High-Level Synthesis for Intel and Xilinx FPGAs

Version 1.0

Executive Summary

Missing Link Electronics (MLE) has been an early adopter of High-Level Synthesis (HLS) for FPGAs. In particular for Domain-Specific Architectures which aim to accelerate algorithms and communication protocols HLS delivers on the promised benefits:

- Increased productivity as we can focus on the behavior and let HLS do the scheduling and resource mapping
- Better portability across FPGA device families and even across FPGA device vendors

This MLE Technical Brief describes our findings when using HLS to accelerate a telecommunications network protocol accelerator with FPGA. Driven by the project's need for short Time-to-Market major portions have been implemented in C/C++ using HLS. And given the application's large unit volume it was important to evaluate cost/performance across a set of Intel and Xilinx FPGA devices.

Our example uses Intel and Xilinx HLS to implement a specialized Packet FIFO. This Packet FIFO is then integrated as a particular design block into a block-based top-level design. Despite the fact that Intel HLS and Xilinx HLS behave quite differently, and do require special code, we did see a benefit from using HLS compared to "classical" RTL design using VHDL and/or Verilog HDL.

Hence, we encourage the reader to follow a similar approach.

1. High-Level Synthesis for FPGA Design

FPGAs offer a massively parallel architecture with advantages in terms of performance, cost and power consumption over conventional processors. However, the creation of RTL designs for FPGAs take significantly longer compared to software development, as design and verification are time-consuming processes (and, no, we do not refer to the longer compile times here, just to the “time to write code”).

High Level Synthesis (HLS) provides a level of abstraction for IP development by translating C++ specifications into RTL code. This accelerates IP creation because verification is done already at the C level and the synthesis tool generates efficient RTL code for a selected FPGA architecture. Xilinx and Intel provide extensive support for complex algorithmic descriptions with their HLS compilers. Libraries for integer, fixed point or floating point numbers as well as data types with arbitrary precision are provided. Furthermore interfaces for the standard interfaces AXI/Avalon are well integrated. RTL generation is device-specific, using on-chip elements such as DSP or memory, providing an efficient result for the platform.

The HLS design flow for Xilinx FPGAs is illustrated in the Figure below. The C/C++/SystemC description of the IP is generated and can be debugged and verified at the C level. The HLS compiler synthesizes the C code into an RTL description and generates the necessary adapters for C/RTL co-simulation. The synthesis result can be influenced by directives/pragmas and explored in terms of resources used, maximum frequency and initiation interval. The toolchain also offers the possibility to export the IP in the standardized IP-XACT format to further simplify the integration in the FPGA design process.

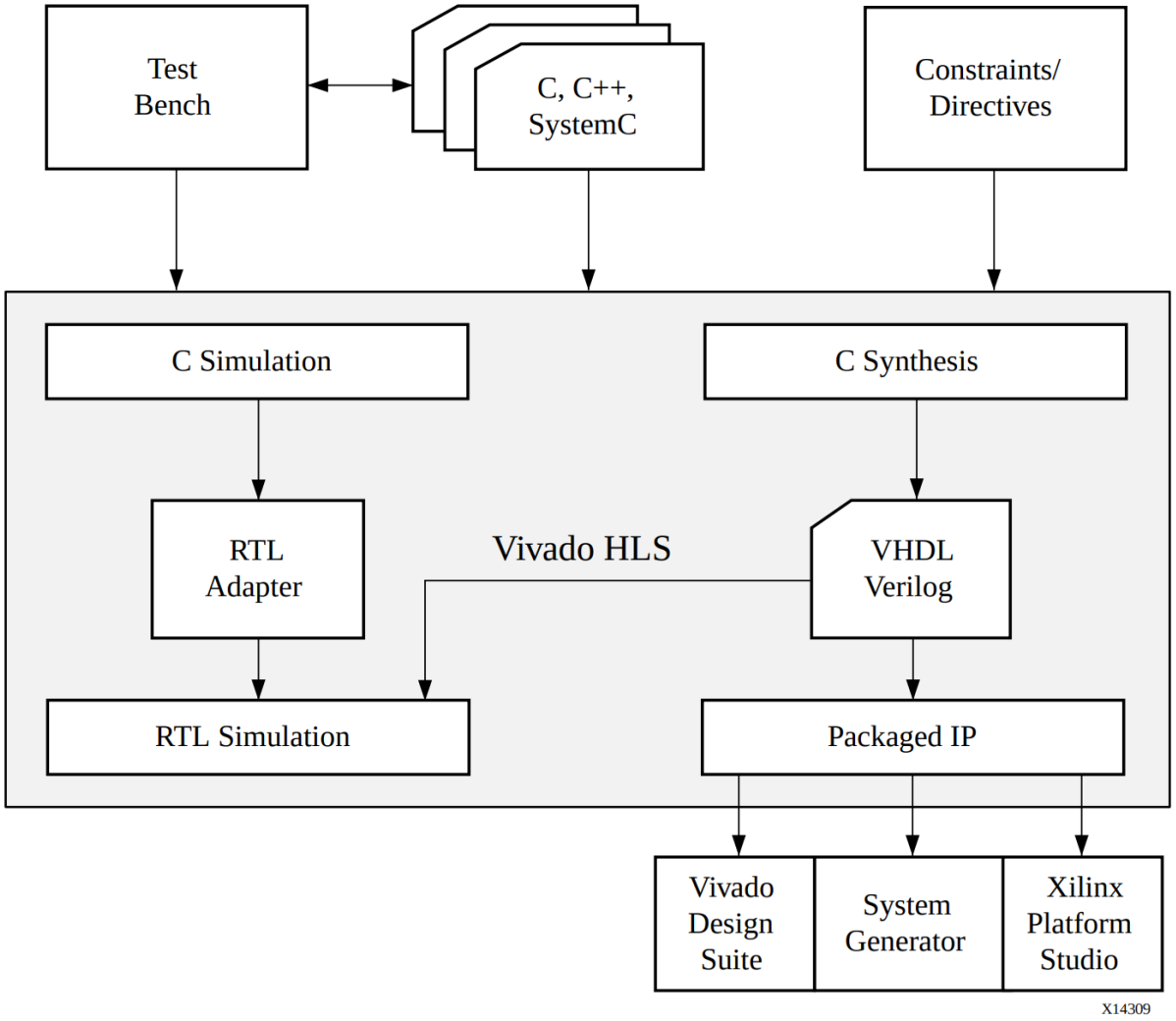

Figure 1 (source https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_1/ug902-vivado-high-level-synthesis.pdf)

2. Network Protocol Offloading Example

The example we are using within this Technical Brief comes from the telecommunication / networking world. In this case, Ethernet Frames according to IEEE 802.1Q Provider Backbone Bridging have to be processed. Sometimes, we refer to those Ethernet Frames simply as “packets”. Obviously, FPGAs can do such packet processing

much more efficiently than CPUs because of the capabilities for “Data-in-Motion” processing: FPGAs enable a fundamentally different model of computation by implementing what is called data flow processing, sometimes called stream processing. Unlike CPUs, which rely on loading/storing data from main memory, FPGA can implement data processing in a massively parallel fashion using deep pipelines similar to a "bucket brigade." For FPGA-based architectures to be fast and efficient, loading/storing data from main memory must be avoided (or at least minimized). The additional advantage of FPGAs is low and very deterministic processing latencies. To achieve data rates of several 10 GigE linerate it is necessary that the data path of the network packets goes through the FPGA and only the control flow is controlled by the software.

This concept shall be highlighted by the following comparison, using the “Baggy Pants” model of IEEE / IETF:

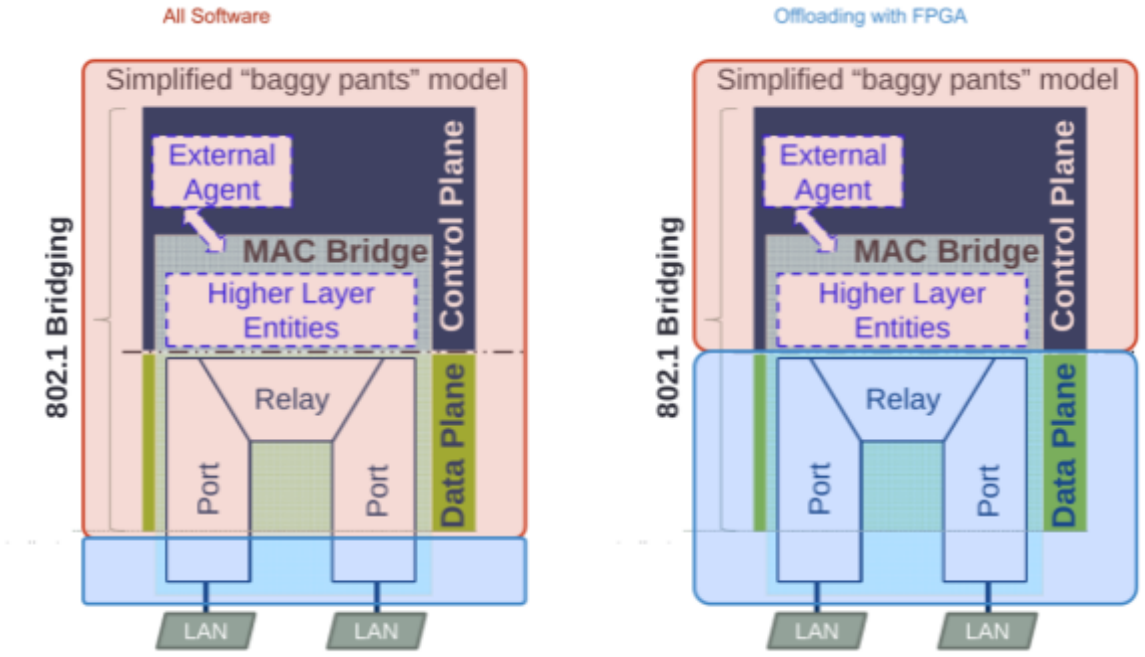

Figure 2

While we have used HLS to design and implement various blocks within the FPGAs, for this example we want to focus on the Packet FIFO block.

3. Packet FIFO Block Design with HLS

At the beginning of the network packet processing pipeline within our FPGA sits the Ethernet Media Access Controller (MAC) block which receives data from the physical interface and generates streaming interface packets. At this point two things need to be considered: First, the MAC block may invalidate packets if bit errors have been detected (for example, by checking the CRC of the Ethernet frame). Second, the MAC block receives Ethernet frames and directly outputs these on the streaming interface; neither throttling nor backpressure from a recipient "down-stream" is supported. Therefore, the interface from the MAC block to the packet processing pipeline is the Packet FIFO. This Packet FIFO is able to discard complete packets if the pipeline signals back pressure or if a packet is marked as defective by the MAC block. The Packet FIFO does not need any information about the protocol structure within the packets, but only works with the packet boundaries of the streaming interface. Here is the block diagram of such a subsystem:

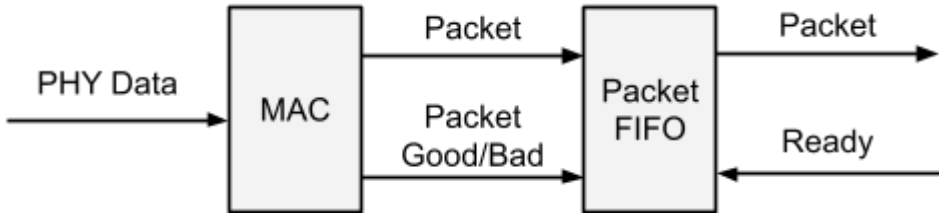

Figure 3

3.1. Functionality of the Packet FIFO

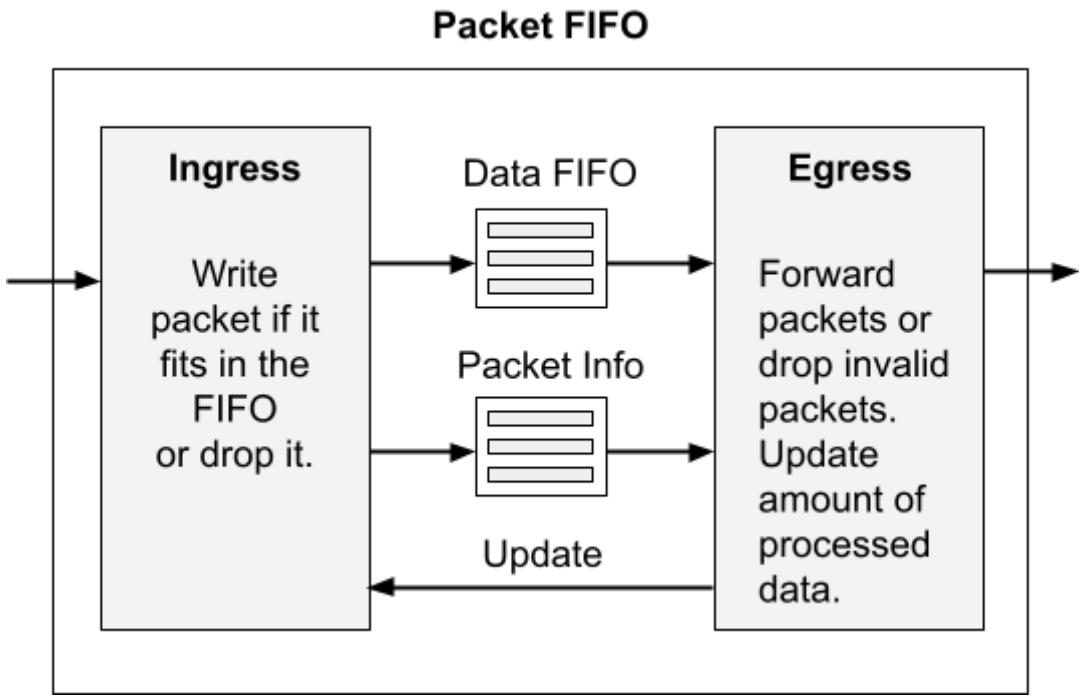

Figure 4

The Packet FIFO implements an internal FIFO that can buffer entire packets, plus read/write procedures that control the reading and writing of the FIFO. A packet passing through the Packet FIFO is written to the internal FIFO if there is enough space for a packet of maximum length. The *Ingress* module maintains a fill counter that is updated when data is taken from the FIFO. If a new packet arrives at the packet FIFO and there is not enough space in the internal FIFO, the packet is not written to the internal FIFO but discarded. When a full packet is written to the internal FIFO, the egress module is notified of an existing packet via the *Packet Info* interface. This information includes whether the MAC invalidated the packet during transmission. The *egress* module takes data words from the internal FIFO when a new packet info is present and the pipeline shows no backpressure. The packet is then forwarded from the internal FIFO or discarded if the packet info indicates an invalid packet.

In order to show the differences of an HLS implementation of the Packet FIFO between Intel and Xilinx and to outline the implementation of the module, some code snippets follow.

3.2. Design with Intel HLS

The Packet FIFO can be implemented as a System of Tasks. The ingress and egress function are executed in parallel by using the launch function.

```
component
hls_always_run_component
void packet_fifo() {
»       ihc::launch<ingress<slave, internal_fifo, packet_info, update> >();
»       ihc::launch<egress<internal_fifo, master, packet_info, update> >();
}
```

Figure 5

The communication of the tasks takes place through objects of the stream class (FIFOs). The stream class can be configured using templates to synthesize an avalon stream interface.

```
ihc::stream_in<ac_int<STREAM_DATA_WIDTH, false>,
»       ·······ihc::usesPackets<true>,
»       ·······ihc::bitsPerSymbol<BITS_PER_SYMBOL>,
»       ·······ihc::usesEmpty<true>,
»       ·······ihc::firstSymbolInHighOrderBits<true> >
»       ·······slave;
```

Figure 6

The depth of the stream buffer can be specified in the declaration.

```
ihc::stream<ac_int<STREAM_DATA_WIDTH, false>,
»       ····ihc::buffer<FIFO_DEPTH> >
»       ····internal_fifo;
```

Figure 7

The streams are declared outside the functions and their type is made known to the functions by template arguments.

```
template <auto &slave, auto &internal_fifo, auto &packet_info, auto &update>
void ingress() {
```

Figure 8

The stream class provides non-blocking read and write functions by which the data flow is realized.

```
»       auto data = slave.tryRead(success, sop, eop, empty);
»       if (!drop & success) {
»       »       internal_fifo.tryWrite(data, sop, eop, empty);
»       »       buffer_fill++;
»       }
```

Figure 9

Variables that should keep their value over the function calls (e.g. counters or flags) are declared static.

An external start/stop interface is omitted, since the packet FIFO should process packets as soon as they are available at the ingress interface. The keyword *hls_always_run_component* is specified together with the component declaration for this purpose.

3.3. Design with Xilinx HLS

In Xilinx HLS, functions in a System of Tasks are executed in parallel by declaring a DATAFLOW region. The interface synthesis is instructed by the INTERFACE pragmas to generate an AXI4 stream interface from the function parameters of the HLS stream class. The internal FIFO is also created as an object of the stream class and the depth of the FIFO is configured by a pragma as well.

```
template <int TDW>
void packet_fifo(hls::stream<axis_klu<TDW, 1> > &slave,
»       »       ·hls::stream<axis_kl<TDW> > &master)
{
#pragma HLS DATAFLOW
#pragma HLS INTERFACE axis port=slave register both
#pragma HLS INTERFACE axis port=master register both

»       static hls::stream<axis_kl<TDW> > internal_fifo;
#pragma HLS stream variable=internal_fifo depth=FIFO_DEPTH

»       //...

»       ingress<TDW>(slave, internal_fifo, packet_info, update);
»       egress<TDW>(internal_fifo, master, packet_info, update);
}
```

Figure 10

The ingress function accepts the stream objects as reference. The PIPELINE II=1 pragma instructs the compiler to reach an initiation interval.

```
template <int TDW>
void ingress(hls::stream<axis_klu<TDW, 1> > &slave,
»       ·····hls::stream<axis_kl<TDW> > &internal_fifo,
»       ·····hls::stream<ap_uint<1> > &packet_info,
»       ·····hls::stream<ap_uint<8> > &update)
{
#pragma HLS PIPELINE II=1
```

Figure 11

The stream class has methods to check if a stream is full or empty, which can be used to control the flow. A packet is read from the slave interface and written to the internal fifo if both interfaces can handle data.

```
»       if (!slave.empty() && !internal_fifo.full()) {
»       »       axis_klu<TDW, 1> beatu;

»       »       s_axis.read(beatu);
»       »       if (!drop) {
»       »       »       internal_fifo.write(axis_kl<TDW>(beatu));
```

Figure 12

3.4. Results

After synthesizing an IP, reports are generated that are useful for analyzing the components in terms of area, loop structure, memory consumption, component pipeline, and maximum frequency. To get an impression of the generated reports, you can find screenshots from the Intel and Xilinx synthesis reports below.

The estimation of resources used is listed for each subcomponent in the Resource Utilization View. Of course, the FPGAs from Xilinx and Intel have different primitives for BRAM/FFs/LUTs and therefore no numerical values shall be compared.

Quartus Estimated Resource Utilization Summary

Name	Source Location	ALM	ALUT	REG	MLAB	RAM	DSP
Quartus Fitter: Full Design (All Components)(?)		TBD		TBD		TBD	TBD
Device		427200		1708800		2713	1518

HLS Estimated Resource Utilization Summary

Name	Source Location	ALM	ALUT	REG	MLAB	RAM	DSP
packet_fifo	packet_fifo.cpp:120		22	11	0	0	0
read_ingress	packet_fifo.cpp:56		155	89	1	1	0
write_egress	packet_fifo.cpp:84		86	64	6	1	0
Compile Estimated: Full Design (All Components)			263	164	7	2	0

Figure 13: Intel Resource Utilization Report

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	-	-	-
FIFO	11	-	279	633	-
Instance	-	-	604	955	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	-	-
Register	-	-	-	-	-
Total	11	0	883	1588	0
Available	1824	2520	548160	274080	0
Utilization (%)	~0	0	~0	~0	0

Figure 14: Xilinx Resource Utilization Report

The Loop Analysis Report verifies that the IP is fully pipelined with an Initiation Interval (II) of 1, which means that the IP is capable of processing one data word per clock cycle. If target II is not achieved, there may be dependencies in the code that prevent the compiler from pipelining the IP. The table also shows the latency, which describes the number of clock cycles required to complete one iteration of a function. Reducing the latency often results in a higher maximum frequency.

Loop Analysis

Name	Source Location	Pipelined	II	Scheduled fMAX	Latency	Speculated Iterations	Max Interleaving Iterations	Brief Info
Component: packet_fifo	packet_fifo.cpp:120							Task function
packet_fifo.B1.start (Component invocation)		Yes	~1	240.0	1	n/a	1	

Figure 15: Intel Loop Analysis Report

Latency

Summary

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
2	2	12.800 ns	12.800 ns	1	1	dataflow

Figure 16

The Schedule Viewer can be used to identify latency bottlenecks in the design. It visualizes the estimated start and end clock for functions, as well as the dependencies and execution order between instructions.

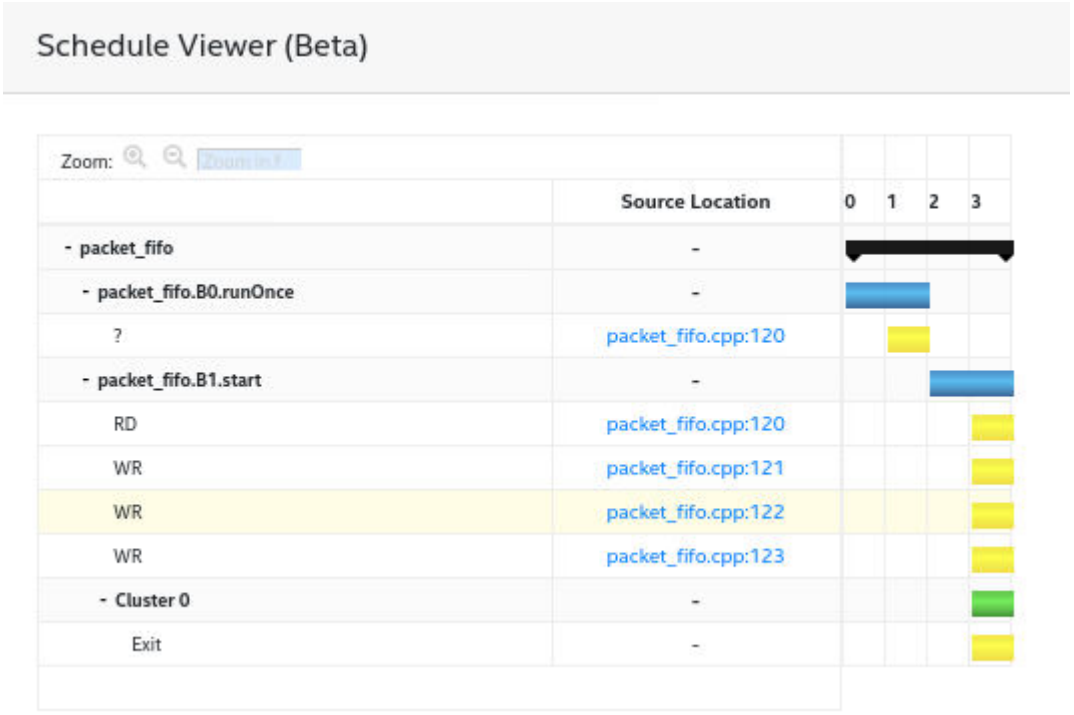

Figure 17: Intel Schedule Viewer

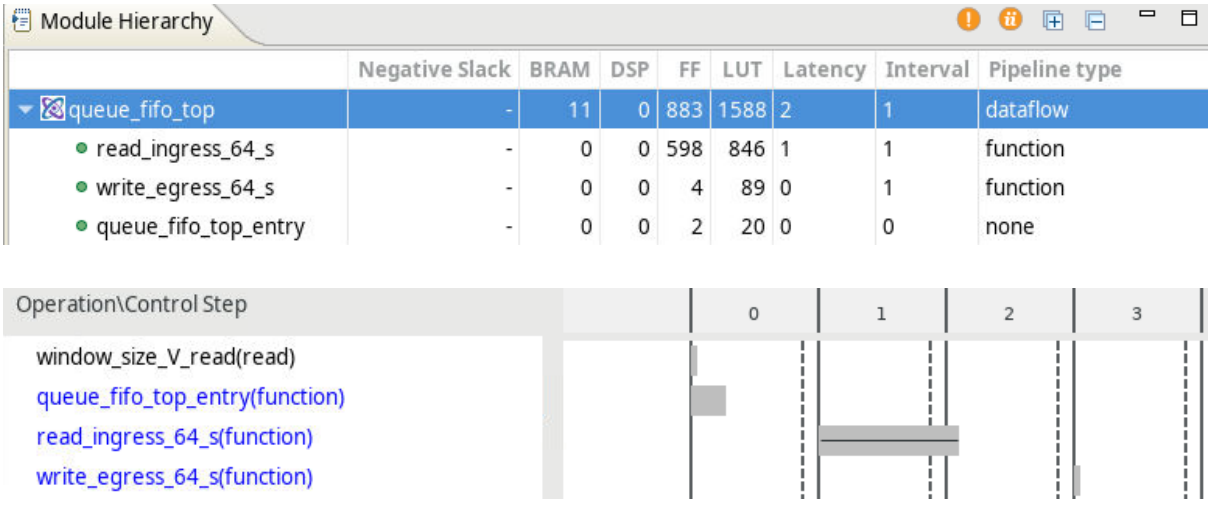

Figure 18 and 19: Xilinx Schedule Viewer. The ‘Packet FIFO’ is called ‘Queue FIFO’ in this implementation.

4. Conclusion and Backgrounder

Despite the fact that Intel and Xilinx HLS both input C code the actual C code implementation looks quite different. However, the concepts for communication between functions and parallel execution remain the same.

One general challenge in HLS is to find a C description that achieves the desired initiation interval of 1 for the pipeline. If there are dependencies on variables in the code, it may not be possible for the compiler to schedule the instructions so that execution takes place within one single clock cycle. Another difficulty can be to achieve the desired frequency for a synthesized circuit. A rewrite of the code may be necessary in these situations.

But the advantages of HLS are obvious. The verification of a module at C level is significantly faster than with the classic RTL approach. Also the abstraction of e.g. memory accesses, handshaking, on-chip streaming interfaces etc. allows a faster and less error-prone implementation of a module. Also the automatic IP-XACT export of the modules allows a good integration into the design flow, both for Intel Quartus as well as for Xilinx Vivado. The automatic generation of driver stubs for memory mapped control and status registers allows HLS IP to be well integrated into the software flow.

5. References

Whether you are new to HLS or whether you have been using it before, below are links to a couple of documents we consider worth reading:

- Bill Jenkins: “Intro to FPGA Flows Overview”, Intel Programmable Solutions Group, https://www.lrz.de/services/compute/courses/archive/2019/2019-05-21_hfpg1s19/
- Xilinx UG998 (v1.0), “Introduction to FPGA Design with Vivado High-Level Synthesis”, July 2, 2013 [At MLE we very much like this “old” one because of the great introductory section] https://www.xilinx.com/support/documentation/sw_manuals/ug998-vivado-intro-fpga-design-hls.pdf

- Ryan Kastner, KJanarbek Matai, Stephen Neuendorffer: “Parallel Programming for FPGAs, the HLS Book”, https://arxiv.org/abs/1805.03648
- Accellera: “Tutorial: Focusing on High-Level Synthesis and Functional Coverage for SystemC”, [Please watch at least Mike Meredith's presentation which starts 18:45], https://www.accellera.org/resources/videos/systemc-tutorial-2019
- Nick Ni (Xilinx): “Two techniques to hardware optimize an image processing algorithm using SDSoC”, [a great visualization about the whys and hows of FPGA-based stream processing] https://www.xilinx.com/video/soc/hardware-image-algorithm-using-sdsoc.html
- TutorialsPoint: “C++ Tutorial”, https://www.tutorialspoint.com/cplusplus/index.htm
- Intel: “Intel High Level Synthesis Compiler Pro Edition - Reference Manual”, March 29, 2021, https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/hls/mnl-hls-reference.pdf
- Xilinx UG902 (v2020.1): “Vivado Design Suite User Guide - High-Level Synthesys”, May 4, 2021, https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_1/ug902-vivado-high-level-synthesis.pdf
- Xilinx XAPP1209 (v1.0.1): “Designing Protocol Processing Systems with Vivado High-Level Synthesis”, August 8, 2014, https://www.xilinx.com/support/documentation/application_notes/xapp1209-designing-protocol-processing-systems-hls.pdf

Authors and Contact Information

Andreas Braun, Sr. Engineer, Missing Link Electronics GmbH
Endric Schubert, PhD, CTO, Missing Link Electronics, Inc.

Missing Link Electronics, Inc.
2880 Zanker Road, Suite 203
San Jose, CA 95134, USA
+1-408-475-1490

Missing Link Electronics GmbH
Industriestrasse 10
89231 Neu-Ulm
Germany
+49-731-141149-0

www.missinglinkelectronics.com

MLE (Missing Link Electronics) is offering technologies and solutions for Domain-Specific Architectures, which focus on heterogeneous computing using FPGAs. MLE is headquartered in Silicon Valley with offices in Neu-Ulm, Germany.