

Implementation Independent Design of a Digital Imaging Algorithm Using Behavioral Synthesis

David J. Pursley
Forte Design Systems
pursley@ForteDS.com

INTRODUCTION

Recently, behavioral design and synthesis has seen a re-emergence in the design community, especially in the design of cutting edge imaging, consumer electronics, and digital signal processing chips and systems-on-chip. In fact, many of the top semiconductor companies in the world have adopted the highly productive behavioral design flow.

The reason is clear. This market space is highly competitive, where the ability to quickly react to a change in specification and still meet time-to-market goals will often define a project's success. The immovable market windows of consumer electronics products often mean that if a design schedule slips, the product is cancelled. The RTL design flow simply doesn't allow the necessary flexibility or productivity offered by a behavioral design flow.

Industrial users have found that behavioral design reduces the design effort by 50% or more while attaining excellent quality of results [Johnson98][Moussa98].

This paper discusses a behavioral design methodology that allows designers to create hardware (ASIC, FPGA or SoC) from an arbitrary implementation independent C/C++ algorithm. We start by defining behavioral synthesis and then place it in the context of a behavioral design flow using industry case studies as examples. The goal of this paper is to give the reader an idea on the methodology benefits of behavioral design.

WHAT IS BEHAVIORAL SYNTHESIS?

A detailed overview of behavioral synthesis can be found in [Meredith04]. Here, we will give only a brief overview.

Behavioral synthesis is an automated design process that interprets an algorithmic description of a desired behavior and creates hardware that implements that behavior.

Starting with an algorithmic description in a high-level language, behavioral synthesis tools automatically create the cycle-by-cycle detail needed for hardware implementation. Most behavioral synthesis approaches leverage the existing logic synthesis toolset by creating a register-transfer level (RTL) implementation from the algorithmic description.

The design description made possible by a behavioral synthesis design flow differs in a number of specific ways from that which is required for traditional logic synthesis. Logic synthesis uses an RTL description of the design. Behavioral synthesis uses a high-level un-timed, or partially timed, functional description. These descriptions can contain large portions of algorithm that, after behavioral synthesis, will be spread over many clock cycles. These algorithms can, and typically do, contain loops and array accesses that are not typically seen in RTL designs.

The behavioral synthesis tool will figure out how best to schedule that over the multiple cycles in order to meet design constraints. It will also determine the datapath, multiplexing and finite state machine needed to implement the design.

As a trivial example, consider the behavioral code in Figure 1.

```

unsigned long example_func (
    unsigned char a, unsigned char b,
    unsigned char c, unsigned char d,
    unsigned char e, unsigned char f )
{
    unsigned long y;
    y = ( ( a* b ) + c ) * ( d * e );
    return y;
}

```

Figure 1: Trivial example of behavioral code

Behavioral synthesis will interpret this as a control-dataflow graph (CDFG) which represents the dependencies of the various operations as shown in Figure 2. Given that CDFG and the list of available functional units (adders, multipliers, etc.) shown in Figure 3, the behavioral synthesis tool could choose to schedule the operations in several different ways, depending on the design goals, such as performance or area.

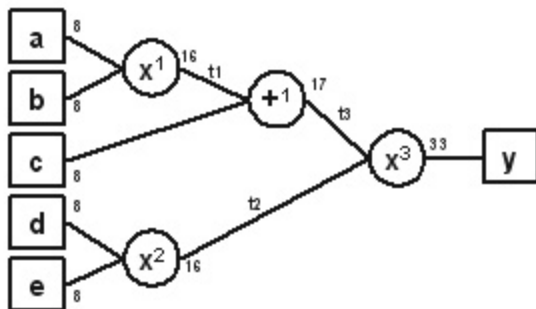


Figure 2: CDFG for trivial example

Functional Unit	Delay	Area
8x8=16	2.78	4896.5
16+16=17	1.99	1440.3
20x20=40	5.88	27692.6

Figure 3: Available functional units

Assuming minimizing area is the design goal, the behavioral synthesis process should select the schedule shown in Figure 4.

Operator	# Needed	Cycle 1	Cycle 2	Cycle 3
$16+16=17$	1		$t1+c$	
$20 \times 20=40$	1	$a*b$	$d*e$	$t2*t3$

Figure 4: Minimum area schedule

After determining that schedule, behavioral synthesis will create the RTL implementation shown in Figure 5. Note the multiplexing that has been added in front of the multiplier because it is being shared. Additionally, the behavioral synthesis would also determine the FSM and registers needed to implement this design.

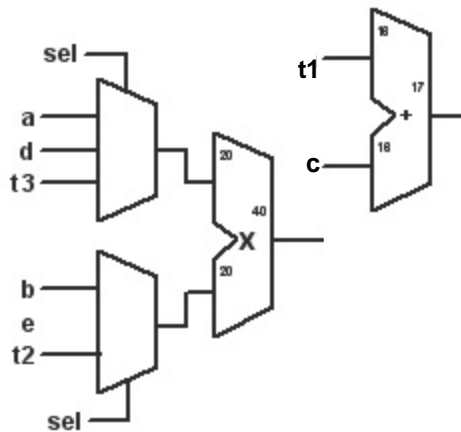


Figure 5: Minimum area RTL

The above was a brief overview of behavioral synthesis and specifically the types of synthesis transformations it does. The remainder of this paper will place behavioral synthesis inside of a behavioral design flow.

BEHAVIORAL DESIGN METHODOLOGY

Behavioral design is the process of taking an algorithm and transforming it to functionally equivalent hardware. While behavioral synthesis is an important step in behavioral design, there are additional tasks that must be done. Below, we outline the general methodology for behavioral design.

As we investigate this methodology, it is important to note that these design steps must be done when targeting any hardware design flow, be it RTL or behavioral. That is, these steps are required to map an algorithm into hardware.

Finally, it is worth mentioning the importance of verification in the behavioral design flow. You need to be able to simulate your design at each stage of the process in order to verify the changes (generally, additional details) committed in each step.

Identify design

Often, the starting point for a behavioral design is a large C or C++ program. This may include thousands of lines of code that can be compiled and executed in order to test the design. The first step in the behavioral design process is to identify the boundaries of the design that will be turned into hardware.

Usually, this step is trivial, but that is not always the case.

Case study: A customer had used behavioral design to implement the next generation of a proprietary blender imaging algorithm. However, they found that the area was significantly larger than the previous generation. By testing the previous generation's RTL design with portions of the new generation's testbench, the customer determined that the previous generation RTL did not implement all the modes in the C specification. (It was later determined this was by design in the previous generation implementation to save area.)

In any case, this illustrates that identifying the design is not always as simple as selecting a function in the C program. Sometimes, the design is a combination of functions or, as in this case study, only a portion of a function. In more complex cases, the design can end up being only select portions of several functions, which may involve a significant amount of recoding in order to allow separation of the testbench and design.

Separating design and testbench

Once the design portion of the C program has been identified, the remainder of the program can generally be treated as testbench. Then, in a SystemC-based methodology, the designer creates separate SC_MODULE's for the design and testbench. It is important to ensure that all of communication between the design and testbench happens through ports, as this is the way it will work in hardware.

Case study: In the case of the proprietary blender imaging algorithm, the main data communication was done as parameters to the function (inputs) and a return value (output). However, closer inspection revealed that other communication was happening through the use of global variables in C. ("Global variables" are variables defined in a

scope above the design function(s).) The testbench would periodically change the mode as well as some other configuration parameters by writing to these global variables.

In order to turn this into hardware, the global variables had to be made visible to the hardware. In this case, the designer simply created a port for each global variable.

If there were many configuration parameters, this could have been done more efficiently by using address and data ports and implementing a small register file in the hardware. Of course, that would have involved additional coding changes. (Often, hardware efficiency comes at the price of having to do more design work. Behavioral design doesn't change that fundamental truth.)

Making the design synthesizable into hardware

The previous steps have identified the design and cleanly separated it from its environment. Now, the real hardware design work starts.

In order to make the behavioral design correctly model hardware, some hardware constructs need to be added. For example, a clock port must be added, and a reset port is almost always desired. Additionally, the designer must decide how the hardware will communicate with its environment. Communication mechanisms could involve adding handshakes for I/O protocols, communicating through shared memories on a bus (which have their own protocols), or simple un-timed reads and writes to ports.

However, it is strongly suggested that the designer does not use the latter frivolously, as I/O accesses without handshakes can often add lock-step timing dependencies between the design and testbench. These can be hard to maintain after behavioral synthesis adds additional clock edges to the design.

Also required in this step is the removal of "pure software" constructs that don't have an efficient analog in hardware. These include unbounded recursion and dynamic memory allocation that can't be statically determined.

Case study: A design group was using behavioral design to implement a proprietary deinterlacing algorithm, which required a significant amount of memory accessing and I/O handshaking. In order to turn the design into hardware, the behavioral design had to be modified to communicate correctly with its environment.

First, the design group had to choose the correct I/O and memory protocols to work with the remainder of the system. Then, in order to make the accesses more modular, they wrapped these protocols inside of function calls, which is a good C programming practice. Most interestingly, they found it easiest to create classes for the ports and memories, and then use the object-oriented nature of C++ to hide all the details of the protocols.

A simple example of various levels of communication abstractions is shown in Figure 6. Note that the original “no protocol” version is actually the exact same code as the “abstracted into classes” version. The only difference is in the definition of the port types.

	No protocol	Detailed protocol	Abstracted into functions	Abstracted into classes
Memory write	Mem[i] = data;	{ Mem_WE = 1; Mem_addr = l; Mem_d = data; Wait(1); Mem_we = 0; Wait(1); }	Mem_put(l, data);	Mem[i] = data;
Port read	Foo = in;	Do { Foo=in.read(); Vld = vld.read(); Wait(1); } while (!vld);	In = Get();	Foo = in;

Figure 6: Protocol modeling at different abstraction levels

Note that whichever level of abstraction they used, they were actually getting a bus-cycle-accurate simulation.

In the end, this allowed them to add hardware detail to the behavioral design without changing the algorithmic code at all. This C++ methodology has been adopted for other designs at this customer.

Optimizing the design

Once the above steps are done, the designer is able to use behavioral synthesis to create multiple RTL implementations of the design to try to meet various Quality of Results goals, such as area, throughput, latency and power consumption.

However, we have often found it is the case that additional design work (as compared to “pushing the button”) must be done in order to meet the desired goals. These are typically the same types of design decisions that would have been done in an RTL flow.

These types of optimizations are extremely design-specific, so this is best illustrated with a final case study.

Case study: In the case of the proprietary deinterlacing design, additional memory access optimizations were required. Calculating the correct values for a given line required reading values from multiple lines in multiple frames, and each frame was stored in a separate frame buffer. In the original algorithm, the calculation for each output pixel accessed all the relevant lines of all of the frames.

However, in the actual hardware architecture, the frames were being stored in DRAM's, and there was a significant latency penalty (~100 cycles) to switch between lines in the DRAM. To improve performance, the designer buffered the lines from each frame so that only one line from each frame is being read for each output line.

For example, instead of reading from N lines of the previous frame for each output line, the improved design read from one line of the previous frame and $N-1$ locally buffered lines (originally read for a previous line). This saved roughly $(N-1)*100$ cycles of overhead caused by DRAM line switching. The cost was the area of the $N-1$ local line buffers.

Note that this was a trade-off that could only be done by a designer. Only the designer knew the different timings of the DRAM based on access patterns. Also, the designer had to decide that increased local storage (and therefore area) was acceptable in order to improve performance.

CONCLUSION

The paper illustrates an effective behavioral design methodology that enables increased productivity and flexibility while maintaining excellent quality of results. When adopting a new methodology, it is important to understand how the work flow will be changed. The above case studies illustrate that while overall design time can be cut in half or more, there is still a significant amount of design work to be done. Perhaps the main difference between the behavioral and RTL design flows is that the bulk of the design time is now spent doing design trade-offs instead of coding.

REFERENCES

[Johnson98] Johnson, David, et.al. "Design automation of a receiver: breaking the RTL cycle Time barrier using Behavioral Compiler." *DesignCon98*, January 1998.

[Meredith04] Meredith, Michael. "A look inside behavioral synthesis." *EEdesign.com*, April 8, 2004.

[Moussa98] Moussa, Imed, et.al. "Comparing RTL and behavioral design methodologies in the case of a 2M-transistor ATM shaper." *36th ACM/IEEE Design Automation Conference*, June 1998.