1. Introduction

In order to determine if a textual description provides enough information for an implementation without ambiguities, an analysis of the additional information provided by a data-flow implementation is needed. The analysis should determine if a data-flow design overview provides enough information that eliminates all the presumed ambiguities.

2. CAL actor language

The CAL actor language is a textual notation that allows representations of the data flow, actors and their functionality. It is designed to allow descriptions of actors by providing statically analyzable information about the behavior of an actor, such as the number of tokens it produces and consumes in each firing, the conditions that enable an action to be fired, the description of the actions priority and dependencies between actions.

Using CAL, a project is designed as an abstract dataflow representation and it is able to produce ready to compile or to synthesize code. At this time, it is possible to produce C, C++, Java and VHDL projects from the abstract design. In order to meet all the constraints of the software and hardware design, the abstract language has limitations. These limitations define the complexity of the operations a FU can implement, the way two or more FUs exchange data and the behavior of each FU.

2.1. CAL concept

From the developer’s point of view, the concepts are quite different from traditional programming language, since the rules of sequentiality don’t fully apply when developing a project using the CAL language. A project is represented as a network of connected elements, that interact by sending data and, if needed synchronization signals among them. A better approach would be to visualize the entire project from the data flow point of view instead of the sequential point of view.

The basic element used to develop a network is the FU (Functional Unit); a network consists of one or more FUs that exchange data.
2.2. Functional Units

The FU represents the data processing elements as part of the network and it has the purpose to consume and produce piece of data called tokens. The communication process between two or more FUs involves the introduction of FIFOs for each communication channel. A FIFO allows the FUs to produce or consume data whenever they are ready to do so and the FIFO conditions are met (FIFO is not full or empty etc).

![Figure 3. Functional Units communication process](image)

Depending on the type of data, one FU provides tokens through the FIFO to the next FU in a serial manner while the next FIFO consumes the tokens. Since the data cannot be sent from one FU to another as a block of data, complex data or pointers, every FU needs functions to serialize and de-serialize data. For this implementation, the type of data can only be of primitive type with a pre-established width (usually the width of the data type, i.e. integer) and the output of a FU has to be of the same type as the input of the next FU on the same communication channel (FIFO).

![Figure 4. Network described using Functional Units including the data exchange process between Functional Units](image)

For example, to send a vector of integer values or a string of characters, all the values will be placed in an orderly fashion in the FIFO.

Each network or sub-network, except for the top-level network has at least an input (a FU that provides a stream of data) and at least an output through a FIFO (that will be used by the next connected FU).

The functionality of a FU is described using actions (the same concept as functions). An action can consume tokens (input), process data and produce tokens (output). In order to describe the order and priority in which the actions are fired a FSM (Finite State Machine) is used. Each time a condition is met, an action is fired and the appropriate function that processes data is called and run.

![Figure 5. States of the FSM that describes the Functional Unit’s behavior](image)
2.3. Networks

A network can describe a complex design, functionality and interaction between FUs and other sub-networks. Since the FUs can act more or less independently, this concept makes it easy to scale the design and to manage the FUs as threads or processes from the software point of view and as different cores from the hardware point of view.

2.4.1. From the software point of view:

One or more FUs can run on the same core, while the other FUs run on different cores, thus increasing the parallelization level. The OS manages the mapping of the FUs on different cores depending on time quanta thus resulting in the less active FUs to be run on the core that is idle or has a low use.

Each FIFO is mapped in a shared memory area where all FUs can write or read data. Access to the FIFO management module and to the shared memory can introduce latencies.

An efficiency analysis could be performed to determine a more efficient level of parallelization. The parameters that determine the efficiency are strongly dependent on the complexity of the operations that a FU implements, the depth and width of the FIFO and the number of the tokens that are consumed or produced at each action that is fired. There is a determinable threshold over which the parallelization introduces an overhead that will make the design less efficient.

2.4.2. From the hardware point of view:

Each FU can be considered a different core on a chip, thus having as a result an easier way to manage and analyze the behavior of each core separately and to debug if needed. Since every FU runs as a separate core and the inputs and outputs are from and to a FIFO, there could be different clock domains for each core. Different clock domains can have different clock signals (different clock periods) on the same platform allowing more efficient power consumption and lower heat signature when the constraints are tight especially for mobile devices. The depth and width of the FIFO are an important issue when resources are limited.

The FUs that implement highly complex operations can run at a higher frequency than the ones that implement less complex operations or whose actions are rarely fired. The efficiency’s threshold is highly dependent on the time which the FUs consume or produce tokens. The FU should provide a token to the next connected FU before the latter needs it but not too early, in order not to fill the FIFO before it is consumed.
2.4. CAL Limitations

Most of CAL’s limitations are introduced by the capabilities to produce both software code and hardware description language code. Designing a hardware module introduces a high number of constraints on resources and on the complexity of the implemented operations. The most important limitations are presented below:

- Each complex operation should be divided into a set of simple operations that together compound the complex operation.
- Access to the data from FIFOs and shared memory area introduces latencies.
- The communication between FUs introduces an overhead – serialize / de-serialize data.
- The FUs can only communicate serial.
- For this implementation, the communication channels can only handle primitive type of data.

2.5. CAL Advantages

- Thinking on an abstract level makes it easy for the designer to focus on the algorithm rather than on the implementation details. The target produced project (as ready to compile code for software and synthesizable code for hardware) can be changed without any modification on the abstract level code – there is no need to change any code in order to switch the target from a software type to another or to hardware design.
- Using the graphical interface, the environment offers a better view on the data flow.
- The design can be easily modified by using drag-and-drop on the FUs – a new FU can be added or removed. The constraint in adding a FU between two other FUs is that the interface (communication channels) of the left FU has to match the interface of the newly added FU and the input interface of the right FU has to match the output interface of the newly added FU. This also allows fast prototyping and testing without the need for detailed knowledge on software or hardware development.
- Since all the functions are designed as Functional Units it makes it easy to reuse the FU in the same project (drag-and-drop) or in a new one.
- The architecture allows a scalable high level of parallelization without needing to implement the parallelization algorithms, data synchronization, data exchange and management between FUs that run simultaneously.
- Allows detailed control over the data exchange mechanisms (FIFO depth, width and control signals).
- If the target project is a hardware design, it allows the designer to control the core operating parameters (constraints for mobile devices).
2.6. Design Flexibility

In order to provide a clear overview on a design, besides the textual description, there could be multiple implementations of the same design or even the same FU. This approach decreases the code reusability and the design’s ability to be reconfigured without modifying the code. Assuming that the project is implemented using FUs or similar modular concepts, the ability to reuse other FUs or to be reconfigured is limited by the interfaces of each FU.

A textual description alone could introduce ambiguity since it does not include all the situations that can occur in the data flow design.

In the example (Fig. 8), FU3 implements an operation that has to be processed based on the results provided by FU1 and has to provide data for FU2. Since the input interface of FU3 does not match the output interface of FU1 and the output interface of FU3 does not match the input interface of FU2, FU3 cannot be placed between FU1 and FU2.

In order to introduce the reconfigurable capabilities to the design, each FU should have the same interfaces (usually data and synchronization signals). This will allow introducing, replacing and removing FUs from the design even during the execution. Since there is only one way that FUs can exchange data, the ambiguity is eliminated.

In the example above (Fig. 9), FU2 has the same input as FU3 and it can be bypassed by configuring SW1 and SW2 accordingly. Having a common communication interface for all the FUs makes the network reconfigurable by configuring the SWs based on data stream analysis.
3. SC3DMC implementation using CAL

In order to implement SC3DMC a deeper analysis of the algorithms and of how they can be divided into FUs is needed. A top level view of the SC3DMC is presented in Fig. 11, where besides the core there are also 3 more FUs that were introduced for testing purposes.

3.1. Description of the top level FUs.

Auxiliary FUs:

- **source**
  Loads data from a file and feeds the data stream by filling the FIFO

- **SPS**
  Provides the Height and Width parameters for the display device

- **display3D**
  It uses system libraries to display a window and OpenGL functions to display the data structures as a mesh.
3.2 Core FUs:

A. decoderDEMUX:

It takes a stream of data as input, analyses the data and depending on the content, feeds the corresponding output and generates the synchronization signals.

The description of the FU’s functionality is represented using a FSM that has the following states:

1. initDEMUX
2. checkStartCode
3. extractHeaderInfo
4. feedHeaderInfo
5. feedDecoders

1. initDEMUX – flushes all variables to zero or default value (ensuring that there will not be any false values for the initial or the next processing stage).
2. checkStartCode – checks the read values for the HEX code F1 01 00 00, code that represents the beginning of a new segment of data to be decoded. If this code is found, the FSM proceeds to the next stage, if not it keeps looping until the code is found in the data stream.
3. extractHeaderInfo – it extracts the size of the data segment and the decoder type information from the data stream (by consuming elements from the input FIFO, this information will not be sent forward to the next FUs).
4. feedHeaderInfo – since some of the header information was extracted, some of the information needs to be encapsulated (serialized) in order to be sent to the next FUs.
5. feedDecoders – Depending on the extracted information, the data stream is demultiplexed and sent to the according decoder (QBCR, SVA, or TFAN). The data is extracted from the input FIFO and redirected to the corresponding output FIFO.

At the FSM level, the conditions are implemented using state priorities that allow a state to be activated if the conditions are met. This method is useful when there could be two next states and the branch decision is made using the input data.

B. decoders:

The FU is a sub-network that implements the three decoders (QBCR, SVA, and TFAN). Each decoder has a separate input (as data streams) and a start decode signal.

The output consists of separate streams of data (separate data for each needed structure) with decoded information for each decoder. The data is not serialized into a single stream to increase the performance. In
some cases multiple data can be read at the same time.

Each decoder, as a part of the sub-network, has separate input and output connections acting independently.

**Decoders Sub-Network structure:**

B.1. decoderQBCR:
This FU is a sub-network that has serial data and synchronization signals as input, and produces the decoded data structures (as parallel serial data). This data is used as input for the decodedDataMerger FU.

![Figure 17. decoderQBCR FU schematic](image)

**Decoder QBCR Sub-Network structure:**

![Figure 18. decoderQBCR sub-network schematic](image)
B.1.1 **decodeQBCRHeader**

In order to extract all the header information from the serial stream and to use it in the decoding process, the data needs to be de-serialized and processed. The header structure provides enough information to extract and convert data types from 8bits serial data to different data types such as bit (Boolean), 32 bits signed or unsigned integers (unfortunately at this time Orcc CAL does not support float data type).

The FSM that defines the behavior of the FU has the following states:

1. **checkSegmentStart** - if the synchronization signal is true (Boolean value) for software and logic one for hardware the decoding process starts by starting the next FSM state or else it remains idle.
2. **initDecoderQBCR** - flushes all variables to zero or default value (ensuring that there will be no false values for the initial or the next processing stage.
3. **decodeHeader** – reads a HEADER_SIZE number of tokens and starts processing them. The function is blocked in read mode until all the tokens are read (some delays can occur when waiting for the FIFO to fill up the necessary number of tokens). Once all the data is available and stored into an array (8bit integer as read from the data stream), it processes the data by applying different operations to obtain the desired data type.

**Example:**

a. to obtain 8 boolean values (ccw, solid, convex, colorPerVertex, normalPerVertex, otherAttributesPerVertex, isTriangularMesh, markerBit) from a byte value the following operations are applied:

\[
ccw := (\text{segmentQBCRData}[8] & \text{CCW\_MASK}) \neq 0; \quad \text{where}
\]

\[
\text{uint (size=8) segmentQBCRData};
\]

\[
\text{uint \text{CCW\_MASK} := 0x00000001};
\]

Another approach with the same result could be an “if condition” to test if the first bit is 1 or 0.

b. To obtain a 32 bits value from 4 bytes (numberOfCoord,numberOfNormal, numberOfCoord,numberOfTexCoord,numberOfColor,numberOfOtherAttributes etc) the following operations are applied (shift and add):

\[
\text{numberOfCoord := segmentQBCRData[9] + (segmentQBCRData[10] \ll 8) + (segmentQBCRData[11] \ll 16) + (segmentQBCRData[12] \ll 24); \quad \text{where}
\]

\[
\text{uint (size=8) segmentQBCRData};
\]

\[
\text{uint ( size = 32 ) numberOfCoord};
\]

Another approach with the same result could be by applying multiply and add operations.
Focusing on the resulted project as software or hardware target, high level operations can be used in the project design with more efficient results for the software approach. A high level operation (such as multiplication or sqrt) will result in either adding logic to the hardware design, thus increasing the area, or replacing it with proprietary modules which depend on the implementation technology.

Since the stream structure is the same for all decoders, this FU can be used to split data for any one of them without modifying the implementation code.

**B.1.2 decoderQBCRData**

This FU is a sub-network that has as multiple inputs the data extracted from the header and produces the decoded data structures (as parallel serial data). This data is used as input for the decodedDataMerger FU.
B.1.2.1 **decoderQBCRDataSplit**

The main purpose of this FU is to extract the data from the serial stream as separate parallel structures and convert data if necessary. Based on the header information, if present, some additional information is extracted (array size, such as Array index information, QP information, quantMin and quantRange information etc).

Additional information is pushed to the next FUs to make the processing possible (example: QP, quantMin, quantRange is sent directly to the inverseQuantization FU).

The size information is useful inside the FU (to split the serial stream correctly) and it is also pushed to the next FUs as tokens to make possible the difference between two sequential data segments.

Depending on the information from the header, the encoded data is de-serialized and pushed to the corresponding output FIFO as tokens (at this point the data is still represented on 8 bits as in the input stream). The output data of this FU is represented as multiple serial streams in parallel.

Since the stream structure is the same for all decoders, this FU can be used to split data for any one of them without modifying the implementation code.

B.1.2.2 **inverseQuantization**

This FU implements all the three inverse quantization types and can be used for any of the three decoder types. Based on the information provided by QP it uses the appropriate inverse function with the quantMin and quantRange as parameters.

The input serial data is represented on 8 bits but after processing it, the resulted data is represented as unsigned integers on 32 bits. The number of read tokens differs depending on the inverse operation type. This operation is not dependent on array size since it uses one element (1, 2 or 3 values) and processes the entire input stream.

B.1.2.3 **decoderQBCRCord, decoderQBCRNormal, decoderQBCRColor and decoderQBCTRTexCoord, decoderQBCROtherAttributes**

For the QBCR implementations these FUs have the purpose to check the serial data and to push the new values to the output FIFO.

The input values are synchronized by reading a certain number of tokens from all the data streams at the same time. The array data is delayed compared to the index data (since the array data is processed by the inverseQuantization module while the index data is pushed directly to the FU).
C. **decodedDataMerger:**

It takes streams of decoded data (corresponding to each decoder) and merges the data into the data structures needed to display the mesh.

*Figure 23. decodedDataMerger FU schematic*
### Functional Units usage

<table>
<thead>
<tr>
<th>FU name</th>
<th>Sub-Network</th>
<th>Used in QBCR</th>
<th>Used in SVA</th>
<th>Used in TFAN</th>
<th>General use</th>
</tr>
</thead>
<tbody>
<tr>
<td>source</td>
<td>NO</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>X</td>
</tr>
<tr>
<td>SPS</td>
<td>NO</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>X</td>
</tr>
<tr>
<td>Display3D</td>
<td>NO</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>X</td>
</tr>
<tr>
<td>decoderDEMUX</td>
<td>NO</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>---</td>
</tr>
<tr>
<td>decoders</td>
<td>YES</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>---</td>
</tr>
<tr>
<td>decodedDataMerger</td>
<td>NO</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>---</td>
</tr>
<tr>
<td>decoderQBCR</td>
<td>YES</td>
<td>X</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>decoderSVA</td>
<td>YES</td>
<td>---</td>
<td>X</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>decoderTFAN</td>
<td>YES</td>
<td>---</td>
<td>---</td>
<td>X</td>
<td>---</td>
</tr>
<tr>
<td>decodeHeader</td>
<td>NO</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>---</td>
</tr>
<tr>
<td>decodeQBCRData</td>
<td>YES</td>
<td>X</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>decodeSVAData</td>
<td>YES</td>
<td>---</td>
<td>X</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>decodeTFANData</td>
<td>YES</td>
<td>---</td>
<td>---</td>
<td>X</td>
<td>---</td>
</tr>
<tr>
<td>decodeDataSplit</td>
<td>NO</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>---</td>
</tr>
<tr>
<td>inverseQuantization</td>
<td>NO</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>---</td>
</tr>
<tr>
<td>inverseBinarization</td>
<td>NO</td>
<td>---</td>
<td>X</td>
<td>X</td>
<td>---</td>
</tr>
<tr>
<td>inversePrediction</td>
<td>NO</td>
<td>---</td>
<td>X</td>
<td>X</td>
<td>---</td>
</tr>
<tr>
<td>decodeQBCRCoord</td>
<td>NO</td>
<td>X</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>decodeQBCRNormal</td>
<td>NO</td>
<td>X</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>decodeQBCRColor</td>
<td>NO</td>
<td>X</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>decodeQBCRTexCoord</td>
<td>NO</td>
<td>X</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>decodeQBCROtherAttributes</td>
<td>NO</td>
<td>X</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
</tbody>
</table>
5. Conclusions

This concept permits an implementation of a flexible design by dividing the top project into FUs and sub-networks that act independently and communicate using a pre-defined method thus making the reconfiguration of the design based on the data stream possible.

In this contribution we presented a first implementation of one component of SC3DMC, namely QBCR by using RVC-CAL. After introducing the main concepts of the language, we employed it for designing the FU needed for 3D mesh decoding as well as the specific network of FU specific to QBCR". We recommend continuing the experiment and implementing a full SC3DMC.

Acknowledgments

The authors gratefully acknowledge the help to understand CAL and the valuable comments from Jérôme Gorin and Mickael Raulet.