eduard Cerny, John Havlicek, Dmitry Korchemny, Surrendra Dudani. SVA: The Power of Assertions in SystemVerilog. Available from: http://what-when-how.com/Tutorial/topic-3571uclo1c/SVA-The-Power-of-Assertions-in-SystemVerilog-428.html [Accessed 20th December 2021].

[8] Arm Limited. AMBA AXI and ACE Protocol Specification AXI3, AXI4, and AXI4-Lite ACE and ACE-Lite – Channel signaling requirements – Read data channel. Available from: https://developer.arm.com/documentation/ihi0022/e/AMBA-AXI3-and-AXI4-Protocol-Specification/Single-Interface-Requirements/Basic-read-and-write-transactions/Channel-signaling-requirements?lang=en#CIACFIIF [Accessed 22nd December 2021].