Automated Configuration of Verification Environments using Specman Macros

DVcon Europe 2018 – Paper number 260-OH22

Milos Mirosavljevic, Veriest Solutions, Belgrade, Serbia (*milosm@veriestS.com*)

Ron Sela, Valens Semiconductors, Hod HaSharon, Israel (*ron.sela@valens.com*)

Dejan Janjic, Veriest Solutions, Belgrade, Serbia (*dejanj@veriests.com*)

Efrat Shneydor, Cadence Design Systems, Petach Tikva, Israel (*efrat@cadence.com*)

***Abstract*—The paper describes a Specman based macro system to automate and facilitate full verification of a 16x16 complex switch project.**

Keywords—macro, verification, switch, Specman

I. INTRODUCTION

Today's ASIC devices delivers increasingly more RTL lines of code, more gates and correspondingly more features wrapped with better efficiency.
These new features represent higher logic complexity, supporting more configurations than ever before.

From a verification point of view, these more-than-ever complex ASICs mean several problems to consider when planning our verification environment:

This paper shares such dilemmas and solutions in a project that had recently been completed in a joint effort of Valens Semiconductors alongside with Veriest Solutions – a design & verification service company.

II. VALENS & HDBASET INTRODUCTION

A. *HDBaseT® In a Glance*

HDBaseT[1] is a standard for the transmission of ultra-high-definition video & audio, Ethernet, controls, USB and up to 100W of power over a single, long-distance, cable. For audiovisual, consumer electronics, and even industrial PCs, this can be a simple category cable (Cat6 or above), for up to 100m/328ft. For medical and government applications, optical fiber is usually preferred, spanning several kilometers. For automotive, HDBaseT can be transmitted over a single unshielded twisted pair (UTP), for up to 15m/50ft, or any other commonly used media (such as STP, HSD, coaxial and fiber).

[1] The HDBaseT standard was defined and contributed by Valens Semiconductors, a fabless company headquartered in Israel and established in 2006.

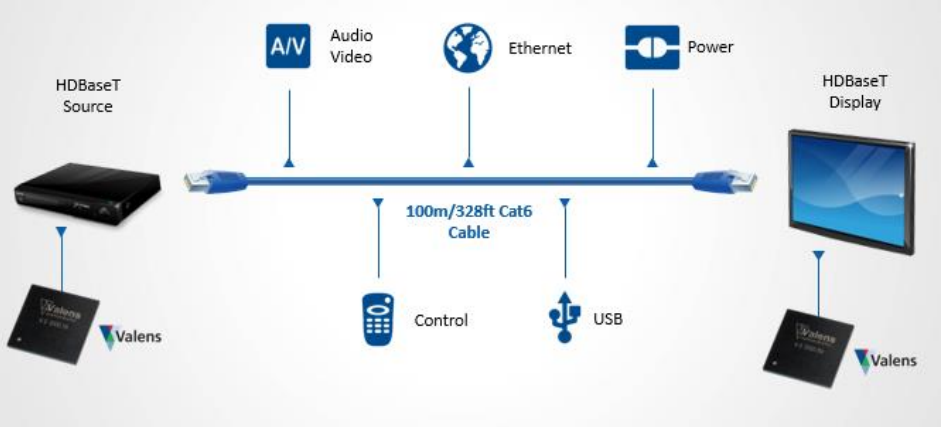


Figure 1:UHD Video & Audio, Ethernet, Power, Controls and USB on 100m/328ft LAN Cable

III. THE PROJECT: HDBASET SWITCH ("T-SWITCH")

HDBaseT Switch ("T-switch") – Main Features

- 16 x ports with 16Gbps per port, non-blocking switch
- Supports HDBaseT wire speed packet switching
- Switching any combination of T-Adaptors / HDBaseT ports
- Each port supports HDCP 2.2 and HDCP 1.4
- Management over Ethernet or I2C
- ValUE management and control system

High level architecture can be seen in the Figure 2.

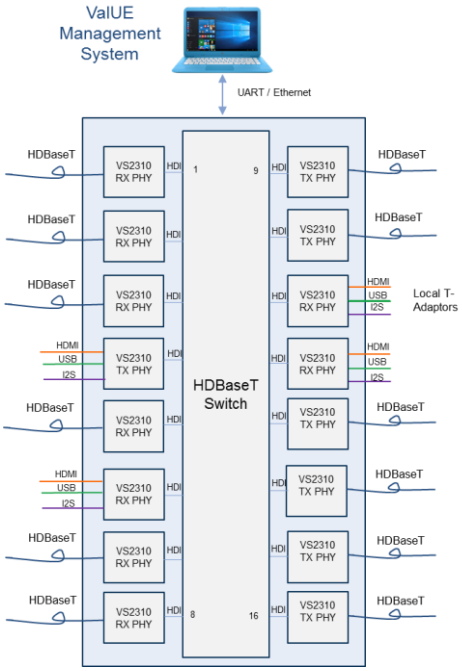


Figure 2: Native and HDBaseT switching

IV. T-SWITCH – VERIFICATION CHALLENGES

When the verification team had to plan how exactly the verification environment for the T-switch device would look like, it had to consider several issues:

How to cover all the different configuration options while, at the same time, meeting shortening schedules demands presented by today's electronics industry

To achieve that, one should leverage concurrently different strategies:

- Implementing new verification methodologies
- Using advanced code techniques

- Harnessing the power of reusability

As ASICs becomes more complex, larger verification teams are needed in order to cope with the task. Most likely the team will be composed of different levels of verification engineers, including senior as well as entry-level engineers. This means that whenever one considers of how to structure the verification environment, one must keep in mind the large variance of the different users of this environment (verification engineers) and as such, to supply an easy-to-use API which solves all configuration matters "under-the-hood", facilitating the end-user the configuration of the system and straightforward access to running actual tests.

In many companies, projects are changing frequently – meaning a project can be cancelled as fast as a new one emerges. This requires the verification group to be agile and adaptive in terms of re-allocating their team members. For the verification team lead who is also responsible for the design of the verification environment, this flexibility requires also to be taken in consideration in advance when planning the verification environment and how it would be used. New engineers whom will join the project during its life cycle might not have enough time and bandwidth to fully understand how to setup the environment correctly and as such will cause irrelevant scenarios and waste of debug time on "bugs" which are not real ones.

V. OUTLINE OF THE VERIFICATION SOLUTION

In order to meet the requirement of driving very complex stimuli in a simple manner, the team relied on capabilities of Specman macros.

Specman macros define parameterized code, which is instantiated by substituting parameters. In principle, a macro is used to extend e and add new language constructs. In general, a macro definition specifies:

- A syntactic pattern that defines a new syntactic construct in the language
- A replacement that specifies e code containing other, already existing constructs of the same category.

The main usage of the macro is providing the test writers easy to use syntax. Syntax that does not require being familiar with the language, the testbench or the methodology. (Next sections will expand on the methodology of using macros).

The output of the macro is a set of rules providing the verification environment information regarding a certain configuration and, accordingly, generates the required stimuli called a *stream*. This stimuli is driven towards the RTL. The project's requirement was to cover many combinations of input streams (each stream conveys different protocol/s type) and a significant challenge was to develop a mechanism which would allow this to be done in a simple way. The switch has 16 ports and they can be configured in a various ways and every port can communicate with every other port, which is 256 of different possibilities for streams between the ports. Every stream itself has multiple options for configuration, which in total leaves thousands of possibilities for stimuli.

Therefore, this "ADD_STREAM" macro provides simple solution to this complex problem.

A block diagram of the ADD_STREAM macro is shown in Figure 3.

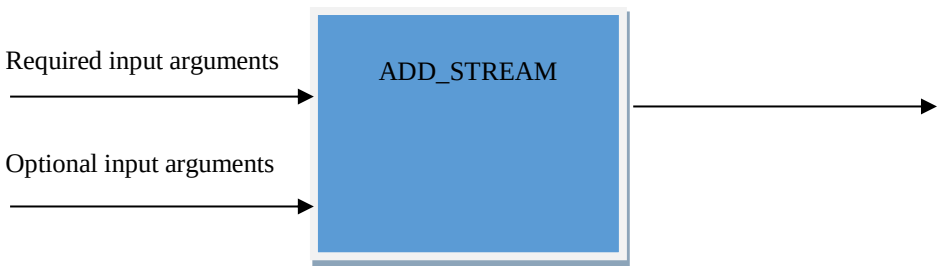


Figure 3: Required and optional input arguments. Result is stimuli towards the RTL

One of the biggest advantages of the macro usage is that it allows new and inexperienced engineers to quickly catch up in the project and create desired scenarios, such as:

Inject HDMI data to the switch port 0 and send this data through the HDCP towards port 14 and back from port 0 (multicast)

Inject data with very low bandwidth to the switch ports 0-7 and send this towards ports 8-15 (checking that low bandwidth will not impact RTL's behavior)

To take it one step further, this mechanism also allowed chip designers to create and run simple tests for sanity checks during RTL development, without too much knowledge about Specman code itself. The ADD_STREAM macro usage is straight-forward and in most cases self-explanatory.

Macro Definition & Strategy

The team had to come up not only with an easy and simple solution, but also define a method to add this code in such a way it will be able to override necessary old configuration fields generated automatically within the verification environment and just before the very first simulation tick. The solution was a new action macro (ADD_STREAM) in such a way that every user would be able to use it under Specman pre-defined *run()* method. This was a major decision for picking this strategy rather than using a sequence mechanism, as sequences are clock-sensitive and cannot achieve the same solution – they must have at least 1 simulation tick for their *body()* method to start working.

In fact, for the most situations, using the ADD_STREAM macro in the test requires around 15 lines of code, as shown below:

```
run() is also {
   ADD_STREAM
   stream_type                 = UNICAST
   src_port                    = 0
   dst_ports                   = {1}
   pkt_types_category_in_strm  = {OTHER_P1}
   specific_p_type_in_category = {PTYPE14}
   priority                    = {PRIORITY_1}
   pkt_type_bw                 = {16000}
   hdcp                        = FALSE
   burst_cycles                = 0
   sid                         = 100
   ayalon_source               = FALSE
   ayalon_dest                 = FALSE
   pkt_num                     = 500;
};
```

It's worth mentioning that this simple macro instantiation translates into the 180 lines of actual code "under-the-hood", starting with the following definition:

```
define <add_stream'action> "ADD_STREAM [ ]stream_type[ ]=[ ]<stream_type'type>\
src_port = <src_port'exp>\
dst_ports = {<dst_port'exp>;...}\
pkt_types_category_in_strm[ ]=[ ]{<pkt_types_category_in_strm'type>;...}[
bist_termination = <bist_termination'type>[ ]bist_port_mode =
<bist_port_mode'type>]\
specific_p_type_in_category[ ]=[ ]{<specific_p_type_in_category'type>;...}\
priority = {<priority'exp>;...}\
pkt_type_bw = {<pkt_type_bw'exp>;...}\
```

```
hdcp = <hdcp'exp>[ hdcp_bypass = <hdcp_bypass'exp>][ hdcp_bfg_en =
<hdcp_bfg_en'exp>][ hdcp_version = {<hdcp_version'type>;...}][ start_delay =
<start_delay'exp>]\
burst_cycles = <burst_cycles'exp>\
sid[ ]=[ ]<sid'exp>[ hdmic_cmd = <hdmic_cmd'type>]\
ayalon_source[ ]=[ ]<ayalon_source'exp>\
ayalon_dest[ ]=[ ]<ayalon_dest'exp>\
pkt_num = <pkt_num'exp>[ add_stream_on_the_fly = <add_stream_on_the_fly'exp>][
remove_stream_on_the_fly = <remove_stream_on_the_fly'exp> remove_delay =
<remove_delay'exp>]" as {
```

That is 1200% of code reduction per one macro usage. Needless to say that writing such 180 lines of code for each case would require higher level of verification skills and time, and would be prone to more errors and inconsistencies.

For completeness and to illustrate the richness of options the macro supports, follows the list of **Required input** arguments to the macro:

- **stream_type**: can be UNICAST, MULTICAST or BROADCAST, to make effective use of well-known forms of data communication.
    - UNICAST: injected data will be sent from 1 source port to 1 destination port. It should be noted that source and destination port can be the same, effectively simulating loopback through one port.
    - MULTICAST: injected data will be sent from 1 source port to multiple destination ports.
    - BROADCAST: injected data will be sent from 1 source port to all other ports, including itself.
- **src_port**: specifies switch port to which data is being injected by the eVC.
- **dst_ports**: unlike the above, this is a list of destination ports, to which data from the source port will be sent to.
- **pkt_types_category_in_strm**: specifies the list of packet type categories which will be present in the stream. Due to large number of packet types used in the project, it was convenient to first group them under the few higher level types. One example for this parameter is HDMI
- **specific_p_type_in_category**: specifies list of packet types, one per each packet type category specified above. For the mentioned HDMI example, this parameter could be one of the following:
    - HDMI_AV_CC
    - HDMI_AV_CG
    - HDMI_AV_GC
    - HDMI_AV_GCG
    - HDMI_AV_PXL
    - HDMI_AV_DATA
- **priority**: specifies priority of the injected packets. This parameter has multiple impacts on the stimuli generation:
    - Bandwidth with which the data will be injected to the RTL. Lower priorities have lower bandwidth due to RTL behavior specification.
    - Generated packet types.
    - Generating proper stimuli distribution (which packet type and with what bandwidth) was one of the major efforts in the project, in order to simulate real life scenario as best as possible. By using the ADD_STREAM macro, test author was able only to specify the priority list, and distribution would be correct.
- **pkt_type_bw**: specifies bandwidth of the input packet injection. This is total bandwidth in the input, which takes into the account the above mentioned specific priorities badnwidths.
- **sid**: specifies Session ID (SID) of the input packets. This parameter is HDBT protocol specific and it defines the routing of the packet inside the switch.

- **hdcp**: specifies whether HDMI/Video stream is going through the HDCP block. Relevant only for simulations related to HDMI. It is disabled by default.
- **ayalon_source/ayalon_dest**: specifies chip type of the source and destination. Has impact on RTL behavior and consequently verification environment.
- **pkt_num**: specifies number of packets to be injected.

As mentioned above, one may also specify **Optional input** arguments to the macro are as follows:

Optional field	Description
bist_termination	Specifies BIST termination type (NONE, INTERNAL, INGRESS, EGRESS). Default is NONE
bist_port_mode	Specifies the same port mode for all BIST stream ports (source & destination ports)
hdcp_bypass	Specifies if HDCP is in bypass mode or not. The default is TRUE.
hdcp_bfg_en	Specifies if HDCP BFG (Basic Format Generator) is enabled. The default is FALSE.
hdcp_version	Specifies HDCP versions of HDCP RX/TX for both SRC and DST port respectively. Default is {VER_1_4; VER_1_4; VER_1_4; VER_1_4}.
hdmic_cmd	Specifies HDMIC command which will be used in HDMIC sequences. The default is RXS.
start_delay	Specifies Start Delay for the stream that is going to be added On-The-Fly. The default is 0.
add_stream_on_the_fly	Specifies if the stream is going to be added On-The-Fly. The default is FALSE.
remove_stream_on_the_fly	Specifies if the stream is going to be removed On-The-Fly. The default is 0.
remove_delay	Specifies Remove Delay for the stream that is going to be removed On-The-Fly. The default is FALSE.

VI. EXAMPLES OF MACRO USAGE

The best way to show how required and optional arguments are used is through examples:

A. *Example1:*

One of the main scenarios the team had to test was the switch under maximum bandwidth through it, which is 288G. Each port should support 18G, 16G in egress direction and 2G in ingress, which is 18G per port, which is 288G per 16 ports. In order to drive required stimuli, total of 16 ADD_STREAM macros were used, encompassed inside for loops, as shown in the Figure 5.

```
run() is also {

 for i from 0 to 7 {
     ADD_STREAM
     stream_type                 = UNICAST
     src_port                    = i
     dst_ports                   = {i+8}
     pkt_types_category_in_strm  = {OTHER_P1;        OTHER_P2;       OTHER_P3}
     specific_p_type_in_category = {PTYPE14;         SPDIF;          PTYPE13}
     priority                    = {PRIORITY_1;      PRIORITY_2;     PRIORITY_3}
     pkt_type_bw                 = {15750;           200;            50}
     hdcp                        = FALSE;
     burst_cycles                = 0
     sid                         = 100+i
     ayalon_source               = FALSE
     ayalon_dest                 = FALSE
     pkt_num                     = 30000;
};
```

```
    for i from 0 to 7 {
        ADD_STREAM
        stream_type                 = UNICAST
        src_port                    = i+8
        dst_ports                   = {i}
        pkt_types_category_in_strm  = {OTHER_P1; OTHER_P2; OTHER_P3}
        specific_p_type_in_category = {PTYPE14; SPDIF; PTYPE13}
        priority                    = {PRIORITY_1; PRIORITY_2; PRIORITY_3}
        pkt_type_bw                 = {1750; 200; 50}
        hdcp                        = FALSE
        burst_cycles                = 0
        sid                         = 200+i
        ayalon_source               = FALSE
        ayalon_dest                 = FALSE
        pkt_num                     = 3750;
    };

   }; // run() is also
```

B. *Example2:*

Consider the following test scenario:

- Drive Video (HDMI) stream from port 0 to ports 1 and 13.
- The stream is going through HDCP block in bypass mode (without encryption)
- HDCP version in all HDCP blocks is 1.4.

Define macro that is going to be set in the test is:

```
ADD_STREAM
stream_type                 = MULTICAST
src_port                    = 0
dst_ports                   = {1;13}
pkt_types_category_in_strm  = {HDMI}
specific_p_type_in_category = {PTYPE14}
priority                    = {PRIORITY_1}
pkt_type_bw                 = {7000} - 7 Gbps
hdcp                        = TRUE
hdcp_bypass                 = TRUE
hdcp_version                = {VER_1_4; VER_1_4; VER_1_4; VER_1_4}
burst_cycles                = 0
sid                         = 100
ayalon_source               = FALSE
ayalon_dest                 = FALSE
pkt_num                     = 200;
```

This macro creates a single MULTICAST stream that is going from Source port 0 to Destination ports 1 and 13. Stream bandwidth is 7 Gbps. **HDCP** macro's input is set to TRUE which means that HDCP block is going to be enabled and configured in all ports related to the created stream (ports 0, 1, and 13).

HDCP blocks in all ports are going to be configured to be in bypass mode and to fit HDCP 1.4 version. What is very important and it is related to the optional macro arguments is that HDCP version is 1.4 by default, same as for HDCP bypass mode which is enabled by default. That means that **hdcp_bypass** and **hdcp_version** optional fields can be skipped in this macro declaration. Only if we want to set different values for these 2 arguments we need to add them.

If we set these arguments in the following way:

```
hdcp_bypass                 = FALSE
hdcp_version                = {VER_1_4; VER_2_2T; VER_2_2T; VER_1_4}
```

HDCP blocks are going to be configured to do encryption/decryption (it is not bypassed anymore). Also, HDCP blocks' versions are changed:

- SRC HDCP RX version – HDCP 1.4
- SRC HDCP TX version – HDCP 2.2T
- DST HDCP RX version – HDCP 2.2T
- DST HDCP RX version – HDCP 1.4

By using these macros in the test it becomes very easy for new/less-experienced verification engineers or even designers to create the desired often very complex scenarios in a very simple way. They do not have to care about the configuration that is controlled with these macros and that will be done "behind the scene".

If there is any limitation to use these macros, dedicated constraints or checkers will find it and suggest to the users that their scenario is not allowed.

C. Example3:

There are some complex scenarios like adding or removing streams on-the-fly. All streams are defined in zero-time. That satisfies 99% of tests so it is the default configuration. Anyway, some tests need to check DUT/environment behavior if new stream is going to be added or if existing stream is going to be removed during run-time.

If the user who is not familiar enough with the environment want to implement such a test, it is possible to do that in very simple way using the macro.

For example, consider the following test scenario:

- Add 2 streams for SRC0 to DST1 path.
- Add on-the-fly the third stream for the same path.
- Remove the stream that was previously added on-the-fly.

Macro for adding new stream:

```
//Add the 1st stream between SRC0 and DST1 at 0 time
ADD_STREAM
stream_type                 = UNICAST
src_port                    = 0
dst_ports                   = {1}
pkt_types_category_in_strm  = {OTHER_P1}
specific_p_type_in_category = {PTYPE14}
priority                    = {PRIORITY_1}
pkt_type_bw                 = {5000
hdcp                        = FALSE
burst_cycles                = 0
sid                         = 100
ayalon_source               = FALSE
ayalon_dest                 = FALSE
pkt_num                     = PKT_NUM;

//Add the 2nd stream between SRC0 and DST1 at 0 time
ADD_STREAM
stream_type                 = UNICAST
src_port                    = 0
dst_ports                   = {1}
```

```
        pkt_types_category_in_strm  = {OTHER_P2}
        specific_p_type_in_category = {SPDIF}
        priority                    = {PRIORITY_2}
        pkt_type_bw                 = {200}
        hdcp                        = FALSE
        burst_cycles                = 0
        sid                         = 101
        ayalon_source               = FALSE
        ayalon_dest                 = FALSE
        pkt_num                     = PKT_NUM;

        //Starting delay for on-the-fly streams
        var start_delay: uint;
        gen start_delay keeping {it in [20000..50000]};

        // ON-THE-FLY - Add 3rd stream between SRC0 and DST1 and later remove it
        ADD_STREAM
        stream_type                 = UNICAST
        src_port                    = 0
        dst_ports                   = {1}
        pkt_types_category_in_strm  = {OTHER_P1}
        specific_p_type_in_category = {PTYPE14}
        priority                    = {PRIORITY_1}
        pkt_type_bw                 = {2000} -- sum of BWs for all 3 streams must
not excide the max BW for that src_dst path
        hdcp                        = FALSE
        start_delay                 = 10000
        burst_cycles                = 0
        sid                         = 102
        ayalon_source               = FALSE
        ayalon_dest                 = FALSE
        pkt_num                     = PKT_NUM
        add_stream_on_the_fly       = TRUE -- this stream will be staeted and it's
SID will be  the SID Routing Table after the remove_delay time has expired
        remove_stream_on_the_fly    = TRUE -- this stream will be stoped and it's
SID will be removed from the SID Routing Table after the remove_delay time has
expired
        remove_delay                = 20000;--stream will be delayed for
remove_delay+5000 ns and SID Routing Table will be reconfigured for this stream
```

All optional inputs that are used in this scenario are added in the 3rd macro. User enables adding/removing streams on-the-fly and defines the time when stream is going to be added (**start_delay** optional input) and the time when stream is going to be removed (**remove_delay** optional input).

This macro is later replaced with the code by the pre-processor, that is used in sequences and checkers so driver will know that it has to wait for specified time before it start driving the new stream. It also knows when the stream is going to stop (removing stream). Dedicated checkers are checking is BW is satisfied per each port. Additionally, the checker also monitors if **start_delay** time is lower than **remove_delay** so stream cannot be removed before it is added, etc.

VII. ADVANTAGES OF USING SPECMAN MACROS

There are multiple advantages (and some limitation) in implementing this kind of solution in Specman:

- Easier to read - Unlike other languages, with e macros the implementer decides of the syntax they give their users. The macro usage does not have to look like a function call. When a macro has so many parameters as this utility requires, this is a big advantage in terms of simplicity.

Following is a comparison between calling it by Macro vs calling it by a Function:

Calling a macro:

```
ADD_STREAM
stream_type                 = UNICAST
src_port                    = 0
dst_ports                   = {1}
pkt_types_category_in_strm  = {OTHER_P1}
specific_p_type_in_category = {PTYPE14}
priority                    = {PRIORITY_1}
pkt_type_bw                 = {16000}
hdcp                        = FALSE
burst_cycles                = 0
sid                         = 100
ayalon_source               = FALSE
ayalon_dest                 = FALSE
pkt_num                     = 500;
```

Calling a function:

```
Add_stream(UNICAST,0,1,OTHER_P1,PTYPE14,PRIORITY_1,16000,FALSE,0,100,FALSE,FALSE,500)
```

The improved readability of the macro is very clear from this example.

- No need to pass arguments like a function - Another advantage of Specman macros can be seen in the following example – since the macro is not in the format of a function, the macro can contain parameters that are list of unknown length. This is implemented using repeating arguments as in the below code example:

```
priority = {<priority'exp>;...}
```

- Future enhancements to this utility can be to encapsulate macro within macro, to make the code even shorter, and easier to maintain. Parts of the macro body that do not depend on the parameters can be cut out into another macro and conditionally called by the main macro.
- It is worth mentioning that during the implementation of the macro, we discovered that Specman macros have limitation of **14** input arguments, so the team utilized a mechanism of optional parameters to increase the number of inputs. By using the optional arguments, the total number of inputs to the macro was increased by roughly 70% (14 →14+10) and it could be higher. Usage of optional arguments allowed the team much more versatility in generating stimuli.
- There is also limitation for number of optional parameters in a macro. Occurrence of ( ), [ ] and syntactic arguments inside <> brackets is limited to 56 per macro. For instance, usage of a single pair of square brackets makes user 2 steps closer to this limit of 56. For example, while defining macro we use square brackets (“**[ ]**”) to define space between 2 inputs in a macro:

```
stream_type[ ]=[ ]<stream_type'type>
```

Each square bracket’s pair provides 0 or multiple spaces. Removing of a single pair of square brackets makes limit 2 sub-matches farther.

Increasing the number of inputs is not only advantage of optional arguments. Optional arguments also simplify usage of the macro. All optional inputs may have their default values so if these values do not need to be changed, such optional inputs don’t need to be declared at all and their default values will be assumed.

VIII. IMPLEMENTATION DETAIL: "DEFINE AS" VS "DEFINE AS COMPUTED"

There are two ways to define macro types:

- **define as**
- **define as computed**

We decided to use "define as" macro, as it offered a comprehensive solution to the problem of generating and driving complex stimuli. While "define as computed" macros offer more possibilities and are as such more powerful, their usage and implementation adds additional layer of complexity, which was not required in the project. The difference is that with a **define as** macro the replacement code is just written in the macro body. With a **define as computed** macro user writes a procedural code that *computes* the desired replacement code text and returns it as a string. It's effectively a method that returns **string**, in which even the **result** keyword can be used to assign the resulting string, just like in any *e* method. A **define as computed** macro is useful when the replacement code is not fixed, and can be different depending on the exact macro argument values or even semantic context. Regardless, it's important to remember that even **define as computed** macros are executed during compilation and not at run time, so they cannot query actual run time values of fields or variables to decide on the resulting replacement code.

The syntax of **"define as"** macro is:

```
define <tag'syntactic-category> "match-expression" as { replacement }
```

The syntax example is:

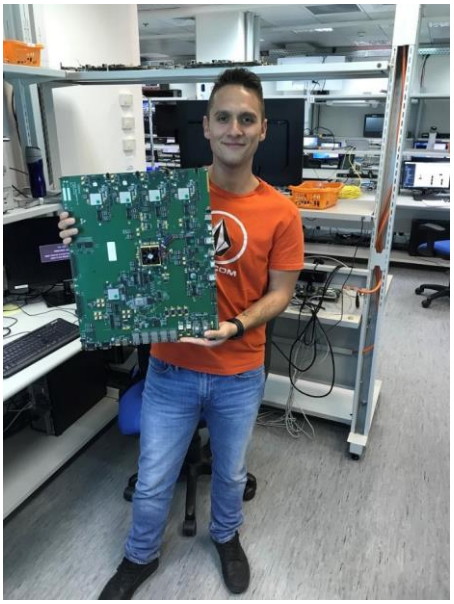
```
<'
    define <add_stream'action> "ADD_STREAM stream_type = <stream_type'type>" as {

   var ingress_stream : stream_s;
   gen ingress_stream keeping {
      .stream_type == <stream_type'type>;
   };
                                          };
'>
```

Another improvement could be to make the macro more generic, not depending on the exact environment hierarchy, as the hierarchy might change in future projects. This can be done using reflection and 'define as computed'. Unlike 'define as' macros, which are simple code replacements, by using 'define as computed' one generates code using procedural code. The way the macro is implemented now requires it to be called only from within the configuring unit, because the replacement code is an action that makes sense only inside a method in the configuration unit. If we use 'define as computed' we will be able to use reflection to query the environment and search for definitions as well instantiations of any type in the environment. For example, instead of assuming that the macro is called from within the unit "config_u", a 'define as computed' macro could find – with reflection - where a unit of a specific type (e.g. – 'config_u') is instantiated, what fields it has, and more. Then – it can call any method of this unit.

```
var s := rf_manager.get_type_by_name( "config_u" );
var all_fields := s.get_declared_fields();
```

As mentioned above, the drawback of using 'define as computed' is that it is more complicated to implement and maintain, as not all engineers are familiar with its syntax.

IX. RESULTS



Largely due the implementation of this ADD_STREAM macro, the project taped-out on-time and silicon was received right-first-time, where all the project goals were achieved:

Exhaustive coverage of the device in multiple stream scenarios

Ability to fully leverage members of the team, senior and junior verification engineers as well as designers, with minimum ramp-up on the project verification setup and maximum utilization of resources.

Some overall project figures:

- The project's life span ~2.5 years
- Over the project course, there were many changes in engineers allocation which led for total of 14 different verification engineers who had been working on this project
- There were total of 7 design engineers whom once finished coding tasks, could join the verification effort and run tests, even without being verification experts.
- The ASIC switch consist of 18,500 registers
- Total tests created were 279
- The Macro solution described in the above document, was used roughly 75% of the total tests whom needed some kind of dataflow stimuli

In the picture, the presenter of this paper holding the device test board.

X. REFERENCES

[1] Specman e Language Reference
[2] HDBaseT Specification Ver2.0