Metrics-based Behavioral Design: A Methodology for Quickly Realizing High Quality Hardware for Signal Processing Applications

David J. Pursley
Forte Design Systems
1501 Reedsdale Street #302
Pittsburgh, PA 15233 USA
+1-412-321-4350 x15
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 systems-on-chip. 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.

This paper discusses a metrics-based behavioral design methodology that allows designers to create hardware (ASIC, FPGA or SoC) from an implementation-independent C/C++ algorithm. We define behavioral design metrics that can be used to quantify certain characteristics of the design, and those metrics are used to predict the impact of certain types of transformations. This metrics-based approach allows designers to concentrate their efforts in areas where they will have the most impact. This makes the overall flow faster, more predictable, and lower risk.

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

We start by defining behavioral synthesis and then place it in the context of a behavioral design flow. Design metrics will be defined and used as a predictor in this methodology. The goal of this paper is to give the reader an understanding of the methodology benefits of metrics-based behavioral design.

Previous work

Behavioral hardware design can be defined as the systematic process of getting from a target-independent behavioral model (often a software model) to optimized hardware. Definition of this process was described in [Pursley05], and we will revisit it briefly below to provide context for this methodology.

A significant amount of work has been done in design metrics, although most has been targeted at the gate-level or below. [Farrahi00] discusses several metrics that can be applied to measure design quality of gate-level designs. [Keating00] proposes design complexity as a metric to measure design quality. Keating's work was done at the netlist-level, and we will propose behavioral complexity metrics in this paper.

The METRICS effort uses datamining to determine predictive metrics on not only design quality but also process quality [Fenstermaker00].

The goal of our line of research is to produce design and process metrics that apply to the behavioral level of hardware design. Given that commercial use of behavioral design processes is fairly new, there is no previous work in this area. The most applicable research is in the area of cyclomatic complexity, a metric of software complexity, dating back to [McCabe76] with much follow-on research.

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.

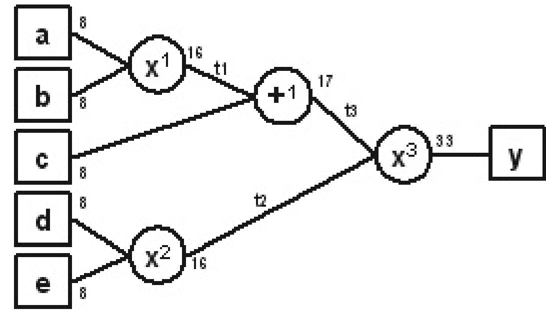

Figure 2: CDFG of operations

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.

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	
20x20=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, behavioral synthesis would also determine the FSM and registers needed to implement this design.

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 and then use behavioral metrics to improve the predictability of the flow.

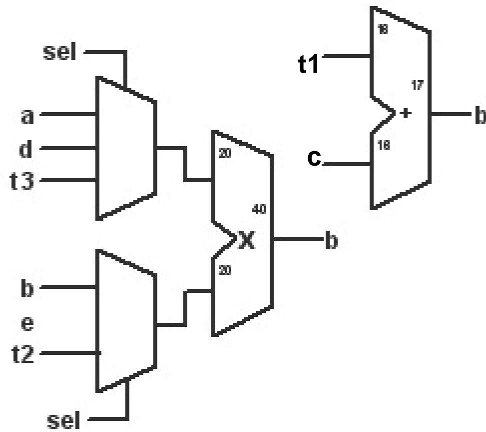

Figure 5: Minimum area RTL

Behavioral Design

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, as shown in Figure 6.

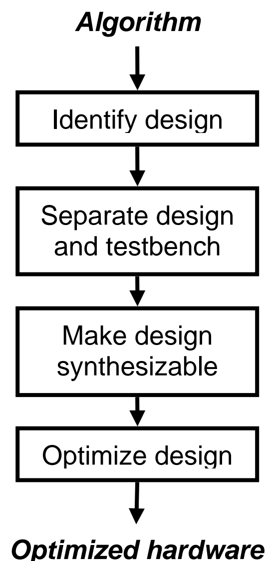

Figure 6: Behavioral design process

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. Designers need to be able to simulate a design at each stage of the process in order to verify the changes (generally, additional details) committed in each step.

<u>Identify design</u>

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.

<u>Separating design and testbench</u>

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.

<u>Making the design synthesizable into hardware</u>

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.

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, such as optimizing memory access patterns.

Metrics-based Behavioral Design

The goal of our line of research is to produce design and process metrics that apply to the behavioral level of hardware design. The final goal is to correlate input metrics, metrics based solely on the inputs to the behavior design process, to final quality of results (QoR) metrics. This will allow designers to have confidence that a given input description will eventually meet design goals after going through the behavioral design shown in Figure 6.

Because behavioral design starts at a higher level of abstraction, farther away from hardware, we are taking a step-by-step approach to creating our metrics-based design process. In brief, this line of research will progress by creating of chain of metrics-based predictors. This is shown in graphical form in Figure 7.

The goal is to find predictive relationships for fairly small steps in the behavioral design process and then link them together in a chain. That is, if A->B ("A implies B") and B->C, then A->C. One challenge in this approach is that there is a confidence factor with each "implies" statement, because each statement is based on statistical relationships. A more correct wording of the relationship is the "Given A, you have an X% chance of getting B." If the level of confidence of each link in the chain is not high enough, the overall predictability of the chain will be very low. This can be mitigated somewhat by collapsing adjacent links and looking for higher confidence relationships once the relevant factors have been identified.

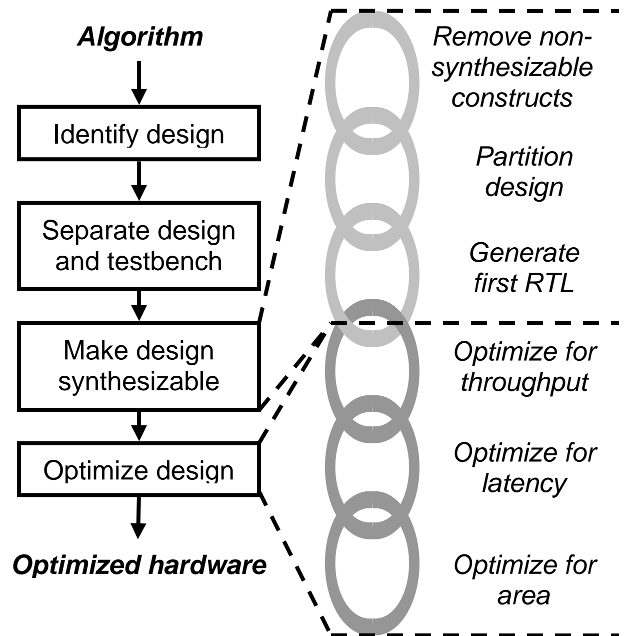

Figure 7: Links of subtasks in behavioral design chain

Results

We started this effort by focusing on metrics for the link in the chain "Generate First RTL." Metrics-based analysis for this subtask will help designers by giving them some predictability to know if the description they are modifying will be processed by behavioral synthesis software. Note that this step is the lynchpin of behavioral design process, as this is when the design turns form being target-independent to a hardware description. Anecdotally, it also appears that this is the most difficult step for new designers unfamiliar with behavioral design.

The ideal metric would be a synthesis tool-independent metric that looks only at the input behavioral source code. One example of this type of metric is cyclomatic complexity, which looks at the overall structure of the control-dataflow graph (CDFG) to compute a quality metric [McCabe76]. However, a fundamental problem with this

approach is that the CDFG is often significantly changed by the compiler-type optimizations applied by the behavioral synthesis tool. For example, loops may be unrolled or merged, functions may be inlined, constant propagation and dead code elimination may cause portions of the CDFG to be optimized away, etc. [Forte05]. An initial analysis confirmed that the cyclomatic complexity of the input behavioral code appeared to be unrelated to run time or memory footprint of behavioral synthesis.

A better metric may be cyclomatic complexity of the CDFG after the bulk of compiler-type optimizations have been applied, but there was no easy way to get that information from the tool we were using. Instead, we used an internal operation count, which included both control and datapath operations. This is similar to counting the number of nodes in the CDFG, although not quite the same. We correlated that metric to normalized CPU time, which is the best indicator of whether or not a job is likely to complete in a timely fashion.

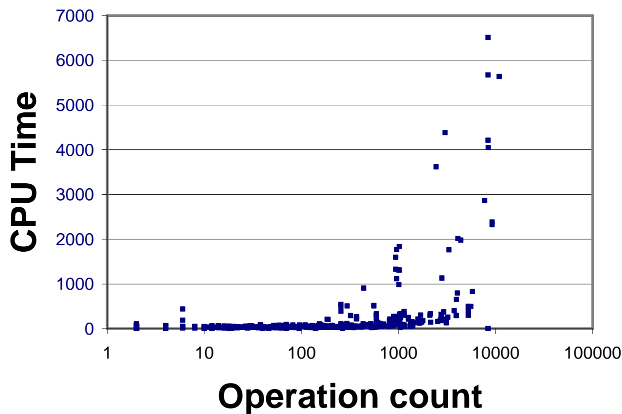

Figure 8 – CPU Time as a function of total operation count (log scale)

Analysis for correlation was performed across a collection of 1200 test cases that were industrial user designs, which was part of a regression suite for the behavioral synthesis tool we were using. The results of this analysis are shown in Figure 8. Note that operation count is plotted in log scale.

Notice that this does show good correlation, between the CPU time and control-datapath operation count, but there were too many points close to zero to determine if a relationship held for smaller operation counts, say less than 700. Making CPU time a logarithmic scale also made the relationship much easier to see for smaller designs, as shown of Figure 9.

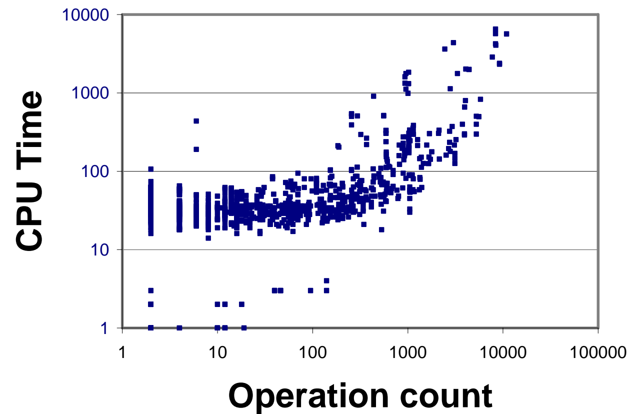

Figure 9 – Same as Figure 8 but with CPU Time on a logarithmic scale.

There appears to be two distinct regions in the graph. There is a nearly flat linear section up to approximately 700 operations. This may suggest that the start-up time dominates actual compute time for small designs, although analysis of why runtimes are affected is beyond the scope of this paper. The second segment starting at 700 operations is approximately linear, increasing with operation count.

Applying a simple piecewise linear model to the data yields an R^2 value of 0.56, indicating moderately strong correlation between CPU time and total operation count.

It is important to note that while the exact equations for CPU time as a function of total operation count will be tool-dependent, the fact that the metric is a predictor is tool-independent.

Future work

There are a number of areas that merit future work. First, cyclomatic complexity should be measured on the CDFG after compiler-type optimizations. While there are some tool challenges in getting that data from a synthesis tool in a way that can be analyzed, this would be a more synthesis tool-neutral metric than the CDFG operation count presented here.

Second, it is likely that correlation of this data can be improved by looking for other correlation factors such as command-line switches to the behavioral synthesis tool, etc. While these may make the results more specific to a given synthesis tool, it may uncover other synthesis tool-independent metrics such as control and dataflow operation count.

Third, the data shown here is form synthesis tool regression testing, which has some limitations. The largest limitation is that only successful tests are in this set of regressions. To accurately judge the predictability of run time from these metrics, you need to try this with designs that run very long or never complete. In such a case, you would hope the model predicts an appropriately long run time. Gathering this type of "real day-to-day usage" data is more problematic, as it strays into the social issues discussed in [Fenstermaker00]

Fourth, we would like to establish a link of metrics-based predictors for each link shown in Figure 7. Note that for the link-based analysis to be used, the predictability of each link may have to be higher than the R^2=0.56 determined for this metric.

Finally, process metrics will need to be introduced at the behavioral level to account for the iterative nature of each of the subtasks. While process metrics in general are not new, determining the tasks and developing a repeatable and verifiable behavioral design process model will be of significant value. While knowing that a given change is likely to increase or decrease QoR after the next behavioral synthesis run is important to know, it is more important to know if this design is likely to ever meet the overall QoR goals after going through the subtasks and iterations in the behavioral design task chain. Process metrics will help in that regard.

Conclusion

In this paper we have presented a methodology for metrics-based behavioral design, including a methodology for determining relevant behavioral metrics. We have used this methodology to determine that control and dataflow operation count is a moderately strong predictor for CPU time of behavioral synthesis tools. This is important because it suggests that a tool-independent metric can be applied to judge the quality of a design (quality being measured as time to first RTL, as defined in Figure 7). While the exact equations for CPU time as a function of total operation count will be tool-dependent, the fact that the metric is a predictor is tool-independent.

References

[Farrahi00] Farrahi, A.H., et.al. "Quality of EDA CAD tools: definitions, metrics, and directions," *IEEE International Symposium on Quality Electronic Design*, March 2000.

[Fenstermaker00] Fenstermake, S., et.al. "METRICS: a system architecture for design process automation," *ACM/IEEE Design Automation Conference*, June 2000.

[Forte05] Forte Design Systems. *Cynthesizer User's Guide v2.5*, http://www.ForteDS.com, August 2005.

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

[Keating00] Keating, M. "Measuring design quality by measuring design complexity," *IEEE International Symposium on Quality Electronic Design*, March 2000.

[McCabe76] McCabe, Thomas J. "A complexity measure," *IEEE Transactions on Software Engineering*, 2(4):308-320, December 1976.

[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." *ACM/IEEE Design Automation Conference*, June 1998.

[Pursley05] Pursley, David J. "Implementation independent design of a digital imaging algorithm using behavioral synthesis," *Embedded Systems Conference 2005*, March 2005.