High-Level Synthesis Walks the Talk: Synthesizing a Complete Graphics Processing Application

Thomas Bollaert
Mentor Graphics, Corp.
8005 SW Boeckman Rd.
Wilsonville, OR 97070
503-685-4752
thomas_bollaert@mentor.com

ABSTRACT
In this paper, we will describe how a complete graphics processing pipeline was implemented using an HLS methodology. As with most real-life applications, this design consists of a complex mix of control logic, datapaths, interfaces, and hierarchy. We will show how these four essential ingredients are addressed in the context of HLS, and we will review the capabilities of current-generation HLS technology and its applicability for complex applications.

In doing so, this paper will focus on the best coding style and suitable abstractions for the various parts of the design. We will dissect and compare the modeling requirements for the control and algorithmic portions of the system. We will conclude by providing guidelines for choosing when high-level models are optimally expressed in a cycle-accurate manner versus the more abstract, purely untimed modeling style, and we will show how to efficiently combine both kinds of models. Thus, the reader will not only see that HLS walks the talk, but they will walk away having learned the correct way to put HLS to work for them today.

Categories and Subject Descriptors
B.5.2 [REGISTER-TRANSFER-LEVEL IMPLEMENTATION]: Design Aids –automatic synthesis, optimization, simulation, verification

General Terms
Algorithms, Design, Experimentation, Languages, Verification

Keywords
High Level Synthesis, Electronic System Level, Design, Verification, Mixed Language Flow, Abstraction, Methodology, Control Logic, Datapaths, Interfaces, Hierarchy, Coding Style, ISP

1. INTRODUCTION
The shift to a higher abstraction is becoming mandatory to address today's ASIC and SoC design challenges. Just as design teams transitioned from gates to RTL in the mid-90s, new thresholds in design complexity are calling for the move from RTL to C++ and SystemC-based modeling, verification, and synthesis.

Consequently, during the past couple of years, high-level synthesis (HLS) has become much more prevalent in design flows, widened its applicability, and entered the mainstream of hardware design [1][2]. However, designers need the know-how to put it into practice in the best possible way.

In this paper, we will show how this is done by describing how a complete graphics processing pipeline was implemented using an HLS methodology. We will demonstrate how today's mature HLS technologies handle the complex mix of control logic, datapaths, interfaces, and hierarchy. We will share the best coding style and suitable abstractions for each of these parts of the design, compare the modeling requirements for the various portions of the system, and provide guidelines for choosing abstraction levels. First, we'll review the primary objectives of designing at higher levels of abstraction.

2. GUIDING PRINCIPLES
The various abstraction levels serve different design needs; for this reason they complement each other to great advantage in a "full-chip" HLS flow. But how does one choose the proper modeling style and most efficient abstraction-level for specific design tasks? The answer to these questions is found in the reason HLS flows are being adopted in the first place. The goal of HLS is to increase design and verification productivity. This primary objective must be kept in mind when making modeling decisions at higher levels of abstraction.

To help with design productivity, models must be kept as abstract as possible. This makes them simpler to write (less lines of code, fewer chances of errors), easier to debug (less details to worry about), and faster to simulate (less simulation overhead).

To help with verification productivity, enough detail must be kept where it matters so design behavior can be predictable and consistent throughout the flow. As a result, the RTL will be guaranteed to match the high-level specification, greatly reducing the burden on the RTL verification team.

The principles of simplicity and sufficient detail are dependent upon two essential parameters that can be abstracted when moving up to a higher level: timing and structure. When determining the levels of timing and structural information to be coded in the source, one should keep these two productivity principles in mind and answer these two basic questions:

- Is the functionality time-dependent or not, and if so, to what extent?
- Do I want to lock down hierarchy and parallelism, or do I want to be able to explore different solutions?

In the following sections, we will show how to answer these questions for the various parts of a complete imaging pipeline and how to most efficiently write the code.

3. AN IMAGE SIGNAL PROCESSOR

With the emergence of smart phones and broadband wireless networks, cameras have quickly evolved from niche features to mandatory functionality for handheld devices. Tightly coupled to the CMOS image sensor, the image signal processor (ISP) defines the image quality of the handheld camera subsystem. In this very dynamic market, differentiation is achieved through proprietary algorithms for defect correction and image improvement [3].

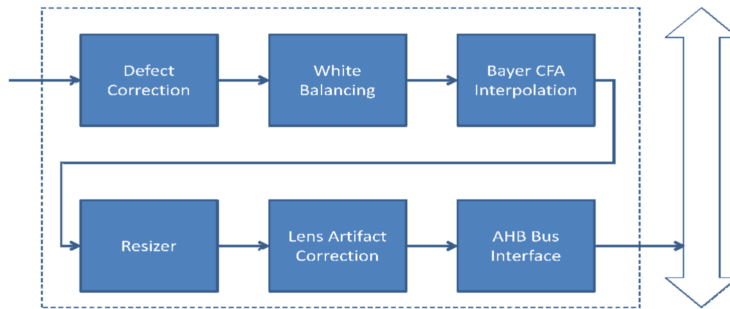

Figure 1. Image signal processor block diagram.

Our reference design implements canonical ISP functions—such as pixel defect correction, white balancing, color filter array (CFA) interpolation, resizing—and various lens artifact correction functions—such as pincushion and barrel distortion. Our design also provides a standard AMBA AHB[4] interface to transfer the image from the ISP to the rest of the system (Fig.1).

In the rest of this article, we will focus on two particular blocks: the image resizer and the AHB bus. These two blocks exhibit the different properties and requirements of algorithmic units and control-logic blocks. As such, they are representative and pedagogical examples.

4. THE IMAGE RESIZER

The resizer block takes an input image and resizes it to a new height and width. The algorithm performs a 4x4 bicubic interpolation; it estimates the color of a pixel in the resized image based on 16 pixels surrounding the closest corresponding pixel in the source image. Line buffers are used to cache the incoming image data and provide the appropriate 16 pixels in parallel to the bicubic kernel. (Fig.2). This allows the resizer to sustain a throughput of 1 pixel per clock on the output. The inputs and outputs of this block are in the form of point-to-point (P2P) pixel streams.

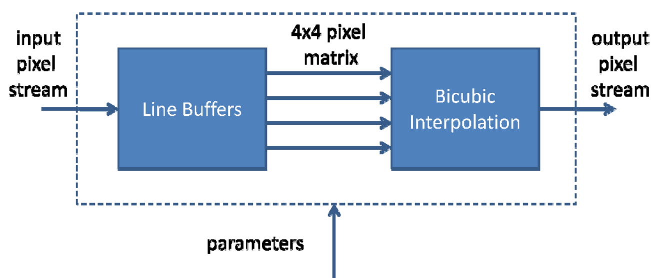

Figure 2. Image resizer block diagram.

4.1 Structure of the High-Level Model

In RTL, a similar block would be decomposed into several sub-blocks and many processes, corresponding to the line buffers and scaling function. The same structural decomposition using dedicated modules and processes is possible in a language like SystemC. With the SC_MODULE macro, SystemC provides a way to explicitly model hierarchy in a way that is easy to understand, looks like Verilog, and is familiar to hardware designers. However, the decomposition of the design into sub-modules becomes counterproductive when taken too far because, by hard-coding structure and parallelism in the source, the potential to explore different implementation alternatives is severely restricted. Moreover, adding superfluous processes and threads in a SystemC model significantly slows down simulation performance due to increased context switching. All of this greatly diminishes some of the major benefits of an HLS flow.

Instead of hard-coding structure and parallelism, modern HLS tools allow users to create arbitrary hierarchical design boundaries from an abstract model. The HLS tool leverages user constraints to partition loops or functions into separate concurrent blocks. Because they are not hard-coded in the source, the boundaries for hierarchy are much more flexible than if expressed with SystemC modules. This flexibility is an advantage when optimizing for performance and area in order to improve QoR.

Following our guiding principles, we kept our model as abstract as possible, avoiding any unnecessary details. The entire resizer is modeled as a single SC_MODULE with only one SC_THREAD implementing both the line buffering and the bicubic interpolation.

```
SC_MODULE(resizer)
{
  public:

  sc_fifo_in<pix_t >  pix_in;
  sc_fifo_out<pix_t > pix_out;
  sc_in<params_t>     params;

  SC_CTOR(resizer) {
    SC_THREAD(proc);
  }

 private:

  void proc ()
  {
    lbuf_c<pix_t> lbuf(IMG_W, IMG_H);

    while(1) {
      if (lbuf.ramping_down()==false)
        lbuf.shiftin(pix_in.read());
      else
        lbuf.shiftin(0);
      if (lbuf.ramping_up()==false) {
        pix_t pix = interpolate(lbuf);
        pix_out.write(pix);
      }
    }
  }
};
```

4.2 Interfaces of the High-Level Model

In RTL, the P2P interfaces would be implemented with a data line and a pair of signals, obeying a typical request/acknowledge protocol. While optional, the handshake is good design practice as it allows building safe data transfers and latency-insensitive designs (LIS) [5].

In SystemC, it would also be possible to model each individual interface signal and their cycle-accurate behavior in the form of a small finite state machine (FSM). However, HLS tools support a more productive modeling approach. The implementation details can be abstracted away; yet the model remains true to the core nature of the protocol: in this case, providing a safe P2P data transfer where no data gets dropped.

The SystemC *sc_fifo* constitutes a very convenient way of modeling P2P connections, easing the shift from clocks and signals to more abstract producers and consumers [6]. Using the *sc_fifos* blocking reads and writes, we guarantee safe data transfers by making simple function calls, and we can delegate the implementation of a corresponding RTL interface to the HLS tool. This approach eliminates the need to develop, debug, and maintain custom interface models; it is faster to simulate than equivalent cycle-accurate models; and it makes the overall modeling experience a lot simpler.

Following our guiding principles, as detailed timing is not of essence in P2P connections, we prefer the more abstract, untimed modeling style. The interfaces of the resizer, and all the P2P connections in the ISP, are modeled with *sc_fifos*. This establishes a simple, safe, and deterministic design approach.

4.3 Memory Architecture of the High-Level Model

An important design aspect that needs to be described is the desired memory architecture. HLS tools provide a number of automatic optimizations and constraints; such as memory splitting, interleaving, and merging. Designers should assume that the memory access patterns in the hardware design will reflect those in the high-level model. Since memory accesses can often be performance bottlenecks in a design, it is essential to write the C++ array accesses in a way that will not limit performance in the RTL.

To model the 1D and 2D sliding windows, we followed the recommendations detailed in chapter 7 of the High-Level Synthesis Blue Book [7].

4.4 Behavior of the High-Level Model

In the resize, as in most other blocks in our ISP, time is not an attribute of functionality. Rather, time, expressed in the form of throughput and latency, is an artifact of the implementation. In other words, the resizer could be implemented in several different ways, resulting in different timing scenarios, yet the result would still be true to its specification.

In HLS, parallelism can be extracted from sequential sources through data flow graph (DFG) analysis and various loop transformations [7][8]. Similarly, time is added during scheduling [7][9].

For example, the interpolation functions are implemented using a sum-of-products (SoP). These are easily modeled in C++ using a simple "for" loop. The loop can be unrolled and/or pipelined to generate different RTL implementations, ranging from a serial multiply-accumulate to a fully parallel architecture with a balanced adder-tree (Fig 3). Thus, there is no advantage in describing timing or this level of parallelism in the source itself, but it does create more work and more overhead: more coding, slower simulations, and harder debug.

```
//interpolator operates on line buffer data
void interpolate (lbuf_c<pix_t> lbuf)
{
  uint2    x = lbuf.x();
  uint2    y = lbuf.y();
  accumt_t t = 0;
  for (int m=0; m<4; m++) {
    for (int n=0; n<4; n++) {
      t += lbuf(m,n) *
           coefs[x][m] *
           coefs[y][n];
    }
  }
  return t;
}
```

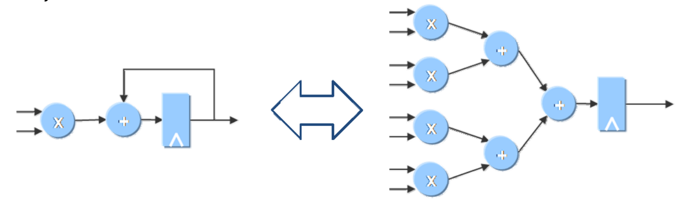

Figure 3. HLS extracts parallelism from sequential sources, targeting multiple architectures from a single abstract model.

5. THE BUS INTERFACE

The bus interface block of our ISP implements an AHB master. It receives the processed image in the form of a stream of pixels and writes it to the system memory using DMA burst transfers.

5.1 Interfaces of the High-Level Model

On one side, the bus interface receives pixels through a P2P connection from the lens artifact correction (LAC) block. The LAC has the same characteristics as the image resizer and is implemented the same way: in a purely untimed fashion with *sc_fifo* interfaces. The bus interface therefore receives the pixel data through an *sc_fifo*. Using this standard and abstract interface allows connecting any block—such as the LAC or the resizer— to the bus interface very easily and in a protocol-agnostic fashion.

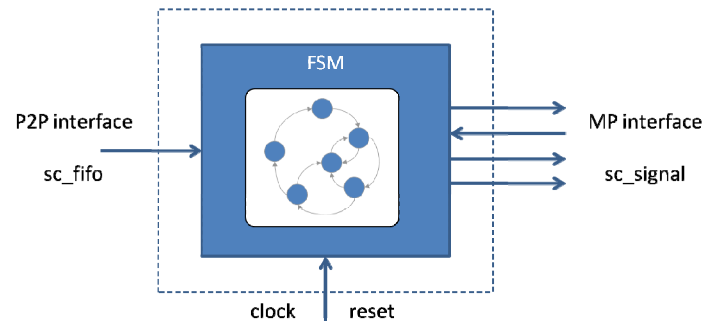

Figure 4. AHB bus interface.

On the other side, the bus interface implements the AMBA AHB master protocol. This is a multi-point (MP) fabric connecting more than two blocks together (Fig.4). For such protocols, timing is obviously an integral part of the functionality. In this case, cycle-by-cycle behavior is very important for verification purposes. With this

level of fidelity, one can accurately simulate and analyze how data is being transferred, what level of traffic is happening at any given time on the bus, and how multiple requests are being arbitrated. Modeling these MP (complex bus) interfaces at the cycle-accurate level provides the ability to properly verify the system and the benefit of a highly predictable path to RTL using HLS. For this reason, we implemented the AHB interface using *sc_signals*.

Yet, even when working with *sc_signals*, we can encapsulate and abstract the complexity of the AHB interface. We achieve this by creating a reusable *AHB_master_if* class that contains all the necessary signals and provides a convenience layer, allowing for the behavioral model to interact with the interface through simple method calls; such as a *burst_write()*.

```
SC_MODULE(AHB_master)
{
  sc_in<bool>           clk;
  sc_in<bool>           rst;
  AHB_master_if         bus_if;
  sc_fifo_in<rgb_t >    pix_in;

  SC_CTOR(AHB_master) {
    SC_METHOD(proc);
    sensitive << clk.pos();
  }

 private:
  int state, i;

  void proc()
  {
    if ( rst.read() ) {
      bus_if.reset();
      state = IDLE;
      i = 0;
    } else {
      switch (state)
      {
        case IDLE:
          if ( pix_in.num_available()>7 )
            state = WAIT_FOR_GRANT;
          break;
        case WAIT_FOR_GRANT:
          bus_if.request();
          i = 0;
          if ( bus_if.is_granted() ) {
            bus_if.start_burst_write(
              BASE_ADDR + i);
            state = BURST;
          }
          break;
        case BURST:
          bus_if.request();
          if ( bus_if.is_ready() ) {
            if ( i < 7 ) {
              bus_if.burst_write(
                BASE_ADDR + i,
                pix_in.read());
            } else {
              bus_if.finish_burst_write(
                pix_in.read());
              state = IDLE;
            }
            i++;
          }
          break;
      }
    }
  }
};
```

5.2 Behavior of the High-Level Model

Our bus interface effectively bridges the untimed world of *sc_fifos* with the timed, cycle-accurate world of *sc_signals*. The untimed domain synchronizes on data availability, while the time domain synchronizes on the edge of a given clock. For deterministic behavior in simulation and synthesis, our bus interface must honor these two synchronization schemes.

In this case, the behavior is most naturally expressed using a finite-state machine (FSM) description, where each state transition occurs on a clock edge, allowing accurate modeling of the AHB protocol. Whenever appropriate, the behavioral FSM interacts with the *sc_fifo* interface using non-blocking calls. The *num_available()* method returns an integer value corresponding to the number of elements in the *sc_fifo* and indicating when it is safe to read data from it.

In our model, when at least eight pixels are present in the FIFO, a burst transfer is initiated knowing that the bus interface will not starve. The burst transfer is decomposed in the subsequent states of the FSM. Upon completion of the transfer, the bus interface FSM polls the *sc_fifo* interface again, checking if enough pixels are available for the next burst.

6. SUMMARY

To summarize, designers need to choose the appropriate level of information on the structural axis (explicit or implicit parallelism) and on the timing axis (cycle-accurate or untimed).

While it is possible to model a full system in a cycle-accurate way with explicit hierarchical boundaries, it is also by far the less productive approach. This would require adding a lot of design detail to the model, which not only results in a greater source of errors, but also in slower simulations and more painful debugging sessions.

The most efficient and productive path to full-chip HLS relies on a mixed-abstraction modeling strategy. The guiding principle is to keep things simple and avoid unnecessary detail in the source. Our recommendations are summarized as follows:

- Model explicit hierarchy at the functional boundary, but keep it uncommitted underneath
- Analyze and write array access patterns to avoid memory bottlenecks
- Keep processing and point-to-point communications strictly untimed
- Use cycle-accurate models for complex, multi-point communications; such as bus interfaces

Only a few years ago, HLS tools were applied only at the block level. Today, mixed-language HLS flows enable full-chip synthesis; including high-quality data paths, control logic, processors, interfaces, and complex interconnects. By paying attention to the proper mix of abstraction levels, HLS users will improve design and verification productivity while delivering high-quality SoCs.

8. ACKNOWLEDGMENTS

Thanks to Todd Burkholder, Senior Writer, Mentor Graphics for editorial support.

9. REFERENCES

[1] University of Oulu Rapid Scheduling of Efficient VLSI Architectures for Next-Generation HSDPA Wireless System Using C Synthesis]

[2] F.Baray, H.Michel, P.Urard and A.Takach. C Synthesis Methodology for Implementing DSP Algorithms, GSPx 2004.

[3] http://www.mentor.com/esl/success/STMicroelectronics-success/fileContent/STMicro_Catapult_C_success_story_11-18-09.pdf.

[4] AMBA, www.arm.com.

[5] L.P. Carloni, K.L. McMillan and A.L. Sangiovanni-Vincentelli, "Theory of Latency-Insensitive Design" in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 20(09): 18, Sept 2001.

[6] Thorsten Grötker, Stan Liao, Grant Martin, Stuart Swan, System design with SystemC, Springer.

[7] M. Fingeroff. “The High-Level Synthesis Blue Book”, Xlibris.

[8] A. Takach, B. Bower, and T. Bollaert. “C based hardware design for wireless applications”. DATE, 2005.

[9] P. G. Paulin and J. P. Knight, “Force-directed scheduling for the behavioral synthesis of ASICs” in IEEE Transactions on Computer-Aided Design, vol. 8, pp. 661–679, June 1989.