Using Vivado-HLS for Structural Design: a NoC Case Study

Zhipeng Zhao
ECE Department
Carnegie Mellon University
Pittsburgh, PA 15213
zzhao1@andrew.cmu.edu

James C. Hoe
ECE Department
Carnegie Mellon University
Pittsburgh, PA 15213
jhoe@ece.cmu.edu

Abstract

There have been ample successful examples of applying Xilinx Vivado's "function-to-module" high-level synthesis (HLS) where the subject is algorithmic in nature. In this work, we carried out a design study to assess the effectiveness of applying Vivado-HLS in structural design. We employed Vivado-HLS to synthesize C functions corresponding to standalone network-on-chip (NoC) routers as well as complete multi-endpoint NoCs. Interestingly, we find that describing a complete NoC comprising router submodules faces fundamental difficulties not present in describing the routers as standalone modules. Ultimately, we succeeded in using Vivado-HLS to produce router and NoC modules that are exact cycle- and bit-accurate replacements of our reference RTL-based router and NoC modules. Furthermore, the routers and NoCs resulting from HLS and RTL are comparable in resource utilization and critical path delay. Our experience subjectively suggests that HLS is able to simplify the design effort even though much of the structural details had to be provided in the HLS description through a combination of coding discipline and explicit pragmas. The C++ source code can be found at http://www.ece.cmu.edu/calcm/connect_hls.

1 Introduction

Xilinx Vivado offers a style of high-level synthesis (HLS) that maps a restricted C function to a hardware module, eschewing the question of supporting the complete C language or mapping a complete C program. Commercial interests aside, this technology has garnered significant interest among researchers across domains as evident in the number of recent papers where Vivado-HLS is used for application development.

In a typical usage, the designer expresses the desired algorithm using C constructs in the function body. Such a C function could have many legal interpretations as a hardware module, differing widely in the timing and structure of the datapath and interface. In mapping C code to a datapath, Vivado-HLS can automatically make default implementation choices where the C specification is silent. Alternatively, the designer has the option to steer many of the mapping decisions using Vivado-HLS provided pragmas [20].

In our own work, we have used Vivado-HLS to develop compute kernels of signal and vision processing algorithms. We have found the combination of HLS design, debug, and analysis environments to indeed improve our productivity without compromising quality relative to what we could have done using register-transfer-level (RTL) Verilog. In our uses, although we are working through C and HLS, we have always approached the design with a skeletal model of the desired datapath in mind. And, thus far, we have been successful in coaxing Vivado-HLS into producing the intended datapath—and thus the expected quality of result—through a combination of pragmas and rewriting/restructuring of the code. *This led us to ask the question, can we use C and Vivado-style HLS*

for pure structural design? Moreover, would there be any benefits in using C and Vivado-style HLS for structural design? [1] (Ans: yes and sometimes.)

By structural design, we mean a design that carries strong explicit notions of synchronous sequential state elements and combinational next-state logic where the designer *wants* to control with precision the design's structure and timing, cycle-by-cycle and bit-by-bit. The difference between structural vs. non-structural descriptions is orthogonal from the style of a language or the language's level of abstraction. Synthesizable RTL subsets of Verilog, VHDL, and SystemC are naturally aligned for structural design descriptions. Chisel [1] and Bluespec [13], which borrow heavily from modern functional programming languages, are very high-level but still structural in nature (requiring explicit declarations of synchronous state variables and their state transition logic). In this paper, we aim to use C and Vivado-HLS to achieve cycle- and bit-level control of the desired datapath structure.

To understand the effectiveness of C and Vivado-HLS for structural design, we chose network-on-chip (NoC) routers as the subject of a design study. Except for perhaps the allocation logic, a NoC router is manifestly "structural". For this study, we set out to replicate the RTL router designs available through CONNECT [15]. To push the limit, we further attempted to describe complete NoCs with routers encapsulated as submodules.

We were ultimately successful in using C/C++ and Vivado-HLS to produce router modules that are exact cycle- and bit-accurate replacements for the full range of diversely parameterized CONNECT family of routers. Moreover, the routers synthesized by Vivado-HLS are comparable with the quality of corresponding CONNECT routers.

At the scope of a router, C and Vivado-HLS proved to be competent in capturing the structural intents and producing the desired outcome. More importantly, we found that the HLS design flow allowed us to address functional correctness separately from structural decisions. Relative to traditional RTL design flow, we were able to more easily and quickly achieve functional correctness due to C's sequential semantics and its development/debug environment. Afterwards, we could also more rapidly explore different structural design alternatives without worrying about breaking functional correctness. This decoupling of the functionality and structural aspects of a design resulted in a clearly perceivable productivity gain, even for this clearly structural design effort where the C and corresponding RTL routers descriptions must carry the same information with respect to the design's structure and timing.

On the other hand, when expanding the design scope to a NoC, we saw great interference from the semantics of C in creating a "netlist" of router submodule instances. We developed a solution involving describing the routers as C++ objects under an imposed coding discipline. All in all, our experience suggests that C/C++ and Vivado-HLS can be used to replicate arbitrary RTL designs of synchronous registers and combinational logic, but the benefits in doing so depend on the hierarchical nature of the design subject.

Outline. Following this introduction, Section 2 offers a brief background review of HLS and the structure of CONNECT routers. Section 3 explains how one could describe a router-like structure in C for HLS. Section 4 next discusses the challenges when expanding the design scope from individual routers to complete NoCs. Section 5 discusses the limiting cases in using C and Vivado-HLS for structural design. Section 6 presents the place-and-route evaluation results comparing HLS routers and NoCs with their CONNECT counterparts. Section 7 offers our conclusions.

2 Background

2.1 High-Level Synthesis

The rapidly growing interest in using FPGAs for compute acceleration has also boosted the interest in HLS as the path to simpler application development and a larger developer pool. Historically, HLS has not been exclusive to the FPGA domain and predates FPGAs [11]. Among the many abstractions and languages considered for high-level or behavioral specification, there has always been a strong emphasis on C for its practical simplicity and immense popularity. Over time, the

---

[1]To be precise, we are interested in C/C++ specifications of behaviors that are readable as C/C++ programs. Synthesizable SystemC, based on processes with sensitivity list, is used as an RTL language in the same way as synthesizable Verilog and VHDL.

range of work in C-to-hardware synthesis has spanned from those that only borrowed C's syntax for an RTL language (e.g., Perle1DC [16]) to those that compiled complete C language programs for execution as hardware (e.g., CASH [2]). At the same time, arguments have been raised against the suitability of C—an untimed, sequential language—for synthesizable hardware description [7].

To reduce the designer's workload, a behavioral specification is intentionally under-constrained to defer structural details and decisions to automation. As such, a given behavioral specification could have multiple correct implementations, varying in their datapath structure and/or timing, with corresponding consequences in their design quality of concern (performance, area, power, etc.). For some behaviors, it is possible that none of the implementations are "good" relative to alternatives such as executing in software. The great challenge for HLS is then in correctly filling in the missing details and decisions to arrive at a good implementation when one exists. Windh et al. offers a survey of the state of the art in HLS [18].

The HLS problem can be simplified by restricting the applicable inputs and/or the range of output structural options. For example, ROCCC specializes on mapping parallelizable loop nests to streaming pipelines [17]. Such specialization allows very high quality results for the intended/anticipated usage scenarios.

To support the complete C language and full programs, one could restrict the language constructs and the program regions synthesized to hardware. LegUp [3] and SDSoC [19] support a hybrid execution where the main program thread is compiled for execution on an embedded processor core. Only specially designated functions—presumably those that make sense to be in hardware—are synthesized to hardware modules. The compiler automatically inserts the required hardware and software interfaces; therefore calls to the hardware accelerated functions are transparent at the source code level. LegUp can also make use of threading semantics to support multiple concurrent, free-running hardware modules [4].

Vivado-HLS, descending from AutoPilot [5], limits itself to only the problem of mapping restricted C functions to modules. Moreover, Vivado-HLS selectively gives emphasis to language constructs and code structures that are most important to hardware-friendly algorithms. For example, Vivado-HLS does not support recursion and only optimizes loops with fixed loop bounds. Lastly, Vivado-HLS relies on an extensive suite of pragmas for designers to explicitly add structural and timing details when the compiler's default outcome is less than desired. As mentioned in the introduction, in the domain of signal and vision processing algorithms, our own work has used C and Vivado-HLS to good effect in arriving at the intended high-quality datapath by exercising strong structural control through code styles and explicit pragmas. This experience motivated our current curiosity in how effective is C and Vivado-HLS for pure structural design. Xilinx Application Note XAPP1209 [9] and XAPP1167 [12] illustrate how to use Vivado-HLS to develop a streaming structure comprising fully pipelined modules that interact only through stylized streaming interfaces. In this paper, we show how one could use Vivado-HLS to develop arbitrary structural designs of synchronous registers and combinational logic. In prior work, Kapre and Gray have used Vivado-HLS to generate standalone router module for Hoplite, a lightweight FPGA overlay NoC [8]. Lahti, et al. showed how to develop a cycle-accurate structural module of a I2C bus controller using Catapult C [10].

2.2 Structure of CONNECT Routers

We elected to use NoC routers for this structural design study. In particular, we set out to replicate the packet-switched routers available from the CONNECT NoC design generator [15]. There is no special significance in choosing routers as the design subject, except that they are structural. The key ideas developed in this paper are generalizable beyond routers.

The CONNECT NoC design generator is parameterized to generate packet-switched routers from a comprehensive design space. Basic router design parameters include the in- and out-degree, flit data width, and flit buffer size. More advanced parameters select major design options in flow control, virtual channel (VC), and allocation. At the next level, CONNECT is parameterized to construct a number of stylized or customized network topologies from routers. CONNECT is available through a GUI web-based portal that produces synthesizable RTL Verilog.[2] Evaluations have shown CONNECT-generated routers and NoCs are competitive with high-quality hand-coded RTL

---

[2]http://www.ece.cmu.edu/calcm/connect. Internally, the RTL design generator is coded in BSV and uses a Bluespec compiler [13] to produce Verilog.
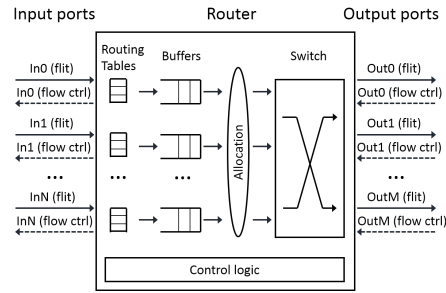
Figure 1: CONNECT router structural template.

designs [15]. Although not necessary for this paper, readers desiring a more thorough background in router and NoC design can refer to [6].

The diverse family of CONNECT routers share a common underlying structure depicted in Figure 1. Generically, a router is a module with some number of input ports and output ports. A flit, comprising data payload bits and route information bits[3], can arrive on an input port from time to time. The job of a router is to emit the flit eventually on an output port according to the flit's route information bits.

Upon entering the router, the flit's eventual output port is determined combinationally (by a look-up table for example) using its route information bits. Because the desired output port is not always immediately available, it is necessary to first buffer the incoming flit. As an optimization (discussed later in Section 5), the data payload and the route information portions of the flits can be physically held in different structures that logically operate in synchronization as one FIFO. On each cycle, the allocation logic considers the flits at the front of all FIFO flit buffers to dispatch a maximum number of flits to their non-conflicting output ports through a crossbar switch; flits not selected are deferred to the next cycle. This datapath from input ports to output ports is pipelined to increase throughput. Finally, because an input port's flit buffer can fill up under output port congestion, an input port needs to communicate with its upstream router using a flow-control protocol to ensure the upstream router never overflows the capacity of the flit buffer.

The basic CONNECT options affect the structure straightforwardly in the number of ports, the width of the datapath, and the depth of the buffers. The advanced CONNECT options primarily affect the combinational decision logic controlling the datapath. The virtual channel options affect structure—associating multiple FIFO flit buffers to each input port—as well as the combinational logic. In Section 3, we first describe how C and Vivado-HLS can be used very effectively to design these kinds of structures. In Section 4, we next describe the challenges and solutions in using C and Vivado-HLS to design NoCs (including, for example, a 2D-Mesh NoC, Figure 2), using router submodules.

3 Describing Structure in C

Although we try to be self-contained, this section is not a good primer for Vivado-HLS. Our usage is not orthodox. We require only a very basic subset of Vivado-HLS features.

Instead of a router, we use the running example of a simple switch module that steers the integer inputs on its two input ports `I1` and `I2` toward the appropriate output ports `Odd` and `Even`. It should be possible to completely understand how to describe the structure of this switch module, and then extrapolate how to describe the CONNECT router structure discussed in Section 2.2 [4]. Below, `switch_try()` is a starting attempt at describing the switch module.[5]

---

[3]It is a unique feature of CONNECT routers that each flit of a packet carry its own route information. This is an optimization to make use of the excess wiring resources expected when mapping regularly tiled structures onto an FPGA fabric [14].

[4]The source code for the examples in this section and the complete router can be found at http://www.ece.cmu.edu/calcm/connect_hls.

[5]For clarity, we omit in the examples the `ap_ctrl_none` pragma on the functions and the `ap_none` pragma on the output variables. They serve only to remove the default control signals that become extraneous in our use.
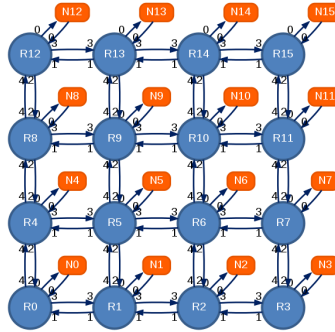
Figure 2: A 4-by-4 2D-Mesh using 5-by-5 routers in the interior, 4-by-4 routers on the edges, and 3-by-3 routers on the corners.

```
1  void switch_try(int I1, int I2, int *Odd, int *Even) {
2    if (I1%2) *Odd=I1;
3    else if (I2%2) *Odd=I2;
4
5    if (!(I1%2)) *Even=I1;
6    else if (!(I2%2)) *Even=I2;
7  }
```

Mapping Function to Module. Vivado-HLS maps the top-level function switch_try() to a corresponding logic module. The pass-by-value arguments I1 and I2 correspond to input ports. The pass-by-reference arguments Odd and Even, only dereferenced for writing, correspond to output ports. The body of the function specifies the calculation of the output values from the input values in each function invocation. This simple example—with no cycles or side-effects in the dataflow between input and output—would by default map to a combinational module provided the chosen target clock period is long enough.[6]

Flow Control.[7] The first attempt switch_try() is deficient in that if inputs I1 and I2 are both even or odd, input I2 is ignored and one of the output value (Even or Odd) is undefined. The next attempt switch_comb() adds flow-control handshakes.

```
1  void switch_comb(vDat I1, vDat I2,
2                vDat *Odd, vDat *Even,
3                bool *acpt1, bool *acpt2) {
4    *acpt1=*acpt2=(*Odd).v=(*Even).v=false;
5
6    if (I1.v && (I1.d%2))
7      {(*Odd).v=true; (*Odd).d=I1.d; *acpt1=true;}
8    else if (I2.v && (I2.d%2))
9      {(*Odd).v=true; (*Odd).d=I2.d; *acpt2=true;}
10   ... repeat 6~9 for Even ...
11 }
```

The inputs I1 and I2, and outputs Even and Odd have type struct vDat {bool v; int d;} where the .v field is used to indicate the validity of the .d data field. The function considers the inputs' validity in making steering decisions; the function also marks the validity of the outputs. Lastly, the function adds two Boolean outputs acpt1 and acpt2 to indicate if the corresponding inputs have been accepted.

---

[6]If the chosen target clock period is too short for the critical path, Vivado-HLS will break the logic into multiple cycles to meet timing. The #pragma HLS LATENCY max=0 can be added to force Vivado-HLS to ignore the target clock period restriction.

[7]Vivado-HLS can automatically introduce select styles of flow control (e.g., streams). Here, we are interested in the ability to build arbitrary flow-control protocols explicitly.

Sequential State and Pipelining. The module synthesized from `switch_comb()` remains combinational by default. Vivado-HLS could be directed to automatically synthesize a streaming pipelined module that would overlap multiple invocations of the function. In other words, the module would proceed ahead to accept a new invocation's inputs each cycle with the corresponding outputs emerging only some pipeline delay later. This streaming execution however is not what we need in the current context. We want to pipeline the path from `I1` and `I2` to `Odd` and `Even`, but we want `acpt1` and `acpt2` to remain combinational to qualify the current inputs `I1` and `I2`. An example of the desired 2-stage pipeline structure is described explicitly as `switch_2stage()` below.

```
1   void switch_2stage(...same as switch_comb...) {
2     static vDat L1, L2;
3     static Path Lreq;
4
5   /* ---- stage 2 ---- */
6     Path grnt=allocate(Lreq);
7     if (grnt.L1xOdd) *Odd=L1;
8     else if (grnt.L2xOdd) *Odd=L2;
9     else (*Odd).v=false;
10    ... repeat 6~9 for Even ...
11
12  /* ---- stage 1 ---- */
13    if (grnt.L1xOdd||grnt.L1xEvn) L1.v=false;
14    if (I1.v && (!L1.v)) { *acpt1=true; L1=I1; }
15    else *acpt1=false;
16    ... repeat 13~15 for I2 and L2 ...
17
18    Lreq=decode(L1,L2);
19  }
```

The synthesized module will execute one invocation of this function to completion once per cycle. In our use of C and Vivado-HLS, a function invocation captures the events of a clock cycle, starting with the combinational propagations based on current state and input values, ending with the synchronous next-state update.

We can introduce sequential states using static variables that retain their values across invocations. The static variables `L1`, `L2`, and `Lreq` are used as pipeline latches (lines 2 and 3). In any given invocation of `switch_2stage()`, `L1` and `L2` holds latched values of `I1` and `I2` from previous invocations. Another pipeline latch `Lreq` holds pre-decode connection requests in `struct Path {bool L1xOdd; bool L1xEven; bool L2xOdd; bool L2xEven;}` which is a bitmap of the four possible connections needed by `L1` and `L2`.

The code corresponding to Stage 2 (lines 6∼10) sets the output `Odd` and `Even` based on the pipeline latches `L1`, `L2`, and `Lreq`. For brevity, we assume a combinational function `Path allocate(Path)` exists to compute a bitmap of which requested connections are granted (line 6).

The code corresponding to Stage 1 (lines 13∼18)—starting only after Stage 2 code is finished using the old values of `L1`, `L2` and `Lreq`—sets the new values of the pipeline latches based on their old values and current input values. Please note that the Stage 1 code is written in "procedural style" where `L1`, `L2` and `Lreq` can be read and written multiple times in describing their "next-state" values combinationally; only the final values at the end of the function are latched. Again for brevity, we assume a combinational function `Path decode(vDat,vDat)` exists to compute a bitmap of the requested connections (line 18).

Hierarchy and Modularity. This `switch_buffered()` example uses an assumed synthesizable finite FIFO class and `switch_2stage()` in a hierarchical structure. Once again, the synthesized module for `switch_buffered()` will execute one invocation to completion once per cycle. The declared static FIFO objects `F1` and `F2` will buffer the inputs before they are switched onto the outputs. In lines 6∼10, the fronts of the FIFOs are presented to `switch_2stage()`. The "call" to `switch_2stage()` sets top-level outputs `Odd` and `Even` directly. In this context, we could transparently replace `switch_2stage()` by `switch_comb()` even though they have different timing, because they obey the same flow-control protocol. Inputs `I1` and `I2` are accepted into `F1`

and `F2`, respectively, as long as the FIFOs are not full (lines 12∼14). The front of a FIFO is popped if its value is accepted (line 15).

```
1  void switch_buffered(...same as switch_comb...) {
2    static FIFO<int> F1, F2;
3    bool okX1, okX2; // combinational "wire" temporaries
4    vDat frontX1, frontX2; // combinational "wire" temporaries
5
6    frontX1.v=!F1.empty(); frontX1.d=F1.front();
7    frontX2.v=!F2.empty(); frontX2.d=F2.front();
8
9    switch_2stage(frontX1, frontX2,
10               Odd, Even, &okX1, &okX2);
11
12   if (!F1.full() && I1.v)
13     {F1.push(I1.d); *acpt1=true;}
14   else *acpt1=false;
15   if (okX1) F1.pop();
16
17   ... repeat 12~15 for I2 and F2 ...
18 }
```

Parameterized Design. This final example shows how C language facilities can be used for more maintainable and scalable design capture. Below, switch_buffered_N() is derived from switch_buffered() for a parameterized number of inputs. We assume a parameterized switch_2stage_N() exists.

```
1  void switch_buffered_N(vDat I[N],
2                     vDat *Odd, vDat *Even,
3                     bool acpt[N]) {
4    #pragma HLS ARRAY_PARTITION variable=I complete dim=1
5    #pragma HLS ARRAY_PARTITION variable=acpt complete dim=1
6
7    static FIFO<int> F[N];
8    bool okX[N];
9    vDat frontX[N];
10
11   for(int i=0;i<N;i++){
12   #pragma HLS UNROLL
13     frontX[i].v=!F[i].empty(); frontX[i].d=F[i].front();
14   }
15
16   switch_2stage_N(frontX, Odd, Even, okX);
17
18   for(int i=0;i<N;i++) {
19   #pragma HLS UNROLL
20     if (!F[i].full() && I[i].v)
21       {F[i].push(I[i].d); acpt[i]=true;}
22     else acpt[i]=false;
23     if (okX[i]) F[i].pop();
24   }
25 }
```

In switch_buffered_N(), the inputs have been declared as an array I[N]. Similarly, the per-input flow-control signal acpt needs to be an array. We include the necessary HLS ARRAY_PARTITION pragma (lines 4 and 5) to instruct Vivado-HLS to synthesize the input array I[N] and the output array acpt[N] as N concurrent ports instead of the default memory-array interfaces.

In the function body, the per-input local variables F, okX and frontX also become arrays parameterized by N. We use a fixed-bound N-iteration loop to scalably describe the operations that were previously repetitiously specified for each non-indexed input I1 and I2. Please note, without the HLS UNROLL pragma (lines 12 and 19), Vivado-HLS will execute the loop sequentially over multiple clock cycles.
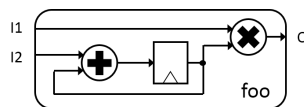
Figure 3: A prototypical Mealy state machine.

Discussions. By elaborating on the same constructions in this section, we have been able to fully replicate the router designs generated by CONNECT, with its full range of parameterizations and features. With careful control of timing and structure, the routers produced through Vivado-HLS are exact replacements of their CONNECT counterparts, bit-for-bit, cycle-for-cycle at the module interface. Moreover, Section 6 will present evaluation results that show the RTL and HLS counterparts are comparable in their resource utilization and critical path delay after place-and-route.

There is no magic in Vivado-HLS. We have been able to arrive at the desired structural design only by introducing the structural information explicitly. Syntax aside, the C functions for the switch examples above (and also for the CONNECT routers) are strongly "register-transfer" in nature. However, in the design study, we have found developing and debugging a structural design is simpler and faster in the Vivado-HLS design flow than using a conventional RTL flow. We believe a major reason is the ability to better separate the concerns for functionality and for performance (as effected through timing and structure) in the Vivado-HLS design flow.

Paying no attention to structural consequences, the example C functions we saw in this section (and same for the CONNECT routers) have natural sequential program readings of the intended functional behaviors. As such, we were able to rely on the convenience of C testbenches and C debugging tools to first establish a high degree of confidence in a design's functional correctness before separately addressing structural and performance design issues. The specifications of functionality and structure in conventional RTL descriptions are too deeply intertwined to do this effectively.

As a final point, we believe our C code for the CONNECT routers would not be out-of-place in a software cycle-based NoC simulator. On the other hand, we do not expect a C router model written expressly for a software simulator—without consideration for synthesis implications—would synthesize to the modeled structure or even be synthesizable at all. While there are many ways to express the same behavior in C, not all of them lead to efficient implementations; it is neither the case in normal C compilation nor in HLS.

4 Composing Modules in C

The Verilog modules of the routers we generated using Vivado-HLS can be readily instantiated in the next enclosing design hierarchy using standard structural design methodologies, whether textual or graphical. In fact, we could rely on CONNECT to generate the desired NoC topology as a Verilog netlist and substitute the router modules from this work for the CONNECT generated counterparts. Bolstered by the positive experience in implementing routers using C and Vivado-HLS, our design study next tried to implement the 4-by-4 2D-Mesh in Figure 2.

This seemingly benign task—making a netlist of multiple submodules—turned out to face much greater resistance from C's semantics. In this section, we discuss the issues and the workarounds in a general context with the help of a prototypical Mealy state machine module (Figure 3). This module `foo` has two integer inputs `I1`, `I2` and an integer output `O`. Inside the module, there is an integer register that accumulates the sequence of values presented on `I2` each cycle. The output `O` is the accumulated sum scaled by `I1`. The corresponding C function `foo()` is below.

```
1  void foo(int I1, int I2, int *O) {
2    static int L=INIT_VAL; // latch
3
4    *O=I1*L; // read current-L
5    L=I2+L; // assign next-L
6  }
```

Function Call ≠ Module Instantiation. Based on Section 3, it should be clear submitting `foo()` as the top-level function to Vivado-HLS produces the desired module `foo` in Figure 3. What happens if we want to instantiate two copies of the module `foo` as submodules within the top-level
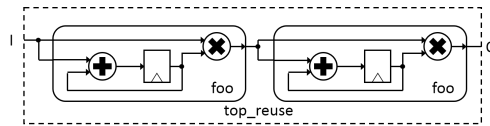
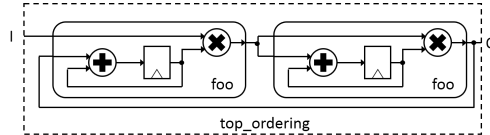Figure 4: Top module `top_reuse` with two instances of the submodule `foo`.

Figure 5: Top module `top_ordering` with two instances of the submodule `foo` where the output of each submodule drives the input of the other.

module `top_reuse` in Figure 4. Since functions and modules both serve as the vehicle for design modularity and reuse in their respective domains, a naive attempt at describing the module `top_reuse` might produce the following, where the function `foo()` is called twice.

```
1  void fxn_reuse_try(int I, int *O) {
2    int tmp; // output of left module
3
4    foo(I,I,&tmp); // left in figure
5    foo(tmp,tmp,O); // right in figure
6  }
```

Unfortunately, calling a C function twice results in two executions of the same function instance; it does not result in two copies of the function. The use of static variable L in `foo()` makes this distinction inescapable.[8] Under C semantics, there is only one instance of `foo()` and one instance of the static variable L. The repeated calls to `foo()` *must* update the same static variable L according to the semantics of C.

Function Evaluation not Reactive. Consider next the top-level module `top_ordering` in Figure 5 again with two instances of the submodule `foo`. This time, the input I2 of the left module is driven by the output O of the right module. Please note that no combinational cycle is formed. A naive attempt at a corresponding C function might lead to the following.

```
1  void fxn_ordering_try(int I, int *O) {
2    int tmp1; // output of left module
3    int tmp2; // output of right module
4
5    foo<1>(I,tmp2,&tmp1); // left in figure
6    foo<2>(tmp1,tmp1,&tmp2); // right in figure
7    *O=tmp2;
8  }
```

We use templatized function with different instance names `foo<1>` and `foo<2>` to workaround the instantiation confusion. However, this example highlights a deeper problem arising from that, in the module `top_ordering`, each submodule's output O drives the input of the other submodule. There is no ordering of calling the functions `foo<1>` and `foo<2>` in `fxn_ordering_try()` that can produce the desired behavior. In the ordering chosen by `fxn_ordering_try()`, `foo<1>` is called with un-initialized variable tmp2 as input; tmp2 is not set until after `foo<2>` is finished.

It may appear the impasse is caused by a cycle in the netlist. The real culprit is more generally the order of evaluation. A combinational value in a logic circuit should re-evaluate spontaneously in *reaction* to changes in its dependent values. In Verilog/VHDL/SystemC, combinational evaluation

---

[8]Note, in Vivado-HLS, multiple calls to a purely combinational function (no side-effect through static or global variables) from a combinational context will result in replicated instances. Multiple calls to a purely combinational function from a sequential context will result in a single instance reused over different clock cycles.

is modeled either as *continuous assign* statements or as processes whose sensitivity list contains all the dependent signals. For reactive evaluations in Verilog/VHDL/SystemC, the order of declaration does not matter; the evaluation order develops dynamically, sometimes including extraneous glitches and redundant evaluations.

The function `fxn_ordering_try()` looks like—but is not—a netlist. The functions `foo<1>` and `foo<2>` are evaluated only when they are called and in the order they are called. The function `foo<1>` is not re-evaluated automatically after `tmp2` is set by `foo<2>`. Explicitly calling `foo<1>` one more time is also incorrect because `foo<1>` would make an erroneous extra update to its internal state `L`. In Verilog/VHDL/SystemC, all synchronous state updates take place at exactly the same moment as the final act in a clock period.

Using C++ Objects for Submodules. The alternative to using functions is to use C++ objects— supported by Vivado-HLS—for modular design and reuse. The notion of object construction better matches that of module instantiation. The module `foo` in Figure 3 can be captured as the object class `foo_class` below.

```
1  class foo_class{
2  public:
3    int L; // latch
4
5    foo_class() {L=INIT_VAL;} // class constructor
6    void query(int I1, int *O) {*O=I1*L;} // output logic
7    void update(int I2) {L=I2+L;}// next-state update
8  };
```

Using this class, we can describe the top-level module in Figure 5 as the function `top_ordering()` below. As in Section 3, the intention is the module synthesized from this function will execute one invocation completely once per cycle. The declaration of static objects `foo1` and `foo2` in `foo_class` correctly convey the notion that we want two distinct and persistent instances of the object `foo_class`.

```
1  void top_ordering(int I, int *O) {
2    static foo_class foo1, foo2;
3
4    int tmp1; // output of left module
5    int tmp2; // output of right module
6
7    /* ---- combinational-query behaviors ---- */
8    foo1.query(I,&tmp1);
9    foo2.query(tmp1,&tmp2);
10   *O=tmp2;
11
12   /* ---- state-update behaviors ---- */
13   foo1.update(tmp2);
14   foo2.update(tmp1);
15 }
```

Notice, in `foo_class`, we decomposed the module's behavior into two methods: a "query" method for the output logic and an "update" method for next-state update. This is a part of a necessary self-imposed coding discipline in our solution. In general, a submodule object class can have multiple member variables as well as multiple query and update methods. A query method output must be purely combinational (i.e., cannot have any side-effect through member or global variables and can only be a function of its input variables and/or member variables).[9] Only an update method can update the member variables of an object; an update method cannot have outputs.

Evaluation Order Discipline. Unlike in a netlist where declaration order does not matter, C++ method invocations must be ordered deliberately. For a small example like `top_ordering()`, we can ascertain by inspection that the chosen ordering of method invocations does lead to the

---

[9]Note it is okay to call the same query method multiple times; each call results in a different combinational logic instance.

desired behavior. (Vivado-HLS would in fact produce the correct structure.) In general, an ordering discipline is needed for correctness and synthesizability.

The order of query method invocations across all objects must obey data dependence—before invoking a query method, its input arguments that are not top-level inputs or static state variables must first be assigned by an earlier query method. Since combinational cycles through query methods are disallowed, a valid dataflow order must exist. All valid dataflow orderings should result in the same synthesis outcome. It is interesting to point out that this same dataflow ordering requirement is almost second-nature for the procedure C code shown in Section 3.

An update method can only be called after all query methods depending on any of the affected member variables have been called. The correct ordering between different update methods of the same object is more subtle and in general depends on what are the effects of the update methods and how they are coded. Alternatively, we propose a general discipline that restricts an object to a single update method that updates all member variables in an object. This update method must be the last method invoked on the object, that is, after all query methods have been called. This general discipline is stricter than necessary but does not require knowledge of an object's internal.

Discussions. Using C++ objects and the above ordering discipline, we can describe and synthesize arbitrary structural designs comprising synchronous registers and combinational logic using Vivado-HLS. This approach is extensible to a hierarchy of modular objects. On the downside, one can imagine the ordering discipline can be cumbersome in larger, more elaborate designs. For the design study, while we were able to describe and synthesize the 2D-Mesh NoC as well as a number of other common topologies, it would have been much easier to use Verilog for the netlist. However, the use of objects and the required ordering discipline is useful and necessary when mixing objects and inline C code. Astute readers may have noticed we already made use of objects in `switch_buffered()` in Section 3 to instantiate FIFO buffers.

A more interesting question is why is the experience so different when describing the structure of a router versus the structure of a NoC. The crux of the answer lies in the fundamental differences in the semantics of a C function and its invocation versus a module and its instantiation. Using a function to capture a purely-combinational module is natural. We also showed using a function to capture a top-level module is effective in Section 3. Keep in mind, however, when working with the complexity and regularity of a router, we naturally created a shallow design hierarchy. We mainly used sub-functions to encapsulate combinational logic (e.g., `allocate()` and `decode()`). In `switch_buffered()` where we called the stateful sub-function `switch_2stage()`, it was only called once—so there is no question of multiple instances and a valid evaluation ordering can be easily built around it. In contrast, in this section, in a more general usage of functions to capture sequential submodules, any illusion that C functions are like hardware modules is inescapably broken [7].

5 Limitations

Using the approaches in Section 3 and 4, one could use C/C++ and Vivado-HLS to describe and synthesize arbitrary structural designs of synchronous registers and combinational logic. However, limitations exist when attempting to replicate RTL designs that make use of macro storage primitives, like FIFOs and memory blocks. First, the primitives available for instantiation and inference in C vs. RTL could be different in structure and in timing behavior. Second, the rules for inferring primitives are different in C vs. RTL synthesis. When facing these differences, sometimes it is possible to still attain the same bit-level and cycle-level behavior by emulating the desired but unavailable primitive with an available one.[10] This emulation will add an overhead cost in logic resources and logic delay.

Consider for example a memory block with one asynchronous read port and one synchronous write port. Many suitably formed Verilog descriptions of this memory block can be automatically mapped to LUT-RAM by Xilinx RTL synthesis tools. Surprisingly, we could not find a way to capture such a basic memory primitive through C and Vivado-HLS. The following function conveys the correct intention if the synthesized module would execute one function invocation completely in each cycle, as in all of the previous examples.

---

[10]In the worst case, one can always resort to emulating the unavailable primitive using registers and combinational logic.

```
1   void ram(int raddr, int *rdata, int waddr, int wdata){
2     static int X[8];
3
4     *rdata=X[raddr];
5     X[waddr]=wdata;
6   }
```

There is no ostensible reason why this should not work.[11] For version 2015.2, Vivado-HLS does map X to LUT-RAM, but even with strong prodding using various pragmas, Vivado-HLS always treats the array read on line 4 as a synchronous read—resulting in a module that corresponds to a memory block with a synchronous read port. If required, one way to achieve the desired asynchronous read timing is to add the pragma `#pragma HLS ARRAY_PARTITION variable=X complete dim=1` to force the array X to be implemented using flip-flops.

Without access to memory with asynchronous read, we could not replicate an optimization in CON-NECT that packs one input port's multiple virtual-channel flit buffers (only on the data payload portion[12]) onto a single random-access memory structure. In HLS, we have to map an input port's multiple flit buffers to separate structures. Thus, our HLS-synthesized routers use more logic to steer the flow to/from the multiple flit-buffer structures. This partially accounts for the higher LUT usage that the HLS-synthesized routers consume relative to CONNECT routers in the synthesis results in the next section.

On the other hand, for the route information portion of the virtual-channel flit buffers, the allocator logic needs to read the first flit of a non-empty flit buffer before deciding whether to dequeue the flit (i.e., only when the flit is selected to advance). In the design study, we implemented the flit buffers using Vivado-supplied `ap_shift_reg` class which allows this behavior (reading-before-dequeuing). In CONNECT, they used a FIFO primitive that requires dequeuing first before the dequeued flit can be read in the next cycle. Both types of FIFO primitives use LUT for storage so they have equal storage efficiency. However, in CONNECT, they have to add an interface shim with flip-flops to convert from dequeuing-before-reading to reading-before-dequeuing. This is why our HLS-synthesized routers consistently use significantly fewer flip-flops than the corresponding CONNECT routers in the synthesis results in the next section.[13]

Finally, in replicating the CONNECT router RTL designs, we replicated the combinational logic only in the truth-table sense. We made no attempt to match up the combinational logic at any more concrete levels of specification. As we will see, this can result in small but inexplicable differences in LUT usage and critical path delay between our HLS-synthesized routers and the CONNECT routers.

6   Evaluation and Analysis

The previous section discussed why the HLS routers are not exactly the same as the CONNECT routers in the lowest RTL design details. These differences have consequences in final implementation quality—resource usage and critical path delay—when mapped onto the FPGA fabric. In this section, we compare the place-and-route implementation quality of the HLS generated routers (Section 3) and NoCs (Section 4) against those produced by CONNECT. The comparison shows the HLS structural design methodology can produce comparable quality as CONNECT Verilog RTL designs.

---

[11]In fact, the desired dual-port asynchronous RAM can be implemented by specifying pragma `#pragma HLS RESOURCE variable=X core=RAM_2P_1S` in earlier Vivado-HLS verion, for example version 2013.2. However, version 2015.2 we used in our work does not support the storage core RAM_2P_1S any more.

[12]Among all the virtual channels of one input port, at most one flit is added and removed per cycle. Thus, the data payload portion of one input port's multiple virtual-channel flit buffers can be time-multiplexed on to one physical structure. The route information portion cannot be similarly packed because the allocation logic needs to examine the route information from all virtual channels in each cycle.

[13]The CONNECT generator could be re-engineered to use SRL16E shift-register FIFOs to nullify this difference.

Table 1: HLS routers place-and-route ratios over CONNECT routers.

Data width		32 bits								128 bits							
Num. VCs		2 VCs				4 VCs				2 VCs				4 VCs			
Buf. Depth		4	8	16	32	4	8	16	32	4	8	16	32	4	8	16	32
2	LUT	0.95	0.97	0.87	0.93	1.24	1.17	0.96	0.91	1.53	1.40	1.28	1.44	2.05	1.93	1.43	1.23
I/O	FF	0.44	0.43	0.38	0.37	0.41	0.48	0.38	0.35	0.44	0.43	0.38	0.37	0.41	0.42	0.35	0.35
Ports	CP	0.85	1.01	0.97	1.03	0.97	0.90	0.87	0.98	0.92	1.09	1.05	1.06	1.04	1.00	0.98	0.95
4	LUT	1.08	1.05	1.12	1.01	1.25	1.19	1.10	1.05	1.72	1.60	1.76	1.64	1.80	1.69	1.46	1.29
I/O	FF	0.48	0.46	0.40	0.39	0.42	0.42	0.36	0.36	0.49	0.46	0.40	0.39	0.42	0.43	0.36	0.35
Ports	CP	0.98	1.12	0.95	1.02	0.99	0.86	0.76	0.89	0.87	0.95	0.87	0.97	0.88	0.88	0.86	0.91
6	LUT	1.07	1.11	0.99	1.31	1.06	1.09	1.00	0.91	1.53	1.55	1.38	1.48	1.54	1.52	1.37	1.13
I/O	FF	0.50	0.48	0.42	0.41	0.43	0.42	0.38	0.38	0.50	0.48	0.42	0.41	0.43	0.43	0.39	0.37
Ports	CP	0.77	0.96	0.84	0.87	0.81	0.82	0.82	0.82	0.79	0.80	0.79	0.83	0.88	0.77	0.75	0.84
8	LUT	1.20	1.18	1.20	1.19	1.18	1.24	1.05	1.01	1.85	1.73	1.75	1.71	1.90	1.74	1.53	1.39
I/O	FF	0.54	0.52	0.45	0.44	0.45	0.44	0.39	0.38	0.54	0.52	0.45	0.46	0.45	0.44	0.39	0.38
Ports	CP	0.79	0.79	0.69	0.73	0.80	0.77	0.76	0.76	0.78	0.80	0.79	0.73	0.79	0.79	0.82	0.75

6.1 Methodology

The comparison of standalone routers samples a space of configurations supported by CONNECT. The space is the cross-product of (a) in/out-degree (2, 4, 6, 8); (b) flit data width (32-bit and 128-bit); (c) number of virtual channels (VCs) (2 and 4); and (b) the depth of flit buffers (4, 8, 16, 32). All of the routers are configured to use credit-based flow control, separable input-first allocator and round-robin arbiter. All of our HLS router examples are based on a common parameterized C++ object class configured through `#define` constants.

The comparison of NoCs samples a range of 16-end-point topologies. The topologies include Ring, DoubleRing, FatTree, Mesh, Torus, HighRadix. As with CONNECT NoCs, these different-topology NoCs are transparently interchangeable in 16-endpoint applications. The Ring topology represents a simple low-cost topology; the HighRadix topology represents a high-cost, high-performance design point.[14] The HLS NoC designs are hand-coded top-level functions that make use of C++ router objects following the ordering discipline prescribed in Section 4. The routers used in this comparison have 2 virtual channels per input port, 8 flits per flit buffer, 32-bit flit data width. All of the routers are configured to use credit-based flow control, separable input-first allocator and round-robin arbiter. The above choices reflect the most commonly chosen router configuration on the CONNECT website. The in/out-degree of the routers is topology dependent.

We evaluate implementation qualities in terms of (1) LUT: number of LUTs consumed; (2) FF: number of flip-flops consumed; and (3) CP: the clock period. Neither the HLS routers nor the CONNECT routers use DSPs or Block-RAMs; flit buffers are implemented using LUTs. The reported values are from place-and-route reports targeting a Xilinx Virtex-7 VX690T FPGA (xc7vx690t, speed grade -2). The HLS routers are synthesized using the HLS flow in Xilinx Vivado 2015.2 with the "evaluate" option (suggested by [9] to run both synthesis and implementation in a single flow). The Verilog RTL from the CONNECT generator is synthesized and implemented using the normal RTL Verilog flow in Xilinx Vivado 2015.2 with the same strategies used in HLS counterparts. For both HLS and RTL synthesis, for each configuration, we swept the target clock period (in increments of 1 ns) in repeated runs and selected the result from the synthesis that yielded the shortest clock period.

6.2 HLS vs. CONNECT

Table 1 gives the ratio of LUT, FF and CP for HLS routers over CONNECT routers. The HLS routers on average use 1.33x LUTs, 0.42x FFs, and can achieve 0.87x CP relative to the corresponding CONNECT routers. The significant reduction in flip-flop usage is explained in Section 5. In terms of LUT usage, the HLS routers compare less favorably in the 128-bit wide configurations than in the 32-bit wide configurations (which are close to parity with CONNECT routers). This can be attributed to the differences in the implementations of the virtual-channel flit buffers for holding the data payloads (also explained in Section 5). More precisely, the difference is not in the LUTs used for storage but in the logic LUTs in the datapath surrounding the different storage structures in use. In configurations that do not use virtual channels (not shown in the table), HLS routers are equally efficient relative to CONNECT routers at 32-bit and 128-bit data width.

---

[14]HighRadix is a customized topology with 8 routers fully-connected and each router supports two nodes, which means each router has 9 I/O ports [14].

Table 2: HLS NoCs place-and-route ratios over CONNECT NoCs.

Network	r_{LUT}	r_{FF}	r_{CP}
Ring	1.01	0.43	1.24
DoubleRing	1.20	0.44	1.07
FatTree	1.13	0.46	1.10
Mesh	1.03	0.44	1.02
Torus	1.09	0.46	1.22
HighRadix	1.23	0.29	0.86

We do not fully understand why the HLS routers can have better critical path delay than CONNECT routers. The critical path delay in all cases is in the allocator combinational logic. The allocator combinational logic in the HLS routers and CONNECT routers are identical in the truth-table sense. They are specified differently of course, and they are subjected to different combinational logic optimizations in their respective synthesis flows. What is most curious is that the gap widens with the complexity of the allocator (higher in/out-degree and more virtual channels). We know this trend is found only in those configurations using virtual channels.

Table 2 reports the HLS NoCs place-and-route ratios in comparison to the equivalent CONNECT NoCs. The ratios are consistent with the comparisons of the underlying router modules discussed earlier. This affirms that the use of C++ objects and the requirement of the ordering discipline from Section 4 do not have inadvertent consequences. At this NoC-level comparison, for the chosen 32-bit-wide router configurations, the HLS NoCs and CONNECT NoCs are closely comparable in overall quality.

7   Conclusions

The answer to the question—***can*** we use C and Vivado-style HLS for structural design—is yes. We showed that arbitrary structural designs of synchronous registers and combinational logic can be captured using the Vivado-HLS methodology. While there may be limitations when trying to replicate specific macro storage primitives, most of the time, there are sufficiently close substitutes to achieve the desired bit-level and cycle-level behavior with only modest overhead. We further demonstrated that the Vivado-HLS methodology is able to produce comparable quality implementation outcomes as a standard RTL methodology.

The answer to the question—***should*** we use C and Vivado-style HLS for structural design—is sometimes. The reference CONNECT RTL designs are generated so we cannot quantify the time and effort involved. Quantifying our own design productivity meaningfully is also very challenging. To provide one quantitative data point, the entire effort reported—including the initial learning curve and the final evaluation analysis—is done by 1 PhD student in 7 months. More to the point though, as mentioned in the discussions in Section 3, the perceived productivity gain in the router portion of the design study is very much noticeable. We attributed this gain to the ability to decouple design concerns for functionality and structure under the Vivado-HLS flow. However, as later mentioned in the discussions in Section 4, the Vivado-HLS methodology has a harder time in handling hierarchical modular structural deigns. In particular, the methodology offers little advantage for pure netlisting. The break-even for adopting Vivado-HLS in these latter uses is highly design and context dependent.

References

[1] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Aviženis, J. Wawrzynek, and K. Asanović. Chisel: constructing hardware in a Scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference*, pages 1216–1225. ACM, 2012.

[2] M. Budiu. *Spatial Computation*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA 15213, Dec. 2003.

[3] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson. LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems. *ACM Trans. Embed. Comput. Syst.*, 13(2):24:1–24:27, Sept. 2013.

[4] J. Choi, S. Brown, and J. Anderson. From software threads to parallel hardware in high-level synthesis for FPGAs. In *Field-Programmable Technology (FPT), 2013 International Conference on*, pages 270–277, Dec 2013.

[5] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. High-level synthesis for FPGAs: From prototyping to deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):473–491, April 2011.

[6] W. J. Dally and B. Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann, 2004.

[7] S. A. Edwards. The challenges of hardware synthesis from C-like languages. In *Design, Automation and Test in Europe*, pages 66–67 Vol. 1, March 2005.

[8] N. Kapre and J. Gray. Hoplite: Building austere overlay NoCs for FPGAs. In *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, Sept 2015.

[9] K. Karras and J. Hrica. Designing protocol processing systems with Vivado high-level synthesis. http://www.xilinx.com/support/documentation/application_notes/xapp1209-designing-protocol-processing-systems-hls.pdf.

[10] S. Lahti, J. Vanne, and T. D. Hämäläinen. Designing a clock cycle accurate application with high-level synthesis. In *IECON 2016 - 42nd Annual Conference of the IEEE Industrial Electronics Society*, pages 4756–4761, Oct 2016.

[11] M. C. McFarland, A. C. Parker, and R. Camposano. The high-level synthesis of digital systems. *Proceedings of the IEEE*, 78(2):301–318, Feb 1990.

[12] S. Neuendorffer, T. Li, and D. Wang. Accelerating OpenCV applications with Zynq-7000 all programmable SoC using Vivado HLS video libraries. https://www.xilinx.com/support/documentation/application_notes/xapp1167.pdf.

[13] R. Nikhil. Bluespec System Verilog: efficient, correct RTL from high level specifications. In *Formal Methods and Models for Co-Design, 2004. MEMOCODE'04. Proceedings. Second ACM and IEEE International Conference on*, pages 69–70. IEEE, 2004.

[14] M. K. Papamichael and J. C. Hoe. CONNECT: Re-examining conventional wisdom for designing NoCs in the context of FPGAs. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '12, pages 37–46, New York, NY, USA, 2012. ACM.

[15] M. K. Papamichael and J. C. Hoe. The CONNECT network-on-chip generator. *Computer*, 48(12):72–79, December 2015.

[16] H. Touati. Perle1DC: a C++ library for the simulation and generation of decperle-1 designs. *DEC-PRL Technical Note*, 4, 1994.

[17] J. Villarreal, A. Park, W. Najjar, and R. Halstead. Designing modular hardware accelerators in C with ROCCC 2.0. In *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on*, pages 127–134, May 2010.

[18] S. Windh, X. Ma, R. J. Halstead, P. Budhkar, Z. Luna, O. Hussaini, and W. A. Najjar. High-level language tools for reconfigurable computing. *Proceedings of the IEEE*, 103(3):390–408, March 2015.

[19] Xilinx. SDSoC environment user guide. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_2/ug1027-sdsoc-user-guide.pdf.

[20] Xilinx. Vivado design suite user guide: High-level synthesis. http://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_2/ug902-vivado-high-level-synthesis.pdf.