JSON-TIL: A tool for generating/reducing boilerplate when creating and composing streaming JSON dataflow accelerators using Tydi interfaces

Jasper Haenen
Student number: 4953266
Computer Engineering Student
Delft University of Technology
Delft, The Netherlands
Email: J.Haenen@student.tudelft.nl

January 2023

Abstract

This report presents JSON-TIL, a tool for automating the design process for parsing JSON streams on FPGAs. The tool is built on the Tydi-JSON project, which demonstrates the Tydi streaming interface specification. The Tydi-JSON project offers a set of parsing components that can be combined together to parse any JSON stream on a FPGA. However, the design process of assembling a combination of components to parse a specific JSON stream can easily become very labor intensive. JSON-TIL is capable of taking a sample of a JSON stream and automatically assembles and connects the necessary components to parse it. JSON-TIL implements a JSON analyzer, Tydi-JSON component tree visualizer, and a TIL generator. The TIL generator outputs a Tydi intermediate language(TIL) file. The TIL output can subsequently be handed over to TIL-VHDL which generates a VHDL project which can be compiled and programmed on a FPGA.

Preface

In this report, I present the results of a 10-week project for my master study in computer engineering at TU Delft. This project was an opportunity for me to get more hands-on experience with VHDL, Rust, and compiler/code generators. I am grateful for the guidance and support provided by my supervisor, professor H.P. Hofstee, and Matthijs Reukers. Their expertise and willingness to help were invaluable throughout the project. I would like to express my appreciation for the time and effort they invested in this project and their preparedness to help me with a problem quickly. This project has been a valuable learning experience and I am proud of the final results.

1 Introduction

When processing streaming data in hardware, the transfer of streams between components is often done through simple bit- and/or byte-oriented interfaces, such as AXI4(-Stream) and Avalon-ST. These interfaces, however, do not reflect the data structures being transferred, and require additional custom logic to organize complex data over transfers. To address this issue, the Accelerated Big Data Systems (ABS) group of TU Delft developed Tydi, a project that aims to standardize interfaces for hardware components used in modern streaming dataflow designs.

Tydi differs from the aforementioned interfaces in that it provides a type system and rules for mapping composite, variable-length data structures onto hardware. This allows transfers of more complex data structures between hardware components. As a demonstration of Tydi's capabilities, the ABS group used Tydi interfaces to create a set of VHDL components for parsing JSON objects. These components can be combined in infinite possible ways to parse different kinds of JSON objects. They were previously used to convert a stream of JSON data to Apache Arrow IPC messages, achieving a throughput exceeding 10 GB/s in some cases, far outperforming CPUs and GPUs and being more energy efficient.

However, while the JSON components are highly reusable, combining them to parse a specific JSON object results in a lot of manual labor in having to write a large amount of boilerplate code specifying the interconnects between components. This increases the design effort while also reducing the readability of the resulting designs, even though the underlying structure is very simple. The tool described in this paper, called JSON-TIL addresses these issues by automating the design process for assembling these components from front to end. It is capable of taking a sample of a JSON stream(*"schema"*) and automatically assembling and connecting all the necessary components to parse such a stream. This is expected to reduce the design effort required for creating a JSON parser on a FPGA, demonstrate the usefulness of Tydi in designing inter-operable components, and help improve the tools and identify any possible shortcomings in the existing tool-set.

This report begins by providing an overview of the background (Section 2) of the previous work JSON-TIL is based on. The main focus of the report is on the implementation of JSON-TIL (Section 3) which describes its components: the JSON-analyzer(Section 3.1), which parses and analyzes the JSON to convert to Tydi-JSON components, a visualizer tool(Section 3.2) that presents the analyzed JSON in a tree of Tydi-JSON components, TIL-generator(Section 3.3), a class that uses the output of the analyzer to create a TIL file that describes the Tydi-JSON streams and interconnections. Additionally, there is a complete example(Section 3.4) that shows the operation of JSON-TIL from front to end.

The report also discusses future developments and improvements that can be made to the tool(Section 4), and concludes by summarizing the key findings and contributions of the report(Section 5).

2 Background

2.1 Tydi

Data exchange between computing system components is a major topic in computer architecture. In the case of processing streaming data in hardware, stream data exchange between components is often transferred over simple bit- and/or byte-oriented interfaces (such as AXI4(-Stream) and Avalon-ST) or case specific custom interfaces are leveraged. There is a lack of a standardized streaming interface specification that reflect the actual data structures being transferred. Peltenburg et al.[1] addressed this by developing Tydi, an interface specification that aims to to standardize interfaces for hardware components used in modern streaming data-flow designs. Tydi differs from the aforementioned interfaces in that it provides a type system and rules for mapping composite, variable-length data structures onto hardware.

Tydi specifies five different logical types to define a streaming interface between two components[2].

- The `Stream` type defines a new physical stream (see Section 2.1.1).
- The `Null` type, this type indicates a transfer of data with the only valid value being $\emptyset$ Null. This
- The `Bits` type, this type indicates a transfer of a group of b bits. Tydi does not specify how these bits have to be interpreted.
- The `Group` type, this type specifies a composition of multiple logical stream types. All logical streams within the group are active at the same time.
- The `Union` type, this type also specifies a composition of multiple logical stream types. However in contrast to the `Group` type only one of the streams can be active at the same time.

These logic types can be combined to define more complex streaming data structures. Allowing Tydi streams to become multi-dimensional by nesting and grouping various logic stream types.

2.1.1 Streams

A Tydi stream type stream is used to define a new physical stream. It is constructed of several parameters describe the resulting physical stream. The parameters are as follows[2]:

- A **logical stream type** parameter indicating the element type carried by the logical stream.
- A **throughput** parameter a positive real number representing the expected amount of elements transferable on the child stream per element in the parent stream. If there is no parent stream, it is the number of elements that should be transferable per clock cycle; This is also called EPC(elements per cycle) or BPC(bits per cycle) in various implementations.
- A **dimensionality** parameter that indicates the amount of nested sequences that are in the current stream.
- A **synchronicity** parameter indicating the relation between the dimensionality information of the parent stream and the child stream. The only value used in Tydi-JSON components is `Sync` which indicates that for each element transferred on the parent, the child has a matching transfer. The other values, `Flatten`, `Desync`, and `FlatDesync`, are not used and thus fall outside the scope.

- A **complexity** parameter which is a number that encodes guarantees on how the sink transfers elements and assumptions that a sink can safely make about the element transfers. A higher complexity indicate fewer guarantees made by the source, source is easier to implement, sink can make less assumptions, and sink is harder to implement.

There are more stream type parameters but they are not used in Tydi-JSON components and such also omitted here as they fall out of scope.

2.1.2 Physical stream

The Tydi physical stream carries a stream of elements and dimensionality information about those elements[2]. The physical stream is parameterized by a set of parameters derived from the stream type. Based on these parameters, the physical stream defines which signals must exist on the interface of the sink and source component. A physical stream consists of the following signals:

- `valid`: Signal from the source signaling if the source is outputting valid data.
- `ready`: Signal from the sink to indicate source if it is ready to receive data.
- `data`: Signal from source with a the stream of data elements.
- `last`: Signal from source that indicates the last transfer of nested sequences of the level equal to the dimensionality of the stream.
- `stai`: Start index; Signal from source indicating the index of the first valid lane.
- `endi`: End index; Signal from source indicating the index of the last valid lane.
- `strb`: Strobe; Signal from source that individually mark data lanes as valid or invalid.
- `user`: Signal from source carrying additional control information along with the stream. However, it is not used in Tydi-JSON.

Table 1 shows the width of the signals depending on the stream parameters. N denotes the number of data lanes, E denotes the amount bits the elements are sized, D denotes the dimensionality of the stream, and scalar denotes a single bit signal.

Name	**Width**
valid	scalar
ready	scalar
data	$N \times \|E\|$
last	$N \times D$
stai	$\lceil log2N \rceil$
endi	$\lceil log2N \rceil$
strb	N

Table 1: Bit width of physical stream signals

2.2 Tydi-JSON

As a demonstration of Tydi, Tydi-JSON was developed by Hadnagy and Brobbel [3] where Tydi interfaces were used to create a set of VHDL components for parsing JSON objects. These components can be combined in infinite possible ways to parse different kinds of JSON objects. This was previously used to convert a stream of JSON data to Apache Arrow IPC messages achieving a throughput exceeding 10 GB/s in some cases, far outperforming CPUs and GPUs and additionally being more energy efficient [4].

Tydi-JSON has several different components that each parses an unique property element or data type of a JSON string. There are six different parsing components:

- A `Boolean parser` which takes in a string `"true"` or `"false"` and outputs a single bit value 1 or 0 respectively.
- An `Integer parser` which takes in a string containing a positive[1] number and converts it to a binary representation of the number of b width, where b is defined by a component parameter.
- A `String value parser` which takes a string as input and also outputs the same string.
- An `Array parser` which handles an input string of `"[.., .., ..]"` and transforms it so that it outputs only the array elements as strings.
- A `Record parser` which handles a key-value pair which are the fields in a JSON object. It handles an input of `"'key': value"` and seperates the string in a key and value.
- A `Key filter` which does not parse but is a filter that matches the key output of a `Record parser` and passes through the value string if the key matches, as seen in Figure 1. The key filter internally has a regex matcher component that matches the incoming key string.

The regex matcher component of the key filter is generated by use of another project *VHDRE* by J. van Straten[5]. It is a Python script that can generate a VHDL component that takes an input string and checks if this string matches a pre-defined Regex. The interface of the Regex matcher component is very similar to a Tydi interface, however it lacks certain control signals specified by Tydi.

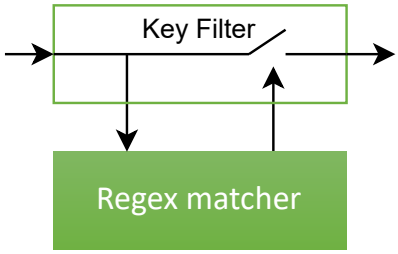

Figure 1: Working principle of the key filter component

The Tydi-JSON components can be stacked so that the nested structure of JSON can be parsed, see Figure 2.

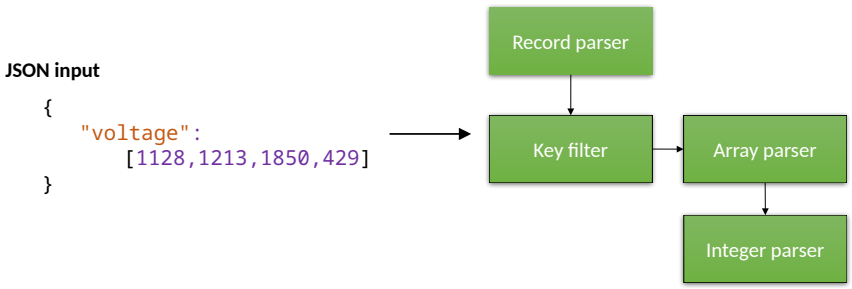

Figure 2: Example of JSON in Tydi-JSON components

Additionally, the Tydi-JSON components have to be parameterized to work properly. These parameters are set in the generic field of the component's VHDL, so henceforth they are called generics. These generics can be split in Tydi-related and component-related parameters. The following component generics are important:

[1]This limitation is due to that the component cannot handle the minus character

- The `EPC` generic is directly mappable to the throughput parameter of a Tydi physical stream and therefore indicates the amount of data elements transferred per cycle.
- The `Outer Nesting` generic tells at which nesting level the component resides in the JSON and is related to the dimensionality of a Tydi physical stream.
- The `Inner Nesting` generic indicates how deeply nested its sub-components are. This generic is only applicable to the `Record` and `Array parser` as those are the only JSON types that support nesting. Moreover, it is not related to the Tydi interface but is only used for the components behavior.
- The `Bit Width` generic, this generic is only used by the `Integer parser` and it specifies the amount of bits are used to represent the integer.

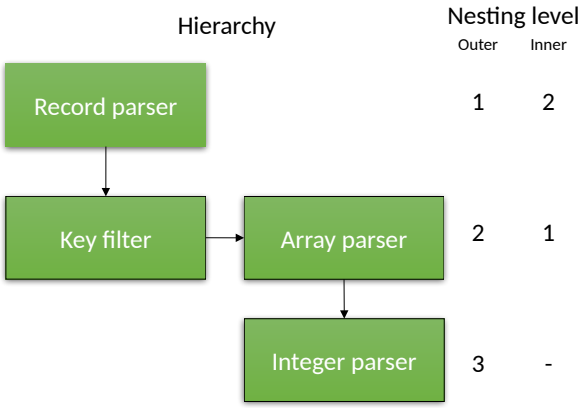

Figure 3: Example of Tydi-JSON components with nesting levels

Figure 3 shows how the components of the example of Figure 2 can be parameterized with their nesting level. There are, however, some aspects about the change in nesting levels and stream dimensionality between components. The `Record` and `Array parser` increase the outer nesting level as their child component resides at different nesting level, this also means that the dimensionality of these component's output stream increases. In contrast, the dimensionality of the output stream is decreased in the `Boolean` and `Integer parser` as the component converts a character stream into a single value. The `Key filter` and `String parser` component do not change outer nesting nor dimensionality. Because the `Key filter` only filters the data and does not transform it. The `String parser` also does not transform the data as the incoming data is already a string of characters.

2.3 TIL-VHDL

TIL-VHDL is a prototype toolchain used for demonstrating an intermediate representation (IR) which enables defining components using the Tydi interface specification. The TIL-VHDL toolchain can subsequently convert the defined components into VHDL component interfaces. The IR and TIL-VHDL toolchain were developed by Reukers during his Master Thesis[6]. The TIL-VHDL toolchain defines a grammar for the IR (TIL), a parser, query system, and back-end targeting VHDL[7]. The IR grammar (TIL) enables defining and connecting components using Tydi streams as interfaces. In TIL, the stream interfaces are called `Streamlets`. In these `Streamlets`, input and output stream types are defined to create a component streaming interface, as seen in Listing 1. The stream types are separately defined and can directly be mapped to Tydi stream types. Moreover, a `Streamlet` can also have some behavioral implementation which can be defined in two ways.

- A linked behavior where the behavior is predefined in a separate VHDL file. The `Streamlet`'s implementation consists of a string pointing to the directory containing the VHDL file. TIL-VHDL will link the interface to the correct VHDL file in this folder.
- An inline behavior where the behavior is defined within the TIL file. This enables the implementation field to instantiate `Streamlets` and connecting input/output streams of `Streamlets` to each other. Additionally, the instances can be connected to the input or output stream of the implementing `Streamlet`.

Listing 1 shows an example of both behavioral implementations.

Listing 1: Example of TIL Streamlet definition

```
type JSONStream = Stream (
    data: Bits(8),
    throughput: 4,
    dimensionality: 2,
    synchronicity: Sync,
    complexity: 8,
);

streamlet array_parser =
(
    input: in JSONStream,
    output: out JSONStream,
){
    # 1. Linked behavioral implementation
    impl: "./vhdl_dir"
};

streamlet top =
(
    input: in JSONStream,
    output: out JSONStream,
){
    # 2. Inline behavioral implementation
    impl: {
        arr_parser_inst_1 = array_parser;
        arr_parser_inst_2 = array_parser;

        input -- arr_parser_inst_1.input;
        arr_parser_inst_1.output -- arr_parser_inst_2.input;
        arr_parser_inst_2.output -- output;
    }
};
```

The example in Listing 1 is however not completely correct yet. This is because in the case of Tydi-JSON, the dimensionality of the output stream must be increased, as an `array_parser` increases the nesting level. However, this would result in a lot of very similar `Stream type` definitions which only differ in dimensionality and make the TIL file less readable. For this reason, a new feature is added to the TIL-VHDL toolchain that handles generics. `Stream types` now support the ability to make the dimensionality generic, see Listing 2. Moreover, the `Streamlet` definition additionally supports general generic definitions which are directly mapped to the generics field of the output VHDL file. Currently, there are 4 generic types:

- An `integer` generic that can hold any integer number.
- A `natural` generic that can hold 0 or a positive integer.
- A `positive` generic that can only hold positive integers.

- A `dimensionality` generic used as an interface generic for stream types. This generic also holds only positive integers.

Listing 2: Example of generic in TIL

```
type JSONStream<d: dimensionality = 2> = Stream (
    data: Bits(8),
    throughput: 4,
    dimensionality: d,
    synchronicity: Sync,
    complexity: 8,
);

streamlet array_parser = <
    EPC: positive = 4,
    OUTER_NESTING_LEVEL: dimensionality = 2,
    INNER_NESTING_LEVEL: natural = 0,
> (
    input: in JSONStream<OUTER_NESTING_LEVEL+1>,
    output: out JSONStream<OUTER_NESTING_LEVEL+2>,
){
    impl: "./vhdl_dir"
};
```

3 JSON-TIL

The JSON-TIL tool aims to automate the generation of the Tydi-JSON VHDL components from front-to-end. Consequently, the tool takes as input a sample of the JSON stream of what will be processed later on the FPGA to determine a *"schema"* of the JSON. With the sample, it directly assembles and generate all necessary components to parse a JSON stream similar to the sample. The sample is necessary because JSON does not specify a way to describe schemas[8].

The JSON-TIL tool consists of three parts:

- A **JSON analyzer** which analyzes the sample and sets up everything for generation.
- A **Visualizer** that can visualize the output from the analysis as a tree of Tydi-JSON components.
- A **TIL generator**, which processes the analyzed JSON and generates a TIL file accordingly.

The flow of the JSON-TIL tool can summarized as follows:

1. The user inputs the JSON sample as a string.
2. The user sets the following generator parameters:
   - The project name.
   - Tydi stream throughput (EPC).
   - A boolean to enable or disable the visualizer.
   - The bit-width of the output of an integer parser.
3. The JSON string is handed over to the JSON-analyzer.
4. If visualization is enabled: a visual tree of Tydi-JSON components is created by the visualizer.
5. The project is generated by the TIL generator.

3.1 JSON analyzer

The JSON analyzer is a class that contain functions to analyze JSON. During the analysis the JSON types are converted into an intermediate component which closely represents its Tydi-JSON equivalent. Additionally, the analyzer class contains a set of managers that are used to register the intermediate components. These managers can be queried after the analysis to generate TIL `Types`, `Streamlets`, etc. The analysis process can be divided into three phases.

- Phase 1: The project starts by parsing the JSON sample.
- A tree traversal through the parsed JSON tree which performs these actions:
  - Phase 2: Converting JSON types to an intermediate component more akin to Tydi-JSON components. This is accomplished by:
    - Determining nesting/(inner-nesting) level of the component
    - Mapping the parsed JSON type to an intermediate component
  - Phase 3: Registering the intermediate component in the TIL managers.

After these analysis steps, it is easy to generate a TIL output as everything is already setup in the managers according to the structure a TIL file.

3.1.1 JSON parser

The first stage of analysis is parsing the JSON sample. By parsing the JSON, the *"schema"* of the JSON can be inferred. This schema can be used to determine the necessary components later. The JSON parsing is performed by an external library, since creating a JSON parser falls out of the time scope of this project; additionally JSON is a popular language format which consequently results in large availability and high quality of parsing libraries. However, the utilization of the JSON parser in this project is not conventional, considering that the parser is used to infer a schema instead of parsing for data. Therefore, it is challenging to find a good JSON library to perform the schema inference. Additionally, since a typed programming language is used for this project, in this case Rust, many libraries expect a pre-defined class definition of the JSON schema to be parsed because it makes it easier to use the JSON in code. However, these parsing libraries are not applicable to the TIL-JSON implementation, as the schema needs to be inferred. After analyzing several libraries, it was opted to use the json-rust library by Hirsz et al.[9], for the following reasons:

- It claims to parse JSON according to the standard defined in RFC 7159[8], the standard JSON specification.
- It does not require any pre-defined schema of the JSON.
- It parses the JSON in a traversable tree, which it makes easy to convert to components later.

Moreover, it is more efficient if the implementation itself infers the schema. Considering that the nesting levels of components additionally need to be inferred, next to the schema.

3.1.2 Tree traversal

After the JSON is parsed with the aforementioned parser. The parsed JSON is now represented as a tree of `JSONValues`, as seen in Figure 4. A `JSONValue` is defined by json-rust, it is an enum object that can be one of the six types defined by the JSON standard. The JSON standard divides these in primitive types (strings, numbers, boolean, and null) and structured types (objects and arrays)[8].

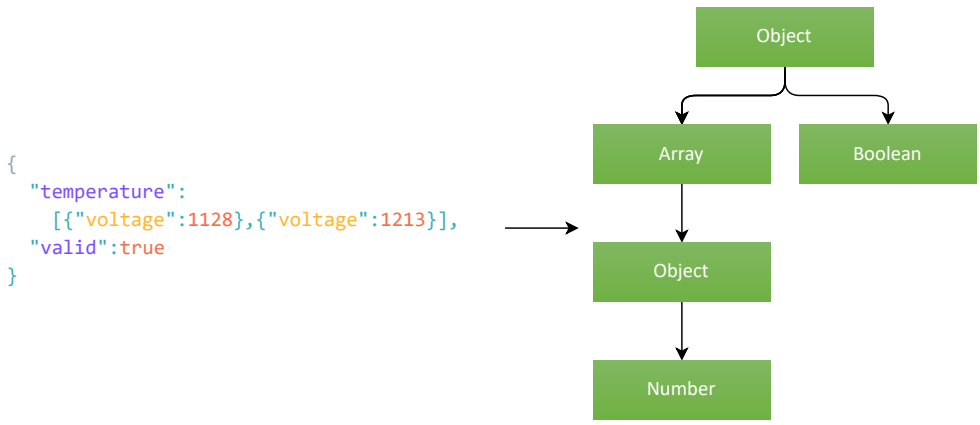

Figure 4: Example of JSON conversion to json-rust tree

To enable generation of Tydi-JSON components, the `JSONValue` tree has to be converted into a tree of objects more similar to Tydi-JSON components, called intermediate components hereafter. This is because `JSONValue` types are not completely directly mappable to Tydi-JSON components. Moreover, the `JSONValue` does not hold any information on nesting levels

which are necessary parameters for Tydi-JSON components. These intermediate components provide a good intermediate representation which can easily be converted from `JSONValues` and ultimately to the final TIL `Streamlets`, `Types`, etc. The intermediate components are constructed by performing a tree traversal on the `JSONValue` tree. The utilized traversal algorithm is a depth-first search analysis, this is used because it makes it easy to determine the inner-nesting levels for the `Array` and `Record` parsers. This traversal starts from the root object and analyzes the components along each branch as deep as possible. Once a leaf component is reached, the analyzer starts to backtrack and start returning an inner-nesting value which starts at zero. During this backtracking two things happen:

- The `JSONValues` are converted to an intermediate component. This intermediate component holds possibly a child component(s); and parameters: a nesting level and possibly an inner-nesting level(`Array` and `Record` parsers).
- This intermediate component is subsequently registered in so called managers. These managers can be queried later to generate the TIL.
- The inner-nesting is incremented, according to the rules defined in Section 2.2, and passed to its parent component.

3.1.2.1 Intermediate component mapping

The conversion of JSON types into intermediate components is mostly a one-to-one conversion for every JSON type. The intermediate components are a set of classes that all implement a trait[2]. This trait forces the intermediate components to implement a set functions that are later queried by the managers. How the functions are implemented differs from component to component, however it ensures the managers that a component with the trait can be registered. The following types are one-to-one conversions:

- `Boolean type` → `Boolean parser`
- `String type` → `String value parser`
- `Array type` → `Array parser`
- `Null type` → `Nothing`[3]
- `Number type` → `Integer parser`[4]
- `Object type` → `Record parser`

Object type mapping A special case is with the `Object` type which is difficult to represent as a pure Tydi component parser in Tydi-JSON. It can be mapped onto a `Record` parser since this parser handles the parsing of key-value pairs. A record parser is not directly matched to a fixed key value.

However, it is possible that not every record in the object has the same inner-nesting level, as seen in the JSON of Figure 4. In the beginning of the development of the tool, this was thought of as a hard limitation; meaning that every key-value pair needed its own record parser. This assumption appeared not to be true as the inner-nesting level is only used in the component to determine buffer sizes, so taking the max inner-nesting suffices. This means that JSON object can be mapped one-to-one to record parsers. Moreover, the key-value pairs inside the object are subsequently mapped onto a `Key filter` with a corresponding regex `Matcher` component. Figure 5 shows an example how JSON can be mapped to Tydi components.

[2]This is a Rust trait which is a collection of methods that can be applied to any struct. However, the struct itself must define the behavior of these methods.

[3]This is represented as the None enum value of the Rust Optional type, this will later indicate that the parent is a leaf of the tree

[4]The number type encapsulates more than only positive integers, Tydi-JSON is, however, not compatible with negative integers and floating point. This means that the mapping might break but is unpreventable.

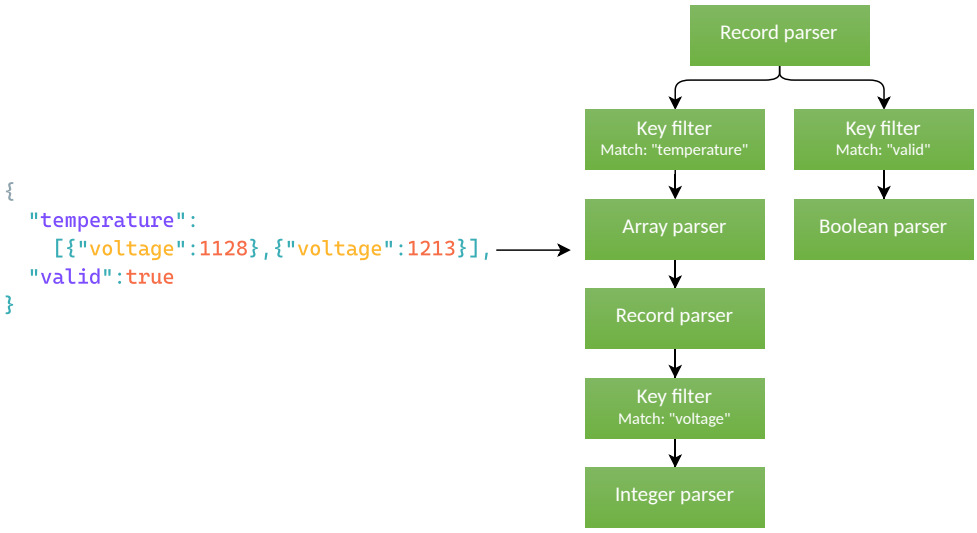

Figure 5: Example of JSON mapping to Tydi-JSON components

However, there is still a problem with representing the record parser as a pure Tydi streamlet. This is because a JSON object can have multiple key-value pairs. Consequently, the `Record` parser will have multiple out-going Tydi streams in that case, as seen in Figure 5. However, this is not possible as the Tydi control signals of child streams collide with each other because they are connected to the same parent signals. This problem is solved in the Tydi-JSON demonstration by inserting a `StreamSync` component between the control signals which essentially performs an `AND` operation on the `valid` and `ready` control signals, thus consolidating them. Section 4.1 will go further into this limitation.

3.1.2.2 Registering in managers

After a JSON Type has been converted to an intermediate component, the intermediate has to be registered in so called managers so that it can be generated later on. The analyzer class consist of 5 managers.

- A **name manager** which generates name for the components and prevent duplicate names.
- A **type manager** which registers the utilized stream types.
- An **instance manager** which registers a Tydi-JSON component as a `Streamlet`.
- A **signal manager** which registers connections between `Streamlet` instances.
- A **file manager** which prepares the additional project files to be generated containing the behavioral logic of the Tydi-JSON components.

Each manager will access the intermediate component to extract the necessary information from the component. These managers can subsequently be queried later by the generator and are designed to be structured like the final TIL file output. The output TIL file will look the following (compare to Listing 1):

- **Type definitions** can be generated directly from **type manager**.
- **Streamlets** can be generated directly from **instance manager**.
- A `Top Streamlet` containing:

– **Streamlet instantiations** can be generated directly from **instance manager**.
– **Signal definitions** can be generated directly from **signal manager**.

The name manager does not appear in the list above as it is only used to name the intermediate components. These names are subsequently processed already in the instance, signal, etc. Additionally, the file manager is not used in the list above as it only generates files that accompany the generated TIL file.

As a demonstration of the managers, Figure 6 shows the process of registering an intermediate component in each manager.

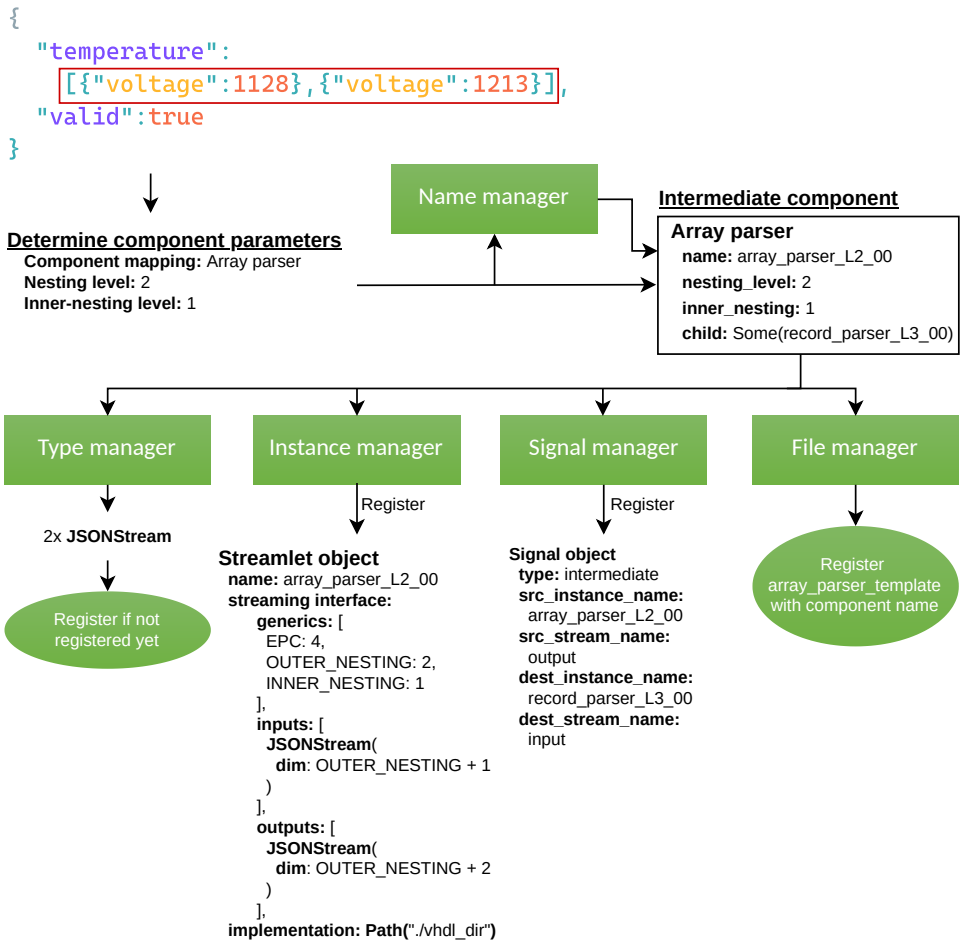

Figure 6: Example of registering a component from JSON

3.1.2.3 Functionality of managers

Name manager
The name manager generates the names for intermediate component while also preventing duplicate names. A new intermediate component will request a name from the name manager during its creation. The naming scheme of components is chosen to be
"{parser_name}_L{nesting_level}_{instance_number}"
, e.g. "array_parser_L1_00". The nesting level is included so that the hierarchy was more easy to verify from the generated TIL file and also it makes identifying components easier. The `instance_number` increments to allow multiple parser of the same type to exist at the same level. This generated name is subsequently saved in the intermediate component so that it can be used by the other managers later.

Type manager
The type manager registers the utilized stream types that will end up in the final TIL file. The type manager has six predefined stream types, these six stream types are describe to be atomic for all streams going in and out of Tydi-JSON components. The stream types have a generic dimensionality so that the stream types can be used by components on any nesting level, instead of generating a new stream type for every component. The throughput of the stream types can be defined by the user by changing the *throughput* generator parameter. The other stream parameters are kept to their equivalent value as defined in the Tydi-JSON components. The 6 predefined type streams of the type manager are as follows:

- A **JSON** `Stream` type, this is the type that most streamlets use. It is a stream of 8-bit ASCII characters representing the JSON string.
- A **Int** `Stream` type, this is the stream type of the output of the integer parser and carries a value of an integer. The integer-width size can be set by the user in the generator parameters.
- A **Bool** `Stream` type, this is the stream type of the output of the boolean parser and carries only a single bit.
- A **Record** `Stream` type, this is the stream type of the output of a records parser, it is the same as a JSON stream however it adds an additional bit to indicate if the output of the parser is the key or the value of the key-value pair.
- A **Matcher string** `Stream` type, this carries the stream of character containing the key in a record to a regex matcher. This differs from the the **JSON** `Stream` type in that it has a fixed dimensionality of 1.
- A **Matcher match** `Stream` type, this is the return stream of the matcher indicating with single bit if there is match or not.

Once an intermediate component is being registered for it types, it delivers the types which that component utilizes in its streaming interface. The type manager subsequently checks if this type is already used and if not registers it. This ensures that in the final TIL only consists of utilized types.

Remark on atomicity of types One could argue that the Matcher string `Stream` type is not atomic, as it can be represented as a JSON `Stream` type with a dimensionality of 1. However up until the near end of the project, it was not permitted by TIL to parameterize the dimensionality of a generic with a value equal to 1. This is now changed, however with this change the stream type remained due to time constraints. The same can be said about the Matcher match type and the Bool type, however these stream differ in that the Bool type

has a fixed throughput of 1, while the throughput of the Matcher match type is equal to the project-wide throughput. It is currently not possible in TIL to make the throughput of a `Stream` type generic, thus forcing these types to be split.

Instance manager
The instance manager registers intermediate components as `Streamlets`. A `Streamlet` component in TIL consists of three sections:

- A name for the `Streamlet`.
- A streaming interface containing:
  - A set of input/output `Streams` of the `Streamlet`
  - An optional set of generics that parameterize the `Streamlet` or input/output `Streams`.
- An implementation for the `Streamlet`.

The manager extracts data from the intermediate component necessary to construct a `Streamlet`. by querying the intermediate component to fill in the required sections defined by TIL, as seen in the list above. These sections are predefined per intermediate component variant and are derived from the Tydi-JSON component implementations. After these function are queried by the manager, it saves it as an `Streamlet` object in a list so that it can be generated later.

Signal manager
A signal indicates a connection between two components. A signal in TIL can consist up to four parameters:

- A **source instance name**: this is the name of the instance that will send data.
- A **source stream name**: this is the name that specifies which stream of the streaming interface of the source instance is used.
- A **destination instance name**: this is the name of the instance that will receive data.
- A **destination stream name**: this is the name that specifies which stream of the streaming interface of the destination instance is used.

The reason that a TIL signal can contain up to four parameters is that the source or destination stream name is not always defined. This occurs when the source or destination stream is a stream of the streaming interface of the `Streamlet` that implements the signals. For instance, Listing 3 specify two signals of which both have a source or destination without specifying an instance, e.g. *"input"* versus *"{inst_name}.input"*. To distinguish between these signals and make generating later easier, the signals are split into three types:

- An **input signal** which does not specify a source instance name. In this project these are signals coming from outside into the FPGA.
- An **intermediate signal** which specifies all four names. This signal type indicates a connection between components inside the FPGA.
- An **output signal** which does not specify a destination instance name. This signal type indicate signals going out of the FPGA.

The process of registering signals proceeds by querying an intermediate component. The intermediate component will subsequently return a list of the outgoing signals from this component. The reason for only returning outgoing signals is that components can only reference their children to get the instance name. This is because each component holds only a reference to their children.

Listing 3: Example of TIL signals

```
streamlet top =
(
    input: in JSONStream,
    output: out JSONStream,
){
    impl: {
        arr_parser_inst = array_parser;

        input -- arr_parser_inst.input; # SIGNAL 1
        arr_parser_inst.output -- output; # SIGNAL 2
    }
};
```

There are some exceptions, a `Key filter` also holds a `Regex matcher`, so additionally has to return the outgoing signal to the matcher. However, the `Matcher` returns a match signal to the `Key filter` which is its parent. But the `Matcher` cannot hold a reference to the `Key filter` as the `Key filter`'s parent holds this reference.[5]. To mitigate this, the `Matcher` only holds the name of the `Key filter` so that it can correctly return an outgoing signal.

Another exception is in components that do not hold children. This can range from components that cannot hold children (`Integer`, `Boolean`, `String` parsers) or components that can hold a child but do not because their child is of JSON type `Null`. These are called leaf components of the component tree, their outgoing signals are subsequently rendered as output signal types. For generation purposes later intermediate and output signals are registered in separate lists. The reason for separating the signals is that later when the top component is generated the output signals have to be registered as output stream in the top component's streaming interface. Input signals are not registered in practice because only out-going signals are returned, but they have their use later.

File manager

The role of the file manager ensures that for each generated component there will exist an accompanying VHDL file containing the correct behavior for TIL-VHDL to link to. These behavioral VHDL files are the same Tydi-JSON files, but they are changed to become a template. A template files have certain variable spots where the text is replaced by e.g. *"${component_name}"*; once the templates are rendered to actual files the file manager can subsequently replace these spots with the correct value. In order to register an intermediate component, the file manager only needs to know which behavioral template the component is using and the name of component. Moreover, it needs to know a project name and project namespace, however these are specified by the tool user before generating. There is one exception for the `Regex matcher` component because this component's behavior depends on the regex that is going to be matched. Therefore it cannot be defined as template beforehand. There already exists a tool to generate these matchers called VHDRE by Van Straten [5], however, this tool has some caveats. The generated matchers did not have Tydi interfaces and there was also no possibility to set a project name. Therefore, a fork of VHDRE has been created for JSON-TIL to solve these issues. Another caveat is that VHDRE is written in Python while this JSON-TIL is written in Rust, however this problem was solved by using the pyo3 library[10] for Rust. This allows to embed the python code into Rust using an embedded python interpreter using the shared library of a Python instance installed on the user's machine. Therefore, this tools allows the rust code to query the VHDRE code and get its result from within the rust code.

[5]This is a constraint in Rust to ensure memory safety. This can be mitigated by using a reference counter wrapper around the key filter, however this was discovered to late in the project to change.

3.2 Visualizer

The visualizer tool serves as a useful aid in verifying that the JSON is correctly interpreted by the analyzer. This is due to the fact that the TIL output can still be quite extensive, particularly when working with a large input JSON sample. The visualizer tool allows for quick verification, which not only benefits users but was also beneficial during the development process. The visualizer uses the dot format which is a file format used to perform graph visualization. The library dot-rust[11] developed by the Rust team is used to convert the intermediate component tree into a visualized tree. This achieved by instructing the library how to traverse the intermediate component tree and implementing function on the intermediate components to create nodes. The generated dot output can be easily visualized with any Graphviz tool. Figure 7 shows an example of an output of the visualizer.

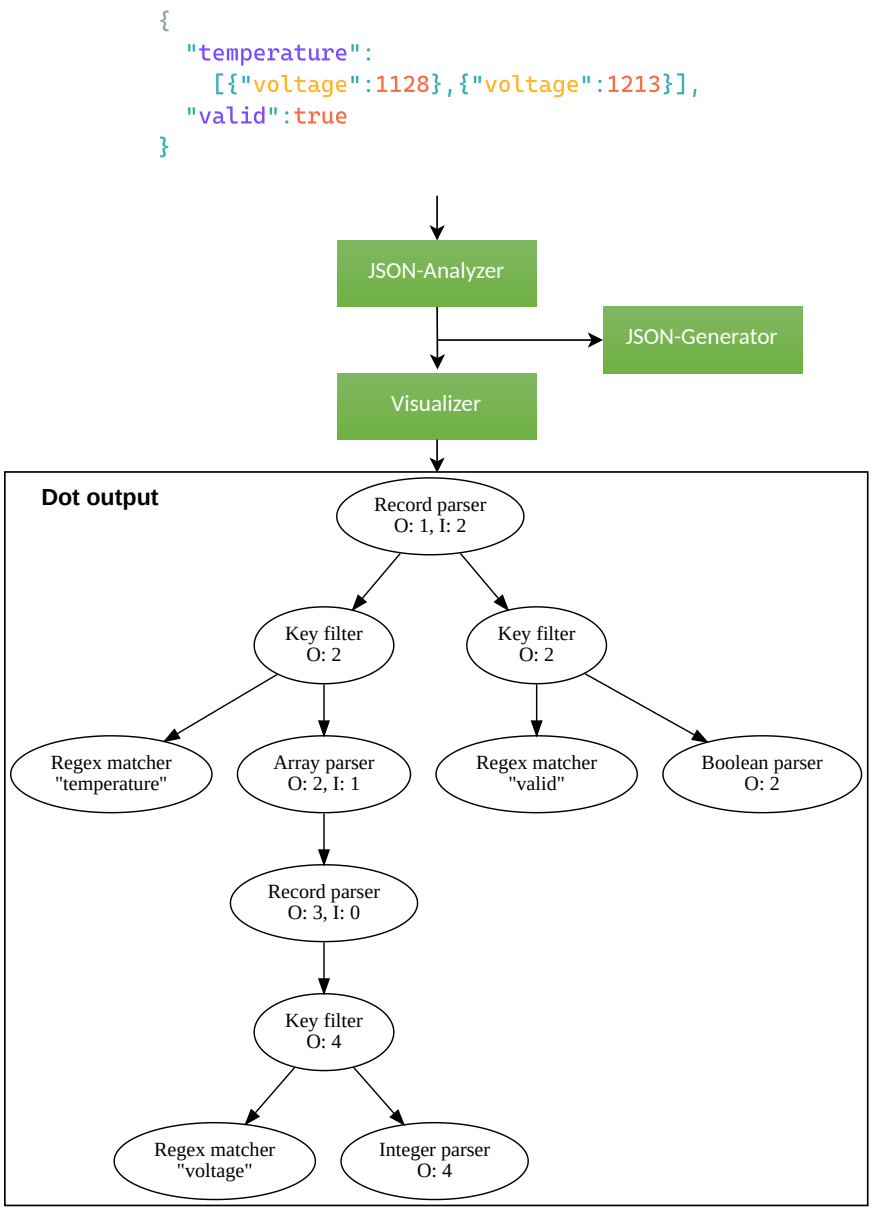

Figure 7: Generated dot visualization of the intermediate component tree from a JSON sample

3.3 TIL generator

After all components have been registered in the different managers, the tool can start generating the necessary files for TIL-VHDL. TIL-VHDL offers multiple options to take a TIL file as input. For JSON-TIL, the project folder input option has been chosen as it was most reliable during testing and development. There are three required folder/files for the project input folder option to work:

- A **TOML** file which describes the properties of the to be generated project. This includes the project name, the location of the TIL file, and the output location for TIL-VHDL to put its generated files.
- A **TIL** file located in the folder specified in the TOML which defines the streamlet definition and interconnections of the Tydi-JSON components.
- A **VHDL** folder that includes all the behavioral code to be linked with the streamlets.

3.3.1 Generating the TIL file

The generation of a TIL file consists of the following steps (see Appendix A for a TIL file example):

1. Generate namespace definition
2. Retrieve stream type and streamlet definition from the type and instance manager respectively.
3. Serialize stream type objects into TIL stream type definitions
4. Serialize streamlet objects into TIL streamlet definitions.
5. Assemble a overarching top component. This top component instantiates the streamlets and connects them together with signals.
6. Serialize top component into a TIL streamlet definition.

The namespace definition of the TIL file is generated from the project name which provided as a generating parameter by the user before generating. This project name is converted by changing the underscores *"_"* in the project name into double semicolons *"::"*.

Retrieving the stream type and streamlet definitions from the analysis is done by querying the type manager and instance manager, both of these managers will subsequently return a list of stream types and streamlet objects. These objects already contain all the necessary information to be generated to TIL definitions. Therefore, these objects can be easily serialized into a string containing the definition in the TIL format and subsequently added to the TIL file.

Assembling the top component The last step to generating the TIL file is to assemble a top component. The top component is used in this project to handle the streams going into and out of the FPGA, instantiating the streamlets, and connecting the instances together using signals. The following steps describe how the top component is assembled:

1. Create a streamlet object with name *"top"*.
2. Create a streaming interface object for the streamlet.
3. Create an input stream called *"input"* and assign it to the streaming inteface.
4. Create an inline implementation for the component indicating the behavior is specified in the TIL file instead of an external VHDL file.

5. Loop over all instances in the instance list of the instance manager and add them to the implementation.
6. Create a signal to connect the input stream to the input stream of the root component of the JSON and add it to the implementation.
7. Query the intermediate signals from the signal manager and add them to the implementation
8. Query the output signals from the signal manager and perform two actions:
    - Add the output signal the implementation
    - Get the output stream type and name from the signal and add it as an output stream to the streaming interface.

With the top component being ready, it can be serialized into a TIL streamlet definition and subsequently added to the TIL file.

3.3.2 Generating project

Finally, the last stage of the tool is generating an output project to hand over to TIL-VHDL. This starts by creating a project folder named after the project name specified in the generation parameters. In the project folder another folder is created called *"src"* with in it the generated TIL file. Lastly, the file manager is queried. The file manager creates another folder, called *"vhdl_dir"*, in the project folder and places a filled-in template files with the behavioral VHDL which are to be linked to the streamlets. Finally, the file manager creates a TOML file specifying the project name, TIL-VHDL output location, and a path pointing to the TIL file in the *"src"* folder. See Figure 8 for the resulting folder structure.

3.4 Example

Figure 8 shows an example of a project generated with the JSON-TIL tool. Appendix A additionally shows, the contents of the generated TIL file of this example.

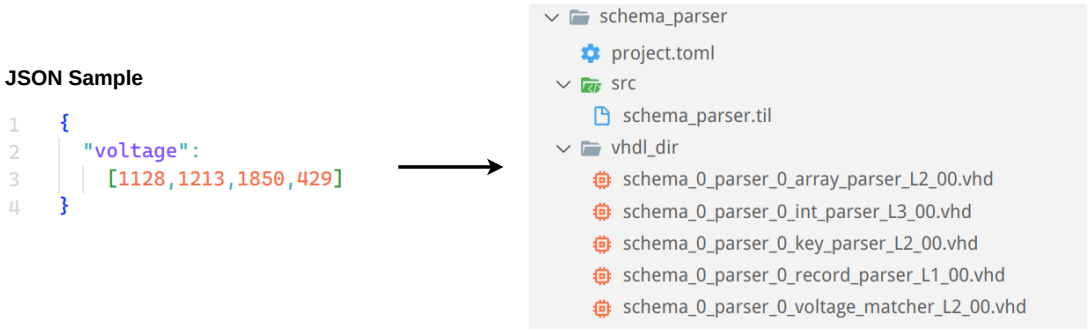

Figure 8: Example of generated project folder from JSON sample

Additionally, Figure 9 shows the tree of intermediate components that is generated by the Visualizer; this tree can be used to verify that the JSON is correctly analyzed. Moreover, it is noteworthy that the *"vhdl_dir"* folder in Figure 8 contains the same components as in the visualized tree in Figure 9, and is the same as the streamlet components in Appendix A.

The project is subsequently handed over to TIL-VHDL which creates a linked VHDL project as seen in Figure 10. A file called *"schema_parser_pkg.vhdl"* is created which contains a definitions of the streaming interfaces of the components(streamlets) in the project, this file enables that once the VHDL is compiled that the streaming interfaces are linked to the

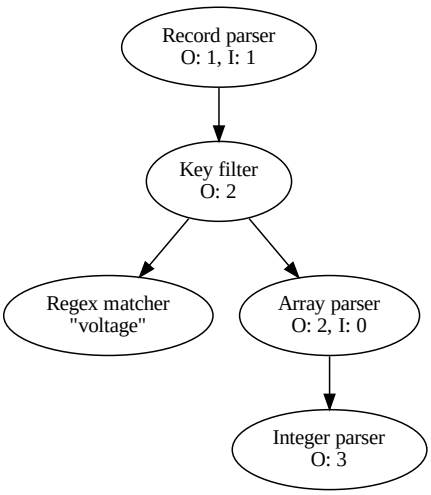

Figure 9: Generated dot visualization of the intermediate component tree from the example JSON sample

correct behavior. To verify the result coming from the VHDL tool, a VHDL testbench has been written by hand, see Appendix B. To perform the verification, a tool called GHDL is used which can compile and simulate VHDL testbenches. The output of the simulation can be seen in Figure 11, indicating a succesful execution of the testbench.

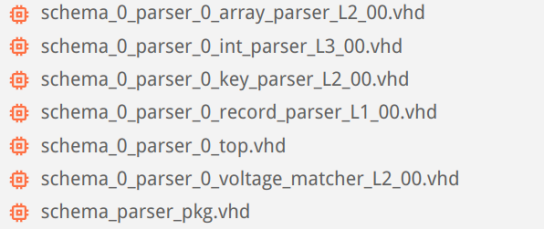

Figure 10: VHDL output of TIL-VHDL

```
12105000000 fs | Test SchemaParser passed!
12105000000 fs | Took 11105000000 fs
---------------+------------------------------------------------------------------
The test took 12105000000 fs. Summary:
 * PASSED SchemaParser
================================================================================
 TEST COMPLETE: result is ***SUCCESS***
================================================================================

Summary:
 * PASSED  work.schema_parser_tc
Test suite PASSED
```

Figure 11: GHDL simulation output

4 Future work

This section discusses potential improvements and developments that can be made to the JSON-TIL tool.

4.1 Branching streams limitation

Currently, the JSON-TIL tool lacks the capability to generate TIL files with branching streams. This limitation means that a single stream source cannot be connected to multiple stream sinks. This is a significant limitation as it prevents to have multiple key-value pairs within a JSON object, which is a common occurrence in JSON. The inability to branch streams is due to the fact that the control signals from all the sinks must in some way be consolidated into a single control signal for the source. The Tydi-JSON demonstration addresses this issue by inserting a StreamSync component between the control signals. This StreamSync component performs a logical AND operation on the Valid and Ready control signals, thereby consolidating them. However, this solution is not easily applicable for the JSON-TIL, as it relies on TIL for VHDL generation. Defining a StreamSync component in TIL is challenging, as it is not easy to describe it using a Tydi interface.

There are plans to introduce multiple intrinsic functions to TIL, one of which will be a synchronizer similar to the StreamSync component[6]. This will resolve the issue and will enable the JSON-TIL tool to generate TIL files with branching streams.. However, until then an approach could be to create a Tydi component with one input stream and multiple output streams for each branch. This component would perform the logical AND operation on the Valid and Ready control signals, while passing through all other Tydi signals. This would enable the capability to branch streams, similar to the StreamSync component, but using Tydi. Creating such a component in TIL is relatively straightforward, however, the behavioral implementation in VHDL poses a challenge. This is due to the fact that a VHDL file cannot have a variable number of output signals but is required as the number of output streams for each StreamSync component can vary. As a result, it is not possible to write this component purely in VHDL and it must be generated through other means.

4.2 Integrating TIL-VHDL

Integrating TIL-VHDL into the JSON-TIL tool would be a good improvement and would make the tool a complete front-to-end automation solution for parsing JSON streams. The current workflow requires the output TIL file to be manually handed over to the TIL-VHDL tool. This added step is a bit tedious as one would have to set-up two tools combining the tool will make the process more streamlined.

The TIL-VHDL tool can be integrated in the JSON-TIL by including the TIL-VHDL parser as an dependency in the project and pointing it to the correct project folder to generate from. This implementation will be fairly similiar to the *"demo-cmd"*[7] project used to demonstrate TIL-VHDL.

4.3 Automatic testbench generation

Another interesting feature would be to generate a VHDL testbench, similar to Appendix B, automatically from the provided JSON sample. This would completely make the tool a front-to-end automation by including a validation step that inserts the sample into the a simulation of generated VHDL and validates if the output from the simulation is equal to the expected output.

The complex part of this is extracting the values from the JSON sample. An approach to solve this is to create a testbench manager. The JSON sample is copied into this manager and while analyzing the component tree and reaching a value type in the JSON(number, boolean, string), the value is registered in the manager as an expected output of the JSON sample. It is important to note that each expected output value has to be assigned to the correct

output stream in the manager. After the analysis, the manager can create a file in the format similar to Appendix B. A stream source should be created and connected to the input of the top component. The JSON sample has to be pushed as a string to this input stream. Additionally, a stream sink must be created for each output stream of the top component. Finally, it checks comparing the output to the expected value should be inserted. It is important to note that these checks should be assigned to the correct output stream.

5 Conclusion

This report has presented JSON-TIL, a tool for automating the design process of parsing JSON streams on FPGAs. JSON-TIL is designed to demonstrate the usefulness of Tydi in designing interoperable components, and help improve the Tydi tools-set and identify shortcomings. The tool is built on the Tydi-JSON project and aims to reduce the manual labor and design effort required for assembling a combination of Tydi-JSON components to parse a specific JSON stream. The tool includes a JSON analyzer, Tydi-JSON component tree visualizer, and a TIL generator.

The tool currently has a limitation in its inability to generate TIL files with branching streams. This is a significant limitation as it prevents the parsing of multiple key-value pairs within a JSON object, which is a common occurrence in JSON. However, the introduction of multiple intrinsic functions to TIL is planned, including a synchronizer. This synchornizer will enable to resolve the limitation and allows the JSON-TIL tool to generate TIL files with branching streams. Another improvement would be the integration of TIL-VHDL into the JSON-TIL tool to make it a complete front-to-end automation solution for parsing JSON streams. Additionally, the implementation of an automatic testbench generation feature would add a validation step and further streamline the front-to-end automation process.

References

[1] J. Peltenburg et al. "Tydi: An Open Specification for Complex Data Structures Over Hardware Streams". In: *IEEE Micro* 40.4 (July 2020), pp. 120–130. ISSN: 0272-1732. DOI: 10.1109/MM.2020.2996373.

[2] M. Brobbel, J. Peltenburg, and J. van Straten. *Tydi: an open specification for complex data structures over hardware streams*. Feb. 2020. URL: https://abs-tudelft.github.io/tydi/index.html.

[3] Á. Hadnagy and M. Brobbel. *tydi-json*. Mar. 2021. URL: https://github.com/teratide/tydi-json.

[4] J. Peltenburg et al. "Tens of gigabytes per second JSON-to-Arrow conversion with FPGA accelerators". In: *2021 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, Dec. 2021, pp. 1–9. ISBN: 978-1-6654-2010-5. DOI: 10.1109/ICFPT52863.2021.9609833.

[5] J. van Straten. *vhdre: a VHDL regex matcher generator*. Jan. 2021. URL: https://github.com/abs-tudelft/vhdre.

[6] M. Reukers. " A Toolchain for Streaming Dataflow Accelerator Designs for Big Data Analytics". MA thesis. Delft University of Technology, Jan. 2022. URL: http://resolver.tudelft.nl/uuid:f2cf3430-328c-4489-9f79-ce1d739eae47.

[7] M. Reukers. *TIL-VHDL*. Jan. 2023. URL: https://github.com/matthijsr/til-vhdl.

[8] *The JavaScript object notation (JSON) data interchange format*. RFC 7159. Mar. 2014.

[9] M. Hirsz. *json-rust*. Mar. 2020. URL: https://github.com/maciejhirsz/json-rust.

[10] PyO3 Project and Contributors. *PyO3*. URL: https://github.com/PyO3/pyo3.

[11] Rust team and contributors. *dot-rust*. URL: https://github.com/przygienda/dot-rust.

A TIL ouput example

```
namespace schema::parser {
type JSONStream<d: dimensionality = 2> = Stream (
    data: Bits(8),
    throughput: 4,
    dimensionality: d,
    synchronicity: Sync,
    complexity: 8,
);

type IntParserStream<d: dimensionality = 2> = Stream (
    data: Bits(64),
    throughput: 1,
    dimensionality: d,
    synchronicity: Sync,
    complexity: 2,
);

type RecordParserStream<d: dimensionality = 2> = Stream (
    data: Bits(9),
    throughput: 4,
    dimensionality: d,
    synchronicity: Sync,
    complexity: 8,
);

type MatcherMatchStream = Stream (
    data: Bits(1),
    throughput: 4,
    dimensionality: 1,
    synchronicity: Sync,
    complexity: 8,
);

type MatcherStrStream = Stream (
    data: Bits(8),
    throughput: 4,
    dimensionality: 1,
    synchronicity: Sync,
    complexity: 8,
);

streamlet int_parser_L3_00 = <
    EPC: positive = 4,
    NESTING_LEVEL: dimensionality = 3,
    BITWIDTH: positive = 64,
> (
    input: in JSONStream<NESTING_LEVEL+1>,
    output: out IntParserStream<NESTING_LEVEL>,
){
    impl: "./vhdl_dir"
};

streamlet array_parser_L2_00 = <
    EPC: positive = 4,
    OUTER_NESTING_LEVEL: dimensionality = 2,
    INNER_NESTING_LEVEL: natural = 0,
> (
    input: in JSONStream<OUTER_NESTING_LEVEL+1>,
    output: out JSONStream<OUTER_NESTING_LEVEL+2>,
){
    impl: "./vhdl_dir"
};

streamlet voltage_matcher_L2_00 = <
```

```
    BPC: positive = 4,
> (
    input: in MatcherStrStream,
    output: out MatcherMatchStream,
){
    impl: "./vhdl_dir"
};

streamlet key_parser_L2_00 = <
    EPC: positive = 4,
    OUTER_NESTING_LEVEL: dimensionality = 2,
> (
    input: in RecordParserStream<OUTER_NESTING_LEVEL+1>,
    matcher_str: out MatcherStrStream,
    matcher_match: in MatcherMatchStream,
    output: out JSONStream<OUTER_NESTING_LEVEL+1>,
){
    impl: "./vhdl_dir"
};

streamlet record_parser_L1_00 = <
    EPC: positive = 4,
    OUTER_NESTING_LEVEL: dimensionality = 1,
    INNER_NESTING_LEVEL: natural = 1,
> (
    input: in JSONStream<OUTER_NESTING_LEVEL+1>,
    output: out RecordParserStream<OUTER_NESTING_LEVEL+2>,
){
    impl: "./vhdl_dir"
};

streamlet top = (
    input: in JSONStream<2>,
    output_int_parser_L3_00_inst: out IntParserStream<3>,
){
    impl: {
        int_parser_L3_00_inst = int_parser_L3_00;
        array_parser_L2_00_inst = array_parser_L2_00;
        voltage_matcher_L2_00_inst = voltage_matcher_L2_00;
        key_parser_L2_00_inst = key_parser_L2_00;
        record_parser_L1_00_inst = record_parser_L1_00;

        input -- record_parser_L1_00_inst.input;
        array_parser_L2_00_inst.output -- int_parser_L3_00_inst.input;

        voltage_matcher_L2_00_inst.output --
                        key_parser_L2_00_inst.matcher_match;

        key_parser_L2_00_inst.matcher_str --
                        voltage_matcher_L2_00_inst.input;

        key_parser_L2_00_inst.output -- array_parser_L2_00_inst.input;
        record_parser_L1_00_inst.output -- key_parser_L2_00_inst.input;
        int_parser_L3_00_inst.output -- output_int_parser_L3_00_inst;
    }
};
}
```

B Testbench example

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

library work;
use work.schema_parser.all;
use work.TestCase_pkg.all;
use work.Stream_pkg.all;
use work.ClockGen_pkg.all;
use work.StreamSource_pkg.all;
use work.StreamSink_pkg.all;
use work.UtilInt_pkg.all;
use work.TestCase_pkg.all;

entity gen_parser_tc is
end gen_parser_tc;

architecture test_case of gen_parser_tc is

  signal clk              : std_logic;
  signal reset            : std_logic;

  constant EPC                    : integer := 4;
  constant INTEGER_WIDTH          : integer := 64;
  constant INT_P_PIPELINE_STAGES : integer := 1;

  signal in_valid         : std_logic;
  signal in_ready         : std_logic;
  signal in_dvalid        : std_logic;
  signal in_last          : std_logic;
  signal in_data          : std_logic_vector(EPC*8-1 downto 0);
  signal in_count         : std_logic_vector(log2ceil(EPC+1)-1 downto 0);
  signal in_strb          : std_logic_vector(EPC-1 downto 0);
  signal in_endi          : std_logic_vector(log2ceil(EPC+1)-1 downto 0);
  signal in_stai          : std_logic_vector(log2ceil(EPC+1)-1 downto 0);

  signal adv_last        : std_logic_vector(EPC*2-1 downto 0) := (others => '0');

  signal out_ready        : std_logic;
  signal out_valid        : std_logic;
  signal out_dvalid       : std_logic;
  signal out_strb         : std_logic;
  signal out_data         : std_logic_vector(INTEGER_WIDTH-1 downto 0);
  signal out_last         : std_logic_vector(2 downto 0);

begin

  clkgen: ClockGen_mdl
    port map (
      clk                       => clk,
      reset                     => reset
    );

  in_source: StreamSource_mdl
    generic map (
      NAME                      => "a",
      ELEMENT_WIDTH             => 8,
      COUNT_MAX                 => EPC,
      COUNT_WIDTH               => log2ceil(EPC+1)
    )
    port map (
      clk                       => clk,
      reset                     => reset,
      valid                     => in_valid,
```

```
    ready                       => in_ready,
    dvalid                      => in_dvalid,
    last                        => in_last,
    data                        => in_data,
    count                       => in_count
  );

  in_strb <= element_mask(in_count, in_dvalid, EPC);

  in_endi <= std_logic_vector(unsigned(in_count) - 1);

  -- TODO: Is there a cleaner solutiuon? It's getting late :(
  adv_last(EPC*2-1 downto 0) <=
          std_logic_vector(shift_left(resize(unsigned'("0" & in_last),
              EPC*2), to_integer(unsigned(in_endi)*2+1)));

  schema_parser_i: schema_0_parser_0_top_com
  port map (
    clk                                 => clk,
    rst                                 => reset,
    input_valid                         => in_valid,
    input_ready                         => in_ready,
    input_data                          => in_data,
    input_strb                          => in_strb,
    input_endi                          => in_endi(log2ceil(EPC)-1 downto 0),
    input_stai                          => in_stai(log2ceil(EPC)-1 downto 0),
    input_last                          => adv_last,
    output_int_parser_L3_00_inst_ready => out_ready,
    output_int_parser_L3_00_inst_data  => out_data,
    output_int_parser_L3_00_inst_valid => out_valid,
    output_int_parser_L3_00_inst_last  => out_last,
    output_int_parser_L3_00_inst_strb  => out_strb
  );

  out_dvalid <= out_strb;

  out_sink: StreamSink_mdl
  generic map (
    NAME                        => "b",
    ELEMENT_WIDTH               => INTEGER_WIDTH,
    COUNT_MAX                   => 1,
    COUNT_WIDTH                 => 1
  )
  port map (
    clk                         => clk,
    reset                       => reset,
    valid                       => out_valid,
    ready                       => out_ready,
    data                        => out_data,
    dvalid                      => out_dvalid
  );

random_tc: process is
  variable a        : streamsource_type;
  variable b        : streamsink_type;

begin
  tc_open("SchemaParser", "test");
  a.initialize("a");
  b.initialize("b");

  a.push_str("{""voltage"":␣[1128,1213,1850,429]}\n");

  a.set_total_cyc(0, 20);
  b.set_valid_cyc(0, 20);
  b.set_total_cyc(0, 20);
```

```
    a.transmit;
    b.unblock;

    tc_wait_for(10 us);

    tc_check(b.pq_ready, true);
    tc_check(b.cq_get_d_nat, 1128, "1128");
    b.cq_next;
    while not b.cq_get_dvalid loop
      b.cq_next;
    end loop;
    tc_check(b.cq_get_d_nat, 1213, "1213");
    b.cq_next;
    while not b.cq_get_dvalid loop
      b.cq_next;
    end loop;
    tc_check(b.cq_get_d_nat, 1850, "1850");
    b.cq_next;
    while not b.cq_get_dvalid loop
      b.cq_next;
    end loop;
    tc_check(b.cq_get_d_nat, 429, "429");

    tc_pass;
    wait;
  end process;

end test_case;
```